CHAPTER 14

PARAMETERIZED DESIGN: PRINCIPLE

Design reuse is one of the major goals in developing VHDL code. Ideally, we want to design some common modules that can be shared by many applications. Since every application is different, it is desirable that a module can be customized to some degree to meet the specific need of an application. Customization is normally specified by explicit or implicit parameters, and we call this parameterized design. The most important parameter is the "width" of the module, which describes the number of bits of the data signal, as in a 24-bit adder. VHDL provides several mechanisms to pass and infer parameters and includes several language constructs to describe the replicated structure. In this chapter, we examine these basic mechanisms and constructs and use simple examples to illustrate their use. More detailed and comprehensive parameterized designs and case studies are discussed in Chapter 15.

14.1 INTRODUCTION

As the size of digital systems continues to grow, designing every system from scratch requires a tremendous amount of time and effort. One way to increase productivity and efficiency is design reuse. Many applications use parts of common functionalities. We can design and verify these parts once, store them in a library and then reuse them in other applications. As we discussed in Chapter 13, VHDL provides a versatile and powerful framework to facilitate the hierarchical design methodology and to accommodate predesigned components. Thus, once the commonly used parts are developed, design reuse can readily be incorporated into the VHDL environment.
While circuits share some parts of common functionalities, the exact specification of the part differs. For example, many applications need a binary counter. The basic construction of counters is similar, but the numbers of bits and the direction of the counting sequence depend on the need of a specific application. The chance that a fixed-size counter, say, an 11-bit up counter, will be reused is very small. On the other hand, if we develop a counter module that can be customized with different numbers of bits and counting directions, it can be utilized by many applications. The customization is normally done by describing certain circuit aspects with external parameters, and thus we call this parameterized design.

VHDL supports parameterized design in several ways. First, it provides mechanisms to pass parameters into an entity and to extract information from objects inside the entity. Second, most operators of VHDL and overloaded operators of the std_logic and numeric_std packages are defined over unconstrained arrays, which are "implicitly parameterized." Finally, VHDL has two language constructs, for generate and for loop, that can be used to describe replicated structures. The desired circuit width can be obtained by properly specifying the index range of these constructs.

14.2 TYPES OF PARAMETERS

In a parameterized design, we can broadly divide the parameters into width parameters and feature parameters. They are discussed in the following subsections.

14.2.1 Width parameters

For design-reuse purposes, we can classify a system's input and output signals into data signals and non-data signals. The clearest example is an FSMD system. The external signals that flow into and out of the data path are the data signals, and the clock and reset signals as well as the command and status signals are the non-data signals. For example, consider the sequential multiplier of Section 11.3.3. The a_in, b_in and r are the data signals and the clk, reset, start and ready are the non-data signals. Some combinational circuits, such as a multiplier or a barrel shifter, contain only data signals.

The widths of data signals normally can be modified to meet different requirements whereas the non-data signals need little or no revision. Again, consider the sequential binary multiplier. We can modify the design to process 16-, 24- or 32-bit operands. The width of the data signals (i.e., a_in, b_in and r) as well as the internal signals and registers will change accordingly. On the other hand, the non-data signals (i.e., clk, reset, start and ready) remain the same.

The width parameters of a parameterized design specify the sizes (i.e., number of bits) of the relevant data signals. A system may need one or more parameters to describe the sizes of input and output signals as well as the sizes of internal signals and registers. For example, the sequential binary multiplier requires one independent width parameter to specify the size of the operands (and the size of the product can be derived accordingly). The FIFO buffer requires two independent width parameters, one for the number of bits in a word and one for the number of words in a buffer.

The main goal of parameterized design is to describe the desired design in terms of the width parameters so that the same VHDL description can be used for applications with different size requirements. Since the sizes of the data signals can be increased or decreased, we also call this scalable design.
14.2.2 Feature parameters

In addition to width, we can use parameters to specify the structure or organization of a design. We call these feature parameters. The feature parameters are defined on an ad hoc basis. We normally use feature parameters to include or exclude certain functionalities (i.e., features) from the implementation or to select one particular version of the implementation.

A feature parameter is generally used to specify small variations within a design. For example, we can specify whether to include an output buffer for the output signal of an FSM, or whether to use a synchronous or asynchronous reset signal for a counter.

In theory, we can also use the feature parameters to select totally different implementations. For example, a counter may have several possible implementations, and we can use a parameter to choose binary counter–based implementation, Gray counter–based implementation, or LFSR-based implementation. To accommodate this, the corresponding VHDL code almost has the description of three independent designs. It may be better to code the three implementations in three separate architecture bodies and use a configuration to instantiate the desired implementation.

There is no definite rule about the use of the feature parameters and the configuration. When a feature parameter leads to significant modification or addition of the non-feature code, it is probably time to use separate architecture bodies and configurations. An example is given in Section 14.6.3.

14.3 SPECIFYING PARAMETERS

A parameterized design needs a mechanism to specify the parameters. There are several ways to do this in VHDL, including generics, array attribute and unconstrained array. Generics behave somewhat like parameters passing between the main program and a routine in a traditional programming language. Array attribute and unconstrained array derive the needed parameter values indirectly from a signal or port declaration.

14.3.1 Generics

We discussed generics in Section 13.3. They can be thought of as symbolic constants that are passed into the entity declaration. When the entity is used later as a component, the generics are assigned values during component instantiation.

Although a generic can assume any data type, only the integer data type is allowed in the IEEE 1076.6 RTL synthesis standard. While the integer data type is used mainly with a width parameter, we can also utilize it as a flag to specify the desired feature. For example, we can use the values of 0 and 1 to specify whether a buffer is needed for an output signal. Since the width parameter cannot be negative, we sometimes use the natural data type, which is a subtype of integer, for a generic.

In this chapter, we use the reduced-xor circuit to illustrate various concepts. The reduced-xor circuit applies xor operation over the elements of an array. For example, assume that the input signal is \( a_3a_2a_1a_0 \). The reduced-xor circuit performs the \( a_3 \oplus a_2 \oplus a_1 \oplus a_0 \) operation. In Section 5.6.2, this circuit was implemented by using a for loop statement. Since the original code was written with reuse in mind, it can easily be converted to parameterized design.

To utilize a generic, we need to replace the constant declaration in the original code with a generic declaration in the entity declaration. The parameterized code is shown in Listing 14.1.
Listing 14.1  Parameterized reduced-xor circuit using a generic

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity reduced_xor is
  generic(WIDTH: natural); -- generic declaration
  port(
    a: in std_logic_vector(WIDTH-1 downto 0);
    y: out std_logic
  );
end reduced_xor;

architecture loop_linear_arch of reduced_xor is
  signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  process(a,tmp)
  begin
    tmp(0) <= a(0); -- boundary bit
    for i in 1 to (WIDTH-1) loop
      tmp(i) <= a(i) xor tmp(i-1);
    end loop;
    y <= tmp(WIDTH-1);
  end process;
end loop_linear_arch;
```

14.3.2 Array attribute

A VHDL attribute provides information about a named item, such as a data type or a signal. We have used the `event` attribute, as in clk'event, to express the changing edge of the clk signal. There is a set of attributes associated with an object of an array data type. Let s be a signal with an array data type. The following attributes provide some information about the array:

- `s'left, s'right`: the left and right bounds of the index range of s.
- `s'low, s'high`: the lower and upper bounds of the index range of s.
- `s'length`: the length of the index range of s.
- `s'range`: the index range of s.
- `s'reverse_range`: the reversed index range of s.

Recall that the std_logic_vector, unsigned and signed data types are defined as array types. The attributes can be applied to the signals defined with these data types. For example, consider the following signals:

```vhdl
signal s1: std_logic_vector(31 downto 0);
signal s2: std_logic_vector(8 to 15);
```

The attributes of `s1` are

- `s1'left = 31; s1'right = 0;`
- `s1'low = 0; s1'high = 31;`
- `s1'length = 32;`
- `s1'range = 31 downto 0`
- `s1'reverse_range = 0 to 31`

The attributes of `s2` are
• s2'left = 8; s2'right = 15;
• s2'low = 8; s2'high = 15;
• s2'length = 8;
• s2'range = 8 to 15
• s2'reverse_range = 15 downto 8

These attributes provide information about the width and boundary of a signal. This information can be used as parameters in VHDL code. For example, we can rewrite the reduced-xor code in Listing 14.1 using the 'length attribute, as shown in Listing 14.2. The a'length returns the size of the a signal and plays the role of the previous WIDTH generic.

Listing 14.2 Parameterized reduced-xor circuit using an attribute

architecture attr_arch of reduced_xor is
  signal tmp: std_logic_vector(a'length-1 downto 0);
begin
  process(a, tmp)
  begin
    tmp(0) <= a(0);
    for i in 1 to (a'length-1) loop
      tmp(i) <= a(i) xor tmp(i-1);
    end loop;
  end process;
y <= tmp(a'length-1);
end attr_arch;

The range of the for loop can also be expressed in other attributes:
• for i in a'low+1 to a'high loop
• for i in a'right+1 to a'left loop

The last signal assignment statement of the code accesses the leftmost bit of the tmp signal. We can use the 'left attribute to obtain the left bound of the signal and rewrite the statement as

y <= tmp(tmp'left);

Since the WIDTH generic is included in the entity declaration, the relevant boundaries can be expressed clearly and concisely by the WIDTH generic, as in Listing 14.1. Use of the attributes is somewhat redundant and even cumbersome in this example. The real application of the array attributes is with the unconstrained array, which is discussed in the next subsection.

14.3.3 Unconstrained array

The std_logic_vector, unsigned and signed data types are the three main array types used in this book. They are defined as an unconstrained array internally. For example, in the std_logic_1164 package, the std_logic_vector data type is defined as follows:

type std_logic_vector is array(natural range <>) of std_logic;

It indicates that the data type of the index value must be natural, but it does not specify the exact bounds. If an object is declared with an unconstrained array data type, we must specify its index range (i.e., a constraint) when the data type is used, as 15 downto 0 in
signal x: std_logic_vector(15 downto 0);

The port declaration is considered a special case. The unconstrained array can be declared without specifying the range. For example, we can describe a register with no explicit range, as shown in Listing 14.3.

**Listing 14.3  Unconstrained D FF**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity unconstrain_dff is
  port(
    clk: std_logic;
    d: in std_logic-vector;
    q: out std_logic-vector
  );
end unconstrain_dff;

architecture arch of unconstrain_dff is
begin
  process (clk)
  begin
    if (clk'event and clk='l') then
      q <= d;
    end if;
  end process;
end arch;
```

Note that the data type for the d and q ports is std_logic-vector and no range is specified. The actual range of the std_logic-vector data type is inferred when an instance of unconstrain_dff is instantiated. The ranges of the actual signals become the ranges of the d and q signals. For example, the dff16 instance is instantiated as a 16-bit register in the code segment shown below.

```vhdl
signal din, qout: std_logic-vector(15 downto 0);
signal clk: std_logic;

dff16: unconstrain_dff
  port map(clk=>clk, d=>din, q=>qout);
```

In this mechanism, we can think that the width parameter is embedded in the actual signal and passed to the entity declaration when the corresponding component is instantiated.

Since no range is specified for d and q, the boundaries of the two signals will not be checked in the analysis stage. The following code segment is syntactically correct:

```vhdl
signal din: std_logic-vector(15 downto 0);
signal qout: std_logic-vector(7 downto 0);

dff_error: unconstrain_dff
  port map(clk=>clk, d=>din, q=>qout);
```

The error can only be detected during the elaboration or execution stage of the code. To make the design more robust, we may need to add error-checking code in the unconstraindff description to ensure that d and q have the same range when the component is instantiated.

The previous reduced-xor circuit can also be described without using an explicit range in the a signal. The VHDL code is shown in Listing 14.4. The description is basically patterned after the code in Listing 14.1. In the new code, the generic declaration is removed and the range of the a signal is omitted. The width parameter is inferred from the 'length attribute of the a signal and then declared as a constant in the declaration of the architecture body.

Listing 14.4 Parameterized reduced-xor circuit using an unconstrained array

```
library ieee;
use ieee.std_logic_1164.all;
entity unconstrain_reduced_xor is
  port(
    a: in std_logic_vector;
    y: out std_logic
  );
end unconstrain_reduced_xor;

architecture arch of unconstrain_reduced_xor is
  constant WIDTH : natural := a'length;
signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  process (a, tmp)
  begin
    tmp(0) <= a(0);
    for i in 1 to (WIDTH-1) loop
      tmp(i) <= a(i) xor tmp(i-1);
    end loop;
  end process;
  y <= tmp(WIDTH-1);
end arch;
```

The code appears to be correct at first glance. For example, if we map the a signal to an actual signal with the data type of std_logic_vector(7 downto 0) during component instantiation, the code functions as expected. However, since the range of the a signal is inferred from the actual signal, it is same as the actual signal. For an 8-bit actual signal, the following range specification formats are possible:

- std_logic_vector(7 downto 0)
- std_logic_vector(0 to 7)
- std_logic_vector(15 downto 8)
- std_logic_vector(8 to 15)

The code does not work properly for the last two formats.

One way to fix the problem is to assign the a signal to an internal signal of known format and use that signal in the code. This scheme is shown in Listing 14.5. We first assign a into an internal aa signal, whose range is specified as WIDTH-1 downto 0, and use it in the remaining architecture body.
Listing 14.5  Improved parameterized reduced-xor circuit using an unconstrained array

```vhdl
architecture better_arch of unconstrain_reduced_xor is
  constant WIDTH : natural := a'length;
  signal tmp: std_logic_vector(WIDTH-1 downto 0);
  signal aa: std_logic_vector(WIDTH-1 downto 0);
beginaa <= a;
  process(aa,tmp)
  begin
    tmp(0) <= aa(0);
    for i in 1 to (WIDTH-1) loop
      tmp(i) <= aa(i) xor tmp(i-1);
    end loop;
y <= tmp(WIDTH-1);
end better_arch;
```

14.3.4 Comparison between a generic and an unconstrained array

A generic and an unconstrained array are two mechanisms to convey width parameter information. The unconstrained array mechanism uses attributes to infer the relevant information from the actual signals. Since the width parameter is derived automatically, this mechanism is more general and flexible than the generic mechanism. However, the flexibility also introduces more opportunities for errors, as shown in the examples in Section 14.3.3.

To develop robust and reliable code for an unconstrained array, we must consider the different formats of range specifications and the potential width mismatch between various signals. These require comprehensive error-checking code to cover possible erroneous conditions. This code may become very involved and unnecessarily complicate the developing and coding process and even overshadow the real design issues. Unless a module is extremely general and widely used, the generic mechanism is satisfactory. We prefer to use the generic mechanism in this book. When the mechanism is more rigid, it clearly specifies the range, direction and width of each signal and avoids many subtle erroneous conditions. This allows us to focus on development of real hardware rather than on error checking.

14.4 CLEVER USE OF AN ARRAY

The logical, relational and arithmetic operators of VHDL and the overloaded operators of the std_logic_1164 and numeric_std packages are defined over unconstrained arrays and thus can be applied to arrays of any size. We can think that they are "implicitly parameterized." For example, consider the following code segment:

```
r <= a - b when a > b else
   a + b;
```

Since the +, - and > operators can accommodate any array sizes, the code is implicitly parameterized.

In a more sophisticated code, an element or a slice of array may be referred and a signal may be assigned or compared with a constant vector value. One key to developing parameterized design is to refrain from fixed-size references. Instead, the references should be expressed in terms of attributes or width parameters. We actually have followed this
practice from the beginning of the book. One early coding guideline is to use symbolic constants instead of hard literals. In a properly coded program, we can convert a regular design into a parameterized design by replacing symbolic constants with expressions derived from attributes and generics. The following subsections discuss techniques to achieve this goal and present several examples to illustrate use of these techniques. Lots of regular code can be modified and converted to parameterized descriptions by cleverly using the array data type.

14.4.1 Description without fixed-size references

As in input and output ports, we can classify the internal signals as data and non-data signals. Data signals normally have an array data type, such as std_logic_vector, unsigned or signed. To achieve parameterized design, we should try to use the width parameters or attributes to describe operations that involve data signals. Following are some techniques to avoid a fixed-size description.

Using named association for aggregates A signal or variable is frequently assigned with a fixed value, as in the initiation of a sequential system. For example, the following is the initialization statement of an 8-bit counter:

```vhdl
q_reg <= "00000000";
```

This statement must be revised every time when the width of the counter is modified. A better alternative is to use named association:

```vhdl
q_reg <= (others => '0');
```

This statement will remain the same regardless of the width of the counter. Other frequently used constant aggregates include all 1’s (i.e., "11...11"):

```vhdl
q_reg <= (others => '1');
```

and a single 1 in the LSB (i.e., "00...01"):

```vhdl
q_reg <= (0=>'1', others => '0');
```

The aggregate has to be assigned to an object of known size and cannot be used in an expression. For example, the following code segment attempts to check whether a is all-zero and is invalid:

```vhdl
signal a: std_logic_vector(WIDTH-1 downto 0)

x <= '1' when (a=(others=>'0')) else ...
```

One way to correct the problem is to use the range attribute to provide the size information:

```vhdl
x <= '1' when (a=a'range=>'0')) else ...
```

Another somewhat cumbersome, but more descriptive way is to define a constant for the all-zero conditions:

```vhdl
constant ZERO std_logic_vector(WIDTH-1 downto 0)
:= (others=>'0');

signal a: std_logic_vector(WIDTH-1 downto 0)

x <= '1' when (a=ZERO)) else ...
```
Using an integer and conversion function in an expression  If an object is with
the unsigned or signed data types, we can express a constant in integer format since the
relational and arithmetic operators are overloaded with the integer or natural data type. For
example, assume that the a signal is with the unsigned data type. Instead of using a
constant in the unsigned data type, as in
\[ x \leq '1' \text{ when } (a='00000110') \text{ else } ... \]
we can express the constant in the natural data type, as in
\[ x \leq '1' \text{ when } (a=6) \text{ else } ... \]
The constant 6 will be converted to the proper number of bits, and thus no revision is needed
when the width of the a signal changes.

If a constant is assigned to an object, we can convert the integer to the designated data
type by the width parameter. For example, assume that the x signal is with the unsigned
data type of WIDTH bits. A constant, say 6, can be assigned to x as
\[ x \leq \text{to-unsigned}(6, \text{WIDTH}); \]
When the integer 6 is converted to the unsigned type, the number of bits is automatically
adjusted with the WIDTH parameter.

We can do this for an object with the std_logic_vector data type with additional type
casting. For example, if we assume that the x signal is with the std_logic_vector data
type, the statement becomes
\[ x \leq \text{std_logic_vector(to-unsigned}(6, \text{WIDTH})); \]
Similarly, the previous a=ZERO expression can also be written as follows without using
the constant declaration
\[ a=\text{std_logic_vector(to-unsigned}(0, \text{WIDTH})) \]
Of course, the numeric_std package has to be invoked to use the unsigned data type.

Using the width parameter to refer to a slice or element in an array  Some
VHDL code must make reference to a single element or a slice of an array. The reference is
frequently the MSB or the LSB and is sometimes dependent on the width of the array. For
example, assume that src is with the signed(7 downto 0) data type and we use alias to
refer to the sign of the src signal:
\[ \text{alias sign: std_logic := src(7);} \]
Instead of a hard literal, we can use an attribute to refer to the MSB bit:
\[ \text{alias sign: std_logic := src(src'left);} \]
If the data type of src is already parameterized as signed(WIDTH-1 downto 0), we can
code the statement as
\[ \text{alias sign: std_logic := src(WIDTH-1);} \]
Similarly, instead of expressing rotating right 1 bit as
\[ \text{dest} \leq \text{src}(0) \& \text{src}(7 \text{ downto } 1); \]
we can write
\[ \text{dest} \leq \text{src(src'right)} \& \text{src(src'left downto src'right+1}); \]
This statement can work only if the size of src is larger than 2 and its range is in descending (i.e., downto) order. If the data type of src is already parameterized as signed (WIDTH-1 downto 0), the rotation operation can be coded in a more descriptive fashion with the WIDTH parameter:
\[
dest <= src(0) & src(WIDTH-1 downto 1);
\]

### 14.4.2 Examples

**Reduced-xor circuit** Several codes were developed for the fixed-size reduced-xor circuit in Section 7.4.1. In Listing 7.17, we used an auxiliary internal signal to represent the intermediate results and described the circuit in a compact, array format. The code can easily be converted to a parameterized design by replacing the constant with a generic. The entity declaration is the same as the one shown in Listing 14.1, and the architecture body is shown in Listing 14.6.

```
Listing 14.6 Parameterized reduced-xor circuit using a clever array representation

architecture array_arch of reduced_xor is
    signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
    tmp <= (tmp(WIDTH-2 downto 0) & '0') xor a;
y <= tmp(WIDTH-1);
end array_arch;
```

**Reduced-and circuit** The reduced-and circuit applies and operation to the elements of an array. For example, if the input signal is \(a_3a_2a_1a_0\), the reduced-and circuit generates the result of \(a_3 \cdot a_2 \cdot a_1 \cdot a_0\).

While the reduced-and circuit can be implemented using methods similar to those of the reduced-xor circuit, we use a different approach in this example. The design is based on the observation that the reduced-and circuit returns '1' only when all the inputs are '1'. The code is shown in Listing 14.7. The key is the Boolean condition \(a=(a'\text{range}=>'1')\). We use the 'range' attribute to obtain the range of the \(a\) signal and then construct an aggregate \((a'\text{range}=>'1')\), in which all elements are assigned to '1' (i.e., "1...1"). The \(a=(a'\text{range}=>'1')\) expression returns true only when the \(a\) signal consists of only 1's.

```
Listing 14.7 Parameterized reduced-and circuit using a clever array representation

library ieee;
use ieee.std_logic_1164.all;
entity reduced_and is
    generic(WIDTH: natural);
    port(
        a: in std_logic_vector(WIDTH-1 downto 0);
y: out std_logic
    );
end reduced_and;
architecture array_arch of reduced_and is
begin
    y <= '1' when a=(a'\text{range}=>'1') else '0';
end array_arch;
```
**Serial-to-parallel converter** A serial-to-parallel converter accepts input data serially and stores the data in a shift register. Since the output of the register can be accessed simultaneously, the serial data is converted into parallel format. A parameterized design is shown in Listing 14.8.

**Listing 14.8** Parameterized serial-to-parallel converter using a clever array representation

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity s2p_converter is
  generic(WIDTH: natural);
  port(
    clk: in std_logic;
    si: in std_logic;
    q: out std_logic_vector(WIDTH-1 downto 0)
  );
end s2p_converter;
architecture array_arch of s2p_converter is
  signal q_reg, q_next: std_logic_vector(WIDTH-1 downto 0);
begin
  process(clk)
  begin
    if (clk'event and clk='l') then
      q_reg <= q_next;
    end if;
  end process;
  q_next <= si & q_reg(WIDTH-1 downto 1);
  q <= q_reg;
end array_arch;
```

**Adder with status circuit** In Section 7.5.3, we discussed a general adder circuit that contains a carry-in signal and various status output signals. To process these extra signals, the adder is expanded by 2 bits internally. We can convert this circuit into a parameterized design, as shown in Listing 14.9. Note that the WIDTH generic is used to access the various elements of the array.

**Listing 14.9** Parameterized adder with status circuit

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity para_adder_status is
  generic(WIDTH: natural);
  port(
    a, b: in std_logic_vector(WIDTH-1 downto 0);
    cin: in std_logic;
    sum: out std_logic_vector(WIDTH-1 downto 0);
    cout, zero, overflow, sign: out std_logic
  );
end para_adder_status;
architecture arch of para_adder_status is
  signal a_ext, b_ext, sum_ext: signed(WIDTH+1 downto 0);
```
signal ovf: std_logic;
alias sign_a: std_logic is a_ext(WIDTH);
alias sign_b: std_logic is b_ext(WIDTH);
alias sign_s: std_logic is sum_ext(WIDTH);
begin
  a_ext <= signed('0' & a & '1');
b_ext <= signed('0' & b & cin);
sum_ext <= a_ext + b_ext;
  ovf <= (sign_a and sign_b and (not sign_s)) or
         ((not sign_a) and (not sign_b) and sign_s);
cout <= sum_ext(WIDTH+1);
sign <= sign_s when ovf='0' else
         not sign_s;
zero <= '1' when (sum_ext(WIDTH downto 1)=0
                   and ovf='0') else
        '0';
overflow <= ovf;
sum <= std_logic_vector(sum_ext(WIDTH downto 1));
end arch;

Ring counter  We studied two possible implementations of an 8-bit ring counter in Section 9.2.2. The first implementation uses the reset signal to initialize the counter to "0000000011". The parameterized version is shown in the reset-arch architecture of Listing 14.10. The second implementation is a self-correcting design. In the original code, a '1' is inserted into the serial input when all 7 MSBs are 0's. For the parameterized version, the WIDTH-1 MSBs need to be checked. The VHDL code is shown in the self_correct-arch architecture of Listing 14.10. We create the r_high alias to represent the WIDTH-1 MSBs and use the r_high=(r_high'range=>'0') expression to check the all-zero condition.

Listing 14.10  Parameterized ring counter

library ieee;
use ieee.std_logic_1164.all;
entity para_ring_counter is
generic(WIDTH: natural);
port(
  clk, reset: in std_logic;
  q: out std_logic_vector(WIDTH-1 downto 0)
);
end para_ring_counter;

architecture using asynchronous initialization
architecture reset_arch of para_ring_counter is
  signal r_reg: std_logic_vector(WIDTH-1 downto 0);
  signal r_next: std_logic_vector(WIDTH-1 downto 0);
begin
  register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= (0=>'1',others=>'0');
    elsif (clk'event and clk='1') then
      r_next <= r_reg;
      r_reg <= r_next;
    end if;
  end process;
end reset_arch;
r_reg <= r_next;
end if;
end process;

-- next-state logic
r_next <= r_reg(0) & r_reg(WIDTH-1 downto 1);

-- output logic
q <= r_reg;
end reset_arch;

architecture using self-correcting circuit
architecture self_correct_arch of para_ring_counter is
signal r_reg: std_logic_vector(WIDTH-1 downto 0);
signal r_next: std_logic_vector(WIDTH-1 downto 0);
signal s_in: std_logic;
alias r_high: std_logic_vector(WIDTH-2 downto 0) is r_reg(WIDTH-1 downto 1);
begin
-- register
process(clk, reset)
begin
if (reset='1') then
 r_reg <= (others=>'0');
elif (clk'event and clk='1') then
 r_reg <= r_next;
end if;
end process;
-- next-state logic
s_in <= '1' when r_high=(r_high'range=>'0') else
 '0';
r_next <= s_in & r_reg(WIDTH-1 downto 1);
-- output logic
q <= r_reg;
end self_correct_arch;

14.5 FOR GENERATE STATEMENT

The generate statements are concurrent statements with embedded internal concurrent statements, which can be interpreted as a circuit part. There are two types of generate statements. The first type is the for generate statement, which is used to create a circuit by replicating the hardware part. The second type is the conditional or if generate statement, which is used to specify whether or not to create an optional hardware part. The generate statements are especially useful for the parameterized design. This section discusses the for generate statement and the next section covers the conditional generate statement.

Many digital circuits can be implemented as a repetitive composition of basic building blocks. They frequently exhibit a regular structure, such as a one-dimensional cascading chain, a tree-shaped connection or a two-dimensional mesh. Since we can easily expand the structure by increasing the number of iterations, these circuits are natural for the parameterized design. The for generate statement is used to describe this kind of circuit.
14.5.1 Syntax

The simplified syntax of the for generate statement is:

```
gen_label:
for loop_index in loop_range generate
    concurrent statements;
end generate;
```

The for generate statement is somewhat similar to the basic for loop statement discussed in Chapter 4. The for generate statement repeats the loop body of concurrent statements for a fixed number of iterations. The loop_range term specifies a range of values between the left and right bounds. The range has to be static, which means that it has to be determined by the time of execution (synthesis). It is normally specified by the width parameters. The loop_index term is used to keep track of the iteration and takes a successive value from loop_range in each iteration, starting from the leftmost value. The index automatically takes the data type of loop_range's element and does not need to be declared. The gen_label term is mandatory. It is the label used to identify this particular generate statement.

The loop body contains a collection of concurrent statements, which may include other generate statements. The concurrent statements describe a stage of the iterative circuit. A stage description is composed of two main ingredients. One is the description of the basic building block and the other is the input--output connection pattern between the blocks. The connection pattern is normally specified by a collection of internal signals, which are represented as a one- or two-dimensional array with loop_index in their index expression.

The key to designing an iterative circuit is to identify the basic block and connection pattern of a stage. To determine the connection pattern and to describe the relationship between the input and output signals of successive stages, we can first draw a small-scale circuit diagram, label a few specific connection signals and then derive the general relationship.

14.5.2 Examples

**Binary decoder** A binary 2-to-2^n decoder is a circuit that asserts one of the 2^n possible output signals. The codes of a 2-to-2^2 decoder are shown in Chapters 4 and 5. While these codes are simple, none can be modified for parameterized design.

One way to view the binary decoder is to treat each bit of the decoded output as the result of a constant comparator. The decoded bit is asserted when the value of the input signal matches the hardwired constant value. The block diagram of a 2-to-2^2 decoder is shown in Figure 14.1.

Note that since one input of the comparator is a constant (i.e., hardwired), it can be simplified during synthesis. This diagram can easily be replicated with a different input width. In the i-th stage, code(i) is asserted when the binary value of the a input is equal to i. This can be translated into the VHDL statement:

```
code(i) <= '1' when a=std_logic_vector(unsigned(i)) else '0';
```

The parameterized VHDL code using the for generate statement is shown in Listing 14.11.
Listing 14.11 Parameterized binary decoder using a for generate statement

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity bin_decoder is
    generic(WIDTH: natural);
    port(
        a: in std_logic_vector(WIDTH-1 downto 0);
        code: out std_logic_vector(2**WIDTH-1 downto 0)
    );
end bin_decoder;

architecture gen_arch of bin_decoder is
begin
    comp_gen:
        for i in 0 to (2**WIDTH-1) generate
            code(i) <= '1' when i=to_integer(unsigned(a)) else '0';
        end generate;
end gen_arch;
```

Note that we have to include the numeric_std package to use the unsigned data type.

**Reduced-xor circuit** As discussed in Section 7.4.1, the reduced-xor circuit can be implemented as a cascading chain or a tree. The block diagram of an 8-bit cascading-chain implementation is shown again in Figure 14.2. The diagram exhibits a regular, iterative pattern and thus is a good match for the for generate statement description. The building block is the xor gate. We divide the chain into stages and number the stages from left to right, starting at the 0th stage. The internal signals connect the output of the current stage to the input of the next stage. These signals are arranged as an array and the `tmp(i)` signal represents the output of the `(i-1)`th stage and the input to the `i`th stage, as shown in Figure 14.2. From the diagram, we can see that there is a clear relationship among the two input signals and the output signal of an xor gate. For the `i`th stage, the three signals can be expressed as

\[ \text{tmp}(i+1) \leq \text{tmp}(i) \text{ xor } a(i+1); \]
Figure 14.2  Block diagram of a reduced-xor circuit.

Note that the statement shows the basic building block (i.e., xor gate) and the interconnection between blocks.

Base on the equation, we can derive the VHDL code using a for generate statement. The code is shown in Listing 14.12. The loop body iterates WIDTH-1 times and thus infers WIDTH-1 xor gates.

Listing 14.12  Parameterized reduced-xor circuit using a for generate statement

```vhdl
architecture gen_linear_arch of reduced_xor is
  signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  tmp(0) <= a(0);
  for i in 1 to (WIDTH-1) generate
    tmp(i) <= a(i) xor tmp(i-1);
    y <= tmp(WIDTH-1);
  end generate;
end gen_linear_arch;
```

In an iterative structure, the boundary stages interface to the external input and output signals, and sometimes their connections are different from the regular blocks. Note that we use two special statements to handle the boundary signals of the leftmost and rightmost stages.

The code here and the array-based code in Listing 14.6 actually specify the same circuit structure. While the former describes the design stage by stage, the latter lumps the signals together in a single array.

Serial-to-parallel converter  We discussed a simple serial-to-parallel converter in Section 14.4.2. It is composed of a series of cascading D FFs, and the conceptual diagram of a 4-bit implementation is shown in Figure 14.3. Each stage consists of a D FF and next-state logic, which is a wire that connects the output of the previous D FF to the input of the current D FF.

The first VHDL description is shown in Listing 14.13. To accommodate the naming convention, we extend the q_reg signal by one extra bit and assign the external si signal to q_reg(WIDTH). Since q_reg(WIDTH) is not assigned inside the for generate statement, only WIDTH-1 D FFs will be inferred. The loop body consists of two concurrent statements. One is the process for the D FF and the other is the next-state logic. Even the next-state
logic is very simple; we still follow synchronous design practice and separate it from the memory element.

Listing 14.13 Parameterized serial-to-parallel converter using a for generate statement

```vhdl
architecture gen_proc_arch of s2p_converter is
    signal q_next: std_logic_vector(WIDTH-1 downto 0);
    signal q_reg: std_logic_vector(WIDTH downto 0);
begin
    q_reg(WIDTH) <= si;
    for i in (WIDTH-1) downto 0 generate
        process(clk)
        begin
            if (clk'event and clk='l') then
                q_reg(i) <= q_next(i);
            end if;
        end process;
        q_next(i) <= q_reg(i+1);
    end generate;
    output
    q <= q_reg(WIDTH-1 downto 0);
end gen_proc_arch;
```

Alternatively, we can also define the DFF as an entity and use it through component instantiation. The VHDL codes for the DFF and the alternative description are shown in Listing 14.14. The code is essentially the structural description of the block diagram in Figure 14.3.

Listing 14.14 Alternative serial-to-parallel converter using a for generate statement

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dff is
    port(
        clk: in std_logic;
```
architecture arch of dff is
begin
  process (clk)
  begin
    if (clk'event and clk='l') then
      q <= d;
    end if;
  end process;
end arch;

architecture using component instantiation
architecture gen_comp_arch of s2p_converter is
  signal q_reg: std_logic_vector (WIDTH downto 0);
  component dff
    port(
      clk: in std_logic;
      d: in std_logic;
      q: out std_logic
    );
  end component;
begin
  q_reg(WIDTH) <= si;
  dff_gen:
  for i in (WIDTH-1) downto 0 generate
    dff_array: dff
      port map (clk=>clk, d=>q_reg(i+1), q=>q_reg(i));
  end generate;
  q <= q_reg(WIDTH-1 downto 0);
end gen_comp_arch;

14.6 CONDITIONAL GENERATE STATEMENT

14.6.1 Syntax

The conditional generate statement is used to specify an optional circuit that can be included or excluded in the final implementation. It can be used to realize the feature parameters of a parameterized design. The simplified syntax of the conditional generate statement is

    gen_label:
    if boolean_exp generate
      concurrent statements;
    end generate;

The boolean_exp is an expression that returns a value with the boolean data type. If it is true, the internal concurrent statements are invoked, which means that the circuit described by the concurrent statements will be included in the implementation. If the expression is false, no concurrent statement is invoked, and thus the corresponding circuit is excluded.
from the implementation. Note that there is no else branch. If we want to include one of the two possible circuits in an implementation, we must use two separate if generate statements. The gen_label term is the label and is mandatory.

For synthesis purposes, the boolean_exp expression must be static so that synthesis software knows whether the corresponding concurrent statements should be included in the physical implementation. The expression is normally described in terms of generics.

14.6.2 Examples

Reduced-xor circuit revisited One common use of the conditional generate statement is to describe the “irregular” stages in a for generate statement. Consider the VHDL code for the reduced-xor circuit in Listing 14.12. The first and last stages are different from others because they interface with the external input and output signals, which use different name conventions. Two statements are used to rename the signals:

\[
\text{tmp}(0) <= \text{a}(0);
\]
\[
y <= \text{tmp}(\text{WIDTH}-1);
\]

To eliminate these statements, we can use conditional generate statements inside the for generate statement. Each Boolean expression of a conditional generate statement represents a specific condition and specifies what kind of circuit should be generated for the corresponding stages. In this design, there are three kinds of stages: the leftmost stage, regular middle stages and the rightmost stage. The if generate statement can check the stage number and then generate a circuit that matches the naming convention accordingly. The VHDL code is shown in Listing 14.15.

Listing 14.15 Parameterized reduced-xor circuit with a conditional generate statement

```vhdl
architecture gen_if_arch of reduced_xor is
signal tmp: std_logic_vector(WIDTH-2 downto 1);
begnin
xor_gen:
  for i in 1 to (WIDTH-1) generate
    -- leftmost stage
    left_gen: if i=1 generate
      tmp(i) <= a(i) xor a(0);
    end generate;
    -- middle stages
    middle_gen: if (i < i) and (i < (WIDTH-1)) generate
      tmp(i) <= a(i) xor tmp(i-1);
    end generate;
    -- rightmost stage
    right_gen: if i=(WIDTH-1) generate
      y <= a(i) xor tmp(i-1);
    end generate;
end gen_if_arch;
```

Up-or-down free-running binary counter An up-or-down binary counter is a counter that can be instantiated in a specific mode. Note that the “or” here means that only one mode of operation, either counting up or counting down but not both, can be implemented in the final circuit. We use the UP generic as the feature parameter to specify the desired mode.
The counter counts up if UP is 1 and counts down otherwise. Since there are two possible features, the boolean data type will be more appropriate for the UP generic. However, since the IEEE RTL synthesis standard and some software accept only the integer data type and its subtypes, the natural type is used.

The conceptual block diagram of this counter is shown in Figure 14.4(a). We use dashed blocks to indicate the optional features of a circuit, such as the incrementor and decrementor in the diagram. In this particular example, only one of the dashed blocks will be used in synthesis, and thus there is no output confliction for the r_next signal. The VHDL code is shown in Listing 14.16.

**Listing 14.16**  Up-or-down free-running binary counter

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity up_or_down_counter is
  generic(
    WIDTH: natural;
    UP: natural
  );
port(
```
Parameterized Design: Principle

clk, reset: in std_logic;
q: out std_logic_vector(WIDTH-1 downto 0);
end up_or_down_counter;

architecture arch of up_or_down_counter is
signal r_reg: unsigned(WIDTH-1 downto 0);
signal r_next: unsigned(WIDTH-1 downto 0);
begin
  -- register
  process(clk, reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  inc_gen: -- incrementor
  if UP=1 generate
    r_next <= r_reg + 1;
  end generate;
  dec_gen: -- decrementor
  if UP/=1 generate
    r_next <= r_reg - 1;
  end generate;
  -- output logic
  q <= std_logic_vector(r_reg);
end arch;

The two next-state logics are described by the two separated if generate statements. Note that the Boolean expressions of the two statements are complementary, and thus only one circuit will be generated.

For comparison purposes, let us examine a dual-mode binary counter that counts in both up and down directions. The mode is specified by an additional mode input signal. This implementation includes an incrementor and a decrementor, and uses a multiplexer to select the desired result, as shown in Figure 14.4(b). The corresponding VHDL code is listed in Listing 14.17.

Listing 14.17  Up-and-down free-running binary counter

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity up_and_down_counter is
  generic (WIDTH: natural);
  port(
    clk, reset: in std_logic;
    mode: in std_logic;
    q: out std_logic_vector(WIDTH-1 downto 0)
  );
end up_and_down_counter;
Counter with an optional output buffer

An output buffer can remove glitches from the signal. Since the buffer is only needed for certain applications, it will be convenient to include the buffer as an optional part of the circuit. This can be achieved by using a feature parameter and conditional generate statements. Consider a binary counter that has a pulse output signal that is activated when the counter reaches 0. We can use the BUFF generic to indicate whether a buffer should be inserted. The conceptual diagram is shown in Figure 14.5 and the VHDL code is shown in Listing 14.18.

Listing 14.18 Counter with an optional output buffer
FSM with a selectable clear signal  In a sequential circuit, we usually include a clear signal to perform system initialization. The clear signal can be either synchronous or asynchronous, and the choice sometimes depends on the target device technology. To make the code portable, it is beneficial to include a generic to specify the type of clear signal to be synthesized. Implementing the asynchronous clear is straightforward. We just replace the state register by a register with an asynchronous reset signal. Implementing the synchronous clear needs to revise the next-state logic of the sequential circuit. To minimize the modification, we can wrap the original next-state logic with a 2-to-1 multiplexer. The conceptual diagram is shown in Figure 14.6. The initial state value (assume that it is idle) will be routed to the register if the synchronous clear signal is asserted.
We use the memory controller FSM of Chapter 10 to demonstrate the design. To accommodate the “clear” feature, we must selectively generate circuits in two places, as shown in Figure 14.6. One is the register, which can be a register with or without the asynchronous reset signal. The other is the optional multiplexer to route the idle value to the state_next signal. We introduce the SYNC generic to specify the desired type of clear and use two if generate statements to create the corresponding circuits. The VHDL code is shown in Listing 14.19.

Listing 14.19 Memory controller FSM with a selectable clear signal

```
library ieee;
use ieee.std_logic_1164.all;
entity clr_mem_fsm is
generic (SYNC: integer);
port(
  clk, clear: in std_logic;
  mem, rw, burst: in std_logic;
  oe, we: out std_logic
);
end clr_mem fsm ;

architecture mult_seg_arch of clr_mem_fsm is
type mc_state_type is
  (idle, read1, read2, read3, read4, write);
signal state_reg, state_i, state_next: mc_state_type;
beg
  state register
  reset_ff_gen: — register with asynchronous clear
  if SYNC/=1 generate
  begin
    if (clear='1') then
      state_reg <= idle;
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
    end if;
```

Figure 14.6 Block diagram of FSM with a selectable clear signal.
no_reset_ff_gen: register without asynchronous clear
if SYNC=1 generate
  process(clk)
  begin
    if (clk'event and clk='l') then
      state_reg <= state_next;
    end if;
  end process;
end generate;

next-state logic
process(state_reg,mem,rw,burst)
begin
  case state_reg is
    when idle =>
      if mem='l' then
        if rw='l' then
          state_i <= read1;
        else
          state_i <= write;
        end if;
      else
        state_i <= idle;
      end if;
    when write =>
      state_i <= idle;
    when read1 =>
      if (burst='l') then
        state_i <= read2;
      else
        state_i <= idle;
      end if;
    when read2 =>
      state_i <= read3;
    when read3 =>
      state_i <= read4;
    when read4 =>
      state_i <= idle;
  end case;
end process;
no_sync_clr_gen: without mux
if SYNC/=1 generate
  state_next <= state_i;
end generate;
sync_clr_gen: with mux
if SYNC=1 generate
  state_next <= idle when clear='l' else
    state_i;
end generate;

Moore output logic
process(state_reg)
begin
14.6.3 Comparisons with other feature-selection methods

In addition to the conditional generate statement, there are two other methods to create a circuit with a selectable feature. One is to create a full-featured circuit and then connect some input control signals to constant values to permanently enable the desired feature. The other is to use the configuration construct. Their uses and differences are discussed in the following subsections.

Comparison to a full-featured circuit  The full-featured scheme can be explained best by an example. Consider the up-and-down counter from Listing 14.17. The counter has an external control signal, mode, which specifies the direction of the counting. The code implies that the next-state logic consists of an incrementor, a decrementor and a multiplexer, as shown in Figure 14.4(b). Although there is no feature parameter in this design, we can imitate the UP generic of the up-or-down counter by connecting the mode signal to a constant value.

For example, assume that we need a 16-bit up counter in a design. To use the parameterized up-or-down counter, we can use the following component instantiation to create the instance:

```vhdl
count16up: up_or_down_counter
generic map(WIDTH=>16, UP=>1);
port (clk=>clk, reset=>reset, q=>q);
```

To create the same counter instance using an the up-and-down counter, we can map the mode signal to '1'. The component instantiation becomes

```vhdl
count16up: up_and_down_counter
generic map(WIDTH =>16);
port (clk=>clk, reset=>reset, mode=>'1', q=>q);
```

Since the mode signal is tied to '1', the counter always counts up, just as in the previous up-or-down counter instance.

Although the two instances have the same functionality, they are two different circuits. The up-or-down counter instance creates a circuit with only the needed features. The up-
and-down counter instance creates a circuit that consists of all features and uses an external control signal to selectively enable a portion of the circuit.

This difference will also be reflected in the processing of the VHDL program. Recall that the processing is divided into analysis, elaboration and execution (synthesis) stages. The conditional generate statement is processed in the elaboration stage and the unneeded circuit is removed. The synthesis software only needs to synthesize the selected portion. On the other hand, while the code from the full-featured scheme is processed, the entire VHDL code will be passed to the synthesis stage. It is the synthesis software's responsibility to propagate the constant signal through the circuits and eliminate the unused portion through logic optimization. This will increase the processing time. For a complex description, the software may not be able to eliminate all the unneeded logic in the final implementation.

In general, use of the feature parameters and conditional generate statements is better than the full-featured approach because it clearly identifies the optional part, and the unused portion of the circuit is removed before synthesis.

**Comparison to configuration** The selected hardware creation can also be achieved by configuration. We can construct multiple architecture bodies, each containing a specific feature. Instead of using the feature generic, we can select the desired feature by configuring the entity with a proper architecture body.

For example, for the previous up-or-down free-running counter, we can eliminate the UP generic and construct one architecture body with a counting-up sequence and another with a counting-down sequence. The VHDL code is shown in Listing 14.20.

**Listing 14.20** Up-or-down counter with two architecture bodies

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity updown_counter is
  generic (WIDTH: natural);
  port(
    clk, reset: in std_logic;
    q: out std_logic_vector (WIDTH-1 downto 0)
  );
end updown_counter;

architecture for the count-up sequence
architecture up-arch of updown_counter is
  signal r_reg: unsigned(WIDTH-1 downto 0);
  signal r_next: unsigned(WIDTH-1 downto 0);
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  r_next <= r_reg + 1;
  -- output logic
```
```vhdl
q <= std_logic_vector(r_reg);
end up_arch;

-- architecture for the count-down sequence
architecture down_arch of updown_counter is
signal r_reg: unsigned(WIDTH-1 downto 0);
signal r_next: unsigned(WIDTH-1 downto 0);
begin
  -- register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  -- next-state logic
  r_next <= r_reg - 1;
  -- output logic
  q <= std_logic_vector(r_reg);
end down_arch;
```

We can create a configuration declaration unit or add the configuration specification to bind the architecture body with the desired feature. This can also be done via the component instantiation in VHDL 93. For example, we can create a 16-bit up counter as follows:

```vhdl
count16up: work.updown_counter(up_arch)
generic map(WIDTH =>16);
port(clk=>clk, reset=>reset, q=>q);
```

Conversely, we can merge the logic from several architecture bodies into a single body and use a feature generic and conditional generate statements to select the desired portion. This essentially replaces the configuration with a feature parameter.

There is no rule about when to use a feature parameter and when to use a configuration construct. In general, code with a feature parameter is more difficult to develop and comprehend because we are essentially describing several different versions of the circuit in the same code. The code of the two architecture bodies of the previous example is clearer and more descriptive than the code with the UP generic. On the other hand, if we use a separate architecture body for each distinctive feature, the number of architecture bodies will grow exponentially and becomes difficult to manage. For example, if we want a binary counter to count up or down, to be equipped with either synchronous or asynchronous clear, and to include a buffered and unbuffered output pulse, we must create eight architecture bodies to cover all possible combinations.

In general, when a feature parameter leads to significant modification or addition of the no-feature code and starts to make the code incomprehensible, it is probably a good idea to use separate architecture bodies and the configuration construct.
14.7 FOR LOOP STATEMENT

14.7.1 Introduction

The for loop statement is a sequential statement and is the only sequential loop construct that can be synthesized. The simplified syntax of the for loop statement is

```plaintext
for index in loop_range loop
    sequential statements;
end loop;
```

The syntax and operation of the for loop statement are similar to those of the generate loop statement except that the loop body is composed of sequential statements. As in the generate loop statement, the loop_range must be static.

The for loop statement is more general and flexible because of the sequential statements. In addition to the statements discussed in Chapter 5, the sequential statements also include the exit statement, which skips the remaining iterations of the loop, and the next statement, which skips the remaining part of the current iteration. The exit and next statements are discussed in the next section.

The basic way to synthesize a for loop statement is to unroll or flatten the loop. Unrolling a loop means to replace the loop structure by explicitly listing all iterations. Since the range is static, the number of iterations is fixed. Once a for loop statement is unrolled, the code is converted to a sequence of regular sequential statements, which can be synthesized accordingly. To derive an effective design, we need to know the implication of various language constructs on the underlying hardware. The examples in the next subsection show the implementation issues of the for loop statement.

14.7.2 Examples of a simple for loop statement

**Binary decoder** The structure of the binary decoder is discussed in Section 14.5.2. We can use the diagram in Figure 14.1 as a reference and derive a for loop statement to describe the basic building block and interconnection pattern. The code is very similar to the for generate version in Listing 14.11 and is shown in Listing 14.21. Note that the conditional signal assignment statement in Listing 14.11 is replaced by the if statement.

```plaintext
Listing 14.21  Parameterized binary decoder using a for loop statement

architecture loop_arch of bin_decoder is
begin
    process (a)
    begin
      for i in 0 to (2**WIDTH-1) loop
        if i=to_integer(unsigned(a)) then
          code(i) <= '1';
        else
          code(i) <= '0';
        end if;
      end loop;
    end process;
end loop_arch;
```
Reduced-xor circuit In Section 14.3.1, the reduced-xor circuit is described using a for loop statement in Listing 14.1. The code is patterned after the version that uses the for generate statement in Listing 14.12.

Serial-to-parallel converter The serial-to-parallel converter discussed in Section 14.5.2 can also be described using a for loop statement. The first version is shown in Listing 14.22. It is patterned after the code using the for generate statement in Listing 14.13.

Listing 14.22 Parameterized serial-to-parallel converter using a for loop statement

```vhdl
architecture loop1_arch of s2p_converter is
  signal q_next: std_logic_vector(WIDTH-1 downto 0);
  signal q_reg: std_logic_vector(WIDTH downto 0);
begin
  q_reg(WIDTH) <= si;
  process(clk,q_reg)
  begin
    for i in WIDTH-1 downto 0 loop
      -- D FF
      if (clk'event and clk='l') then
        q_reg(i) <= q_next(i);
      end if;
      -- next-state logic
      q_next(i) <= q_reg(i+1);
    end loop;
  end process;
  q <= q_reg(WIDTH-1 downto 0);
end loop1_arch;
```

Since the for loop statement is a sequential statement, it must be enclosed within a process. The loop body, which contains both the D FF and next-state logic, is enclosed within the same process accordingly. Note that both clk and q_reg signals are listed in the sensitivity list. An alternative is to split the register and the next-state logic and describe the structure in two separate for loop statements. The revised code is shown in Listing 14.23.

Listing 14.23 Parameterized serial-to-parallel converter using separate for loop statements

```vhdl
architecture loop2_arch of s2p_converter is
begin
  process(clk)
  begin
    for i in WIDTH-1 downto 0 loop
      if (clk'event and clk='l') then
        q_reg(i) <= q_next(i);
      end if;
    end loop;
  end process;
  process(si,q_reg)
  begin
    q_next(WIDTH-1) <= si;
  end process;
```

In Section 14.5.2, the code in Listing 14.14 uses component and instantiation to describe the structure of the serial-to-parallel converter. Since the component instantiation statement is a concurrent statement, this approach cannot be duplicated for the for loop statement.

14.7.3 Examples of a loop body with multiple signal assignment statements

Only sequential signal assignment statements can be used inside the loop body of a for loop statement. Recall that a signal can be assigned multiple times inside a process and only the last assignment takes effect. We can use this property to develop more abstract description. Two examples are shown below.

**Priority encoder**  Recall that a priority encoder is a circuit that returns the binary code of the highest-priority request. In the parameterized version, we assume that the request signals are arranged as an array of \( r(\text{WIDTH}-1 \text{ downto 0}) \), and priority is given in descending order (i.e., the \( r(\text{WIDTH}-1) \) signal has the highest priority). In addition to the binary code, the \text{bcode} \) signal, the output includes the \text{valid} \) signal, which is asserted when at least one request signal is activated. One possible VHDL code of a parameterized priority encoder is shown in Listing 14.24.

**Listing 14.24** Parameterized priority encoder using a for loop statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity prio_encoder is
generic(WIDTH: natural);
port(
  r: in std_logic_vector(WIDTH-1 downto 0);
  bcode: out std_logic_vector(log2c(WIDTH)-1 downto 0);
  valid: out std_logic
);
end prio_encoder;

architecture loop_linear_arch of prio_encoder is
  constant B: natural := log2c(WIDTH);
signal tmp: std_logic_vector(WIDTH-1 downto 0);
begin
  — binary code
  process(r)
  begin
    bcode <= (others=>'0');
    for i in 0 to (WIDTH-1) loop
      if r(i)='1' then
        bcode <= std_logic_vector(to_unsigned(i,B));
      end if;
    end loop;
  end process;
end loop_linear_arch;
```
For an input with \( n \) request signals, the number of bits of the binary code is \( \lceil \log_2 n \rceil \). To express the range of the \( \text{bcode} \) signal, we can use the \( \log_{2c} \) function derived in Chapter 12. We assume that it is stored in the \texttt{util-pkg} package and include the use statement to make it visible.

The major part of the program is the first for loop statement, which iterates from the lowest index to the highest index. When the corresponding request is asserted, its binary code will be assigned to the \( \text{bcode} \) signal. Recall that the last signal assignment takes effect in a process. The \( \text{bcode} \) signal is assigned with the binary code of the highest index, which represents the highest-priority request. If none of the request is asserted, the \( \text{bcode} \) assumes the default assignment of all 0's.

Unlike the previous binary decoder and reduced-xor examples, in which the program codes are derived from the actual circuit structures, this program is based on an abstract, behavioral description of a priority encoding circuit. We drive the circuit structure from the VHDL code, but not the other way around.

To derive the conceptual implementation, we first need to unroll the loop. Assume that \( \text{WIDTH} \) is 4. The flattened code becomes

```vhdl
bcode <= "00"
if r(0)= '1' then
  bcode <= "00";
end if;
if r(1)= '1' then
  bcode <= "01";
end if;
if r(2)= '1' then
  bcode <= "10";
end if;
if r(3)= '1' then
  bcode <= "11";
end if;
```

The code performs a sequence of assignments to the same signal. As discussed in Section 5.4.1, this kind of code is equivalent to an if statement with multiple elsif branches, which implies a priority routing network. The conceptual diagram of the flattened code can be derived using the procedure in Section 5.4.1 and is shown in Figure 14.7. It is basically a cascading chain of 2-to-1 multiplexers.
The valid signal is obtained by performing an or operation on all request signals. It is essentially a reduced-or circuit, and its implementation is similar to that of the reduced-xor circuit.

**Multiplexer** A multiplexer routes the designated input signal to the output port. In the parameterized version, we assume that the input signals are arranged as an array, from \( a(WIDTH-1) \) to \( a(0) \), and the selection signal of the multiplexer uses the index of the array to specify the designated signal. One possible VHDL code is shown in Listing 14.25.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity muxi is
  generic(WIDTH: natural);
  port(
    a: in std_logic_vector(WIDTH-1 downto 0);
    sel: in std_logic_vector(log2c(WIDTH)-1 downto 0);
    y: out std_logic
  );
end muxi;

architecture loop_linear_arch of muxi is
begin
  process(a, sel)
  begin
    y <= '0';
    for i in 0 to (WIDTH-1) loop
      if i = to_integer(unsigned(sel)) then
        y <= a(i);
      end if;
  end process;
end loop_linear_arch;
```
The code is based on an abstract behavioral description. It infers a cascading priority routing networks, similar to that of the previous priority encoder. Although the code is simple and clear, it leads to bulky and inefficient hardware implementation. A better alternative is shown in Chapter 15.

14.7.4 Examples of a loop body with variables

Variables can be used in sequential statements and thus can be used in the body of a for loop statement. Since a variable assignment takes effect immediately, it is useful when an object inside the loop needs to be updated in each iteration. Two examples are shown below.

Reduced-xor circuit with variables

The reduced-xor circuit can be described using a variable. The VHDL code is in Listing 14.26.

Listing 14.26 Parameterized reduced-xor circuit using a variable

```vhdl
architecture loop_linear_var_arch of reduced_xor is
begin
  process(a)
  variable tmp: std_logic;
  begin
    tmp := a(0);
    for i in 1 to (WIDTH-1) loop
      tmp := a(i) xor tmp;
    end loop;
    y <= tmp;
  end process;
end loop_linear_var_arch;
```

The code is more like a program in a traditional programming language. Although it is more abstract and descriptive, deriving the conceptual implementation for this code requires more effort. The key is to convert the variables into constructs that can be mapped into a hardware entity. We first unroll the loop and then rename the variable to avoid self-reference.

Assume that WIDTH is 4. The flattened code is

```vhdl
tmp := a(0);
tmp := a(1) xor tmp;
tmp := a(2) xor tmp;
tmp := a(3) xor tmp;
y <= tmp;
```

To avoid self-reference, the variable is given a new name in each statement and the new names are propagated through subsequent statements:

```vhdl
tmp0 := a(0);
tmp1 := a(1) xor tmp0;
tmp2 := a(2) xor tmp1;
tmp3 := a(3) xor tmp2;
y <= tmp3;
```
We can now interpret that each variable is a connection wire and derive the conceptual diagram accordingly.

For comparison purposes, we also unroll the code of Listing 14.1, which uses signals in the body of the for loop statement. The code becomes

```vhdl
tmp(0) <= a(0);
tmp(1) <= a(1) xor tmp(0);
tmp(2) <= a(2) xor tmp(1);
tmp(3) <= a(3) xor tmp(2);
y <= tmp3;
```

Note that the appearances of the two flattened codes are very similar. The tmp variable can be thought of as shorthand to replace an array of signals, which are needed to express the intermediate values.

**Population counter** The population counter counts the number of 1's from the elements of an array input signal. A fixed-size circuit was discussed in Section 7.5.5. One way to derive the parameterized version is to use a for loop statement and a variable to keep track of the occurrences of 1's. The abstract VHDL code is shown in Listing 14.27.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity popu_count is
  generic(WIDTH: natural);
  port(
    a: in std_logic_vector(WIDTH-1 downto 0);
    count: out std_logic_vector(log2c(WIDTH)-1 downto 0)
  );
end popu_count;

architecture loop_linear_arch of popu_count is
begin
  process(a)
  variable sum: unsigned(log2c(WIDTH)-1 downto 0);
  begin
    sum := (others=>'0');
    for i in 0 to (WIDTH-1) loop
      if a(i)= '1' then
        sum := sum + 1;
      end if;
    end loop;
    count <= std_logic_vector(sum);
  end process;
end loop_linear_arch;
```

Deriving the conceptual implementation of this code involves more work. Assume that WIDTH is 3. The loop can be unrolled into three iterations:

```vhdl
sum := 0;
if a(0)= '1' then
  sum := sum + 1;
```
end if;
if a(1)= '1' then
    sum := sum + 1;
end if;
if a(2)= '1' then
    sum := sum + 1;
end if;
count <= sum;

Unlike the previous reduced-xor example, the following simple renaming will not work properly:

sum0 := 0;
if a(0)= '1' then
    sum1 := sum0 + 1;
end if;
if a(1)= '1' then
    sum2 := sum1 + 1;
end if;
if a(2)= '1' then
    sum3 := sum2 + 1;
end if;
count <= sum3;

To correctly rename the signals, we must include the else branch, which implies that the sum variable remains unchanged. The revised, unrolled code is

sum := 0;
— 1st stage
if a(0)= '1' then
    sum := sum + 1;
else
    sum := sum;
end if;
— 2nd stage
if a(1)= '1' then
    sum := sum + 1;
else
    sum := sum;
end if;
— 3rd stage
if a(2)= '1' then
    sum := sum + 1;
else
    sum := sum;
end if;
count <= sum;

We can easily rename the variables now and the code segment becomes

sum0 := 0;
— 1st stage
if a(0)= '1' then
    sum1 := sum0 + 1;
else
    sum1 := sum0;
The basic building block of each stage is now clear. The corresponding diagram of the flattened and renamed code is shown in Figure 14.8. Note that the diagram also exhibits a cascading-chain structure.

14.7.5 Comparison of the for generate and for loop statements

Both the for generate and for loop statements are used to describe replicated structures. The major difference is the type of statements in their loop bodies. The for generate statement can only use concurrent statements, and the for loop statement can only use sequential statements.

Because there is a clear mapping between concurrent statements and hardware parts, the circuit involved in a stage can be easily visualized. When a for generate statement is used, we frequently start a design with a conceptual diagram of a few stages. The diagram is used to identify the basic building block and connection pattern, and then the code of the loop body is derived accordingly. We sometimes create an entity for the basic building block and then describe the replicated structure by instantiating the component instances.

On the other hand, because of the sequential semantics and the existence of variables, the body of the for loop statement can be more general and versatile. While this allows us to develop more abstract code, it may also lead to an unnecessarily complex implementation or even an unsynthesizable description. The synthesis issue is discussed in Section 14.9.
14.8 EXIT AND NEXT STATEMENTS

The exit and next statements are sequential statements used inside a for loop statement to alter the regular iterations of the loop. The exit statement stops execution of a for loop statement and exits the loop immediately. The remaining iterations will be skipped. The next statement stops execution of the current iteration and jumps to the beginning of the next iteration. The remaining statements of the iteration will be skipped. While the two statements can sometimes be useful for modeling, they are difficult to synthesize. Many synthesis software packages do not support these two statements. The following subsections discuss the use and conceptual implementation of the two statements and the alternative coding style.

14.8.1 Syntax of the exit statement

The syntax of the exit statement is

```
exit when boolean_expr;
```

The `boolean_expr` term is a Boolean expression indicating the exiting condition. When it is evaluated as true, the exit takes effect and the execution skips the remaining iterations of the loop.

Note that the `when boolean_expr` portion is optional. It is not needed if the exit statement is associated with a condition of an if statement, as in the following code segment:

```
if (boolean_expr) then
  ...
  exit;
else
  ...
```

14.8.2 Examples of the exit statement

**reduced-and circuit** The reduced-and circuit was discussed in Section 14.4.2. We use a different approach in this example. One property of the and operation is that \( x \cdot 0 = 0 \). Thus, the reduced-and operation can return '0' as soon as the first '0' element of the input array is found. The VHDL code in Listing 14.28 is based on this observation. The code uses a for loop statement to check the values of the array's elements and uses the exit statement to terminate the loop after the first '0' element is found.

```
Listing 14.28  Parameterized reduced-and circuit using an exit statement

architecture exit_arch of reduced_and is
begin
  process(a)
  variable tmp: std_logic;
  begin
    tmp := '1';  — default output
    for i in 0 to (WIDTH-1) loop
      if a(i)='0' then
        tmp := '0';
        exit;
      end if;
    end for;
  end process;
end;
```
end loop;
y <= tmp;
end process;
end exit_arch;

If this code is developed as a routine in a traditional programming language, it is an effective approach since the execution does not need to go through all iterations of the loop and can cut one half of the execution time in average. However, in synthesis, we cannot dynamically create the circuit according to the input pattern. Instead, the synthesized circuit must accommodate all possible input combinations. Using the exit statement actually introduces additional hardware overhead and complicates the synthesis process. This is discussed in more detail in the next subsection.

**Leading-zero counting circuit**  The leading-zero counting circuit is a combinational circuit that counts the number of leading 0’s in front of an input signal. One possible implementation is to use a for loop statement to keep track of the number of consecutive 0’s and use an exit statement to terminate the loop when the first ’1’ is encountered. The VHDL code is shown in Listing 14.29.

**Listing 14.29**  Parameterized leading-zero counting circuit using an exit statement

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity leading0_count is
generic (WIDTH: natural);
port (a: in std_logic_vector(WIDTH-1 downto 0);
zeros: out std_logic_vector(log2c(WIDTH)-1 downto 0)
);
end leading0_count;
architecture exit_arch of leading0_count is
begin
process (a)
variable sum: unsigned(log2c(WIDTH)-1 downto 0);
begin
sum := (others=>'0'); — initial value
for i in WIDTH-1 downto 0 loop
if a(i)='1' then
exit;
else
sum := sum + 1;
end if;
end loop;
zeros <= std_logic_vector(sum);
end process;
end exit_arch;
```

This code infers multiple adders and is not an efficient design. It is only for demonstration of the use of the exit statement.
14.8.3 Conceptual Implementation of the exit statement

Since the hardware cannot expand or shrink dynamically to accommodate the input pattern, the exit statement cannot be implemented directly in hardware. However, we can emulate the effect of the exit statement using a special circuit to bypass some iterations.

The “bypassing” can best be explained by an example. Consider the VHDL code of the leading-zero counting circuit in Listing 14.29. The exit statement specifies that the remaining iterations of the loop should be skipped if the condition \( a(i) = '1' \) is met. We can achieve the same effect using an array of flags, which indicate whether the execution of the corresponding stages should be bypassed. The revised VHDL code is shown in Listing 14.30.

```
architecture bypass_arch of leading0_count is
  signal bypass: std_logic_vector(WIDTH downto 0);
begin
  process(a, bypass)
  variable sum: unsigned(log2c(WIDTH) - 1 downto 0);
  begin
    — initial value
    sum := (others=>'0');
    bypass(WIDTH) <= '0';
    — bypass flags
    for i in WIDTH-1 downto 0 loop
      if a(i)='1' then
        bypass(i) <= '1';
      else
        bypass(i) <= bypass(i+1);
      end if;
    end loop;
    — counting 1's
    for i in WIDTH-1 downto 0 loop
      if bypass(i)='0' then
        if a(i)='0' then
          sum := sum + 1;
        end if;
      end if;
    end loop;
    zeros <= std_logic_vector(sum);
  end process;
end bypass_arch;
```

The flags are implemented by the bypass signal. The leftmost element, \( \text{bypass}(\text{WIDTH}) \), is set to '0'. An element of the bypass signal is set to '1' when the condition \( a(i) = '1' \) is met, and the condition will be propagated to the subsequent elements. For example, if \( \text{bypass}(3) \) is set to '1', \( \text{bypass}(2), \text{bypass}(1) \text{ and } \text{bypass}(0) \) will be set to '1' as well.

The bypass signal is then used to control the increment operation of each stage. In the \( i \)th stage, the increment operation is performed only if the corresponding \( \text{bypass}(i) \) is not set (i.e., is '0'). Once an element of the bypass signal is set to '1', all the subsequent increment operations will be skipped and the values of the \text{sum} variable will remain unchanged in the remaining iterations. This essentially achieves the effect of the exit statement without actually using it inside the for loop statement.
Note that when the bypass(i) signal is '0', the a(i) must be '0'. Thus, checking the a(i)='0' condition is not needed in the second for loop statement, and it can be simplified to

```vhdl
for i in WIDTH-1 downto 0 loop
    if bypass(i)= '0' then
        sum := sum + 1;
    end if;
end loop;
```

Since there is no exit statement in the revised code, we can derive the conceptual diagram by unrolling the two for loops. Assume that WIDTH is 3. The diagram is shown in Figure 14.9. The upper loop can be simplified to a chain of or gates, as shown at the bottom of the diagram. The bottom loop is similar to the population counter of Figure 14.8, as shown at the top of the diagram. The bypass(i) signal specifies whether to skip the incrementing operation by routing either the original or the incremented input to the next stage. Note that once a bypass value is set to '1' in a stage, the '1' will be propagated through the descending stages, and thus all subsequent incrementing operations are skipped.

The example shows that the use of the exit statement actually introduces additional "bypass overhead" to the original circuit. This overhead makes synthesis more difficult and may increase the size of the final implementation. Some synthesis software packages may not be able to handle a for loop with the exit statement.

### 14.8.4 Next statement

The syntax of the next statement is

```vhdl
next when boolean_expr;
```

When the boolean_expr term is evaluated as true, the next statement takes effect and the execution skips the remaining statements of the iteration. Like the exit statement, the when boolean_expr portion is not needed if the next statement is used with an if statement.

The VHDL code of the population counter of Section 14.7.4 can be revised by using the next statement, as shown in Listing 14.31. When a(i) is '0', the next statement skips the remaining statements of the loop (i.e., the sum := sum + 1 statement).
Listing 14.31  Parameterized population counter using a next statement

```
architecture next_arch of popu_count is
begin
  process (a)
  begin
    variable sum: unsigned(log2c(WIDTH)-1 downto 0);
    sum := (others=>'0');
    for i in 0 to (WIDTH-1) loop
      next when a(i)='0';
      sum := sum + 1;
    end loop;
    count <= std_logic_vector(sum);
  end process;
end next_arch;
```

The next statement is somewhat similar to the “opposite” of an if statement with only a then branch. The former skips some statements when the corresponding condition is met, while the latter executes some statements when the corresponding condition is met. Based on this observation, we can convert the next statement to a modified if statement. For example, consider the following VHDL segment:

```
for ...
  sequential statement 1;
  next when boolean_exp;
  sequential statement 2;
end loop;
```

It can be rewritten as

```
for ...
  sequential statement 1;
  if (not boolean_exp) then
    sequential statement 2;
  end if
end loop;
```

The if statement is preferred because it is more descriptive and modular. The revised code also shows that the implementation of the next statement should be similar to that of an if statement without an else branch, as in Listing 14.27.

### 14.9 SYNTHESIS OF ITERATIVE STRUCTURE

VHDL provides a variety of mechanisms to describe the iterative structure. From the synthesis’s point of view, the key is to identify the circuit involved in a stage. Once it is done, the circuit can be replicated to a specific number set by the width parameters. The synthesis software can process the flattened description as a regular circuit.

When deriving code with for generate statements, we normally first draw a sketch diagram of the hardware and then derive the VHDL description accordingly. This is partially due to the semantics of the concurrent statements, which prevents us from thinking in terms of sequential programming constructs. Ideally, we should apply the same approach when using for loop statements. Unfortunately, since sequential statements are more abstract and closer to the statements of traditional programming languages, it is easy to use for loop
statements to write abstract, sequential codes. This frequently leads to bulky, unnecessarily complex implementation. Thus, when a for loop statement is used, we should be conscious of the implications on the underlying hardware.

The for loop and for generate statements provide an easy mechanism to describe the iterative structure. Unfortunately, the simple description does not always lead to efficient implementation. The examples of this chapter utilize a single-level for generate or for loop statement, which translates into a one-dimensional structure. Except for special cases, such as the binary decoder, the one-dimensional structure leads to a cascading-chain type of circuit. This kind of topology is difficult to handle during placement and routing and may introduce a large propagation delay, especially when the chain is very long. A more effective two-dimensional tree- or mesh-shaped structure is more desirable. This issue is discussed in the Chapter 15.

14.10 SYNTHESIS GUIDELINES

- Use a generic to specify the width parameter.
- If an unconstrained array is used for parameterized design, take the range and direction of the array into consideration.
- Use the if generate statement for small feature variation.
- The for loop statement should be considered as a construct to describe a circuit with a replicated structure.
- A single-level for generate or for loop statement normally leads to a one-dimensional cascading-chain structure.

14.11 BIBLIOGRAPHIC NOTES

While many texts cover the VHDL generate and loop constructs, few literatures provide in-depth discussions of parameterized design. One place to find good examples is in the package body of the IEEE numerid_std package. The source file can normally be found in the directory where the IEEE library resides. The functions and overloaded operators of the package are defined over the unsigned and signed data types with no explicit range specification, and the needed parameters are derived from attributes. Because the codes include comprehensive error-checking, they are quite complex. Another source is the VHDL code of “Library of Parameterized Modules” (LPM). It is an early, not-so-successful attempt to develop parameterized device-independent VHDL modules. The VHDL package should still be available via internet search.

Problems

14.1 Consider a 1-bit incrementor cell that adds 1 to the input operand. It has two 1-bit input signals, a and cin, which represent the input operand and carry-in respectively, and two 1-bit output signals, s and cout, which represent the sum and carry-out respectively.

(a) Derive the function table for this circuit.
(b) Derive the VHDL code for this circuit using only simple signal assignment statements and logical operators.
(c) Derive the block diagram of a 4-bit incrementor using four incrementor cells.

14.2 Follow the block diagram of Problem 14.1(c) to design a parameterized incrementor in which the width of the input operand is specified by a generic. Derive the VHDL code using the for generate statement. Use a simple signal assignment statement in the loop body, and no VHDL arithmetic operator is allowed.

14.3 Repeat Problem 14.2, but create the 1-bit incrementor cell as a component and use component instantiation in the loop body.

14.4 Repeat Problem 14.2, but use conditional generate statements for the boundary cells.

14.5 Repeat Problem 14.2, but use the for loop statement.

14.6 Repeat Problem 14.2, but apply the clever-use-of-array techniques discussed in Section 14.4. No for generate or for loop statement is allowed.

14.7 Repeat Problem 14.2, but use no generic. Declare the data type of the input port as std_logic_vector with no explicit range specification. Make sure that the code can work with different formats of specification when the component is instantiated.

14.8 Follow the technique of the reduced-and circuit of Listing 14.4.2, and derive a parameterized VHDL code for the reduced-or circuit.

14.9 For the memory controller FSM circuit in Section 10.7.2, the output signal can be unbuffered or buffered. The buffered output uses the look-ahead output buffer scheme. Derive a VHDL code that includes both schemes and use the BUF generic as a feature parameter to specify which buffer scheme to use.

14.10 Consider the priority encoder code of Listing 14.24. Rewrite the code using a for generate statement.

14.11 Consider the population counter code of Listing 14.27. Rewrite the code using a for generate statement.

14.12 Consider the reduced-and code of Listing 14.28. Follow the conceptual implementation procedure discussed in Section 14.8.3 to replace the exit statement with flag signals.

(a) Derive the VHDL code.

(b) Draw the conceptual diagram.

(c) Prove that the conceptual diagram actually performs the reduced-and operation.
This Page Intentionally Left Blank
After learning the basic language constructs for the parameterized design, we study more sophisticated circuit examples in this chapter. In addition to parameterization, the emphasis is on the efficiency and performance of the circuits. The main focus is on the derivation of efficient parameterized RT-level modules that can be used as building blocks of larger systems.

15.1 INTRODUCTION

Parameterization is not directly related to the efficiency of a digital circuit. However, it frequently leads to inefficient design for several reasons. First, a parameterized description relies on a small set of language constructs, mainly the for generate and for loop statements. We have less freedom to describe the intended circuit structure. Second, because the for loop statement is similar to the loop constructs found in the traditional programming languages, we tend to develop behavioral descriptions and become less aware of the underlying hardware organization.

We constructed several parameterized modules in Chapter 14. Because of the regular, repetitive nature of these circuits, they are described by a for loop or for generate statement. While the code is simple and easy to understand, a single for loop or for generate statement generally describes a one-dimensional cascading structure. This kind of structure introduces a long propagation delay and thus penalizes the performance. Since synthesis software can only perform certain local transformation, it is not able to restructure and optimize the
entire chain. This is particularly problematic for parameterized description since we may instantiate a module with a large parameter.

To derive efficient parameterized modules, we must pay particular attention to the topology of the underlying structure, as discussed in Section 7.4. The description should help the synthesis process to derive a more effective implementation. In this chapter, we discuss the data types used to describe scalable two-dimensional structure, illustrate the design and description of various RT-level components, and study several more difficult application examples.

15.2 DATA TYPES FOR TWO-DIMENSIONAL SIGNALS

One-dimensional arrays, which include std_logic_vector of the std_logic_1164 package, and unsigned and signed of the numeric_std package, are the primary data types used in our codes. They are natural matches to represent a multiple-bit signal. To enhance portability and improve readability, we prefer to use these predefined data types and avoid the multidimensional array in general. In a parameterized design, a circuit may exhibit a scalable two-dimensional structure and require two-dimensional data types to represent the internal signals or I/O ports. While the data types of VHDL are flexible and versatile, no two-dimensional array data type is defined in the std_logic_1164 or numeric_std package. Thus, we must create a user-defined data type for two-dimensional signals.

Because of the lack of a common synthesis standard, software support varies and multidimensional array data types are not accepted by all software tools. This section discusses use of genuine VHDL two-dimensional data types, and two work-arounds, which are the array-of-arrays and emulated two-dimensional array data types. We can choose the scheme that is supported by the software in hand.

15.2.1 Genuine two-dimensional data type

The array data type in VHDL is defined as a collection of elements of the same data types. Its definition is very general. The simplified syntax is

```
type data_type_name is array (range_1, range_2, ...) of element_data_type;
```

The range terms inside the parentheses specify the boundaries of the array, and the number of range terms corresponds to the dimensions of the array. The data type is known as a constrained array if the ranges are fixed and is known as an unconstrained array otherwise.

**Constrained array**  The definition and use of a user-defined two-dimensional data type can best be explained by an example. Consider a 4-by-6 SRAM (i.e., an SRAM that contains four words and each word is 6 bits wide). The structure of the SRAM is a natural match for a two-dimensional data type:

```vhd
constant ROW natural := 4;
constant COL natural := 6;
type sram_4_by_6_type is array (ROW-1 downto 0, COL-1 downto 0) of std_logic;
```

This is a constrained array since its ranges are fixed.

We can assign a two-dimensional constant to a signal with this data type. As in a one-dimensional array, both positional and named association can be used for the aggregate. Some examples are
DATA TYPES FOR TWO-DIMENSIONAL SIGNALS

signal t1, t2, t3: sram_4_by_6_type;

— *positional association*

\[
t1 <= ("000000", "010101", "000111", "111111");
\]
— *"101010" to all rows*

\[
t2 <= (others=>'101010');
\]
— all 0's

\[
t3 <= (others=>'0');
\]

We can use the two indexes, in the form of \((i,j)\), to access a particular element of the array, as in the following examples:

\[
signal t4: sram_4_by_6_type;
signal e1, e2, e3: std_logic;
\]

\[
t4(0,0) <= '1';
t4(1,2) <= e1 and e2;
e3 <= t4(3,5);
\]

In an SRAM, we frequently want to access a word, which corresponds to a row in the two-dimensional data type. Unfortunately, there is no build-in VHDL mechanism to specify a particular dimension. The work-around is to use a for loop or for generate statement to iterate through the individual elements. An example of using the for loop statement is

\[
signal t5: sram_4_by_6_type;
signal vl: std_logic_vector(COL-1 downto 0);
\]

\[
process(...) 
begin 
  for i in COL-1 downto 0 loop 
    vl(i) <= t5(i,i); 
  end loop;
\]

**Unconstrained array** An unconstrained array does not specify the boundary of the range when the data type is defined. This information is provided later when the data type is used. An unconstrained two-dimensional array of element type of `std_logic` can be defined as

\[
type std_logic_2d is 
  array (natural range <>, natural range <>) of std_logic;
\]

This is more effective since it can accommodate two-dimensional arrays of various sizes. For example, assume that three different sizes are used in a design. If the constrained array is used, we need three data types:

\[
type array_4_by_6_type is 
  array (3 downto 0, 5 downto 0) of std_logic;
type array_16_by_32_type is 
  array (15 downto 0, 31 downto 0) of std_logic;
type array_8_by_2_type is 
  array (7 downto 0, 1 downto 0) of std_logic;
\]
signal s1: array_4_by_6_type;
signal s2, s3: array_16_by_32_type;
signal s4, s5: array_8_by_2_type;

On the other hand, the code will be much simpler if an unconstrained array is used:

```vhdl
type std_logic_2d is
  array (natural range <> , natural range <> ) of std_logic;
signal s1: std_logic_2d(3 downto 0, 5 downto 0);
signal s2, s3: std_logic_2d(15 downto 0, 31 downto 0);
signal s4, s5: std_logic_2d(7 downto 0, 1 downto 0);
```

If we include the std_logic_2d data type in a package, this data type can be used in VHDL code after the package is invoked. In fact, the std_logic_vector, unsigned and signed data types are defined as an unconstrained one-dimensional array. The utility package discussed in Section 13.5.3 actually follows this practice and includes the two-dimensional data type in the package declaration. The definition of the std_logic_2d data type is very general, and its dimension can be specified as generic parameters in the port declaration of an entity and in the signal declaration of an architecture body. A simple example is

```vhdl
use work.util_pkg.all;
entity ... generic (ROW: natural;
  COL: natural );
  port (p1, p2: in std_logic_2d(ROW-1 downto 0, COL-1 downto 0);
    ...
  );
  architecture ...
  signal sig1, sig2:
    std_logic_2d(ROW-1 downto 0, COL-1 downto 0)
    ...
```

While this is an elegant scheme, the std_logic_2d data type may not be accepted by some synthesis software because of the lack of support for multidimensional arrays.

### 15.2.2 Array-of-arrays data type

The array in VHDL is very flexible, and the data type of the element of an array can also be an array. Thus, a two-dimensional structure can be defined as a one-dimensional array whose element's data type is also a one-dimensional array. We call this an array-of-arrays data type. For example, we can replace the previous sram_4_by_6_type data type with an array-of-arrays data type:

```vhdl
constant ROW natural:=4;
calculate COL natural:=6;
type sram_aoa_46_type is array (ROW-1 downto 0) of
  std_logic_vector(COL-1 downto 0);
```

This data type is a one-dimensional array with four elements whose data type is a six-element one-dimensional array (i.e., std_logic_vector(5 downto 0)).
The structures of sram_4_by_6_type and sram_aoa_46_type are very similar. In fact, the constant assignment for the two data types are identical:

```vhdl
signal t1, t2, t3: sram_aoa_46_type;

-- positional association
t1 <= ("000000", "010101", "001111", "111111");
-- "101010" to all rows
t2 <= (others=>'101010');
-- all 0's
t3 <= (others=>(others=>'0'));
```

We can also access a single bit in an array-of-arrays data type. Instead of \((i, j)\), the index is in the form of \((i)(j)\). The example in Section 15.2.1 becomes

```vhdl
signal t4: sram_aoa_46_type;
signal e1, e2, e3: std_logic;
... t4 (0) (0) <= '1';
t4 (1) (2) <= e1 and e2;
e3 <= t4 (3) (5);
```

Since a row of an array-of-arrays data type is an element of a one-dimensional array, to access a row is much easier. For example, to retrieve a word from the previous SRAM, we can just use the first index:

```vhdl
signal t5: sram_aoa_46_type;
signal v1: std_logic_vector(COL-1 downto 0);
... v1 <= t5 (1);
```

Thus, for this particular application, an array-of-arrays data type is more natural than a genuine two-dimensional data type.

The major limitation of the array-of-arrays data type is that the data type of its element must be a constrained array. At best, we can only define a data type as

```vhdl
type std_logic_aoa_N is
  array (natural range <>) of std_logic_vector(N-1 downto 0);
```

In other words, only the first dimension (i.e., the number of rows) can be left unconstrained.

If an array-of-arrays data type is defined and used inside the architecture body, it is still possible to pass the two-dimensional parameters via generics. A simple example is
Figure 15.1 Emulation of a two-dimensional 4-by-3 array with a one-dimensional array.

array (ROW-1 downto 0) of std_logic_vector (COL-1 downto 0);
signal sig1, sig2: aoa_RC_type;

However, if an array-of-arrays data type is used for a port declaration, only the first dimension can be parameterized. Since the size of the second dimension must be fixed in port declaration, the array-of-arrays data type is not as flexible as the std_logic_2d data type. This is the most severe constraint of using an array-of-arrays data type in a parameterized design.

Because an array-of-arrays data type is a one-dimensional array, more synthesis software accepts this form.

15.2.3 Emulated two-dimensional array

Emulated two-dimensional array is a scheme that imitates a two-dimensional structure using a one-dimensional array. In this scheme, we introduce no new data type but cleverly interpret a one-dimensional array as a two-dimensional structure. For example, consider a 4-by-3 two-dimensional array. We can enumerate the four rows in a single list, as shown in Figure 15.1. The relationship between the one-dimensional index, n, and the two-dimensional indexes, i and j, is characterized by a simple equation:

\[ n = i \times 3 + j \]

Since the regular one-dimensional array data type is used to describe a two-dimensional structure, we call it an emulated two-dimensional array.

Let us consider the 4-by-6 SRAM example again. Although no new data type is needed, it will be handy to define a function to calculate the indexes. The code is

\[
\begin{align*}
\text{constant ROW natural} & := 4; \\
\text{constant COL natural} & := 6; \\
\text{data type is std_logic_vector (ROW*COL-1 downto 0);} \\
\text{function ix (r, c: natural) return natural is} \\
\text{begin} \\
\text{return (r*COL + c);} \\
\text{end ix;} \\
\end{align*}
\]

To access a single bit in the emulated array, we can invoke the ix function to calculate the corresponding position in the one-dimensional array. We can replace the index \((i,j)\) used in a genuine two-dimensional array with the indexing function, \(ix(i,j)\). The indexing example in Section 15.2.1 becomes

\[
\begin{align*}
\text{signal t4: std_logic_vector (ROW*COL-1 downto 0);} \\
\text{signal e1, e2, e3: std_logic;} \\
\end{align*}
\]
DATA TYPES FOR TWO-DIMENSIONAL SIGNALS

A row in the SRAM corresponds to a slice in the one-dimensional array. We can access the entire row after determining the upper and lower boundaries of the row. For the \( i \)th row, the index of the upper boundary is \( \text{ix}(i, \text{COL}-1) \) and the lower boundary is \( \text{ix}(i, 0) \), and thus we can use the range \( \text{ix}(i, \text{COL}-1) \) \textbf{downto} \( \text{ix}(i, 0) \) to access the row. The example in Section 15.2.1 retrieves the first word of the SRAM. It can be modified as follows:

```vhdl
signal \( t5 \) : std_logic_vector (ROW*COL-1 downto 0);
signal \( v1 \) : std_logic_vector (COL-1 downto 0);

\( v1 \) <= \( t5(\text{ix(1, COL-1)} \) downto \( \text{ix(1, 0)} \));
```

Assigning a constant to the emulated array is just assigning a constant to a regular one-dimensional array. We can use the concatenation operator to make the code clearer and consistent with other schemes. The constant expression in Section 15.2.1 can be modified as follows:

```vhdl
signal \( t1 \), \( t2 \), \( t3 \) : std_logic_vector (ROW*COL-1 downto 0);

\( t1 \) <= "000000" & "010101" & "000111" & "111111";

\( t2 \) <= "101010" & "101010" & "101010" & "101010";

\( t3 \) <= (others=>'0');
```

Because the emulated array uses a predefined one-dimensional array data type, the parameters for two dimensions can be passed via generics for the port declaration and signal declaration. An example is shown below.

```vhdl
entity . . .
generic(
  ROW: natural;
  COL: natural
);
port(
  p1, p2: in std_logic_vector(ROW*COL-1 downto 0);
  . . .);
architecture . . ;
function \( \text{ix}(r, c: \text{natural}) \) return \text{natural} is
begin
  return (r*COL + c);
end \( \text{ix} \);
signal \( \text{sig1, sig2} \) : in std_logic_vector(ROW*COL-1 downto 0);
  . . .
```

The emulated array involves numerous calculations to map a two-dimensional index into a one-dimensional index. However, these calculations are static and thus can be determined when the VHDL code is elaborated. No physical circuit will be inferred for this purpose.
15.2.4 Example

In Chapter 14, we presented a parameterized multiplexer in Listing 14.25. In this design, the number of input ports is specified by a generic but the number of bits per port is fixed (i.e., 1 bit). A more general description should add an additional parameter to specify the number of bits of a port as well. Let the number of input ports be P and the number of bits per port be B. The a input signal now represents a two-dimensional P*B-bit signal. The following codes illustrate how to use the three two-dimensional data types to implement the new multiplexer.

Implementation with a genuine two-dimensional array  The first description uses the genuine two-dimensional std_logic_2d data type. The VHDL code is shown in Listing 15.1. The util_pkg package is needed for the std_logic_2d data type and the log2c function.

Listing 15.1 Parameterized two-dimensional multiplexer using a genuine array

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity mux2d is
  generic(
    P: natural;  -- number of input ports
    B: natural  -- number of bits per port
  );
  port(
    a: in std_logic_2d(P-1 downto 0, B-1 downto 0);
    sel: in std_logic_vector(log2c(P-1 downto 0);
    y: out std_logic_vector(B-1 downto 0)
  );
end mux2d;

architecture two_d_arch of mux2d is
begin
  process(a, sel)
  begin
    y<=(others=>'0');
    for i in 0 to (P-1) loop
      if i=to_integer(unsigned(sel)) then
        for j in 0 to (B-1) loop -- B-bits of the port
          y(j)<=a(i,j);
        end loop;
      end if;
    end loop;
  end process;
end two_d_arch;
```

The code is basically patterned after the one-dimensional multiplexer code in Listing 14.25. An extra inner for loop statement is added to route B bits from an input port to the output.
Implementation with an emulated array  The second description uses an emulated array, and the VHDL code is shown in Listing 15.2. The code also includes the util_pkg package since the log2c function is needed. It follows the description in Listing 15.1 but with several simple modifications:

- Use the regular std_logic_vector data type.
- Define the ix function.
- Use a(ix(i,j)) to replace a(i,j).

**Listing 15.2** Parameterized two-dimensional multiplexer using an emulated array

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity mux_emu_2d is
  generic(
    P: natural; — number of input ports
    B: natural — number of bits per port
  );
  port(
    a: in std_logic_vector(P*B-1 downto 0);
    sel: in std_logic_vector(log2c(P)-1 downto 0);
    y: out std_logic_vector(B-1 downto 0)
  );
end mux_emu_2d;

architecture emu_2d_arch of mux_emu_2d is
  function ix (r, c: natural) return natural is
    begin
      return (r*B + c);
    end ix;
  begin
    process(a, sel)
    begin
      y<=(others=>'0');
      for r in 0 to (P-1) loop
        if r=to_integer(unsigned(sel)) then
          for c in 0 to (B-1) loop — B-bits of the port
            y(c) <= a(ix(r,c));
          end loop;
        end if;
      end loop;
    end process;
  end emu_2d_arch;
```

Since we can specify a slice of array in the emulated array scheme, the inner loop

```vhdl
for c in 0 to (B-1) loop
  y(c) <= a(ix(r,c));
end loop;
```

can be replaced by

```vhdl
y <= a(ix(r,B-1) downto ix(r,0));
```
**Implementation with an array of arrays** Because the element data type of an array of arrays must be a constrained array, the array-of-arrays data type is not general enough to be used in the port declaration of the two-dimensional multiplexer. However, this data type can still be used inside the architecture body. We can use the previous emulated array in the entity declaration and then convert the input into the array-of-arrays data type in the architecture body. The VHDL code of the architecture body is shown in Listing 15.3.

Listing 15.3 Parameterized two-dimensional multiplexer using an array of arrays

```vhdl
architecture a_of_a_arch of mux_emu_2d is
  type std_aoa_type is
    array(P-1 downto 0) of std_logic_vector(B-1 downto 0);
  signal aa: std_aoa_type;
  begin
    -- convert to array-of-arrays data type
    process(a)
    begin
      for r in 0 to (P-1) loop
        for c in 0 to (B-1) loop
          aa(r)(c) <= a(r*B+c);
        end loop;
      end loop;
    end process;
    -- mux
    process(aa, sel)
    begin
      y<=(others=>'0');
      for i in 0 to (P-1) loop
        if i = to_integer(unsigned(sel)) then
          y <= aa(i);
        end if;
      end loop;
    end process;
  end a_of_a_arch;
```

The first process is for type conversion. It is static, and no physical circuit should be inferred. The second process describes the actual multiplexer. The code is identical to one-dimensional multiplexer code in Listing 14.25. The only difference is that the element data type of the aa signal is std_logic_vector(B-1 downto 0), and the element data type of the a signal of the one-dimensional multiplexer is std_logic. From this point of view, the array-of-arrays data type is the most concise representation of the underlying circuit structure.

**15.2.5 Summary**

Ideally, we wish to select a two-dimensional representation that can effectively describe the underlying circuit structure and be universally accepted by synthesis software. It cannot be easily achieved due to the intrinsic limitation of VHDL and the variation on synthesis software support. However, since these representations describe the same two-dimensional structure, conversion between the representations is fairly straightforward. We should select a scheme that works with the synthesis software in hand and properly document the use of these data types in the VHDL code so that they can be easily modified when needed.
In the remainder of this chapter, we use the std_logic_2d data type in general and use the array-of-arrays data type if it closely matches the underlying structure.

15.3 COMMONLY USED INTERMEDIATE-SIZED RT-LEVEL COMPONENTS

We discussed the level of abstraction in Section 1.4. The focus of this book is on the RT level, in which the main parts are intermediate-sized components. Most synthesis software contains predesigned modules for relational operators and addition and subtraction operators, and these modules are inferred and instantiated during synthesis. There are many other intermediate-sized RT-level components that are frequently encountered in a large design, including reduction circuit, decoder, encoder, multiplexer, barrel shifter and multiplier. Since these components are common building parts that are needed in many applications, they are good candidates to be parameterized.

As discussed in Section 7.4, the efficiency of a circuit relies heavily on its basic structure and underlying topology. A good description helps the synthesis process to derive a more effective implementation. To describe a parameterized multidimensional circuit is more involved. The key to designing this type of circuit is to identify a general pattern and then use for loop or for generate statements to describe the desired connection pattern. The following procedure helps us to achieve this goal:

- Draw a small-scale diagram with basic building blocks.
- Derive a proper index for the connection signals in each stage.
- Identify the general relationship between the signals in successive stages.
- Identify the connection patterns between boundary stages and I/O ports.
- Derive the VHDL code accordingly.

The remaining section illustrates the design and derivation of several RT-level components.

15.3.1 Reduced-xor circuit

In Chapter 14, we constructed a parameterized reduced-xor circuit using various VHDL language constructs, as in Listings 14.1, 14.6 and 14.12. These codes essentially describe the same cascading circuit of Figure 14.2. For an n-bit input, the critical path includes n xor gates. We can rearrange the cascading chain into a tree-shaped structure, as discussed in Section 7.4.1, and reduce the critical path to log n xor gates.

For a non-parameterized design, we can use parentheses to force the desired order of evaluation and thus implicitly construct a tree-shaped circuit, as shown in Listing 7.18. Translating this approach into a parameterized description is not feasible. We need to explicitly specify the connection pattern in VHDL code. The circuit diagram of a tree-shaped eight-input reduced-xor circuit is shown in Figure 15.2. This is a two-dimensional structure. We first divide the tree into stages and number the stages from right to left. Each stage now contains multiple xor gates. We treat each xor gate as a row and number the rows from top to bottom. An xor gate can be identified with a two dimensional index (s, r), which represents the rth row of the sth stage. The corresponding output signals of the xor gate is named p_{s,r}. We can label all the interconnection signals according to this naming convention, as shown in Figure 15.2. Note that the input signals to the leftmost stage are also named following the same convention to make a homogeneous diagram.

The key to describing a repetitive structure is to identify the relationship of the signals between successive stages. Let us examine the xor gate in the rth row of the sth stage. Its two inputs are from the the 2rth row and (2r+1)th row of the left stage (i.e., the (s+1)th stage).
Figure 15.2 Tree-shaped reduced-xor circuit.

The factor 2 in a row’s index reflects the fact that the number of rows is reduced by half in each stage. The input–output relationship of this xor gate can be described as

\[ p_{s,r} = p_{s+1,2r} \oplus p_{s+1,2r+1} \]

After identifying the key relationship, we can convert the circuit into VHDL code. The two-dimensional structure implies that we need a two-dimensional data type for the \( p \) signal and a nested generate statement for the structure, with the outer statement for iteration in terms of the stages and the inner statement for iteration in terms of the rows. Since an xor gate has two inputs, the number of rows is reduced by half at each stage. For an input of \( n \) bits, the implementation needs \( \log_2 n \) stages and there are \( 2^n \) rows in the \( s \)th stage.

The VHDL code is shown in Listing 15.4. The entity declaration is the same as the one in Chapter 14 and is included for clarity. We assume that the width of the input is in a power of 2. The code uses a nested two-level for generate statement for the general structure and an additional for generate statement to convert the input signal to the internal naming convention.

Listing 15.4 Parameterized tree-shaped reduced-xor circuit with input of \( 2^n \) bits

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.util_pkg.all;
entity reduced_xor is
  generic(WIDTH: natural);
  port(
    a: in std_logic_vector(WIDTH-1 downto 0);
    y: out std_logic
  );
end reduced_xor;

architecture gen_tree_arch of reduced_xor is
  constant STAGE: natural := log2c(WIDTH);
  signal p: std_logic_2d(STAGE downto 0, WIDTH-1 downto 0);
begin
  -- rename input signal
  in_gen: for i in 0 to (WIDTH-1) generate
```
p(STAGE,i) <= a(i);
end generate;
-- replicated structure
stage_gen:
for s in (STAGE-1) downto 0 generate
row_gen:
for r in 0 to (2**s-1) generate
  p(s,r) <= p(s+1,2*r) xor p(s+1,2*r+1);
end generate;
end generate;
-- rename output signal
y <= p(0,0);
end gen_tree2_arch;

If the number of input bits is not a power of 2, the input stage may appear irregular. One way to handle the input of arbitrary width is to create a full-sized reduced-xor tree and tie the unused inputs to 0's. Since $x \oplus 0 = x$, there is no effect on functionality. These 0 inputs are static, and the redundant xor gates will be removed during synthesis. Thus, the padding 0's should have no adverse impact on the physical implementation. The revised VHDL code is shown in Listing 15.5. An if generate statement is added. The input to the leftmost stage will be padded with 0's if its number is not a power of 2.

**Listing 15.5** Parameterized tree-shaped reduced-xor circuit with input of arbitrary bits

```vhdl
architecture gen_tree2_arch of reduced_xor is
  constant STAGE: natural := log2c(WIDTH);
  signal p:
  std_logic_2d(STAGE downto 0, 2**STAGE-1 downto 0);
begin
  -- rename input signal
  in_gen:
  for i in 0 to (WIDTH-1) generate
    p(STAGE,i) <= a(i);
  end generate;
  -- padding 0's
  pad0_gen:
  if WIDTH < (2**STAGE) generate
    zero_gen:
    for i in WIDTH to (2**STAGE-1) generate
      p(STAGE,i) <= '0';
    end generate;
  end generate;
  -- replicated structure
  stage_gen:
  for s in (STAGE-1) downto 0 generate
    row_gen:
    for r in 0 to (2**s-1) generate
      p(s,r) <= p(s+1,2*r) xor p(s+1,2*r+1);
    end generate;
  end generate;
  -- rename output signal
  y <= p(0,0);
end gen_tree2_arch;
```
The design can also be coded with a for loop statement, as shown in Listing 15.6.

**Listing 15.6** Parameterized tree-shaped reduced-xor circuit using for loop statement

```vhdl
architecture loop_tree_arch of reduced_xor is
  constant STAGE: natural := log2c(WIDTH);
  signal p:
    std_logic_2d(STAGE downto 0, 2**STAGE-1 downto 0);
begin
  process(a,p)
  begin
    for i in 0 to (2**STAGE-1) loop
      if i < WIDTH then
        p(STAGE,i) <= a(i);  -- rename input signal
      else
        p(STAGE,i) <= '0';  -- padding 0's
      end if;
    end loop;
    for a in (STAGE-1) downto 0 loop
      for r in 0 to (2**s-1) loop
        p(s,r) <= p(s+1,2*r+1) xor p(s+1,2*r+1);
      end loop;
    end loop;
    end process;
    -- rename output signal
    y <= p(0,0);
  end loop_tree_arch;
end loop_tree_arch;
```

### 15.3.2 Binary decoder

We discussed the design of a parameterized binary decoder in Section 14.7.2. The code in Listing 14.21 represents a one-dimensional vertical structure, as shown in Figure 14.1. Since the decoding of each output bit is done in parallel, the code is better than the codes of a cascading chain. However, the parallel vertical structure introduces a large number of input signals and may hinder the placement and routing process.

An alternative is to construct a larger decoder with a collection of smaller decoders that are arranged as a two-dimensional tree. This example illustrates the construction with 1-to-2^l decoders. The block diagram and the function table of the 1-to-2^l decoder are shown in Figure 15.3(a). An enable signal, en, is added to the decoder to accommodate the construction. When it is not asserted, the decoder is disabled with an all-zero output. The logic equations for this circuit are very simple:

\[
\begin{align*}
y_0 &= en \cdot a' \\
y_1 &= en \cdot a
\end{align*}
\]

The block diagram of a 3-to-2^3 decoder with 1-to-2^1 decoders is shown in Figure 15.3(b). In this scheme, the input signal is decoded in stages, from the MSB to the LSB. The leftmost stage (i.e., stage 2) decodes the a_2 bit, and its output enables either the top or bottom part of the downstream decoding stages. The next stage decodes the a_1 bit and enables one-half of its downstream decoding stages. Thus, after two stages, only one-fourth of the downstream
decoding stages is enabled. For an \( n \)-to-\( 2^n \) decoder, this operation repeats for each bit until all the bits are decoded and one out of \( 2^n \) output bits is asserted.

Note that there is an additional enable signal, \( en \), in the input of the parameterized module. If the \( en \) signal is not asserted, it disables the leftmost \( 1 \)-to-\( 2^1 \) decoder, which, in turn, disables all downstream \( 1 \)-to-\( 2^1 \) decoders. None of the output bits will be asserted.

The VHDL description is shown in Listing 15.7, and the entity declaration of Chapter 14 is included for clarity. It is coded with a nested two-level for loop statement. The two inner sequential signal assignments are based on the logic equations of the \( 1 \)-to-\( 2^1 \) decoder.

**Listing 15.7** Parameterized tree-shaped binary decoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.util_pkg.all;
entity tree_decoder is
  generic(WIDTH: natural);
  port(
    a: in std_logic_vector(WIDTH-1 downto 0);
    en: std_logic;
    code: out std_logic_vector(2**WIDTH-1 downto 0)
  );
```

![Symbol and function table of a 1-to-2^1 decoder](image)

(a) Symbol and function table of a 1-to-2^1 decoder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>en</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

![3-to-2^3 decoder using 1-to-2^1 decoders](image)

(b) 3-to-2^3 decoder using 1-to-2^1 decoders

Figure 15.3 Tree-shaped binary decoder.
end tree_decoder;

architecture loop_tree_arch of tree_decoder is
constant STAGE: natural := WIDTH;
signal p:
  std_logic_2d(STAGE downto 0, 2**STAGE-1 downto 0);
begin
  process (a, p)
  begin
    — leftmost stage
    p(STAGE, 0) <= en;
    — middle stages
    for s in STAGE downto 1 loop
      for r in 0 to (2**((STAGE-s)-1)) loop
        p(s-1, 2*r) <= (not a(s-1)) and p(s, r);
        p(s-1, 2*r+1) <= a(s-1) and p(s, r);
      end loop;
    end loop;
    — last stage and output
    for i in 0 to (2**STAGE-1) loop
      code(i) <= p(0, i);
    end loop;
  end process;
end loop_tree_arch;

15.3.3 Multiplexer

A parameterized multiplexer was designed in Chapter 14 and the code is shown in Listing 14.25. The code represents a one-dimensional cascading priority routing network and thus is not an ideal structure.

Tree-shaped multiplexer One scheme to derive a two-dimensional structure is to divide the multiplexing into stages that are controlled by the individual bits of the selection signal. The block diagram of an 8-to-1 multiplexer is shown in Figure 15.4. It consists of three stages of 2-to-1 multiplexers. At each stage, the selection signals of the 2-to-1 multiplexers are tied together and connected to a bit of the selection signal, sel, of the 8-to-1 multiplexer. The LSB of the sel signal is connected to the leftmost stage (i.e., stage 2). It selects one-half of the eight possible inputs and routes them to the next stage. The selection process repeats two more times until a single input is routed to the output.

The operation of this circuit can be understood by examining an example. Routing with the sel signal of "110" is shown in Figure 15.5. We use a "binary subscript" to make the routing process clearer. For example, the $a_{6}$ input is expressed as $a_{110}$. The routing is done as follows:

- Stage 2 (the leftmost stage): The LSB of the sel signal is '0' and thus input signals with index "xx0", which include $a_{000}$, $a_{010}$, $a_{100}$ and $a_{110}$, are selected and routed to the next stage.
- Stage 1 (the middle stage): The second LSB of the sel signal is '1' and thus signals with index "x1x", which include $a_{010}$ and $a_{110}$, are selected and routed to the next stage.
Figure 15.4  Tree-shaped 8-to-1 multiplexer.

Figure 15.5  Routing with sel = "110".
• Stage 0 (the rightmost stage): The MSB of the sel signal is '1' and thus the signal with index "lxx", which is $a_{110}$, is selected and routed to the output.

We can develop the VHDL code following the basic connection pattern of Figure 15.5. Note that the basic structure of the multiplexer is similar to the tree-shaped reduced-xor circuit of Section 15.3.1. Thus, the code of the reduced-xor circuit can be modified for the multiplexer. The VHDL code using the for loop statement is listed in Listing 15.8.

Listing 15.8  Parameterized tree-shaped multiplexer

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity muxl is
  generic(WIDTH: natural);
  port(  
a: in std_logic_vector(WIDTH-1 downto 0);
  sel: in std_logic_vector(log2c(WIDTH)-1 downto 0);
y: out std_logic
  );
end muxl;

architecture loop-tree-arch of muxl is
  constant STAGE: natural:= log2c(WIDTH);
signal p:
    std_logic_2d( STAGE downto 0, 2**STAGE-1 downto 0);
begin
  process(a,sel,p)
  begin
    for i in 0 to (2**STAGE-1) loop
      if i < WIDTH then
        p(STAGE,i) <= a(i);  -- rename input signal
      else
        p(STAGE,i) <= '0';  -- padding 0's
      end if;
    end loop;
    -- replicated structure
    for s in (STAGE-1) downto 0 loop
      for r in 0 to (2**s-1) loop
        if sel((STAGE-1)-s)='0' then
          p(s,r) <= p(s+1,2*r);
        else
          p(s,r) <= p(s+1,2*r+1);
        end if;
      end loop;
    end loop;
    -- rename output signal
    y <= p(0,0);
  end process;
end loop-tree-arch;
```

The code is identical to that in Listing 15.6 except that we replace the xor gate

$$p(s,r) <= p(s+1,2*r) \text{ xor } p(s+1,2*r+1);$$
with a 2-to-1 multiplexer:

\[
\text{if } \text{sel}((\text{STAGE}-1)-s)='0' \text{ then}
\]
\[
p(s,r) \leftarrow p(s+1,2*r);
\]
\[
\text{else}
\]
\[
p(s,r) \leftarrow p(s+1,2*r+1);
\]
\[
\text{end if;}
\]

**Behavioral description** If the input of a multiplexer is represented as an array, as in the code of Listing 15.8, the multiplexing can be considered as an indexing function that uses the \text{sel} signal as an index to select an element from the array. Based on this observation, we can derive the behavioral VHDL code, as shown in Listing 15.9.

**Listing 15.9** Behavioral description of a multiplexer

```vhdl
architecture beh_arch of mux1 is
begin
  y <= a(to_integer(unsigned(sel)));
end beh_arch;
```

We have used the complex index expressions before. However, these expressions are *static*, which means that their values are determined during the elaboration process, and no physical circuit will be inferred. On the other hand, the index expression in the *beh_arch* architecture depends on the \text{sel} input. This implies that the expression is *dynamic* and will infer a multiplexing circuit.

In the ideal case, the synthesis software recognizes this expression, and a predesigned, optimized multiplexer is inferred from the device library accordingly. We can use a simple one-line code to obtain an efficient implementation. However, not all synthesis software accepts the dynamic expression in array index, and thus the code is less portable.

**Two-dimensional description** In Section 15.2.4, we extended the multiplexer to accommodate two-dimensional input data. The code follows the cascading priority routing network of the one-dimensional design and suffers the same performance problem.

We can follow the process in Section 15.2.4 and extend the tree-shaped multiplexer to accept two-dimensional input data as well. The extension requires the use of a three-dimensional data type to represent the internal signal. This can be done by defining a new genuine data type like std_logic_2d or creating a new index function to emulate the three-dimensional data type with a one-dimensional array.

Alternatively, we can construct a two-dimensional multiplexer by duplicating the existing one-dimensional multiplexers. The VHDL code is shown in Listing 15.10. The \text{a} signal is converted into an array-of-arrays data type internally, and a for generate statement creates multiple instances of one-dimensional multiplexers.

**Listing 15.10** Two-dimensional multiplexer using one-dimensional multiplexers

```vhdl
architecture from_mux1d_arch of mux2d is
type aoa_transpose_type is
  array(B-1 downto 0) of std_logic_vector(P-1 downto 0);
signal aa: aoa_transpose_type;
component mux1 is
generic (WIDTH: natural);
port(
a: in std_logic_vector(WIDTH-1 downto 0);
```
15.3.4 Binary encoder

A binary encoder is a circuit that converts a one-hot input into a binary representation. The width of the input is normally a power of 2, and only 1 bit of the input is asserted. The function table of an 8-to-3 binary encoder is shown in Table 15.1. One unique characteristic of a binary encoder is the number of don’t-care input combinations. For an n-bit input, \(2^n - n\) combinations are not used. This can lead to significant circuit reduction.

The circuit can easily be constructed by observing the function table. The logic expressions of the previous 8-to-3 binary encoder are

\[
\begin{align*}
    b_2 &= a_7 + a_6 + a_5 + a_4 \\
    b_1 &= a_7 + a_6 + a_3 + a_2 \\
    b_0 &= a_7 + a_5 + a_3 + a_1
\end{align*}
\]
Deriving an abstract parameterized code for the binary encoder is not very hard. However, this kind of description tends to "overspecify" the circuit. For example, the priority encoder code of Listing 14.24 can also be used to describe a binary encoder. Although the circuit functions correctly, the overspecification leads to unnecessary circuit complexity.

One way to describe a more efficient implementation is to follow the pattern of the previous or expressions. Close observation shows that the $a_k$ bit will be included in the or expression of $b_i$ if the following condition is met:

\[
\frac{k}{2^i} \mod 2 = 1
\]

For example, let $i = 1$. For an 8-to-3 binary encoder, the range of $k$ is between 0 and 7, and the condition is satisfied when $k$ is 7, 6, 3 and 2. Thus, the or expression of $b_1$ can be written as $a_7 + a_6 + a_3 + a_2$.

To accommodate the condition, we create a mask table mirroring the desired patterns and apply the pattern to enable the desired bits. For example, the mask table of the previous 8-to-3 binary encoder is

- "11110000"
- "11001100",
- "10101010",

To obtain $b_2$, we can perform the and operation between the a input and the first row of the mask table and then perform reduced-or operation over the result. This scheme is coded in Listing 15.11. We define a function, gen_or_mask, to generate the mask table with an array-of-arrays data type and then use it to disable the unneeded bits. The circuit is described by a nested two-level for loop statement. The outer loop iterates through the $\log_2 n$ output bits, and the inner loop performs the reduced-or operation over the masked input. The code for the reduced-or circuit represents a cascading structure. If needed, we can revise it to make a tree-shaped implementation, as the reduced-xor circuit in Section 15.3.1. This is probably not necessary since the synthesis software should be able to handle such a simple circuit.

**Listing 15.11** Parameterized binary encoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.util_pkg.all;
entity bin_encoder is
  generic (N: natural);
  port(
    a: in std_logic_vector(N-1 downto 0);
    bcode: out std_logic_vector(log2c(N)-1 downto 0);
  );
end bin_encoder;

architecture para_arch0 of bin_encoder is
  type mask_2d_type is array(log2c(N)-1 downto 0) of std_logic_vector(N-1 downto 0);
  signal mask: mask_2d_type;
  function gen_or_mask return mask_2d_type is
    variable or_mask: mask_2d_type;
  begin
    for i in (log2c(N)-1) downto 0 loop
```


for k in (N-1) downto 0 loop
    if (k/(2**i) mod 2) = 1 then
        or_mask(i)(k) := '1';
    else
        or_mask(i)(k) := '0';
    end if;
end loop;
end loop;
return or_mask;
end function;

begin
    mask <= gen_or_mask;
    process (mask, a)
        variable tmp_row: std_logic_vector(N-1 downto 0);
        variable tmp_bit: std_logic;
        begin
            for i in (log2c(N)-l) downto 0 loop
                tmp_row := a and mask(i);
                -- reduced or operation
                tmp_bit := '0';
                for k in (N-1) downto 0 loop
                    tmp_bit := tmp_bit or tmp_row(k);
                end loop;
                bcode(i) <= tmp_bit;
            end loop;
        end process;
    end para_arch0;

Note that the gen_or_mask function and the mask operation are static. The masked bits will become 0’s during elaboration process and be removed from the physical circuit during synthesis.

15.3.5 Barrel shifter

In Section 7.4.4, we studied the design of a fixed-size 8-bit rotating-right circuit. It consists of three stages of shifting-multiplexing circuits. According to the value of the control signal, the input can be either passed directly to the output or shifted by a fixed amount. The amount of shifting doubles in each stage, from $2^0$ to $2^1$ and $2^2$. The 3-bit selection signal controls the three shifting-multiplexing circuits. After an input signal passes through three stages, the total shifted amount is the summation of the three individual stages set by the selection signal.

This is an efficient implementation for several reasons. First, as the number of inputs increases, the number of stages grows on the order of $O(\log_2 n)$. The length of the critical path grows in the same order, and thus its performance is much better than the cascading chain. Second, the circuit exhibits a regular two-dimensional structure and thus is easier for the synthesis and placement and routing software to obtain better results. Finally, recall that shifting a fixed amount requires only reconnection of the input and output signals. The shifting–multiplexing circuit is essentially a simple 2-to-1 multiplexer. Because of the regular structure, this scheme can be extended easily to accommodate parameterized design.
To make the parameterized shifting circuit more flexible, we include a feature parameter to indicate the type of shift operation, which can be shifting left, rotating left, shifting right and rotating right. The design starts with the shifting–multiplexing module. The basic block diagram is shown in Figure 15.6(a). The VHDL code of the parameterized shifting–multiplexing module is shown in Listing 15.12. The code includes three parameters. The WIDTH generic specifies the size of the circuit, the S_AMT generic specifies the amount to be shifted and the S_MODE generic specifies the type of shifting operation. Four if generate statements generate the desired amount of shifting or rotation, and the result is passed to a 2-to-1 multiplexer. Note that the shifted amount is determined by the S_AMT generic and thus is static. The shifting/rotation circuit involves only reconnection of the signals.

Listing 15.12 Parameterized fixed-size shifting–multiplexing module

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.util_pkg.all;
entity fixed_shifter is
  generic(
    WIDTH: natural;
    S_AMT: natural;
    S_MODE: natural
  );
  port(
    s_in: in std_logic_vector(WIDTH-1 downto 0);
    shift: in std_logic;
    s_out: out std_logic_vector(WIDTH-1 downto 0)
  );
end fixed_shifter;

architecture para_arch of fixed_shifter is
  constant L_SHIFT: natural :=0;
  constant R_SHIFT: natural :=1;
  constant L_ROTAT: natural :=2;
  constant R_ROTAT: natural :=3;
  signal sh_tmp, zero: std_logic_vector(WIDTH-1 downto 0);
begin
  zero <= (others=>'0');
  -- shift left
  l_sh_gen:
  if S_MODE=L_SHIFT generate
    sh_tmp <= s_in(WIDTH-S_AMT-1 downto 0) &
      zero(WIDTH-1 downto WIDTH-S_AMT);
  end generate;
  -- rotate left
  l_rt_gen:
  if S_MODE=L_ROTAT generate
    sh_tmp <= s_in(WIDTH-S_AMT-1 downto 0) &
      s_in(WIDTH-1 downto WIDTH-S_AMT);
  end generate;
  -- shift right
  r_sh_gen:
  if S_MODE=R_SHIFT generate
    sh_tmp <= zero(S_AMT-1 downto 0) &
```
The block diagram of a general 8-bit three-stage barrel shifter is shown in Figure 15.6(b). Each stage is a shifting–multiplexing module, and the $i$th bit of the $amt$ signal is connected to the $shift$ signal of the $i$th stage. The amount of shifting is determined by the stage and is $2^i$ for the $i$th stage. The VHDL code is shown in Listing 15.13. We assume that the value of input (i.e., the WIDTH parameter) is a power of 2.

**Listing 15.13** Parameterized barrel shifter

```
library ieee;
use ieee.std_logic_1164.all;
use work.util_pkg.all;
entity barrel_shifter is
```

(a) Block diagram of a shifting–multiplexing module

(b) Block diagram of an 8-bit three-stage barrel shifter

```
s_in(WIDTH-1 downto S_AMT);
end generate;
— rotate right
r_rt_gen:
if S_MODE=R_ROTAT generate
    sh_tmp <= s_in(S_AMT-1 downto 0) & s_in(WIDTH-1 downto S_AMT);
end generate;
— 2-to-1 multiplexer
s_out <= sh_tmp when shift='1' else s_in;
end para_arch;
```
```vhdl
5  generic(
    WIDTH: natural;
    S_MODE: natural
  );
port(
  a: in std_logic_vector(WIDTH-1 downto 0);
  amt: in std_logic_vector(log2c(WIDTH)-1 downto 0);
  y: out std_logic_vector(WIDTH-1 downto 0)
);
end barrel_shifter;
architecture para_arch of barrel_shifter is
constant STAGE: natural := log2c(WIDTH);
type std_aoa_type is array(STAGE downto 0) of
  std_logic_vector(WIDTH-1 downto 0);
signal p: std_aoa_type;
component fixed_shifter is
  generic(
    WIDTH: natural;
    S_AMT: natural;
    S_MODE: natural
  );
port(
  s_in: in std_logic_vector(WIDTH-1 downto 0);
  shft: in std_logic;
  s_out: out std_logic_vector(WIDTH-1 downto 0)
);
end component;
begi
  p(0) <= a;
  stage_gen:
    for s in 0 to (STAGE-1) generate
      shift_slice: fixed_shifter
      generic map(WIDTH=>WIDTH, S_MODE=>S_MODE, S_AMT=>2**s)
      port map(s_in=>p(s), s_out=>p(s+1), shft=>amt(s));
    end generate;
y <= p(STAGE);
end para_arch;
```

### 15.4 MORE SOPHISTICATED EXAMPLES

We study more sophisticated design examples in this section, including a reduced-xor-vector circuit and cell-based combinational multiplier, which exhibit more complex two-dimensional structures, and a priority encoder and FIFO, which are constructed using pre-designed parameterized RT-level components.
15.4.1 Reduced-xor-vector circuit

The reduced-xor-vector circuit was explained in Section 7.4.2. It performs the xor operation over successive ranges of the input. For example, for a 4-bit input $a_3a_2a_1a_0$, the circuit returns four values: $a_0, a_1 \oplus a_0, a_2 \oplus a_1 \oplus a_0$ and $a_3 \oplus a_2 \oplus a_1 \oplus a_0$.

**Cascading-chain structure** We discussed two implementations in Section 7.4.2. The linear cascading implementation requires a minimal number of gates, and its VHDL code is very simple. The code of Listing 7.21 takes advantage of the VHDL array construct and can easily be modified to accommodate a parameterized design. The revised code is shown in Listing 15.14.

```
library ieee;
use ieee.std_logic_1164.all;
use work.util.pkg.all;
entity reduced_xor_vector is
  generic(N: natural);
  port(
    a: in std_logic_vector(N-1 downto 0);
    y: out std_logic_vector(N-1 downto 0)
  );
end reduced_xor_vector;

architecture linear_arch of reduced_xor_vector is
  signal p: std_logic_vector(N-1 downto 0);
begin
  p <= (p(N-2 downto 0) & '0') xor a;
  y <= p;
end linear_arch;
```

The cascading structure experiences a large propagation delay. For an $N$-bit input, the critical path includes $N$ xor gates.

**Parallel-prefix structure** A more efficient structure was shown in Figure 7.8(b), which reduces the critical path to $\log_2 N$ xor gates and achieves the maximal amount of sharing. The interconnection is arranged according to a special class of structures based on the parallel-prefix algorithm.

The connection structure of this circuit is more involved. To better understand the connection pattern, we rename the signals in the circuit diagram of Figure 7.8(b) and add some pass-through boxes. The revised diagram is shown in Figure 15.7.

Assume that a reduced-xor-vector circuit has $N$-bit input and $N = 2^n$. The circuit can be divided into $n$ stages, each containing $2^n$ blocks (rows). A block can be an xor gate or an empty pass-through box. We number the stages from left to right and the rows from top to bottom. For the $i$th row in the $s$th stage, its output is labeled as $p_{si}$. An 8-bit circuit is shown in Figure 15.7.

Closer observation of the diagram shows that it follows a simple pattern. Consider the $s$th stage:

- The stage is divided into $2^{n-s}$ modules. Each module contains $2^s$ blocks and is shown as a shaded rectangle in Figure 15.7.
- The top-half blocks of the module are pass-through boxes. The input of a box is connected to the output from the same row of the preceding stage.
• The bottom-half blocks of the module are xor gates. One input of an xor gate is connected to the output from the same row of the preceding stage. The other input is the same for all xor gates in the module. It is from the output whose row index is one smaller than the index of the top xor gate in the module.

For example, consider the second stage in the diagram. We can divide it into two $2^2$ modules. In the first module, the top half of the first module, whose outputs are labeled $p_{20}$ and $p_{21}$, is connected to $p_{10}$ and $p_{11}$. The outputs of the bottom half of the module are labeled $p_{22}$ and $p_{23}$. In addition to the $p_{12}$ and $p_{13}$ signals, the xor gates share a common input, the $p_{11}$ signal. The second module has a similar pattern. Note that the $p_{15}$ signal is connected to the xor gates whose outputs are labeled $p_{26}$ and $p_{27}$.

The VHDL code is shown in Listing 15.15. We assume that the number of elements of the $a$ input is a power of 2.

### Listing 15.15 Parameterized parallel-prefix reduced-xor-vector circuit

```vhdl
architecture para_prefix_arch of reduced_xor_vector is
constant ST: natural := log2c(N);
signal p: std_logic_2d(ST downto 0, N-1 downto 0);
begn
process(a,p)
begn
    -- rename input
    for i in 0 to (N-1) loop
        p(0,i) <= a(i);
    end loop;
    -- main structure
    for s in 1 to ST loop
```

---

**Figure 15.7** Parallel-prefix reduced-xor-vector circuit.
The main structure is described by a nested three-level for loop statement. The outer loop specifies the iterations over ST stages:

```plaintext
for s in 1 to ST loop
  for k in 0 to (2**(ST-s)-1) loop
    — 1st half: pass-through boxes
    for i in 0 to (2**(s-1)-1) loop
      p(s, k*(2**s)+i) <= p(s-1, k*(2**s)+i);
    end loop;
    — 2nd half: xor gates
    for i in (2**(s-1)) to (2**s-1) loop
      p(s, k*(2**s)+i) <=
      p(s-1, k*(2**s)+i) xor p(s-1, k*(2**s-2**s+2**(s-1)-1));
    end loop;
  end loop;
  — rename output
  for i in 0 to N-1 loop
    y(i) <= p(ST,i);
  end loop;
end process;
end para_prefix_arch;
```

The first inner loop iterates through the pass-through boxes and the second inner loop iterates through the xor gates. Note that the loop index represents half of the number of the blocks in a module.

### 15.4.2 Multiplier

Multiplication is a frequently needed arithmetic operation and its synthesis is not supported by all software. Two fixed-size implementations were discussed earlier, including an adder-based combinational multiplier in Section 11.6 and a sequential multiplier in Section 7.5.4. In this section, we convert the previous implementations to parameterized modules and also introduce a more efficient cell-based design.

**Sequential multiplier** The sequential multiplier utilizes a simple shift-and-add algorithm to iterate additions sequentially through a single adder. Since the algorithm can be applied for any input width, the design can be easily parameterized.

The original fixed-size 8-bit multiplier code is shown in Listing 11.8. Various array boundaries, initial values, and test conditions are based on the input width. To convert the code into a parameterized design, we just need to represent these values in terms of the WIDTH generic. The revised code is shown in Listing 15.16.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity seq_mult_para is
  generic(WIDTH: natural);
  port(
    clk, reset: in std_logic;
    start: in std_logic;
    a_in, b_in: in std_logic_vector(WIDTH-1 downto 0);
    ready: out std_logic;
    r: out std_logic_vector(2*WIDTH-1 downto 0)
  );
end seq_mult_para;

architecture shift_add_better_arch of seq_mult_para is
  constant C_WIDTH: integer:=log2c(WIDTH)+1;
  constant C_INIT: unsigned(C_WIDTH-1 downto 0)
    :=to_unsigned(WIDTH,C_WIDTH);
  type state_type is (idle, add_shift);
  signal state_reg, state_next: state_type;
  signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
  signal n_reg, n_next: unsigned(C_WIDTH-1 downto 0);
  signal p_reg, p_next: unsigned(2*WIDTH downto 0);
  alias pu_next: unsigned(WIDTH-1 downto 0) is p_next(2*WIDTH downto WIDTH);
  alias pu_reg: unsigned(WIDTH downto 0) is p_reg(2*WIDTH downto WIDTH);
  alias pl_reg: unsigned(WIDTH-1 downto 0) is p_reg(WIDTH-1 downto 0);
begin
  -- state and data registers
  process(clk, reset)
  begin
    if reset='1' then
      state_reg <= idle;
      a_reg <= (others=>'0');
      n_reg <= (others=>'0');
      p_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      a_reg <= a_next;
      n_reg <= n_next;
      p_reg <= p_next;
    end if;
  end process;
  -- combinational circuit
  process(start, state_reg, a_reg, n_reg, p_reg, a_in, b_in,
    n_next, p_next)
  begin
    a_next <= a_reg;
Adder-based combinational multiplier

The adder-based combinational multiplier uses an array of adders to perform additions in parallel, as discussed in Section 7.5.4. The revised block diagram of Section 9.4.3 illustrates the repetitive nature of this design. Our parameterized design is based on this structure. The block diagram is repeated in Figure 15.8. We modify the internal signal names to help us identify the input and output relationships of each stage.

To increase the flexibility of this module, we include two parameters, N and WITH_PIPE, in this design. The N generic specifies the width of the operand, and the WITH_PIPE generic indicates whether to add a pipeline to the multiplier. If the pipeline is desired, registers will be inserted between the stages.

The VHDL code is shown in Listing 15.17. Two array-of-arrays data types are defined for the internal signals. The std_aoan_type data type is used for the propagated operands, and the std_aoa_2n_type data type is used to represent the partial product and the bit product. The code includes three major parts. The first part is composed of two if generate statements, which either generate buffer registers between stages or serve as a direct connection. The second part is the process that generates the bit product vector. The bit product in the ith
Figure 15.8  Adder-based combinational multiplier with new signal labels.
stage is represented by the $bp(i)$ signal, which is in the form of $0 \cdots 0 a_{n-1} b_i \cdots a_0 b_i \cdots 0$. There are $N - i$ and $i$ padding 0's in the front and end respectively. The process includes two for loop statements, one for the two boundary bit products (i.e., $bp(0)$ and $bp(1)$) and the other for regular stages. The third part specifies the addition operation in each stage. It includes a for generate statement for the middle stages and special signal connections for the first and the last stages.

Listing 15.17  Parameterized adder-based combinational multiplier

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity multn is
  generic(
    N: natural;
    WITH_PIPE: natural
  );
  port(
    clk, reset: std_logic;
    a, b: in std_logic_vector(N-1 downto 0);
    y: out std_logic_vector(2*N-1 downto 0)
  );
end multn;

architecture n-stage_pipe_arch of multn is
  type std_aoa_n_type is
    array(N-2 downto 1) of std_logic_vector(N-1 downto 0);
  type std_aoa_2n_type is
    array(N-1 downto 0) of unsigned(2*N-1 downto 0);
  signal a_reg, a_next, b_reg, b_next: std_aoa_n_type;
  signal bp, pp_reg, pp_next: std_aoa_2n_type;
begin
  — part 1
  — without pipeline buffers
  g_wire:
  if (WITH_PIPE/=l) generate
    a_reg <= a_next;
    b_reg <= b_next;
    pp_reg(N-1 downto 1) <= pp_next(N-1 downto 1);
  end generate;
  — with pipeline buffers
  g_reg:
  if (WITH_PIPE=l) generate
    process(clk,reset)
    begin
      if (reset = '1') then
        a_reg <= (others=>(others=>'0'));
        b_reg <= (others=>(others=>'0'));
        pp_reg(N-1 downto 1) <= (others=>(others=>'0'));
      elsif (clk'event and clk='1') then
        a_reg <= a_next;
        b_reg <= b_next;
        pp_reg(N-1 downto 1) <= pp_next(N-1 downto 1);
    end process;
  end generate;
end n-stage_pipe_arch;
```
end if;
end process;
end generate;

-- part 2

-- bit-product generation
process(a, b, a_reg, b_reg)
begin
  -- bp(0) and bp(1)
  for i in 0 to 1 loop
    bp(i) <= (others=>'0');
    for j in 0 to N-1 loop
      bp(i)(i+j) <= a(j) and b(i);
    end loop;
  end loop;

  -- regular bp(i)
  for i in 2 to (N-1) loop
    bp(i) <= (others=>'0');
    for j in 0 to (N-1) loop
      bp(i)(i+j) <= a_reg(i-1)(j) and b_reg(i-1)(i);
    end loop;
  end loop;
end process;

-- part 3

-- addition of the first stage
pp_next(1) <= bp(0) + bp(1);
a_next(1) <= a;
b_next(1) <= b;

-- addition of the middle stages
for i in 2 to (N-2) generate
  pp_next(i) <= pp_reg(i-1) + bp(i);
a_next(i) <= a_reg(i-1);
b_next(i) <= b_reg(i-1);
end generate;

-- addition of the last stage
pp_next(N-1) <= pp_reg(N-2) + bp(N-1);
renam output
y <= std_logic_vector(pp_reg(N-1));
end n_stage_pipe_arch;

Cell-based carry-ripple combinational multiplier  The previous adder-based multiplier utilizes “coarse” RT-level parts, namely the 2N-bit adders. The alternative is to use a 1-bit full-adder cell as the basic building block. This allows us to explore the “fine” structure of the multiplier and derive a more efficient circuit.

The multiplication of two 4-bit binary numbers is shown in Figure 15.9. The operation can be considered as the summation over the $a_i b_j$ terms, which are aligned in a specific two-dimensional pattern.

The $a_i b_j$ term returns a 1-bit value, and the addition of any two terms can be done by a 1-bit adder, which is commonly known as a full adder. The input of a full adder includes two 1-bit operands, $a_i$ and $b_j$, and a 1-bit carry-in, $c_i$, and the output includes a sum bit, $s_o$, and a carry-out, $c_o$. The gate-level VHDL description is shown in Listing 15.18. For
**Figure 15.9** Multiplication as a summation of $a_i b_j$ terms.

In most ASIC technologies, there is a predesigned full-adder cell in the device library, and it will be inferred during synthesis.

### Listing 15.18 1-bit full adder

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity fa is
  port(
    ai, bi, ci: in std_logic;
    so, co: out std_logic
  );
end fa;

architecture arch of fa is
begin
  so <= ai xor bi xor ci;
  co <= (ai and bi) or (ai and ci) or (bi and ci);
end arch;
```

To summate the $a_i b_j$ terms, we can arrange the full-adder cells according to the two-dimensional structure of multiplication operation in Figure 15.9. Two common arrangements are *carry-ripple architecture* and *carry-save architecture*. We study the carry-ripple multiplier in this subsection and the carry-save multiplier in the next subsection.

The block diagram of a 4-bit carry-ripple multiplier is shown in Figure 15.10. Because the carry is propagated (i.e., rippled) from the LSB to the MSB stage by stage, this arrangement is known as the *carry-ripple architecture*. In the diagram, each full adder cell is given an index and expressed as $FA_{ij}$, indicating that the cell is located in the $i$th row and the $j$th column. For a non-boundary cell, such as $FA_{21}$ and $FA_{22}$ in the diagram, the input and output signals of the $FA_{ij}$ cell follow a specific pattern:

- The $ci$ port is connected to the $c_{i,j}$ signal.
- The $co$ port is connected to the $c_{i+1,j}$ signal, which becomes the carry-in of the $FA_{i+1,j}$ cell.
- The $so$ port is connected to the $s_{i,j}$ signal, which is connected to the $bi$ port of the $FA_{i+1,j-1}$ cell.
- The $ai$ port is connected to the $a_i b_j$ term.
- The $bi$ port is connected to the $s_{i-1,j+1}$ signal, which is the $so$ signal of the $FA_{i-1,j+1}$ cell.
Figure 15.10 Cell-based carry-ripple combinational multiplier.

The boundary cells are located in the top and bottom rows, and the leftmost and rightmost columns. Their connections are modified as follows:

- **Top row**: The bi port of the FA_{1,j} cell is connected to the a_0 b_{j+1} term. Note that the b_4 bit does not exist and the leftmost term (i.e., a_0 b_4 in the diagram) is used for the naming convention. The a_0 b_4 term is actually connected to '0'.
- **Bottom row**: The so ports of the cells and the co port of the leftmost cell form the top portion of the final result.
- **Rightmost column**: The ci port of the FA_{10} cell is connected to '0'. The so ports of the cells form the lower portion of the final result.
- **Leftmost column**: The bi port of the FA_{14} cell is connected to the co port from the leftmost cell in the previous row. In other words, the c_{i,3} signal is used in the place of the s_{i,3} signal.

Once identifying the normal and boundary connection patterns and the signal naming convention, we can derive the VHDL description accordingly. The code is shown in Listing 15.19. We define an array-of-arrays type for the internal bit-product, carry and sum signals. The code is divided into several segments. The first segment is a nested two-level for generate statement that generates the ab signal, which consists of all a_i ∙ b_j terms. The second segment specifies the connection patterns for the leftmost and rightmost columns. The third segment specifies the input signal of the top row. The fourth segment is a nested two-level for generate statement that instantiates the two-dimensional N-by-(N - 1) full-adder cells of the middle rows. The last segment uses the sum signals of the bottom row and rightmost column to form the final result.

```
library ieee;
use ieee.std_logic_1164.all;
entity mult_array is
generic(N: natural);
port(
a_in, b_in: in std_logic_vector(N-1 downto 0);
y: out std_logic_vector(2*N-1 downto 0)
);
PARAMETERIZED DESIGN: PRACTICE

Although the appearance of this code is different from that of the previous adder-based code in Listing 15.17, the circuit it describes is very similar. Each row of the full-adder cells in Figure 15.10 forms a 4-bit ripple adder. Thus, this code actually describes a ripple adder-based combinational multiplier.
The fine granularity does provide more information about the underlying implementation and helps us better understand the operation of this circuit. For example, our previous pipelined design inserts pipeline registers for the sum output of the adders, as shown in Figure 15.11. These are not the optimal locations since no signal can be passed to the next row until the slowest carry bit (i.e., the MSB) becomes available.

A better division can be obtained by examining the signal propagation in the cell-level diagram. If we assume that the propagation delay of a full-adder cell is $T_{fa}$ and the delay of obtaining $a_i \cdot b_j$ is negligible, the signal propagation from the LSB of the top row to the MSB of the bottom row is shown in Figure 15.12. The propagation is shown as a set of contour lines, each representing an increment of a delay of $T_{fa}$. Recall that a good pipelined design should divide the combinational circuit into stages of similar propagation delays. The pipeline registers should be inserted along these contour lines.

The contour lines also help us to identify the critical paths. One path is marked as a thick dashed line in Figure 15.12. For an $N$-bit multiplier, there are $N - 1$ rows, each consisting of $N$ full-adder cells. The critical path includes $N$ cells in the top row and two cells of each remaining $N - 2$ rows. Thus, the propagation delay is

$$NT_{fa} + 2(N - 2)T_{fa} = (3N - 4)T_{fa}$$

**Cell-based carry-save combinational multiplier** The carries of the carry-ripple architecture form a cascading chain and introduce a large propagation delay. Instead of propagating the carry to the next cell in the same row, an alternative is to "save" the carry outputs and pass them to the cells in the next row, where they are summed in parallel. This is known as the carry-save architecture. The block diagram of a 4-bit carry-save combinational multiplier is shown in Figure 15.13. In the first three rows, a full-adder cell adds the $a_i b_j$ term and the sum bit (i.e., $s_0$) and the carry-out bit (i.e., $c_0$) from the previous row, and passes the results to the next row. The arrangement in each row represents a carry-save adder. The cells in the last row are arranged as a regular carry-ripple adder,
Figure 15.12  Propagation delay contour lines of a carry-ripple multiplier.

Figure 15.13  Cell-based carry-save multiplier.
which summates the carry-out signals from the last carry-save adder and forms the final result.

The derivation of the VHDL code is similar to that of the cell-based carry-ripple multiplier. We first identify the connection pattern of a non-boundary cell and then specify the special requirements for the cells in the first and last rows and the leftmost and rightmost columns. The complete VHDL code is shown in Listing 15.20.

**Listing 15.20** Parameterized cell-based carry-save combinational multiplier

```vhdl
architecture carry_save_arch of mult_array is
  type two_d_type is
    array(N-1 downto 0) of std_logic_vector(N-1 downto 0);
  signal ab, c, s: two_d_type;
  signal rs, rc: std_logic_vector(N-1 downto 0);
  component fa
    port(
      ai, bi, ci: in std_logic;
      so, co: out std_logic
    );
  end component;
begin
  -- bit product
  g_ab_row:
  for i in 0 to N-1 generate
    g_ab_col: for j in 0 to (N-1) generate
      ab(i)(j) <= a_in(i) and b_in(j);
    end generate;
  end generate;
  -- leftmost column
  g_N_col:
  for i in 1 to (N-1) generate
    s(i)(N-1) <= ab(i)(N-1);
  end generate;
  -- top row
  s(0) <= ab(0);
  c(0) <= (others=>'0');
  -- middle rows
  g_fa_row:
  for i in 1 to (N-1) generate
    g_fa_col: for j in 0 to (N-2) generate
      u_middle: fa
        port map
          (ai=>ab(i)(j), bi=>s(i-1)(j+1), ci=> c(i-1)(j),
           so=>s(i)(j), co=>c(i)(j));
    end generate;
  end generate;
  -- bottom row ripple adder
  rc(0) <= '0';
  g_acell_N_row:
  for j in 0 to (N-2) generate
    unit_N_row: fa
      port map (ai=>s(N-1)(j+1), bi=>c(N-1)(j), ci=> rc(j),
              so=>rs(j), co=>rc(j+1));
  end generate;
end carry_save_arch;
```

Figure 15.14 Propagation delay contour lines of a carry-save multiplier.

---

output signal

g_out:
  for i in 0 to (N-1) generate
    y(i) <= s(i)(0);
  end generate;
  y(2*N-2 downto N) <= rs(N-2 downto 0);
  y(2*N-1) <= rc(N-1);
end carry_save_arch;

The propagation of the carries is much easier to trace for the carry-save multiplier. The propagation delay contour lines and the critical path are shown in Figure 15.14. For an N-bit multiplier, the critical path includes N - 1 cells in the bottom row and one cell of each remaining N - 1 rows. Thus, the propagation delay becomes

\[(N - 1)Tfa + (N - 1)Tfa = (2N - 2)Tfa\]

This value is about two-thirds of the delay of the previous ripple-carry multiplier. Furthermore, since the single ripple adder in the last row accounts for one-half of the delay, we can replace it with a faster adder architecture to further improve the performance.

Because of the clear propagation delay contour lines, we can easily divide the carry-save multiplier into stages of identical delays and convert it to a pipelined design. The sketch of the location of the pipeline registers is shown in Figure 15.15. The cells in the last row are rearranged for clarity. To reduce cluttering, the pipeline registers for the operands are not included.
Figure 15.15 Pipelined carry-save multiplier.
### 15.4.3 Parameterized LFSR

The LFSR was discussed in Section 9.2.3. Its feedback circuit is simple and involves only one or three xor gates, as shown in Table 9.1. Despite its simplicity, the xor expression depends on the size of the shift register and is determined on an ad hoc basis. One way to parameterize the xor expression is to list all of the expressions in a table. Each row of the table corresponds to a specific size and indicates which register bits are needed in the expression. For example, the feedback expression of a 5-bit LFSR is \( q_2 \oplus q_0 \), and the corresponding row is "00101". The table can be considered as a mask table, and the pattern in each row can be used to enable or disable the corresponding bits. Consider the previous example. The "00101" pattern can function as a mask. After performing a bitwise and operation between the mask pattern and \( q_4q_3q_2q_1q_0 \), we obtain \( 00q_20q_0 \). The feedback circuit can be obtained by applying reduced-xor operation (i.e., \( 0 \oplus 0 \oplus q_2 \oplus 0 \oplus q_0 \)) over the result. Since \( x \oplus 0 = x \), the 0's will be removed during synthesis, and the expression will be simplified to \( q_2 \oplus q_0 \).

There is no algorithm to generate the mask table. It must be exhaustively listed. Following Table 9.1, we can define the mask table as a constant of a two-dimensional array-of-arrays data type:

```vhdl
type tap_array_type is array(2 to MAX_N) of std_logic_vector(MAX_N-1 downto 0);
constant TAP_CONST_ARRAY: tap_array_type :=
(2 => (1|0=>'1', others=>'0'),
3 => (1|0=>'1', others=>'0'),
4 => (1|0=>'1', others=>'0'),
5 => (2|0=>'1', others=>'0'),
   . . .);
```

The MAX_N term is a constant. It specifies the maximal range of the parameter.

Section 9.2.3 shows that we can use additional logic in the feedback path to include the all-zero pattern and make an \( n \)-bit LFSR circulate through all \( 2^n \) states. This can be made as an option in a parameterized LFSR.

The complete VHDL code is shown in Listing 15.21. There are two generics: \( N \), which specifies the size of the LFSR, and \( \text{WITH}_0 \), which specifies whether the all-zero pattern should be included. The MAX_N is chosen to be 8, and thus the range of \( N \) is between 2 and 8. The MAX_N can be enlarged by adding additional rows to TAP_CONST_ARRAY.

#### Listing 15.21 Parameterized LFSR

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity lfsr is
   generic(
      N: natural;
      \text{WITH}_0: natural
   );
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end lfsr;
```
architecture para_arch of lfsr is
  constant MAX_N: natural := 8;
  constant SEED: std_logic_vector(N-1 downto 0)
                := (0=>'1', others=>'0');
  type tap_array_type is array(2 to MAX_N) of
                 std_logic_vector(MAX_N-1 downto 0);
  constant TAP_CONST_ARRAY: tap_array_type :=
    (2 => (1|0=>'1', others=>'0'),
     3 => (1|0=>'1', others=>'0'),
     4 => (1|0=>'1', others=>'0'),
     5 => (2|0=>'1', others=>'0'),
     6 => (1|0=>'1', others=>'0'),
     7 => (3|0=>'1', others=>'0'),
     8 => (4|3|2|0=>'1', others=>'0'));
signal r_reg, r_next: std_logic_vector(N-1 downto 0);
signal fb, zero, fzero: std_logic;
begin
  — register
  process(clk,reset)
  begin
    if (reset='1') then
      r_reg <= SEED;
    elsif (clk'event and clk='1') then
      r_reg <= r_next;
    end if;
  end process;
  — next-state logic
  process(r_reg)
  begin
    constant TAP_CONST: std_logic_vector(MAX_N-1 downto 0)
                        := TAP_CONST_ARRAY(N);
    variable tmp: std_logic;
    tmp := '0';
    for i in 0 to (N-1) loop
      tmp := tmp xor (r_reg(i) and TAP_CONST(i));
    end loop;
    fb <= tmp;
  end process;
  — with all-zero state
  gen_zero:
  if (WITH_ZERO=1) generate
    zero <= '1' when r_reg(N-1 downto 1)=
            (r_reg(N-1 downto 1)’range=>’0’)
            else
               '0';
    fzero <= zero xor fb;
  end generate;
  — without all-zero state
  gen_no_zero:
  if (WITH_ZERO/=1) generate
    fzero <= fb;
  end generate;
  r_next <= fzero & r_reg(N-1 downto 1);
The xor feedback circuit is implemented by a for loop statement, in which the reduced-xor operation is performed over the masked register output. The optional logic to process the all-zero pattern is implemented by two if generate statements. One statement generates the logic, and the other just reconnects the original feedback signal.

15.4.4 Priority encoder

A parameterized priority encoder was described in Listing 14.24. The code maps to a one-dimensional cascading priority routing network, and thus the performance suffers. One way to improve the performance is to construct the circuit using a collection of smaller priority encoders and multiplexers, as discussed in Section 7.4.3. The structure is quite complex.

An alternative way is to first convert the input into one-hot code and then pass the code into a regular binary encoder. For example, if an 8-bit input is "00110101", it will be converted to "00100000" and then encoded as a one-hot input. The conversion process can be explained by an example. Consider an 8-bit priority encoder whose input is $a_7, a_6, \ldots, a_0$ and $a_7$ has the highest priority. Let the corresponding one-hot code be $t_7, t_6, \ldots, t_0$. For the $t_i$ bit to be asserted, the $a_i$ bit must be '1' and all the upper bits, which include $a_7, a_6, \ldots, a_{i+1}$, must be '0'. This can be translated into a logic expression:

$$t_i = a_i \cdot a'_7 \cdot a'_6 \cdots a'_{i+1}$$

The logic expression represents a variant of reduced-and operations. As for the reduced-xor circuit, we can describe the reduced-and circuit as a tree to improve its performance. The specific pattern of the and operations also provides an opportunity for further optimization. Let us first list all logic expressions:

- $t_7 = a_7$
- $t_6 = a_6 \cdot a'_7$
- $t_5 = a_5 \cdot a'_7 \cdot a'_6$
- $t_4 = a_4 \cdot a'_7 \cdot a'_6 \cdot a'_5$
- $t_3 = a_3 \cdot a'_7 \cdot a'_6 \cdot a'_5 \cdot a'_4$
- $t_2 = a_2 \cdot a'_7 \cdot a'_6 \cdot a'_5 \cdot a'_4 \cdot a'_3$
- $t_1 = a_1 \cdot a'_7 \cdot a'_6 \cdot a'_5 \cdot a'_4 \cdot a'_3 \cdot a'_2$
- $t_0 = a_0 \cdot a'_7 \cdot a'_6 \cdot a'_5 \cdot a'_4 \cdot a'_3 \cdot a'_2 \cdot a'_1$

If we ignore the first non-inverted element, the expressions become

- $a'_7$
- $a'_7 \cdot a'_6$
- $a'_7 \cdot a'_6 \cdot a'_5$
- $\ldots$
- $a'_7 \cdot a'_6 \cdot a'_5 \cdot a'_4 \cdot a'_3 \cdot a'_2$
- $a'_7 \cdot a'_6 \cdot a'_5 \cdot a'_4 \cdot a'_3 \cdot a'_2 \cdot a'_1$
The pattern is similar to the output of the reduced-xor-vector circuit discussed in Section 15.4.1. We can duplicate the code in Listing 15.15 to describe a reduced-and-vector circuit to take advantage of the sharing opportunity. The VHDL code is shown in Listing 15.22.

Listing 15.22 Parameterized parallel-prefix reduced-and-vector circuit

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.util_pkg.all;
entity reduced_and_vector is
  generic(N: natural);
  port( a: in std_logic_vector(N-1 downto 0);
       y: out std_logic_vector(N-1 downto 0));
end reduced_and_vector;

architecture para_prefix_arch of reduced_and_vector is
  constant ST: natural := log2c(N);
signal p: std_logic,2d(ST downto 0, N-1 downto 0);
begin
  process(a,p)
  begin
    -- rename input
    for i in 0 to (N-1) loop
      p(0,i) <= a(i);
    end loop;
    -- main structure
    for s in 1 to ST loop
      for k in 0 to (2**(ST-s)-1) loop
        -- 1st half: pass-through boxes
        for i in 0 to (2**(s-1)-1) loop
          p(s, k*(2**s)+i) <= p(s-1, k*(2**s)+i);
        end loop;
        -- 2nd half: and gates
        for i in (2**(s-1)) to (2**s-1) loop
          p(s, k*(2**s)+i) <=
          p(s-1, k*(2**s)+i) and
          p(s-1, k*(2**s)+2**(s-1)+1);
        end loop;
      end loop;
    end loop;
    -- rename output
    for i in 0 to (N-1) loop
      y(i) <= p(ST,i);
    end loop;
  end process;
end para_prefix_arch;
```

After developing the reduced-and-vector circuit, we can derive the VHDL code, as shown in Listing 15.23. The code uses the reduced-and-vector circuit and simple glue logic to generate the one-hot code and then pass it to a binary encoder. Two for loop statements are used to reverse the order of the input to match the convention used in the reduced-and-
vector circuit. Since the critical paths of the parallel-prefix reduced-and-vector circuit and the optimized binary encoder circuits are on the order of $O(\log_2 n)$, the performance of this circuit is much better than that of the cascading design.

<table>
<thead>
<tr>
<th>Listing 15.23 Parameterized priority encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>library ieee;</td>
</tr>
<tr>
<td>use ieee.std_logic_1164.all;</td>
</tr>
<tr>
<td>use ieee.numeric_std.all;</td>
</tr>
<tr>
<td>use work.util_pkg.all;</td>
</tr>
<tr>
<td>entity prio_encoder is</td>
</tr>
<tr>
<td>generic(N: natural);</td>
</tr>
<tr>
<td>port(</td>
</tr>
<tr>
<td>a: in std_logic_vector(N-1 downto 0);</td>
</tr>
<tr>
<td>bcode: out std_logic_vector(log2c(N)-1 downto 0)</td>
</tr>
<tr>
<td>);</td>
</tr>
<tr>
<td>end prio_encoder;</td>
</tr>
<tr>
<td>architecture para_arch of prio_encoder is</td>
</tr>
<tr>
<td>component reduced_and_vector is</td>
</tr>
<tr>
<td>generic(N: natural);</td>
</tr>
<tr>
<td>port(</td>
</tr>
<tr>
<td>a: in std_logic_vector(N-1 downto 0);</td>
</tr>
<tr>
<td>y: out std_logic_vector(N-1 downto 0)</td>
</tr>
<tr>
<td>);</td>
</tr>
<tr>
<td>end component;</td>
</tr>
<tr>
<td>component bin_encoder is</td>
</tr>
<tr>
<td>generic(N: natural);</td>
</tr>
<tr>
<td>port(</td>
</tr>
<tr>
<td>a: in std_logic_vector(N-1 downto 0);</td>
</tr>
<tr>
<td>bcode: out std_logic_vector(log2c(N)-1 downto 0)</td>
</tr>
<tr>
<td>);</td>
</tr>
<tr>
<td>end component;</td>
</tr>
<tr>
<td>signal a_not_rev: std_logic_vector(N-1 downto 0);</td>
</tr>
<tr>
<td>signal a_vec, a_vec_rev, t: std_logic_vector(N-1 downto 0);</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>― reverse a</td>
</tr>
<tr>
<td>gen_reverse_a:</td>
</tr>
<tr>
<td>for i in 0 to (N-1) generate</td>
</tr>
<tr>
<td>a_not_rev(i) &lt;= not a(N-1-i);</td>
</tr>
<tr>
<td>end generate;</td>
</tr>
<tr>
<td>― reduced and operation</td>
</tr>
<tr>
<td>unit_token: reduced_and_vector</td>
</tr>
<tr>
<td>generic map(N=&gt;N)</td>
</tr>
<tr>
<td>port map(a=&gt;a_not_rev, y=&gt;a_vec_rev);</td>
</tr>
<tr>
<td>― reverse the result</td>
</tr>
<tr>
<td>gen_reverse_t:</td>
</tr>
<tr>
<td>for i in 0 to (N-1) generate</td>
</tr>
<tr>
<td>a_vec(i) &lt;= a_vec_rev(N-1-i);</td>
</tr>
<tr>
<td>end generate;</td>
</tr>
<tr>
<td>― form one-hot code</td>
</tr>
<tr>
<td>t &lt;= a and (‘1’ &amp; a_vec(N-1 downto 1));</td>
</tr>
<tr>
<td>― regular binary encoder</td>
</tr>
<tr>
<td>unit_bin_code: bin_encoder</td>
</tr>
</tbody>
</table>
15.4.5 FIFO buffer

Implementation of a four-word FIFO buffer was discussed in Section 9.3.2. The code can be modified for a parameterized design. To achieve better performance, we use the previously developed modules to implement the circuit. The basic organization of the parameterized buffer is similar to that in Section 9.3.2, and its block diagram is shown in Figure 15.16. In the top level, the FIFO buffer is divided into a FIFO control circuit and a register file, which contains one write port and one read port. The control circuit contains two counters for the read and write pointers and the logic to generate full and empty status. The register file consists of a register array and a decoder to generate the proper enable signal and a multiplexer to route the desired value to output. The main components of the design hierarchy is shown in Figure 15.17.

For parameterized FIFO, we normally want to specify the width of a word (i.e., the number of bits in a word) and the size of the buffer (i.e., the number of words in the buffer). In our code, the B generic is used for the number of bits in a word. For simplicity, the buffer size is specified indirectly by the number of address bits of the buffer, represented by the W generic. To provide more flexibility and achieve better efficiency, we include a feature parameter, the CNT_MODE generic, to indicate whether binary or LFSR counters are used for the read and write pointers. Note that the sizes of the buffer for the binary and LFSR counter options are $2^W$ and $2^W - 1$ respectively.

The top-level VHDL code is shown in Listing 15.24. It is the instantiation of two components and a simple glue logic for the write enable signal of the register file. The codes of the register file and FIFO control circuit are discussed in the following two subsections.
Figure 15.17  Design hierarchy of a FIFO buffer.

Listing 15.24  Parameterized FIFO buffer top-level instantiation

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity fifo_top_para is
  generic(
    B: natural;  -- number of bits
    W: natural;  -- number of address bits
    CNT_MODE: natural -- binary or LFSR
  );
  port(
    clk, reset: in std_logic;
    rd, wr: in std_logic;
    w_data: in std_logic_vector (B-1 downto 0);
    empty, full: out std_logic;
    r_data: out std_logic_vector (B-1 downto 0)
  );
end fifo_top_para;

architecture arch of fifo_top_para is
  component fifo_sync_ctrl_para
generic(
    N: natural;
    CNT_MODE: natural
  );
  port(
    clk, reset: in std_logic;
    wr, rd: in std_logic;
    full, empty: out std_logic;
    w_addr, r_addr: out std_logic_vector(N-1 downto 0)
  );
  end component;
  component reg_file_para
generic(
    W: natural;
    B: natural
  );
  port(

```
The operation and implementation of a fixed-size register file was discussed in Section 9.3.1. It consists of a register array, write-enable decoding logic and an output multiplexing circuit. The parameterized code can simply follow the skeleton of the fixed-size VHDL code in Listing 9.15 and replace the original segments with a parameterized register array and the predeveloped parameterized decoder and multiplexer. The array-of-arrays data type is a natural match for the register array. However, since the input data type of the parameterized multiplexer is a genuine two-dimensional array, the output of the register array must first be converted to the proper data type and then mapped to the input of the multiplexer. The complete VHDL code is shown in Listing 15.25.

Listing 15.25 Structural description of a parameterized register file

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;

entity reg_file_para is
  generic(
    W: natural;
    B: natural
  );
  port(
    clk, reset: in std_logic;
    wr_en: in std_logic;
    w_data: in std_logic_vector(B-1 downto 0);
    w_addr, r_addr: in std_logic_vector(W-1 downto 0);
    r_data: out std_logic_vector(B-1 downto 0)
  );
end component;

signal r_addr : std_logic_vector(W-1 downto 0);
signal w_addr : std_logic_vector(W-1 downto 0);
signal f_status, wr_fifo: std_logic;

begin
  u_ctrl: fifo_sync_ctrl_para
    generic map(W=>W, CNT_MODE=>CNT_MODE)
    port map(clk=>clk, reset=>reset, wr=>wr, rd=>rd,
      full=>f_status, empty=>empty,
      w_addr=>w_addr, r_addr=>r_addr);
  wr_fifo <= wr and (not f_status);
  full <= f_status;
  u_reg_file: reg_file_para
    generic map(W=>W, B=>B)
    port map(clk=>clk, reset=>reset, wr_en=>wr_fifo,
      w_data=>w_data, w_addr=>w_addr,
      r_addr=> r_addr, r_data => r_data);
end arch;
```
architecture str_arch of reg_file_para is
component mux2d is
generic(
P: natural;  -- number of input ports
B: natural  -- number of bits per port
);
port(
a: in std_logic_2d(P-1 downto 0, B-1 downto 0);
.sel: in std_logic_vector(log2c(P)-1 downto 0);
y: out std_logic_vector(B-1 downto 0)
);
end component;
component tree_decoder is
generic(WIDTH: natural);
port(
a: in std_logic_vector(WIDTH-1 downto 0);
en: std_logic;
.code: out std_logic_vector(2**WIDTH-1 downto 0)
);
end component;
constant W_SIZE: natural := 2**W;
type reg_file_type is array (2**W-1 downto 0) of std_logic_vector(B-1 downto 0);
signal array_reg: reg_file_type;
signal array_next: reg_file_type;
signal array_2d: std_logic_2d(2**W-1 downto 0, B-1 downto 0);
signal en: std_logic_vector(2**W-1 downto 0);
begins
-- register array
process(clk, reset)
begin
if (reset='1') then
   array_reg <= (others=>(others=>'0'));
elsif (clk'event and clk='1') then
   array_reg <= array_next;
end if;
end process;
-- enable decoding logic for register array
u_bin_decoder: tree_decoder
   generic map(WIDTH=>W)
   port map(en=>wr_en, a=>w_addr, code=>en);
-- next-state logic of register file
process(array_reg, en, w_data)
begin
for i in (2**W-1) downto 0 loop
   if en(i)='1' then
      array_next(i) <= w_data;
   else
      array_next(i) <= array_reg(i);
   end if;
end loop;
end process;
end reg_file_para;
end process;

process (array_reg)
begin
    for r in (2**W-1) downto 0 loop
        for c in 0 to (B-1) loop
            array_2d(r,c)<=array_reg(r)(c);
            end loop;
        end loop;
    end process;

-- convert to std_logic_2d
process
begin
    for r in (2**W-1) downto 0 loop
        for c in 0 to (B-1) loop
            array_2d(r,c)<=array_reg(r)(c);
        end loop;
    end loop;
end process;

-- read port multiplexing circuit
read_mux: mux2d
    generic map(P=>2**W, B=>B)
    port map(a=>array_2d, sel=>r_addr, y=>r_data);
end str_arch;

Register file operation can be consider as accessing an array with a dynamic index (i.e., using a signal as an index), and some synthesis software may recognize this type of description. If this is the case, the behavioral VHDL code can be used for the register file, as shown in Listing 15.26.

Listing 15.26 Behavioral description of a parameterized register file

architecture beh_arch of reg_file_para is
    type reg_file_type is array (2**W-1 downto 0) of std_logic_vector(B-1 downto 0);
    signal array_reg: reg_file_type;
    signal array_next: reg_file_type;
begin
    -- register array
    process(clk, reset)
    begin
        if (reset='1') then
            array_reg <= (others=>(others=>'0'));
        elsif (clk'event and clk='l') then
            array_reg <= array_next;
        end if;
    end process;
    -- next-state logic for register array
    process(array_reg, wr_en, w_addr, w_data)
    begin
        array_next <= array_reg;
        if wr_en='1' then
            array_next(to_integer(unsigned(w_addr))) <= w_data;
        end if;
    end process;
    -- read port
    -- read port
    process
    begin
        r_data <= array_reg(to_integer(unsigned(r_addr)));
    end process;
end beh_arch;

FIFO Controller We choose the look-ahead configuration of Section 9.3.2 for the parameterized FIFO controller because LFSR counters can be used to achieve better performance. The main task is to derive parameterized code to determine the counter’s successive value.
Since the look-ahead configuration requires the next value of the counter, the predeveloped parameterized LFSR counter of Section 15.2.1 cannot be used directly. Instead, we must create a customized module for this purpose. This module is essentially the next-state logic of the parameterized LFSR of Listing 15.2.1. The VHDL code is shown in Listing 15.2.7.

Listing 15.2.7 Parameterized LFSR next-state logic

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity lfsr_next is
  generic(N: natural);
  port(q-in: in std_logic_vector(N-1 downto 0);
      q-out: out std_logic_vector(N-1 downto 0)
  );
end lfsr_next;

architecture para_arch of lfsr_next is
  constant MAX-N: natural:= 8;
type tap_array_type is
    array(2 to MAX-N) of std_logic_vector(MAX-N-1 downto 0);
  constant TAP_CONST_ARRAY: tap_array_type:=
    (2 => (1\0=>'1', others=>>'0'),
     3 => (1\0=>'1', others=>>'0'),
     4 => (1\0=>'1', others=>>'0'),
     5 => (2\0=>'1', others=>>'0'),
     6 => (1\0=>'1', others=>>'0'),
     7 => (3\0=>'1', others=>>'0'),
     8 => (4\2\0=>'1', others=>>'0'));
signal fb: std_logic;
begin
  -- next-state logic
  process(q_in)
    constant TAP_CONST: std_logic_vector(MAX-N-1 downto 0)
      := TAP_CONST_ARRAY(N);
    variable tmp: std_logic;
  begin
    tmp := '0';
    for i in 0 to (N-1) loop
      tmp := tmp xor (q_in(i) and TAP_CONST(i));
    end loop;
    fb <= not(tmp); -- exclude all 1's
  end process;
  q-out <= fb & q_in(N-1 downto 1);
end para_arch;
```

There is a minor modification over the original code. The feedback xor expression is inverted before it is appended to the MSB of the output. The purpose is to replace the all-zero state with the all-one state (i.e., the "11...11" pattern, instead of the "00...00" pattern, will be excluded from the circulation). This simplifies the system initialization.
The complete code of the parameterized FIFO controller is shown in Listing 15.28. It is similar to fixed-size code in Listing 9.16 except that two if generate statements are used to generate the desired successive value.

**Listing 15.28 Parameterized FIFO control circuit**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_sync_ctrl_para is
  generic(
    N: natural;
    CNT_MODE: natural
  );
  port(
    clk, reset: in std_logic;
    wr, rd: in std_logic;
    full, empty: out std_logic;
    w_addr, r_addr: out std_logic_vector(N-1 downto 0)
  );
end fifo_sync_ctrl_para;

architecture lookahead_arch of fifo_sync_ctrl_para is
  component lfsr_next is
    generic(N: natural);
    port(
      q_in: in std_logic_vector(N-1 downto 0);
      q_out: out std_logic_vector(N-1 downto 0)
    );
  end component;
  constant LFSR_CTR: natural:=0;
  signal w_ptr_reg, w_ptr_next, w_ptr_succ: std_logic_vector(N-1 downto 0);
  signal r_ptr_reg, r_ptr_next, r_ptr_succ: std_logic_vector(N-1 downto 0);
  signal full_reg, empty_reg, full_next, empty_next: std_logic;
  signal wr_op: std_logic_vector(1 downto 0);
begin
  -- register for read and write pointers
  process(clk, reset)
  begin
    if (reset='1') then
      w_ptr_reg <= (others=>'0');
      r_ptr_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      w_ptr_reg <= w_ptr_next;
      r_ptr_reg <= r_ptr_next;
    end if;
  end process;
  -- statute FF
  process(clk, reset)
  begin
    if (reset='1') then
```
full_reg <= '0';
empty_reg <= '1';
elsif (clk'event and clk='1') then
full_reg <= full_next;
empty_reg <= empty_next;
end if;
end process;

-- successive value for LFSR counter
g_lfsr:
if (CNT_MODE=LFSR_CTR) generate
u_lfsr_wr: lfsr_next
generic map(N=>N)
port map(q_in=>w_ptr_reg, q_out=>w_ptr_succ);
u_lfsr_rd: lfsr_next
generic map(N=>N)
port map(q_in=>r_ptr_reg, q_out=>r_ptr_succ);
end generate;

-- successive value for binary counter
g_bin:
if (CNT_MODE/=LFSR_CTR) generate
w_ptr_succ <= std_logic_vector(unsigned(w_ptr_reg) + 1);
r_ptr_succ <= std_logic_vector(unsigned(r_ptr_reg) + 1);
end generate;

-- next-state logic for read and write pointers
wr_op <= wr & rd;
process(w_ptr_reg, w_ptr_succ, r_ptr_reg, r_ptr_succ, wr_op, empty_reg, full_reg)
begin
w_ptr_next <= w_ptr_reg;
r_ptr_next <= r_ptr_reg;
full_next <= full_reg;
empty_next <= empty_reg;
case wr_op is
when "00" => -- no op
when "01" => -- read
if (empty_reg /= '1') then -- not empty
r_ptr_next <= r_ptr_succ;
full_next <= '0';
if (r_ptr succ=w_ptr_reg) then
empty_next <= '1';
end if;
end if;
when "10" => -- write
if (full_reg /= '1') then -- not full
w_ptr_next <= w_ptr_succ;
empty_next <= '0';
if (w_ptr_succ=r_ptr_reg) then
full_next <= '1';
end if;
end if;
when others => -- write/read;
w_ptr_next <= w_ptr_succ;
r_ptr_next <= r_ptr_succ;
SYNTHESIS OF PARAMETERIZED MODULES

In a parameterized module, the parameter is assigned to a fixed value when the module is instantiated. At the time of synthesis, the software is always performing the synthesis of a fixed-size circuit. From this point of view, parameterized code imposes no additional requirement on actual synthesis.

On the other hand, to facilitate the use of parameters, the expressions tend to be more general and the parameterized code normally needs more "preparation" work, including the flattening of the multidimensional array and the processing and optimization of static expressions. Recall that a static expression is an expression whose value can be calculated when the VHDL code is analyzed. It implies that the expression does not depend on any external signal and that no physical circuit should be inferred from the expression.

In a parameterized code, static expressions are commonly used to express the size of arrays and the range of for generate and for loop statements. They are also used to represent more involved indexing structures, as in the parallel-prefix reduced-xor-vector code of Listings 15.15. In a complex circuit structure, we sometimes use auxiliary static expressions to assist development of the parameterized VHDL codes. For example, the VHDL code of the binary encoder in Listing 15.11 first utilizes an auxiliary gen_or_mask function to generate the static mask and then applies the mask to the input signal (via the and operation) to disable the unneeded elements of the input signal. The function and the and operation are both static. Good synthesis software should be able to calculate the mask, propagate the constants through the and expression, and keep only the needed elements of the input signal for the final or expression.

SYNTHESIS GUIDELINES

- Portability of two-dimensional data type can be an issue since it is not defined in the RTL synthesis standard.

- User-defined genuine unconstrained two-dimensional data types are the most general type.

- User-defined array-of-arrays data types cannot have unconstrained elements and are not general enough to be used in a port declaration.

- Be aware of the difference between static and dynamic expressions. The former should not infer any physical logic during synthesis and can be of assistance in developing parameterized code.

- A one-dimensional cascading-chain structure should be avoided and replaced by more efficient two-dimensional alternatives.
15.7 BIBLIOGRAPHIC NOTES

While developing parameterized VHDL codes relies on an understanding of basic language constructs and some programming skills, developing efficient parameterized codes requires the insight and in-depth knowledge of the problem domain, as demonstrated by the parallel-prefix reduced-xor-vector circuit and carry-save multiplier. The parallel-prefix scheme is a class of algorithms that can be applied to a variety of operations. The dissertation, *Binary Adder Architectures for Cell-Based VLSI and Their Synthesis* by R. Zimmermann of Swiss Federal Institute of Technology, provides a detailed analysis on applying the algorithms to construct addition circuits. Implementing and synthesizing complex arithmetic circuits is an active research topic. The text, *Computer Arithmetic Algorithms* by I. Koren, gives a comprehensive coverage of the algorithm and construction of various arithmetic functions.

Problems

15.1 Consider the parameterized binary decoder in Section 15.3.2. Derive the VHDL code for a 1-to-2^l decoder with an enable signal and rewrite the code using a generate statement and component instantiation.

15.2 The parameterized binary decoder can also be constructed using 2-to-2^2 decoders.
   (a) Derive the VHDL code for a 2-to-2^2 decoder with an enable signal.
   (b) Derive the VHDL code of the parameterized binary decoder using only the 2-to-2^2 decoders of part (a).

15.3 Repeat part (b) of Problem 15.2. Instead of being limited to 2-to-2^2 decoders, use a 1-to-2^l decoder in the leftmost stage if the input of the parameterized decoder has an odd number of bits.

15.4 Consider the parameterized multiplexer in Section 15.3.3. Redesign the multiplexer using 4-to-1 multiplexers and derive the VHDL code accordingly.

15.5 Extend the parameterized multiplexer code in Listing 15.3.3 to accommodate two-dimensional data. We need to define a three-dimensional data type for the internal signals.
   (a) Follow the definition of std_logic_2d and define a genuine three-dimensional data type. Derive the VHDL code using this data type.
   (b) Follow the discussion of the emulated two-dimensional array and define an index function to emulate a three-dimensional array. Derive the VHDL code using this method.

15.6 Consider the parameterized binary encoder in Section 15.3.4. Instead of using for loop statements, rewrite the VHDL code with for generate statements.

15.7 We want to extend the parameterized barrel shifter in Section 15.3.5 by adding one additional mode of shift operation, arithmetic shift right. In this mode, the MSB, instead of '0', will be shifted into the left portion of the output. Modify the VHDL code to include this mode.

15.8 The VHDL code in Listing 15.15, the number of input bits of the parallel-prefix reduced-xor-vector circuit is limited a power of 2. Revise the code so that the number of input bits can be any arbitrary number.

15.9 Discuss the circuit complexity (in terms of the number of two-input xor gates) of the two reduced-xor-vector circuits discussed in Section 15.4.1.
15.10 The code of the adder-based multiplier of Listing 15.17 has a feature parameter to insert pipeline registers to the circuit. The number of stages of the pipeline is the same as the width of the input operand. Modify the code to incorporate an additional parameter that specifies the number of desired pipeline stages.

15.11 In the discussion of the multiplier circuit, the widths of the two input operands (i.e., multiplier and multiplicand) are assumed to be identical. In some application the widths can be different. Let the number of bits of multiplier and multiplicand be MR and MD respectively. Modify the sequential multiplier code of Listing 15.16 for the new requirement.

15.12 Repeat Problem 15.11, but modify the adder-based multiplier of Listing 15.17.

15.13 Repeat Problem 15.11, but modify the cell-base carry-ripple multiplier of Listing 15.19.

15.14 Repeat Problem 15.11, but modify the cell-base carry-save multiplier of Listing 15.20.

15.15 Both the adder-based multiplier of Section 15.4.2 and the carry-save multiplier of Section 15.4.2 can be configured as a pipelined circuit. Assume that the ripple adders are used in the adder-based multiplier. Let both the input width and the number of pipelined stages be N. Compare the delay and bandwidth of the two circuits.

15.16 The parameterized LFSR of Section 15.4.3 can only circulate through \(2^N - 1\) or \(2^N\) patterns. Modify the design so that the LFSR can circulate through M patterns, where M is a separate parameter and \(M < 2^N\). You can create a function that determines the Mth pattern in the LFSR sequence and load the initial value to the register when the LFSR reaches this pattern.

15.17 The register file of Section 15.4.5 has one read port. We want to revise the design so that the number of read ports can be specified by a parameter. To achieve this, the read ports need to be grouped as a single output with a two-dimensional data type. Use the std_logic_2d data type and derive the VHDL code.

15.18 The operation of a stack was discussed in Problem 9.11. Follow the design procedure in Section 15.4.5 to derive VHDL code for a parameterized stack.

15.19 The operation and design of a CAM was discussed in Section 9.3.3. Follow the design procedure in Section 15.4.5 to derive VHDL code for a parameterized CAM.
This Page Intentionally Left Blank
The single most important design principle used in this book is the synchronous methodology, in which all registers are controlled by a common clock signal. Design and analysis so far are based on an ideal clocking scenario. We assume that the entire system can be driven by a single clock signal and that the sampling edge of this clock signal can reach all registers at the same time. In reality, this is hardly possible. We need to take into consideration a non-ideal clock signal and sometimes even have to divide a large system into subsystems with independent clock signals. This chapter discusses the modeling and effect of a non-ideal clock signal, the synchronization of an asynchronous signal, and the interface between two independent clock domains.

16.1 OVERVIEW OF A CLOCK DISTRIBUTION NETWORK

16.1.1 Physical implementation of a clock distribution network

The clock distribution network is the circuit that distributes the clock signal to all FFs in the system. Since the circuit does not perform any logic function, its design and analysis are mainly at the transistor level. In Section 6.5.1, we discussed the low-level model of gates and wires for propagation delay calculation. As shown in Figure 6.15, each input port of a gate and each wire introduce small values of resistance and capacitance. The output port of a cell has to charge or discharge (i.e., “drive”) all capacitors when a signal switches state. The number of input ports driven by a cell is known as fan-out. The driving capability of
a cell depends on the electrical characteristics of the internal transistors. A typical cell can normally drive up to half a dozen cells.

While the basic transistor-level model of the clock distribution network is similar to that in Figure 6.15, the fan-out is much larger. Since all registers are connected to the same clock signal in a synchronous circuit, the fan-out of a clock signal is the number of FFs in the system. It may reach thousands or even tens of thousands in a large system. Thus, the physical implementation of a clock distribution network is very different from that of regular connection wires. Its construction is separated from the routing of regular logic and processed independently.

In addition to the clock signal, the reset signal is connected to all FFs of the system. Thus, construction of the reset network is somewhat similar to that of the clock distribution network. Because the reset signal does not impose many strict timing constraints, its implementation is simpler and less critical.

**Clock synthesis of ASIC devices** In ASIC technology, the clock distribution network is constructed by a process known as clock synthesis, which is a step in the physical design. The clock synthesis uses multiple levels of buffers to increase the driving capability and applies a special routing algorithm to balance the distribution network and minimize the difference in propagation delays. A conceptual three-level clock distribution network is shown in Figure 16.1. We assume that each buffer can drive four input ports. The buffers are used to increase the driving capability and do not perform any logic function.

An example of idealized physical routing of the previous distribution network is shown in Figure 16.2. It is done by a two-level recursive H-shaped network so that the wire length from the clock source to each FF is about the same. While the propagation delay from the clock source to an FF is unavoidable, this routing helps to ensure that the clock signal reaches each FF at about the same time.

![Conceptual clock distribution network](image-url)
Clock distribution networks of FPGA devices

In FPGA technology, a chip usually has one or more prerouted and prefabricated clock distribution networks. If we develop the VHDL code in a disciplined way, the synthesis software can recognize the existence of the clock signal and automatically map it to a prefabricated clock distribution network.

16.1.2 Clock skew and its impact on synchronous design

The construction and analysis of a clock distribution network is essentially a task at the transistor level. At the gate and RT levels, the effect of the clock distribution network is modeled by propagation delays from the clock source to various registers. Because of the variation in buffering and routing, the propagation delays may be different, as shown in the simple example in Figure 16.3. The key characteristic is the difference between the arrival times of the sampling edges, which is known as the clock skew. For multiple registers, we consider the worst-case scenario and define the clock skew as the difference between the arrival times of the earliest and latest sampling edges.

As the size of a circuit and the number of FFs increase, the clock distribution network becomes larger and more complex. Controlling the arrival time of the clock’s sampling edge to each FF becomes more difficult. This introduces larger variations over the arrival times, which, in turn, increase the clock skew. Thus, we can expect that the clock skew increases with the size of the circuit.

To accommodate the existence of clock skew, we have to modify the synchronous design methodology. The modification depends on the size and clock rate of the system. For a small circuit, propagation delays from the clock source to various FFs are small and almost identical, which implies that the rising edge of the clock signals arrives at the register at almost the same time. We can treat this as the ideal clocking scenario and ignore the clock skew.

For a moderately-sized system, the clock skew is normally a small fraction (a few percent) of the clock period. We can treat it as an ideal synchronous system and design it accordingly. However, during the analysis of setup time and hold time constraints, the
clock skew must be taken into consideration. The skew usually introduces tighter timing requirements and reduces system performance. Current technology can support a clock distribution network with up to several tens of thousands of FFs with an acceptable clock skew. Recall that the proper partition size for synthesis is between 5000 and 50,000 gates. There are no problems applying synchronous design methodology inside each partition block. Section 16.2 discusses the effect of small clock skew in timing analysis.

For a fast, large-scale system, the skew may become comparable to the clock period and can no longer be treated as a small deviation of the arrival time. Because of the lack of a common clock, the synchronous design methodology can no longer be applied. One way to deal with this problem is to divide the system into several smaller subsystems and let each subsystem be controlled by an independent clock signal. Whereas the internal operation of a subsystem is synchronous, its interface to other subsystems is asynchronous. Because of the asynchronous interface, timing violations may occur. We need to use special synchronization schemes and protocols to ensure that the control signals and data can be transferred between subsystems reliably. These schemes and protocols are discussed in Sections 16.3 to 16.9.

16.2 TIMING ANALYSIS WITH CLOCK SKEW

Clock skew is the difference between the arrival times of the sampling edges of a clock signal. It has a significant impact on the timing of sequential circuits. In general, clock skew degrades system performance and imposes a tighter hold time margin. In Section 8.6, we analyzed the timing of sequential circuits with an ideal clock and showed conditions to meet setup time and hold time constraints. The following subsections repeat the analysis with the existence of clock skew.

16.2.1 Effect on setup time and maximal clock rate

For a digital system with an ideal clock, there is no clock skew. We bundle all registers together into a single element, as shown in the basic sequential circuit block diagram of Figure 8.5. To express the different arrival times, individual registers must be considered.
Figure 16.4 Next-state logic feedback path with positive clock skew.

A conceptual diagram of two registers and a single feedback path is shown in Figure 16.4. We assume that the lengthy routing wire introduces a delay of $T_{skew}$ and thus the arrival times of the rising edges are different for the clk1 and clk2 signals. In this particular case, the arrival time of the clk2 signal is late by the amount $T_{skew}$. The late arrival is also known as a positive clock skew.

The timing diagram is shown in Figure 16.5. We follow the procedure used in Section 8.6.2 to analyze the new circuit. To satisfy the setup time constraint, the $r2_{next}$ signal must be stabilized before $t_4$. This requirement translates into the condition

$$t_3 < t_4$$

From the timing diagram, we see that

$$t_3 = t_0 + T_{cq} + T_{next(max)}$$

and

$$t_4 = t_5 - T_{setup} = (t_0 + T_c + T_{skew}) - T_{setup}$$

After we substitute $t_3$ and $t_4$, the inequality equation is simplified to

$$T_{cq} + T_{next(max)} + T_{setup} - T_{skew} < T_c$$

and the minimal clock period is

$$T_{c(min)} = T_{cq} + T_{next(max)} + T_{setup} - T_{skew}$$

Note that in this particular case, the existence of clock skew reduces the minimal clock period and actually helps the performance.

The clock skew does not always mean late arrival of the sampling edge. For example, if we switch the locations of the two registers in the previous example, the arrival time of the clk2 signal is ahead of the arrival time of the clk1 signal by the amount $T_{skew}$. The early arrival is also known as a negative clock skew. In this case, $t_4$ becomes

$$t_4 = t_5 - T_{setup} = (t_0 + T_c - T_{skew}) - T_{setup}$$
and the minimal clock period becomes

\[ T_{c_{\text{min}}} = T_{cq} + T_{\text{next}(\text{max})} + T_{\text{setup}} + T_{\text{skew}} \]

This implies that the clock period must be increased by the amount \( T_{\text{skew}} \).

If there are multiple feedback paths, the effect of the clock skew can be positive for some paths and negative for the others. Consider that we add a feedback path from the \( r2 \) register to the \( r1 \) register, as shown in Figure 16.6. For simplicity, we assume that the propagation delays of the next-state logic are identical. The minimal clock periods of the two paths are

\[ T_{cq} + T_{\text{next}(\text{max})} + T_{\text{setup}} - T_{\text{skew}} \]

and

\[ T_{cq} + T_{\text{next}(\text{max})} + T_{\text{setup}} + T_{\text{skew}} \]

respectively. Since the design has to satisfy the worst-case scenario, the clock period of the system must be at least \( T_{cq} + T_{\text{next}(\text{max})} + T_{\text{setup}} + T_{\text{skew}} \).

While the positive clock skew can be used to reduce the minimal clock period in theory, it is very difficult to do this in real design because of the existence of multiple feedback paths and constraints on the placement of registers and routing of the clock distribution network. The clock skew usually has a negative effect on the clock period and thus imposes a penalty on performance.
16.2.2 Effect on hold time constraint

The hold time analysis is similar to that in Section 8.6.3. To satisfy the hold time constraint, we must ensure that

\[ t_h < t_2 \]

From the timing diagram, we see that

\[ t_2 = t_0 + T_{cq} + T_{next(min)} \]

and

\[ t_h = t_0 + T_{hold} + T_{skew} \]

After substitution, the inequality equation can be simplified to

\[ T_{hold} < T_{cq} + T_{next(min)} - T_{skew} \]

Compared to the original inequality equation, the positive clock skew imposes a tighter margin for the hold time constraint.

In the worst-case scenario, \( T_{next(min)} \) can be close to 0 (when the output of one FF is connected to the input of another FF, as in a shift register). The inequality equation becomes

\[ T_{hold} < T_{cq} - T_{skew} \]

Recall that the device manufacture usually guarantees that \( T_{hold} < T_{cq} \), and thus the inequality equation will always be satisfied if there is no clock skew. With a positive clock skew, the margin on the inequality equation will be reduced. It may even lead to a timing violation if \( T_{skew} \) is greater than the safety margin of \( T_{cq} - T_{hold} \).

Unfortunately, there is no fix from the RT-level design. \( T_{hold}, T_{cq} \) and \( T_{skew} \) are the three parameters in the inequality equation. The first two are inherent characteristics of the device, and the third can only be obtained after synthesis of the clock distribution network. Although in theory we can add some redundant combinational logic (such as a pair of cascading invertors) to introduce artificial propagation delay, this approach is delay sensitive and is not recommended. This problem is better left for the physical design. After
construction of the clock distribution network and completion of the placement and routing, accurate clock skew information can be obtained. The physical design software will check for hold time violations. It should correct the problem by rearranging the clock routing or inserting a delay element in the violated path.

The analysis of negative clock skew is similar. The inequality equation becomes

$$T_{\text{hold}} < T_{\text{eq}} + T_{\text{next(min)}} + T_{\text{skew}}$$

The clock skew provides extra slack. Since the device manufacturer usually guarantees that $T_{\text{hold}} < T_{\text{eq}}$, the extra margin provides no additional benefit.

### 16.3 OVERVIEW OF A MULTIPLE-CLOCK SYSTEM

When we apply the synchronous design methodology, all FFs of the system are controlled by a single global clock. However, as digital systems grow more complex, it becomes very difficult or even impossible to follow the pure synchronous design principle. Multiple clocks may exist or become necessary for several reasons:

- **Inherent multiple-clock sources.** A digital system frequently needs to interact with external systems, such as peripheral devices, or to exchange information through communication links. These external systems or links may not use the same clock signal.

- **Circuit size.** As discussed in Section 16.1, the clock skew increases with the size of the circuit and the number of FFs. When a circuit is large, it is not possible to maintain a single clock. We must divide the system into smaller subsystems and use separate clock signals in the subsystems.

- **Design complexity.** A large digital system is frequently composed of several small subsystems of different performance and power criteria. Applying pure synchronous design methodology may introduce unnecessary constraints. For example, consider a system with a 16-bit 20-MHz processor, a fast 100-MHz 1-bit serial network interface and several 1-MHz peripheral I/O controllers. If the pure synchronous design methodology is used, the system must be operated at 100 MHz to accommodate the highest clock rate, even though this clock rate is only used in the serial-to-parallel conversion of the serial network interface. It is clear that this system is "overdesigned" for the processor and I/O controllers. The artificial, unnecessarily high clock rate introduces a tighter timing constraint, complicates the design and synthesis process, and increases the hardware complexity. Utilizing separate clock signals can reduce the circuit complexity and simplify the design process.

- **Power consideration.** The dynamic power of a CMOS device is proportional to the switching frequency of transistors, which is correlated to the system clock frequency. An inflated system clock rate will unnecessarily increase the system's power consumption. If we consider the previous system, synchronous design methodology requires the entire system to be operated at 100 MHz. It will consume much more power than three subsystems with clock rates of 100, 20 and 1 MHz.

As discussed in Section 8.2, the synchronous methodology is the fundamental principle in today's digital system development, and most design and analysis schemes are based on
this methodology. Thus, even with multiple clocks, we still want to apply this methodology as much as possible. The basic approach is to divide a system into multiple synchronous subsystems and design a special interface between the subsystems. This allows us to continuously apply the synchronous methodology to design a much larger system.

In a multiple-clock system, the subsystems can use either a derived clock signal or an independent clock signal. We briefly review the two schemes in the following subsections.

16.3.1 System with derived clock signals

A derived clock signal is a clock signal obtained from a known clock signal. A special clock generation circuit takes the original clock signal, generates new clock signals with different frequencies or phases, and routes them to different subsystems. Each subsystem then utilizes its own clock distribution network to distribute the clock signals to the registers within the subsystem.

In theory, we can apply general RT-level design technique to modify the frequency of a clock signal. For example, we can obtain three lower-frequency clock signals by tapping the output of 3-bit binary counter, as shown in Figure 16.7. There are two problems with this approach. First, because of the clock-to-q delay, there is a skew between the rising edges of original clock signal and the derived clock signals. Second, due to the variation of the clock-to-q delays of the FFs and the unknown wiring delays, it is difficult to determine the exact values of the skews among the three derived clock signals.

To control the skew between the clock signals, the clock generation circuit should be separated from regular logic, and manually analyzed and implemented. Special analog components, such as delay lines and buffers, or even a phase-locked loop (PLL), can be used to minimize the skew.

A system with derived clock signals is subjected to the same setup and hold time constraints. Once the clock skew between the two clock signals is known, we can apply the technique in Section 16.2 to analyze timing. The derivation procedure is similar except that we must take into consideration the difference in the clock periods.
In a multiple-clock system, we use the term clock domain to describe use of the clock signal. A clock domain is a block of circuitry in which the FFs are controlled by the same clock signal. Although a derived clock signal has a different clock frequency or phase, its relationship to the original clock signal is known. The design and analysis techniques of synchronous sequential circuit can be modified and applied in such a system. Because of this, we consider that subsystems with derived clock signals are in the same clock domain. Note that these derive clock signals need their own individual clock distribution networks even though they are in the same clock domain.

### 16.3.2 GALS system

Due to the clock skew of large circuit size or inherent I/O characteristics, it is sometimes difficult or impossible to maintain or find the relationship between the clock signals of subsystems. The clock signals in these subsystems are considered to be independent, and each subsystem constitutes its own clock domain.

Within a clock domain, the circuit operation is completely synchronous and its design follows the synchronous design methodology. Interface between the two clock domains involves two independent clocks and thus is asynchronous. This configuration is sometimes known as a globally asynchronous locally synchronous (GALS) system. After we develop a proper asynchronous interface, this scheme allows us to continuously apply the synchronous methodology to design a much larger system.

The major difficulty in designing a GALS system is the interface of clock domains; i.e., how to exchange information and transfer data between two clock domains (known as domain crossing). Since the circuit in one domain has no clock information about another domain, a signal may switch at the clock's sampling edge of another domain, which leads to a setup or hold time violation. Recall that one main advantage of the synchronous design methodology is that it provides a systematic way to avoid a timing violation. Since a timing violation in the domain crossing is not avoidable, the design must focus on what to do after a timing violation occurs.

The interface between clock domains is very different from a regular synchronous system or a system with derived clock signals. Its design cannot be automated and usually needs detailed manual analysis. It is more difficult and error-prone. Furthermore, the existence of multiple clock domains affects other processes in the development flow and complicates the static timing analysis, formal verification and test circuit synthesis. Thus, before adding an additional clock domain, we must carefully consider the trade-off between the benefits and potential complexities. In general, it is warranted only for a substantially sized subsystem or a critical high-performance subsystem. The subsequent sections discuss the nature of synchronization failure, the design of a synchronization circuit, and the design and implementation of data transfer protocols.

### 16.4 METASTABILITY AND SYNCHRONIZATION FAILURE

One fundamental timing constraint of a sequential circuit is the setup and hold times of an FF. It specifies that the input data to an FF must be stable in a decision window around the sampling edge of the clock signal. Consider the basic sequential circuit block diagram shown in Figure 8.5. The input of the register is the next-state logic’s output, which is obtained from the register’s output and an external input.
METASTABILITY AND SYNCHRONIZATION FAILURE

(a) Input stable during setup and hold time

(b) Input changing during setup or hold time

Figure 16.8 Timing diagrams of a D FF.

Since the register's output is based on the sampling edge of the clock, we have timing information about this signal. Our timing analysis examines the closed loop formed by the register and next-state logic and ensures that no timing violation will occur. Similar analysis can be performed if the external input signal is generated in the same clock domain. On the other hand, if the external input signal comes from another clock domain, as in a GALS system, the subsystem has no information about the timing relation to its clock, and thus the signal is treated as an asynchronous signal. An asynchronous input signal can change any time, including inside the decision window, and cause a timing violation. The following subsections discuss the characteristics of a timing violation.

16.4.1 Nature of metastability

When an input data signal satisfies the timing constraint, the sampled value will be propagated to the FF's output after the clock-to-q \((T_{cq})\) delay, as shown in Figure 16.8(a). On the other hand, if the input signal changes during setup or hold time, it violates the timing constraint and the output response is very different. Assume that the input changes from '0' to '1' during the setup and hold time window. One of three scenarios happens:

- The output of the FF becomes '1'.
- The output of the FF becomes '0'.

• The FF enters a metastable state, and the output exhibits an in-between voltage value.

The first scenario is the desired result and causes no problem. The second scenario implies that the FF just sampled the previous value. If the input remains unchanged, the correct value will be sampled at the next rising edge. Since we make no assumption about the arrival time of the input signal, there will be no ill effect.

The third scenario is the troublesome one. In normal operation, an FF stays in one of the two stable states, and its output voltage is either high or low. They are interpreted as logic 1 or logic 0 if the positive logic is used. When an FF enters a metastable state, its output voltage is somewhere between the low and the high, and cannot be interpreted as either logic 0 or logic 1. If the output of the FF is used to drive other logic cells, the in-between value may propagate to downstream logic cells and lead the entire digital system into an unknown state.

As its name indicates, a metastable state is not really a stable state. A small noise or disturbance will offset its "balance" and force the FF to enter one of the stable states. In other words, the FF will eventually resolve to a stable state. The time required to reach a stable state is known as the resolution time, \( T_r \). The timing diagram is shown in Figure 16.8(b). Theoretical study shows that a bistable device always has a metastable state, and this phenomenon is unavoidable. The only solution is to provide enough time to let the device resolve the situation and reach a stable state.

The resolution time, unfortunately, is not deterministic. It is characterized by a probability distribution function

\[ P(T_r) = e^{-\frac{T_r}{\tau}} \]

In this equation, \( \tau \) is the decay time constant and is determined by the electrical characteristics of the FF. A typical value of today's device technology is around a fraction of a nanosecond. The equation indicates the probability that the metastability condition persists beyond \( T_r \) after the clock edge. It can be interpreted as the probability that the metastability cannot be resolved within \( T_r \) seconds.

16.4.2 Analysis of MTBF\((T_r)\)

Since the timing violation can occur in any asynchronous input, the goal of the design is to confine the metastable condition in an FF and to prevent the in-between value being propagated to the downstream logic. When an FF cannot resolve the metastability condition within the given time, it is known as a synchronization failure.

Because of the stochastic nature of the occurrence of a timing violation and resolution time, analysis of the metastable condition is characterized by a statistical average. We use the average time interval between two synchronization failures to express the reliability of the design. It is known as mean time between synchronization failures (MTBF) and is the main quantity used in metastability timing analysis. MTBF depends on many factors. However, in a realistic design scenario, most factors cannot be easily changed, and the only freedom we have is to provide proper resolution time. Thus, MTBF is frequently expressed as a function of \( T_r \), as in MTBF\((T_r)\).

We can derive the MTBF by calculating the average rate of synchronization failures, \( AF \), which is the reciprocal of MTBF. \( AF \) is defined as the average number of synchronization failures occurring in a 1-second interval. It is determined by two factors:

- \( R_{meta} \): The average rate at which an FF enters the metastable state.
- \( P(T_r) \): The probability that an FF cannot resolve the metastable condition within \( T_r \).
$R_{\text{meta}}$ is determined by the formula

$$R_{\text{meta}} = w \times f_{\text{clk}} \times f_d$$

In this formula, $w$ is the *susceptible time window*, which is a constant determined by the electrical characteristics of the FF. It can be interpreted as a metastability susceptible time interval associated with the triggering edge of the clock signal. For current device technology, the typical value of $w$ is from few picoseconds to a fraction of a nanosecond. The $f_{\text{clk}}$ parameter is the frequency of the clock signal, which is defined as the number of clock cycles per second. During the 1-second interval, there are $f_{\text{clk}}$ triggering edges, and thus the $w \times f_{\text{clk}}$ portion of 1 second is susceptible for the metastability. The $f_d$ parameter is the rate of change in input data, which is defined as the number of input changes per second. We assume that the input data is independent of the clock and that the change can occur at any time. The probability of a single change occurring within a metastability susceptible interval is $w \times f_{\text{clk}}$. Since there are $f_d$ changes in 1 second, the FF will enter the metastability state $w \times f_{\text{clk}} \times f_d$ times per second, as shown in the equation above.

Once an FF enters the metastable state, it takes a certain amount of time to resolve to a stable state. The discussion in Section 16.4.1 shows that the probability that the FF cannot resolve the metastable condition within the given resolution time of $T_r$ is

$$P(T_r) = e^{-\frac{T_r}{\tau}}$$

In other words, when an FF enters the metastable state, it may resolve the condition within the given resolution time. Only $P(T_r)$ of the events persists over $T_r$ and leads to synchronization failure. Since the FF enters the metastability state $R_{\text{meta}}$ times per second on average and only $P(T_r)$ of the entries leads to synchronization failure for the given $T_r$, the average number of synchronization failures per second is $R_{\text{meta}} \times P(T_r)$; that is,

$$AF(T_r) = R_{\text{meta}} \times P(T_r) = w \times f_{\text{clk}} \times f_d \times e^{-\frac{T_r}{\tau}}$$

For a given $T_r$, MTBF($T_r$) becomes

$$\text{MTBF}(T_r) = \frac{1}{AF(T_r)} = \frac{e^{\frac{T_r}{\tau}}}{w \times f_{\text{clk}} \times f_d}$$

Note that the $f_{\text{clk}}, f_d, w$ and $\tau$ parameters are associated with original system specifications and device technology, and revising them can lead to significant design modification. The only freedom we have is to adjust the resolution time ($T_r$) in the synchronization circuit. That is why we normally express MTBF as a function of $T_r$, as in MTBF($T_r$).

In the MTBF calculation, $\tau$ and $w$ depend on the electrical characteristics of the device, and their values can be found in the manufacturer’s data sheet. Note that some FPGA manufacturers define the resolution time as the additional time needed after the regular clock-to-q delay ($T_{cq}$). If we call this time $T_{r2}$, the relationship between $T_r$ and $T_{r2}$ is

$$T_r = T_{cq} + T_{r2}$$

After simple mathematical manipulation, we can easily convert MTBF($T_{r2}$) to MTBF($T_r$), and vice versa.
### Table 16.1 Sample MTBF($T_r$) computation

<table>
<thead>
<tr>
<th>$T_r$</th>
<th>MTBF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0 ns</td>
<td>$4.00 \times 10^{-05}$ sec (0.04 msec)</td>
</tr>
<tr>
<td>2.5 ns</td>
<td>$5.94 \times 10^{-03}$ sec (5.94 msec)</td>
</tr>
<tr>
<td>5.0 ns</td>
<td>$8.81 \times 10^{-01}$ sec (0.88 sec)</td>
</tr>
<tr>
<td>7.5 ns</td>
<td>$1.31 \times 10^{+02}$ sec (131 sec)</td>
</tr>
<tr>
<td>10.0 ns</td>
<td>$1.94 \times 10^{+04}$ sec (5.39 hours)</td>
</tr>
<tr>
<td>12.5 ns</td>
<td>$2.88 \times 10^{+06}$ sec (3.33 days)</td>
</tr>
<tr>
<td>15.0 ns</td>
<td>$4.27 \times 10^{+08}$ sec (1.36 years)</td>
</tr>
<tr>
<td>17.5 ns</td>
<td>$6.34 \times 10^{+10}$ sec (2.01 $\times 10^{3}$ years)</td>
</tr>
<tr>
<td>20.0 ns</td>
<td>$9.42 \times 10^{+12}$ sec (2.99 $\times 10^{5}$ years)</td>
</tr>
<tr>
<td>22.5 ns</td>
<td>$1.40 \times 10^{+15}$ sec (4.43 $\times 10^{7}$ years)</td>
</tr>
<tr>
<td>25.0 ns</td>
<td>$2.07 \times 10^{+17}$ sec (6.58 $\times 10^{9}$ years)</td>
</tr>
<tr>
<td>27.5 ns</td>
<td>$3.08 \times 10^{+19}$ sec (9.76 $\times 10^{11}$ years)</td>
</tr>
<tr>
<td>30.0 ns</td>
<td>$4.57 \times 10^{+21}$ sec (1.45 $\times 10^{14}$ years)</td>
</tr>
<tr>
<td>32.5 ns</td>
<td>$6.78 \times 10^{+23}$ sec (2.15 $\times 10^{16}$ years)</td>
</tr>
<tr>
<td>35.0 ns</td>
<td>$1.01 \times 10^{+26}$ sec (3.19 $\times 10^{18}$ years)</td>
</tr>
</tbody>
</table>

#### 16.4.3 Unique characteristics of MTBF($T_r$)

We have examined various timing parameters, such as propagation delay, setup time and hold time. The metastability resolution time is very different. It is not deterministic and not even bounded, and thus must be characterized by a probability distribution function. Note that the resolution time is random in nature, and MTBF, as its name shows, is an average value. When a system has an MTBF value of 1 year, it does not mean that the synchronization failure always happens once a year. It means that the synchronization failure happens once a year on average. The actual interval can be 1 month, 6 months, 1 year, 2 years, 5 years and so on. A system may fail in a year regardless of whether its MTBF value is 1 year, 10 years or 1000 years. However, the probability of failure for the system with a 1000-year MTBF is much smaller.

Another observation about the resolution time relates to its highly non-linear characteristics. Note that $T_r$ is in the exponent position of the MTBF($T_r$) formula. A small variation over $T_r$ leads to drastic change in the value of MTBF. For example, consider an FF with a $w$ of 0.1 ns and a $\tau$ of 0.5 ns and assume that the system clock frequency ($f_{clk}$) is 50 MHz and the data rate ($f_d$) is 0.1 $f_{clk}$. The resolution time of a synchronizer is normally ranged between a fraction of a clock period to one or two clock periods (discussed in the next section). Table 16.1 lists the MTBF values of $T_r$ from 0 to 35 ns at increments of 2.5 ns.

Note that the period of a 50-MHz clock signal is 20 ns. When no resolution time is provided (i.e., $T_r = 0$), the MTBF is an unacceptable 0.04 ms. If we can use a $T_r$ value of half a clock period (i.e., 10 ns), the MTBF becomes about 5 hours. Because of the exponential rate, each extra 2.5 ns can increase the MTBF more than 100 times. When $T_r$ reaches 17.5 ns, the MTBF reaches about 2000 years. If we provide 1.5 times the clock period (i.e., 30 ns), the MTBF becomes about $10^{14}$ years (for comparison, the age the universe is on the order of $10^{11}$ years, and the appearance of the human being is on the order of $10^{5}$ years).

This phenomenon is a mixed blessing. On the positive side, while the synchronizing failure cannot be eliminated, we can make the probability of occurrence extremely small.
On the negative side, because of the sensitivity of the resolution time, a small decrease in the resolution time can significantly degrade the value of MTBF. Thus, minor revisions in the system, such as the slight increment of the system clock rate or use of an FF with a slightly larger setup time, may lead to a drastic consequence.

16.5 BASIC SYNCHRONIZER

When an asynchronous input causes a setup or hold time violation, the FF may enter the metastable state and its output exhibits an in-between value. If not blocked, the in-between value will be passed to the next stage and gradually propagated through the entire system.

As its name shows, a synchronization circuit (or a synchronizer) is to synchronize an asynchronous input with the system clock. As we learned from the previous sections, no circuit can prevent the occurrence of the metastability of a bistable device. The purpose of a synchronization circuit is to stop the propagation of the in-between value and confine the metastability condition within the synchronizer. Since the metastability condition will eventually resolve itself, the task of a synchronizer is just to provide enough time for the FF to reach a stable state.

The following subsections analyze various configurations of a synchronizer and their MTBFs. In our examples, we assume that the circuit utilizes the FF of Section 16.4.3, and has same clock frequency and data rate; i.e., \( w = 0.1 \text{ ns}, \tau = 0.5 \text{ ns}, f_{clk} = 50 \text{ MHz} \) and \( f_d = 0.1 f_{clk} \).

### 16.5.1 The danger of no synchronizer

We first consider a sequential circuit that has no synchronizer for its asynchronous input, as shown in Figure 16.9(a). If the asynchronous signal causes a timing violation, the system register may enter the metastable state, and the in-between value will be propagated to the next-state logic circuit. We can analyze how frequently the system enters the metastable state using the previous MTBF formula. Since there is no synchronizer, no resolution time is provided (i.e., \( T_r = 0 \)). Substituting this value into the formula, we have \( \text{MTBF}(0) = 0.04 \text{ ms} \). This failure rate is clearly unacceptable.

### 16.5.2 One-FF synchronizer and its deficiency

The first design of a synchronizer is to use a single D FF, as shown in Figure 16.9(b). Let \( T_c, T_{setup} \) and \( T_{comb} \) be the clock period of the system, the setup time of the FF and the propagation delay of the combinational circuit respectively. Consider the path from the synchronizer D FF to the system D FF. The synchronizer provides one clock period for the out-sync signal to resolve, propagate through the combinational logic and satisfy the setup time constraint of the system D FF. The required time for the latter two is \( T_{comb} + T_{setup} \), and the remaining balance can be used to resolve the metastability condition, which is

\[
T_r = T_c - (T_{comb} + T_{setup})
\]

Assume that \( T_{setup} \) of the system register is 2.5 ns. The resolution time of this circuit becomes

\[
T_r = 20 - (T_{comb} + 2.5) = 17.5 - T_{comb}
\]

\( T_r \) and MTBF depend on \( T_{comb} \), the propagation delay of the combinational circuit. For a simple combination circuit, the \( T_r \) will be relatively large. For example, if \( T_{comb} \) is 1 ns,
Figure 16.9  Synchronizers.
$T_r$ becomes 16.5 ns and MTBF(16.5 ns) is about 272 years. On the other hand, a complex combination circuit can drastically reduce the MTBF value. If $T_{comb}$ is 12.5 ns, $T_r$ becomes 5 ns and MTBF(5 ns) is dropped to about 0.88 second.

As discussed earlier, MTBF is extremely sensitive to $T_r$, and a small variation leads to a huge swing in MTBF value. The value of $T_{comb}$ depends on the logic function of the combinational circuit, device technology as well as synthesis and placement and routing process, and thus cannot be determined in advance. A minor modification in the combinational logic, the synthesis process or the placement and routing process can lead to a significant reduction in MTBF and make the system susceptible to synchronization failure. Therefore, this is not a reliable design. A better alternative is to use two D FFs for the synchronizer.

### 16.5.3 Two-FF synchronizer

The previous analysis shows that a maximal resolution time can be obtained if $T_{comb}$ is 0. Since the function of the combinational logic is defined by the original system, we cannot modify it arbitrarily. Instead, we can insert another D FF to form a two-FF synchronizer, as shown in Figure 16.9(c). The resolution time provided by the two FFs inside the synchronizer is

$$ T_r = T_c - T_{setup} $$

If $T_{setup}$ is 2.5 ns, the resolution time becomes

$$ T_r = 20 - 2.5 = 17.5 \text{ ns} $$

The MTBF(17.5 ns) is about 3000 years. In addition to providing more resolution time, this design is also more robust since no logic function or synthesis is involved. The only uncertain factor in this design is the wiring delay, which can be substantial if the two D FFs are located far apart. To minimize this delay, the two D FFs must be placed as close as possible. In physical design, we may need to manually perform the placement and routing for the synchronizer.

The VHDL code for the synchronizer is straightforward, following the block diagram of Figure 16.9(c). The code is shown in Listing 16.1.

---

**Listing 16.1** Two-FF synchronizer

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity synchronizer is
  port(
    clk, reset: in std_logic;
    in_async: in std_logic;
    out_async: out std_logic
  );
end synchronizer;

architecture two_ff_arch of synchronizer is
  signal meta_reg, sync_reg: std_logic;
  signal meta_next, sync_next: std_logic;
begin
  -- two D FFs
  process(clk, reset)
  ```
begin
if (reset='1') then
  meta_reg <= '0';
  sync_reg <= '0';
elsif (clk'event and clk='1') then
  meta_reg <= meta_next;
  sync_reg <= sync_next;
end if;
end process;

-- next-state logic
meta_next <= in_async;
sync_next <= meta_reg;

-- output
out_sync <= sync_reg;

end two_ff_arch;

Because of its simplicity and robustness, the two-FF configuration is the most widely used synchronizer. It is satisfactory in most applications. However, the regular D FF occasionally may not be able to provide sufficient $T_r$. For example, if we increase the system clock by one-third to 66.7 MHz, the clock period is reduced to 15 ns and $T_r$ becomes 12.5 ns. The MTBF is reduced to 3.33 days. To overcome this problem, many ASIC technologies have a special metastability-hardened D FF cell in their libraries. The functionality of this D FF is identical to that of a regular D FF, but its $w$, $\tau$ and $T_{setup}$ are made smaller to increase MTBF. We can use component instantiation in VHDL code to instantiate this type of D FF cell in a synchronizer. Due to its internal implementation, the circuit size of a metastability-hardened D FF cell is several times larger than that of a regular D FF cell and thus should not be used in regular sequential circuits.

16.5.4 Three-FF synchronizer

If the device technology does not provide a metastability-hardened D FF cell, we can increase the resolution time by cascading more D FF cells or artificially enlarging the clock period of the synchronizer. The three-FF synchronizer is shown in Figure 16.9(d). An extra D FF is cascaded with a two-FF synchronizer. The idea behind this design is to use the extra D FF to provide an additional opportunity to resolve the metastability condition.

We can follow the procedure in Section 16.4.2 to calculate the MTBF of this circuit. Recall that $R_{meta}$, the average rate at which the first D FF enters a metastable state, is

$$R_{meta} = w \cdot f_{clk} \cdot f_d$$

Once the FF enters the metastable state, it has a time interval of $T_c - T_{setup}$ to resolve the situation. The probability that the metastability condition persists beyond the current clock cycle is

$$P_1 = e^{-\frac{T_c - T_{setup}}{\tau}}$$

If this situation happens, the metastability condition is sampled and passed to the second FF. The second FF, again, has a time interval of $T_c - T_{setup}$ to resolve the situation, and the probability that the metastability condition persists beyond this clock cycle is

$$P_2 = e^{-\frac{T_c - T_{setup}}{\tau}}$$
The MTBF of this circuit becomes

\[
MTBF = \frac{1}{R_{\text{meta}} \cdot P1 \cdot P2} = e^{\frac{2(T_c - T_{\text{setup}})}{w \cdot f_{\text{clk}} \cdot f_d}}
\]

If we compare this equation to the two-FF synchronizer, which is

\[
MTBF = \frac{1}{P1} = e^{\frac{(T_c - T_{\text{setup}})}{w \cdot f_{\text{clk}} \cdot f_d}}
\]

We can interpret that the three-FF synchronizer increases the resolution time from \(T_c - T_{\text{setup}}\) to \(2(T_c - T_{\text{setup}})\).

Since this term is in the exponent of the equation, its impact is very significant. If \(T_c\) is 20 ns and \(T_{\text{setup}}\) is 2.5 ns, the resolution time increases from 17.5 ns to 35 ns, and the MTBF increases from 2000 thousand years to \(10^{18}\) years. If \(T_c\) is 15 ns, the resolution time increases from 12.5 ns to 25 ns and the MTBF increases from 3 days to about 6 billion years, which is a pretty safe number.

The disadvantage of the three-buffer synchronizer is the delay for the input signal. The extra D FF increases the delay from two clock cycles to three clock cycles. When possible, we should use a metastability-hardened D FF cell rather than using an additional D FF.

We can cascade more D FFs to increase the MTBF. However, because of the effect of the exponential decay, this is seldom needed in reality.

16.5.5 Proper use of a synchronizer

The function of a synchronizer is to provide a non-metastable output value. We must use it properly to obtain a reliable synchronized result. Good design practices can help us to achieve this goal and avoid subtle errors:

- Use a glitch-free signal for synchronization.
- Synchronize a signal in a single place.
- Avoid synchronizing multiple "related" signals.
- Reanalyze the synchronizer after each design change.

These practices are discussed in the following paragraphs.

Use a glitch-free signal for synchronization The asynchronous input signal normally comes from another clock domain. Since the synchronizer has no knowledge about the clock signal in another domain, it can sample the asynchronous input any time. If a glitch exists in the input signal, it may be sampled and synchronized incorrectly as a legitimate value. It is important to pass a glitch-free signal for synchronization. This can be achieved by adding an output buffer when the signal is generated.

Synchronize a signal in a single place The function of a synchronizer is to generate a stable output value. The synchronizer, however, cannot guarantee which value will be reached. For example, if a timing violation occurs when the input changes from '0' to '1', the synchronized input value can be '0' or '1' at the current sampling clock edge. Assume that the input signal does not change. It will be sampled again at the next rising edge of the clock and a stable '1' will be obtained. This implies that the arrival time of a synchronized asynchronous input signal may exhibit a random one-clock delay. We must take the random delay into consideration when using a synchronizer.
An asynchronous input signal may be used in multiple places in a clock domain. It should be synchronized in a single entry point. An example of a poor design is shown in Figure 16.10(a), in which the `in_async` signal and its derivative are synchronized by two individual synchronizers. The potentially random one-clock delay may introduce inconsistent values to the system and lead to incorrect operation. A better alternative is shown in Figure 16.10(b). The signal is synchronized by a single synchronizer, and the system is always fed with the same value.

**Avoid synchronizing related signals** A similar issue is to synchronize related signals. Related signals means that a group of signals are combined to represent a command, state and so on. For example, we may use two signals to represent four possible actions. Because of the random one-clock delay, synchronizing related signals may lead to uncertain results. For example, consider that two related signals changes from "00" to "11". If the two signals switch at about the same time and both transitions cause timing violations, the resolved results can be "00", "01", "10" or "11" for one clock cycle. Although the signal will eventually be settled to "11" in the next clock cycle, the "01" and "10" conditions may exist for one clock cycle. This may cause a serious problem for some applications.

There are two ways to correct the problem. The first is to apply special coding patterns, such as Gray code, to ensure that only one bit changes during the transition. One example
is given in Section 16.9.1. A better, more systematic alternative is to bundle all signals and pass them as a single data item. The data transfer between two clock domains is discussed in Section 16.8.

**Reanalyze the synchronizer after each design change** MTBF is extremely sensitive to the available resolution time, and a small variation can lead to drastic change. For example, consider the two-FF synchronizer discussed in Section 16.5.3. If the original system is running at 50 MHz, the MTBF is about 3000 years. Assume that we upgrade the design using faster functional units and the new system can run at 66.7 MHz, about 33% faster. Since the same device technology is used for the D FFs, \( w \) and \( \tau \) remain unchanged. The MTBF is reduced to a mere 3 days, which is only 0.0002% of the original value. The example demonstrates the sensitivity of the synchronizing circuit. It is good practice to examine the synchronizer after each design modification.

### 16.6 SINGLE ENABLE SIGNAL CROSSING CLOCK DOMAINS

In a GALS system, clock domains are driven by independent clock signals. The clock frequencies and data processing rates of these domains may not be identical. A subsystem can communicate with another subsystem whose clock frequency is 10 times faster or 10 times slower. The function of a synchronizer is to prevent the subsystems from entering the metastable state. Additional control schemes are needed to coordinate the information exchange between the two clock domains. We show how to propagate an enable pulse signal from one clock domain to another clock domain in this section and Section 16.7 and discuss the data transfer in Sections 16.8 and 16.9.

#### 16.6.1 Edge detection scheme

A digital system frequently includes a control signal in the form of an enable pulse, which activates the desired action for a single time. The enable signal of a counter and the start signal of a sequential multiplier are signals of this type. An enable pulse should be sampled by exactly one clock edge. A longer duration may cause errors. For example, a counter may count twice for a single event or a multiplier may load the incorrect operands.

While using an enable pulse between two synchronous subsystems is straightforward, it is much harder to pass the pulse crossing the clock domains. We must consider the synchronization and the difference in clock rates. The following subsections discuss several ad hoc edge detection schemes to regenerate an enable pulse from a slow or a fast clock domain. A more robust scheme that involves feedback signal is discussed in the next section.

**Wide enable signal** If an enable pulse is generated from a slow clock domain, its duration may last for several clock cycles in the current clock domain, and the signal appears as a very wide pulse. A rising-edge detection circuit is needed to regenerate a shorter, synchronized enable pulse in the current clock domain. The block diagram is shown in Figure 16.11(a), which includes a synchronizer and an edge detection circuit.

The rising-edge detection circuit can be designed by using an FSM or direct implementation, as discussed in Section 10.4.1. We use the implementation shown in Figure 10.19 of Section 10.8.1, and its VHDL code is repeated in Listing 16.2.
Figure 16.11  Regeneration of a wide enable signal.

Listing 16.2  Rising-edge detection circuit

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity rising_edge_detector is
  port(
    clk, reset: in std_logic;
    strobe: in std_logic;
    pulse: out std_logic
  );
end rising_edge_detector;
architecture direct_arch of rising_edge_detector is
  signal delay_reg: std_logic;
begin
  -- delay register
  process(clk,reset)
  begin
    if (reset='1') then
      delay_reg <= '0';
  end process;
end direct_arch;
```

(a) Block diagram

(b) Correct circuit diagram

(c) Incorrect implementation
After substituting the gate-level implementation in the blocks, we can obtain a more detailed circuit diagram, as shown in Figure 16.11(b).

The VHDL code for the complete enable pulse regeneration circuit is shown in Listing 16.3. We intentionally use the component instantiation and create two component instances in the top-level description to highlight the use of a synchronizer and to differentiate it from a regular sequential circuit. After each design change, the synchronizer instance must be reexamined and, if needed, replaced, to ensure the proper MTBF.

**Listing 16.3** Enable pulse regenerator for a wide enable signal

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity sync_en_pulse is
  port(
    clk, reset: in std_logic;
    en_in: in std_logic;
    en_out: out std_logic
  );
end sync_en_pulse;

architecture slow_en_arch of sync_en_pulse is
  component synchronizer
    port(
      clk, reset: in std_logic;
      in_async: in std_logic;
      out_sync: out std_logic
    );
  end component;
  component rising_edge_detector
    port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      pulse: out std_logic
    );
  end component;
  signal en_strobe: std_logic;
begn
  sync: synchronizer
  port map (clk=>clk, reset=>reset, in_async=>en_in,
            out_sync=>en_strobe);
  edge_detect: rising_edge_detector
  port map (clk=>clk, reset=>reset, strobe=>en_strobe,
            pulse=>en_out);
end slow_en_arch;
```
We may be tempted to use the second D FF of the synchronizer to function as the edge detection circuit to save a D FF and to reduce the propagation delay, as shown in Figure 16.11(c). This is a poor design since the unresolved signal may leak through the and cell and propagate to the downstream logic.

**Narrow enable signal** Handling an enable pulse from a fast clock domain is more difficult. For example, if the pulse is generated from a domain whose clock frequency is eight times faster than the frequency of the current clock domain, the duration of the enable pulse is only one-eighth of the period of the current clock signal. The sampling edge of the D FF of the synchronizer is likely to miss the narrow pulse.

Since the signal cannot be sampled by the clock edge, no synchronous design method can solve this problem. We must turn to ad hoc techniques to "stretch" the pulse until it is sampled by the current clock. One possible design is shown in Figure 16.12. In this design, the enable pulse is used as the clock for the stretcher D FF. When a pulse arrives, the stretcher D FF is loaded with ‘1’. The output of the D FF is then passed to the synchronizer. After the pulse is synchronized, the asserted synchronizer output clears the first D FF via the asynchronous reset. Due to the random one-clock delay of the synchronizer, the duration of the synchronized output can be one or two clock periods, and thus an edge detection circuit is needed to ensure correct operation. Because the first D FF is driven by a different clock signal, it should be excluded for the regular timing analysis and testing circuit. The VHDL code of the revised architecture body is shown in Listing 16.4.

![Figure 16.12 Regeneration of a narrow enable signal.](image)

Listing 16.4 Enable pulse regenerator for a narrow enable signal

```vhdl
architecture fast_en_arch of sync_en_pulse is
  component synchronizer
    port(
      clk, reset: in std_logic;
      in_async: in std_logic;
      out_sync: out std_logic
    );
  end component;
  component rising_edge_detector
    port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      pulse: out std_logic
    );
  end component;
```
Since the function of the first D FF depends only on the rising edge, not on the duration, of the incoming pulse, this scheme can be applied to a wide enable pulse as well. Note that the incoming enable pulse must be glitch-free to prevent false triggering.

### 16.6.2 Level-alternation scheme

An alternative to the ad hoc pulse-stretching circuit is to slightly modify the interface between the two clock domains and use the alternation of the output level to carry the information. In this scheme, the sending subsystem toggles the output value when an enable pulse is generated and thus embeds the pulse information into the signal transition edges. The block diagram is shown in Figure 16.13(a). The circuit is a T FF, which toggles its output after each time the en signal is asserted. When an enable pulse arrives, the en_level signal switches state, as shown in the top and middle parts of the timing diagram in Figure 16.13(c). The corresponding VHDL segment is

```vhdl
-- D FF
process (clk, reset)
begin
    if (reset='1') then
        t_next <= '0';
    elsif (clk'event and clk='1') then
        t_reg <= t_next;
    end if;
end process;
-- next-state logic
  t_next <= not (t_reg) when en='1' else
  t_reg;
-- output logic
en_level <= t_reg
```
In the domain that receives the enable pulse, it needs a synchronizer and a dual-edge detection circuit that can detect both the rising and falling edges of an input signal. The edge detection circuit senses the change in signal level and converts it back to a single one-clock-period pulse. We can derive the dual-edge detection circuit by using an FSM or direct implementation. One possible direct implementation is to perform an xor operation over the current input value and the previous input value stored in a D FF, as shown in Figure 16.13(b). The output waveform of the regenerator is demonstrated in the bottom part of the timing diagram in Figure 16.13(c). For clarity, the synchronizer delay is not included in the diagram.

The VHDL codes for the dual-edge detection circuit and the architecture body of the revised enable pulse regeneration circuit are shown in Listings 16.5 and 16.6 respectively. Note that the new dual_edge_detector component is used in the architecture body.

**Listing 16.5** Dual-edge detection circuit

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity dual_edge_detector is
  port(
```

Listing 16.6 Enable pulse regenerator using the level-alternation scheme

architecture level_arch of sync_en_pulse is
component synchronizer
port(
    clk, reset: in std_logic;
    in_async: in std_logic;
    out_sync: out std_logic
);
end component;
component dual_edge_detector
port(
    clk, reset: in std_logic;
    strobe: in std_logic;
    pulse: out std_logic
);
end component;
signal en_strobe: std_logic;
begin
    sync: synchronizer
    port map (clk=>clk, reset=>reset, in_async=>en_in, out_sync=>en_strobe);
    edge_detect: dual_edge_detector
    port map (clk=>clk, reset=>reset, strobe=>en_strobe, pulse=>en_out);
end level_arch;
16.7 HANDSHAKING PROTOCOL

While the pulse regeneration schemes of Section 16.6 can handle an enable signal with different widths, they cannot control the rate at which the enable pulses are generated. For example, consider a sending subsystem with a clock frequency that is eight times faster than that of a receiving subsystem. The previous schemes can regenerate the enable pulse in the receiving subsystem even when the input's width is only one-eighth that of the clock period. However, if the enable pulse is generated every four clock cycles, the rate is too fast for the receiving subsystem to process, and some pulses will be lost when crossing the domains. In order to function properly, the sending subsystem needs some knowledge of the receiving subsystem and issues the enable pulse accordingly.

To develop a more robust scheme, we must utilize a feedback signal from the receiving subsystem to communicate its status and establish a rule, which is known as a protocol, between the two subsystems. The following subsections discuss a four-phase and a two-phase handshaking protocols. While these protocols can be used to regulate the rate of the arriving enable pulses, their major applications are associated with the data transfer between two clock domains. This subject is discussed in the next section.

16.7.1 Four-phase handshaking protocol

The most commonly used scheme to coordinate operations between two clock domains is the four-phase handshaking protocol. This protocol makes no assumptions about the relative data processing rates between the clock domains and thus can accommodate a wide range of applications. In this protocol, the two subsystems are designated as the talker and the listener respectively. The talker and the listener exchange information via the req signal, which is the request signal from the talker to the listener, and the ack signal, which is the acknowledge signal from the listener to the talker. The simplified block diagram is shown in Figure 16.14(a).

The basic operation sequence (i.e., the handshaking procedure) of the four-phase handshaking protocol is illustrated in Figure 16.14(b). It consists of the following steps:

1. The talker activates the req signal.
2. When the listener detects activation of the req signal, it activates the ack signal to inform the talker.
3. When the talker senses activation of the ack signal, it deactivates the req signal.
4. After the listener detects deactivation of the req signal, it deactivates the ack signal accordingly.
5. Once the talker senses deactivation of the ack signal, it returns to the initial state.

The talker can issue a new request if needed.

In this protocol, the listener provides feedback information via the ack signal to let the talker know that a change is detected in the req signal, and the talker can respond accordingly. Note that there is no assumption about the operation speed of the listener and the talker. The talker must keep the req signal asserted until the ack signal is activated. The talker does not need to make any assumptions about the operation speed or the clock rate and can send a signal to a subsystem with unknown characteristics.

Note that we can combine the talker and the listener and treat them as a single system. The values of the req and ack signals define the "system state." When the req and ack signals are "00", the system is in the idle or initial state. During the handshaking process, they change to "10", "11" and "01" and eventually return to "00", the original state. We call the protocol *four-phase handshaking* because the sequence progresses through four distinctive states.

Since the req and ack signals cross the clock domains, two synchronizers are needed in the actual implementation. The more detailed block diagram of the handshaking scheme is shown in Figure 16.15. In the actual implementation, we use the .in, .out and .sync suffixes to indicate that the corresponding signal is an asynchronous input signal, output signal and synchronized input signal respectively.

The protocol can be implemented by two separate FSMs, one for the talker and one for the listener. Their ASM charts are shown in Figure 16.16. We assume that the talker FSM also has an input command, start, and an output status, ready. The FSM initializes the handshaking operation when the start signal is activated and asserts the ready signal when it is in the idle state. When the sending subsystem wants to issue an enable pulse across the clock domain, it checks the ready signal to ensure that the talker FSM is idle and then activates the start signal for one clock cycle. After the talker FSM senses the
Figure 16.16  ASM charts of the talker and listener of the four-phase handshaking protocol.
HANDSHAKING PROTOCOL

The start signal, it moves to the s_req1 state, in which the req_out signal is activated. The FSM then stays in the s_req1 state until activation of the acknowledge signal, ack_sync. It then moves to the s_req0 state and deactivates the req_out signal. The FSM returns to the idle state after it senses deactivation of the ack_sync signal.

The listener FSM is similar to the talker FSM except that it contains no start signal and thus can only respond to the talker FSM.

Because the ack_out and req_out signals are to be synchronized by a different clock domain, they must be glitch-free. This can be achieved by adding proper output buffers. Since they are designed as Moore outputs in the FSMs, we use the look-ahead output buffer scheme discussed in Section 10.7.2. The VHDL codes of the two FSMs are shown in Listings 16.7 and 16.8 respectively.

Listing 16.7 Talker FSM of the four-phase handshaking protocol

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity talker_fsm is
port(
    clk, reset: in std_logic;
    start, ack_sync: in std_logic;
    ready: out std_logic;
    req_out: out std_logic
);
end talker_fsm;

architecture arch of talker_fsm is
    type t_state_type is (idle, s_req1, s_req0);
    signal state_reg, state_next: t_state_type;
    signal req_buf_reg, req_buf_next: std_logic;
begin
    -- state register and output buffer
    process(clk,reset)
    begin
        if (reset='1') then
            state_reg <= idle;
            req_buf_reg <= '0';
        elsif (clk'event and clk='1') then
            state_reg <= state_next;
            req_buf_reg <= req_buf_next;
        end if;
    end process;
    -- next-state logic
    process(state_reg,start,ack_sync)
    begin
        ready <= '0';
        state_next <= state_reg;
        case state_reg is
        when idle =>
            if start='1' then
                state_next <= s_req1;
            end if;
            ready <= '1';
        when s_req1 =>
            state_next <= s_req0;
            ready <= '0';
        when s_req0 =>
            state_next <= idle;
            ready <= '0';
        end case;
    end process;
end arch;
```
if ack_sync='1' then
    state_next <= s_req0;
end if;
when s_req0 =>
if ack_sync='0' then
    state_next <= idle;
end if;
end case;
end process;

when s_req0 =>
if ack_sync='0' then
    state_next <= idle;
end if;
end case;
end process;

state-next <= s-req0;
state-next <= idle;
end case;
end process;

process(state-next)
begin
    case state-next is
        when idle =>
            when s-reql =>
                req-buf-next <= '0';
            when s-req0 =>
                req-buf-next <= '1';
        end case;
        when s-req0 =>
            req-buf-next <= '0';
        end case;
    end case;
end process;

req-out <= req-buf-reg;
end arch;

Listing 16.8 Listener FSM of the four-phase handshaking protocol

library ieee;
use ieee.std_logic_1164.all;
entity listener_fsm is
    port(
        clk, reset: in std_logic;
        req-sync: in std_logic;
        ack-out: out std_logic
    );
end listener_fsm;

architecture arch of listener_fsm is
    type l_state_type is (s_ack0, s_ack1);
    signal state_reg, state_next: l_state_type;
    signal ack_buf_reg, ack_buf_next: std_logic;
begin
    -- state register and output buffer
    process(clk,reset)
    begin
        if (reset='1') then
            state_reg <= s_ack0;
            ack_buf_reg <= '0';
        elsif (clk'event and clk='1') then
            state_reg <= state_next;
            ack_buf_reg <= ack_buf_next;
        end if;
    end process;
    -- next-state logic
To complete the design, synchronizers are needed for the input signals. Again, to emphasize the unique characteristics of the synchronization circuits, we separate them from the regular sequential circuits and instantiate the synchronizers in the top level. The VHDL codes of the complete design follow the block diagram in Figure 16.15 and are shown in Listings 16.9 and 16.10 respectively.

Listing 16.9 Talker of the four-phase handshaking protocol

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity talker_str is
  port(
    clkt: in std_logic;
    resett: in std_logic;
    ack_in: in std_logic;
    start: in std_logic;
    ready: out std_logic;
    req_out: out std_logic
  );
end talker_str;

architecture str_arch of talker_str is
  signal ack_sync: std_logic;
  component synchronizer
    port(
      clk: in std_logic;
      in_async: in std_logic;
    )
  end component;
begin
  -- look-ahead output logic
  process(state_next)
  begin
    case state_next is
      when s_ack0 =>
        ack_buf_next <= '0';
      when s_ack1 =>
        ack_buf_next <= '1';
    end case;
  end process;
  ack_out <= ack_buf_reg;
end arch;
```
CLOCK AND SYNCHRONIZATION: PRINCIPLE AND PRACTICE

Listing 16.10 Listener of the four-phase handshaking protocol

library ieee;
use ieee.std_logic_1164.all;

entity listener_str is
  port(
    clk1: in std_logic;
    reset1: in std_logic;
    req_in: in std_logic;
    ack_out: out std_logic
  );
end listener_str;

architecture str_arch of listener_str is
  signal req_sync: std_logic;
  component listener_fsm
    port ( 
      clk: in std_logic;
      req_sync: in std_logic;
      reset: in std_logic;
      ack_out: out std_logic
    );
  end component;
  component synchronizer
    port ( 
      clk: in std_logic;
      in_async: in std_logic;
      reset: in std_logic;
      out_sync: out std_logic
    );
  end component;
begin
  sync_unit: synchronizer
    port map (clk=>clkt, reset=>resett, in_async=>ack_in, out_sync=>ack_sync);
  fsm_unit: talker_fsm
    port map (clk=>clkt, reset=>resett, start=>start,
              ack_sync=>ack_sync, ready=>ready,
              req_out=>req_out);
end str_arch;
We can use this protocol to pass an enable pulse across the clock domain by connecting the en signal to the start signal of the talker FSM. When an enable pulse arrives, the talker initiates the handshaking operation. When the listener detects the activation edge of the req_in signal, it can also generate an output pulse, which corresponds to the regenerated enable pulse in the new clock domain. Since the sending subsystem cannot generate another enable pulse until the handshaking operation is completed, the sending subsystem will not overrun the receiving subsystem.

At first glance, the four phases may appear to be somewhat redundant. We may be tempted to discard the second half of the handshaking to simplify the FSMs and let the talker and listener return to the initial state automatically. Let us consider what happens if this is done. Assume that the talker and the listener deactivate the req and ack signals automatically after the system reaches the "11" phase. There will be no problem if the deactivations are done simultaneously, as shown in the timing diagram of Figure 16.17(a). However, since the two subsystems are driven by different clocks, this is hardly possible. If the talker is much slower, the listener may be fooled into thinking that the asserted req signal is the initiation of a new request, as shown in the timing diagram of Figure 16.17(b). At time $t_2$, the listener deactivates the ack signal. It then senses the activation of the req signal and mistakenly treats the condition as a new round of handshaking and responds accordingly. Thus, the same incoming request pulse will be incorrectly regenerated again.

On the other hand, if the listener is too slow, the talker may start to send a new request when the ack signal is still asserted, as shown in the timing diagram of Figure 16.17(c). At time $t_3$, the talker mistakenly thinks that the handshaking is completed and starts a new round shortly after. Since the listener still processes the first request, the new request will be lost. These examples show that all steps are needed in the original four-phase handshaking protocol.

### 16.7.2 Two-phase handshaking protocol

In the four-phase handshaking protocol, the talker and the listener exchange information on two separate occasions. One is during the first half of the handshaking, activation and acknowledgment of the req signal, and the other is during the second half, deactivation and acknowledgment of the req signal. Some applications, such as sending an enable pulse across the domain, require only a single exchange of information. In these applications, the req signal (e.g., the enable signal) has already been successfully detected and regenerated in the first half. The purpose of the second half is to ensure that the system can return safely to the initial state.

We can make the handshaking scheme more efficient by including only a single information exchange in the protocol. In this scheme, we do not require the system to return to the original state and define that the system is idle when the req and ack signals are
both '0' or both '1'. The system will alternate between the two representations of the idle state.

The operation sequence of the new protocol includes the following steps:
1. The talker activates the req signal.
2. When the listener detects activation of the req signal, it activates the ack signal to inform the talker.
3. After the talker senses activation of the ack signal, it knows that the handshaking is completed and the system reaches the idle state.

Note that the values of the req and ack signals are "11". When a new round of handshaking is initiated, the system starts from the "11" state and the steps are:
1. The talker deactivates the req signal.
2. When the listener detects deactivation of the req signal, it deactivates the ack signal to inform the talker.
3. After the talker senses deactivation of the ack signal, it knows that the handshaking is completed and the system reaches the idle state.

Note that the values of the req and ack signals are "00" now, and thus the system returns to its initial state. The timing diagram is shown in Figure 16.18. Although the appearance of the four-phase and two-phase timing diagrams are similar, interpretation of the req and ack signals (i.e., system state) is very different.
We can follow the previous procedure to derive the talker and listener FSMs for the two-phase handshaking protocol. The revised talker and listener ASM charts are shown in Figure 16.19. Note that the talker FSM stays in the s.req1 and s.req0 states until a new round of handshaking is initiated (i.e., when the start signal is '1'). Closer observation shows that the idle and s.req0 states of the talker FSM are equivalent, and we can merge the two states and remove the idle state.

As in the four-phase handshaking system, two synchronizers are needed for the acknowledge and request signals in the final implementation.

16.8 DATA TRANSFER CROSSING CLOCK DOMAINS

Data transfer between synchronous subsystems is just passing data from one register to another register, and the operation takes one clock cycle. Data transfer between two clock domains is more complicated. As in passing a single enable signal, it involves two issues, which are synchronization of the data signals and regulation of the data transfer rate.

In most applications, the interface between clock domains includes command signals, data lines and address lines. As we discussed in Section 16.5.5, synchronizing related signals is difficult and error-prone. A better alternative is to bundle all signals and use an enable signal to coordinate the access of the bundled signals. Basically, the sending subsystem activates the bundled signals, waits until they are stabilized, and then activates an enable signal to inform the receiving subsystem to access the bundled signals. Since the bundled signals are stabilized when accessed, no timing violation will occur. Only the enable signal is subjected to the metastability condition and needs to be synchronized. Instead of worrying about the synchronization of all signals, we need only focus on the enable signal.

Since clock frequencies and data processing rates are likely to be different in two clock domains, resolving the synchronization problem alone cannot guarantee reliable data transfer. We also need a mechanism to control the rate of data transfer to ensure that no information is lost or duplicated during the transaction. We can incorporate the data transfer into the earlier handshaking protocols and divide the transfer into three categories:

- Four-phase handshaking transfer
- Two-phase handshaking transfer
- One-phase transfer

The four-phase handshaking transfer has the highest overhead but is most robust. It assumes that the two subsystems have minimal information about each other. One-phase transfer uses a single enable signal with no feedback. It involves minimal overhead, but its operation is based on the assumption that the two subsystems have prior knowledge of the other's timing characteristics.
Figure 16.19 ASM charts of the talker and listener of the two-phase handshaking protocol.
For an asynchronous subsystem, storing data into another subsystem is known as a *push* operation and retrieving data from another subsystem is known as a *pull* operation. Many applications process data stage after stage, and thus the push operation is more common.

### 16.8.1 Four-phase handshaking protocol data transfer

The `req` and `ack` signals of the handshaking protocol form a special signaling mechanism and can be associated with various operations in the talker and listener. They can be used to perform push, pull or combined operations.

**Basic one-direction data transfer** Let us first consider the basic push operation, in which the talker transfers one data word to the listener. The conceptual block diagram and a representative timing diagram are shown in Figure 16.20. The basic handshaking sequence remains the same, and the talker places data on the data bus according to activation and deactivation of the `req` signal. The operation follows the basic handshaking sequence:

1. The talker activates the `req` signal and also places the data on the data bus.
2. The listener detects activation of the `req` signal and understands that data is available. After retrieving and processing the data, it activates the `ack` signal.
3. When the talker senses activation of the `ack` signal, it deactivates the `req` signal and removes the data from the data bus.
4. The listener deactivates the `ack` signal accordingly.
5. Once the talker senses deactivation of the `ack` signal, it knows the data transfer is completed and a new one can be initiated.

A possible implementation of the talker and listener is shown in Figure 16.21. We assume that the data line is a tri-state bus. The talker can place the data word on the bus by asserting the `tri_oe` signal, the enable signal of the tri-state buffer. As discussed above, the data is placed on the bus when the `req` signal is asserted. This can be achieved by asserting the `tri_oe` signal in the `s_req1` state of the talker FSM. Note that when the data is on the bus,
the req_out signal is also asserted. We can actually use the req_out signal to control the
tri-state buffer.

The listener has a register for the input data and retrieves the data word by asserting the
data_en signal, the enable signal of the register. The data_en signal can be asserted when
the listener detects activation of the req_syn signal. Since the req_syn signal is delayed by
two D FFs of the synchronizer, its activation is at least one clock cycle later than activation
of the req and data signals. Thus, the data signal should be stabilized when the req_sync
signal is activated and thus no timing violation will occur. We can modify the code of the
listener FSM in Listing 16.8 to include the data_en signal as an output signal:

```
state_next <= state_reg;
data_en <= '0';
case state_reg is
  when s_ack0 =>
    if req_sync='1' then
      state_next <= s_ack1;
data_en <= '1';  -- activate enable signal
    end if;
  when s_ack1 =>
    ...
end case;
```

Note that this design is only for demonstration purposes. Since the data transfer is not
bidirectional, the tri-state buffer is not actually needed. The push operation should function
properly as long as the desired data is placed on the data bus when the req_out signal is
asserted.

The basic pull operation is similar to the push operation except that the listener provides
the data and the talker retrieves the data. The simplified block diagram and timing diagram
are shown in Figure 16.22. After sensing activation of the req signal, the listener places
the data on the data bus and activates the ack signal. Once detecting activation of the ack
signal, the talker retrieves the data and deactivates the req signal. The listener then removes
the data and deactivates the ack signal accordingly. Again, because the ack_syn signal is
Bidirectional data transfer

The four-phase handshaking protocol can also incorporate more sophisticated operation. The talker can bundle additional information, such as the commands and address lines, push them to the listener and later pull the result back. The listener retrieves the bundled signals, processes the data according to the command and activates the ack signal when the operation is done. The following example illustrates the use of handshaking to access an eight-word register file in a different clock domain. We assume that a system consists of a processor and an I/O controller, which reside in different clock domains. The processor can read data from or write data to the eight-word register file of the I/O controller through an asynchronous interface based on the four-phase handshaking protocol. The talker and listener are in the processor’s clock domain and the I/O controller’s clock domain respectively. The basic block diagram is shown in Figure 16.23. To reduce the clutter, only the main components and connections of the data paths are shown.

In this system, the processor first checks the ready signal to ensure that the talker is not busy and then initiates the access by activating the start signal of the talker accordingly. When asserting the start signal, the processor also uses the rw signal to indicate the type of operation ('1' for read and '0' for write), places the address of the register file on the addr line and, in the case of a write operation, places data on the data line. After detecting the start signal, the talker of the asynchronous interface loads the address, the rw control signal and data (if needed) into its internal registers and starts the handshaking and data transfer operation.

The bundled signals include a 3-bit address line, a control signal, pull, and an 8-bit data line. Since the pull and push operations are mutually exclusive, the data line can be shared and thus is bidirectional.

The data path of the talker includes a register for the address, a register for the rw signal and two data registers to store the transmitted and received data. The data path of the listener is an eight-word register file. In a realistic scenario, the I/O controller should also be able...
Figure 16.23  Block diagram of a push-and-pull system using the four-phase handshaking protocol.
to access the register file, and thus all signals should be multiplexed. For simplicity, the signals from the I/O controller to the register file are not shown.

The basic handshaking sequence of this circuit remains the same, and thus the state transition is similar to the FSMs of Section 16.7.1. The talker and listener FSMs also function as the control paths that control operation of the two data paths. In the idle state, the talker FSM checks the start signal. If it is asserted, the FSM moves to the s_req1 state and stores the relevant information to the registers. The remaining operation of the data path depends on the type of access. Let us first consider the push operation. In the s_req1 state, the talker activates the req_out signal and enables the tri-state buffer. The data is placed in the data line accordingly. Note that since the address line and the pull signal are not shared, they are connected to the listener data path during the entire operation.

When the listener FSM detects activation of the req_sync signal, it also checks the pull signal, whose '0' value indicates a push operation. At the next rising edge of the clock, the FSM moves to the s_ack1 state, and the data will be stored into the location specified by the addr line. Note that the req_sync signal is delayed by the D FFs of the synchronizer. All other signals are already stabilized when its activation is detected. The FSM also activates the ack_out signal when entering the s_ack1 state.

After the talker FSM senses activation of the ack_sync signal, it moves to the s_req0 state, deactivates the req_out signal and disables the tri-state buffer. The talker and listener then proceed as in regular four-phase handshaking protocol to return to the initial state.

For the pull operation, the tri-state buffer of the talker is always disabled. When the listener FSM detects activation of the req_sync signal and assertion of the pull signal, it knows that the transaction is a pull operation. At the next rising edge of the clock, the listener FSM moves to the s_ack1 state, activates the ack_out signal, and enables the tri-state buffer to place the register's output on the data line. When the talker FSM senses activation of the ack_sync signal, it knows that the data is also available. At the next rising edge of the clock, the talker FSM moves to the s_req0 state, deactivates the req_out signal and stores the data into the 12t register. The talker and listener then proceed to return to the initial state.

The VHDL codes for the talker and listener interfaces are shown in Listings 16.11 and 16.12 respectively.

**Listing 16.11**  Talker interface of a push-and-pull system

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity talker_interface is
  port(
    clkt, reset: in std_logic;
    start, rw: in std_logic; -- read or write to i/o
    ack_sync: in std_logic;
    ready: out std_logic;
    req_out: out std_logic;
    d_t21: in std_logic_vector(7 downto 0);
    addr_in: in std_logic_vector(1 downto 0);
    d_12t: out std_logic_vector(7 downto 0);
    pull: out std_logic;
    addr: out std_logic_vector(1 downto 0);
    d: inout std_logic_vector(7 downto 0)
  );
end talker_interface;
```
architecture arch of talker_interface is
    type t_state_type is (idle, s_req1, s_req0);
signal state_reg, state_next: t_state_type;
signal req_buf_reg, req_buf_next: std_logic;
signal t_tri_en: std_logic;
signal l2t_next, l2t_reg: std_logic_vector(7 downto 0);
signal t2l_next, t2l_reg: std_logic_vector(7 downto 0);
signal rw_next, rw_reg: std_logic;
signal addr_next, addr_reg: std_logic_vector(1 downto 0);
begin
  —---------------------
  — talker FSM
  —---------------------
  — state register and output buffer
  process(clkt, resett)
  begin
    if (resett='1') then
      state_reg <= idle;
      req_buf_reg <= '0';
    elsif (clkt'event and clkt='1') then
      state_reg <= state_next;
      req_buf_reg <= req_buf_next;
    end if;
  end process;
  — next-state logic
  process(state_reg, start, ack_sync)
  begin
    ready <= '0';
    state_next <= state_reg;
    case state_reg is
      when idle =>
        if start='1' then
          state_next <= s_req1;
        end if;
        ready <= '1';
      when s_req1 =>
        if ack_sync='1' then
          state_next <= s_req0;
        end if;
      when s_req0 =>
        if ack_sync='0' then
          state_next <= idle;
        end if;
    end case;
  end process;
  — look-ahead output logic
  process(state_next)
  begin
    case state_next is
      when idle =>
        req_buf_next <= '0';
      when s_req1 =>
DATA TRANSFER CROSSING CLOCK DOMAINS

req_buf_next <= '1';
when s_req0 =>
  req_buf_next <= '0';
end case;
end process;
req_out<= req_buf_reg;

---talker data path
---data register
process(clkt,resett)
begin
  if (resett='1') then
    t2l_reg <= (others=>'0');
    t2t_reg <= (others=>'0');
    addr_reg <= (others=>'0');
    rw_reg <= '0';
  elsif (clkt'event and clkt='1') then
    t2l_reg <= t2l_next;
    t2t_reg <= t2t_next;
    addr_reg <= addr_next;
    rw_reg <= rw_next;
  end if;
end process;

---data path next-state logic
process(state_reg,t2l_reg,t2t_reg,addr_reg,rw_reg,d_t2l,#
  addr_in,d,rw,start,ack_sync)
begin
  t2l_next <= t2l_reg;
  t2t_next <= t2t_reg;
  addr_next <= addr_reg;
  rw_next <= rw_reg;
  t_tri_en <= '0';
  case state_reg is
    when idle =>
      rw_next <= rw;
      addr_next <= addr_in;
      if (start='1' and rw='0') then
        t2l_next <= d_t2l;
      end if;
    when s_req1 =>
      if (rw_reg='0') then
        t_tri_en <= '1';
      end if;
      if (ack_sync='1') and (rw_reg='1') then
        t2t_next <= d;
      end if;
    when s_req0 =>
      t_tri_en <= '0';
  end case;
end process;

--output
d <= t2l_reg when t_tri_en='1' else (others=>'Z');
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity listener_interface is
5   port(
    clkl, resetl : in std_logic;
    req_sync : in std_logic;
    ack_out : out std_logic;
    pull : in std_logic;
    addr : in std_logic_vector(1 downto 0);
    d : inout std_logic_vector(7 downto 0)
    );
end listener_interface;

architecture arch of listener_interface is
10 type l_state_type is (s_ack0, s_ack1);
signal state_reg, state_next: l_state_type;
signal ack_buf_reg, ack_buf_next: std_logic;
signal l_tri_en, r_en: std_logic;
type r_file_type is array (3 downto 0) of
15 std_logic_vector(7 downto 0);
signal r_file_reg: r_file_type;
begin
20 ----------------
25 listener FSM
25 ----------------
25 state register and output buffer
process(clkl, resetl)
begin
30 if (resetl='1') then
  state_reg <= s_ack0;
  ack_buf_reg <= '0';
elsif (clkl'event and clkl='1') then
  state_reg <= state_next;
  ack_buf_reg <= ack_buf_next;
end if;
end process;
25 next-state logic
30 process(state_reg, req_sync)
begin
35 state_next <= state_reg;
case state_reg is
  when s_ack0 =>
    if req_sync='1' then
      state_next <= s_ack1;
    end if;
end process;
DATA TRANSFER CROSSING CLOCK DOMAINS

when s_ack1 =>
  if req_sync='0' then
    state_next <= s_ack0;
  end if;
end case;
end process;

— look-ahead output logic
process(state_next)
begin
  case state_next is
    when s_ack0 =>
      ack_buf_next <= '0';
    when s_ack1 =>
      ack_buf_next <= '1';
  end case;
end process;

ack_out<= ack_buf_reg;

— listener data path

process(clkl,resetl)
begin
  if (resetl='1') then
    for i in 0 to 3 loop
      r_file_reg(i) <= (others=>'0');
    end loop;
  elsif (clk1'event and clkl='1') then
    if r_en='1' then
      r_file_reg(to_integer(unsigned(addr))) <= d;
    end if;
  end if;
end process;

— enable logic
process(state_reg,req_sync,pull)
begin
  l_tri_en <= '0';
  r_en <= '0';
  case state_reg is
    when s_ack0 =>
      if (req_sync='1') then
        if (pull='0') then — push
          r_en <= '1';
        end if;
      end if;
    end if;
    when s_ack1 =>
      if (pull='1') then
        l_tri_en <= '1';
      end if;
  end case;
end process;

— output
d <= r_file_reg(to_integer(unsigned(addr))))
As in the handshaking code of Section 16.7.1, we must add two synchronizers for the request and acknowledge signals to complete the implementation.

**Performance of four-phase handshaking data transfer** The strength of four-phase handshaking is that it makes a minimal assumption about the two subsystems. It will function properly even if a subsystem has no knowledge of the clock frequency and the data processing rate of other subsystems. However, there is a high overhead associated with this protocol. Assume that the clock period of the talker and listener are $T_{c,t}$ and $T_{c,l}$ respectively. We can estimate the required time to complete one data transfer. During a data transfer, each FSM traverses all its states and then returns to the initial state. Since the talker and listener FSMs have three and two states respectively, it takes $3T_{c,t} + 2T_{c,l}$. Because both the ack and req signals cross the clock domain, synchronizers are needed. If we assume that two-FF synchronizers are used, the synchronization requires up to two clock cycles whenever a signal is synchronized. The ack signal is used twice in the talker FSM, and the synchronization introduces an overhead of $4T_{c,t}$. Similarly, synchronization of the req signal introduces an overhead of $4T_{c,l}$. Thus, it takes $7T_{c,t} + 6T_{c,l}$ to complete one data transfer, which is very slow compared with the one-clock synchronous data transfer.

### 16.8.2 Two-phase handshaking data transfer

The two-phase handshaking protocol can reduce the overhead by half. However, since only a single handshaking occurs in the protocol, this scheme is less flexible and imposes certain constraints on the data transfer.

Let us first consider the push operation. The data transfer can be embedded in the two-phase handshaking protocol as follows:

1. The talker activates the req signal and places data on the data bus.
2. The listener detects activation of the req signal. It retrieves the data and activates the ack signal.
3. Once the talker senses activation of the ack signal, it removes the data from the data bus.

The first push operation is done at this point. Note that both the req and ack signals are '1'. When the talker wants to push the next data, the handshaking continues from this state:

1. The talker deactivates the req signal and places data on the data bus.
2. The listener detects deactivation of the req signal. It retrieves the data and deactivates the ack signal.
3. Once the talker senses deactivation of the ack signal, it removes the data from the data bus.

Note that after two push operations, the req and ack signals will be '0' and the system returns to the original state.

The block diagram for the two-phase push operation is identical to the four-phase push operation, as shown in Figure 16.20(a). A representative timing diagram is shown in Figure 16.24(a). Unlike the four-phase push operation, the req signal remains unchanged when the talker removes the data from the data bus.

Using the two-phase handshaking protocol to perform a pull operation is more difficult. The two-phase operation only allows the listener to signal the talker that it has placed the
data on the data bus. There is no explicit signaling mechanism to let the listener know when the data is retrieved and when the data can be removed from the bus. One way to overcome the problem is to embed this information in the next operation. When the talker initiates a new data transfer, it implicitly indicates that the data from the previous pull operation has been retrieved. Thus, when the listener detects the transition of the req signal of the next operation, it can safely remove the data from the data bus. The timing diagram is shown in Figure 16.24(b). Note that data must stay on the data bus for a long time if the two pull operations are far apart.

In the four-phase handshaking protocol, the two handshaking operations are used to indicate the initiation and completion of the data transfer. The values of the req and ack signals represent the system state and can be used to indicate the status of the data line as well. The talker and the listener, or any other subsystems that have access to the two signals, can determine the status of the data line via the two signals. This feature is important if the data line is shared. For example, in the push-and-pull design of Section 16.8.1, the push and pull are done in the same data line. The talker and listener need to know the status of the line to avoid bus fighting. On the other hand, in the two-phase handshaking protocol, handshaking operation is used only for initiation of the data transfer. The status of the data line cannot be determined by the req and ack signals, and thus the line cannot be shared. If we want to use the two-phase handshaking protocol for the previous push–pull design, separate data lines are needed for the push and pull operations.

### 16.8.3 One-phase data transfer

If the characteristics of the listener are known in advance, we can customize the data transfer timing and eliminate the acknowledge signal. Since there is no feedback, the request signal behaves like the enable pulse discussed in Section 16.6.
Let us first consider the push operation. The req signal now functions as an enable signal to inform the listener of the availability of the data. Since there is no feedback from the listener, the talker relies on prior knowledge about the listener to calculate the minimal assertion time for the data signal. The talker asserts the req signal and places the data on the data bus for a predetermined interval. The listener will detect activation of the req signal and retrieve the data within this interval. We can use the schemes discussed in Section 16.6 to regenerate an enable pulse from the req signal. A representative timing diagram is shown in Figure 16.25(a). If we assume that the listener is always available, the listener needs about three clock cycles (i.e., $3T_{cyc}$) to store the data into a register. The interval includes two clock cycles to synchronize the req signal and one clock cycle to store the data.

The basic pull operation can be done in a similar fashion. The listener knows in advance how long the data should be put on the data line, and the talker knows when the data should be available. After activating the req signal, the talker waits for a predetermined amount of time and then retrieves data from the data line. A representative timing diagram is shown in Figure 16.25(b).

16.9 DATA TRANSFER VIA A MEMORY BUFFER

Although the handshaking protocol provides a reliable mechanism to transfer data across clock domains, it is not an efficient scheme. Each transaction involves a large overhead, and thus this method is good only for small, random exchanges of information between two subsystems. It is not an effective way to move a large amount of data between the two clock domains. A better alternative is to use a memory buffer as temporary storage. Instead of direct interactions, the two subsystems store and retrieve data via the memory buffer. Two common configurations are the asynchronous FIFO buffer and shared memory. These configurations cannot eliminate the metastable condition but can significantly reduce the overhead associated with data transfer.

16.9.1 FIFO buffer

A FIFO buffer is like a one-directional pipe. The sending subsystem puts the data in one end of the pipe, and the receiving subsystem retrieves the data from the other end of the pipe. Figure 16.25 Timing diagrams of push and pull operations using one-phase protocol.
pipe. In Section 9.3.2, we discussed the operation and design of a synchronous FIFO buffer, in which the two subsystems are controlled by the same clock signal. The operation of an asynchronous FIFO is similar, but the sending and receiving subsystems are controlled by clocks from different clock domains.

In an asynchronous FIFO, the read pointer (counter) is controlled by the clock signal from the receiving subsystem and the write pointer (counter) is controlled by the clock signal from the sending subsystem. Since the operation of these counters only involves the clock signal from its own domain, the counters impose no synchronization problem. The difficulty comes from the full and empty status signals. As discussed in Section 9.3.2, there are several different methods to obtain the status. These methods need information from both the sending and receiving subsystems and thus involve the signals from two clock domains. The main task of implementing an asynchronous FIFO is to design a circuit that generates reliable, properly synchronized status signals.

One possible implementation is to follow the synchronous FIFO organization discussed in Figure 9.14. For a synchronous FIFO with an \( n \)-bit address space (i.e., \( 2^n \) words), it is constructed as follows:

- Use two \((n + 1)\)-bit binary counters as the pointers, one for the read pointer and one for the write pointer.
- Use two \( n \)-bit binary counters (which are the \( n \) LSBs of the \((n + 1)\)-bit counters) as the read and write addresses to access the designated element of the memory array.
- Compare the two \((n + 1)\)-bit counters to obtain full and empty status.

To use this scheme in an asynchronous environment, we must ensure that the comparison circuit can generate the full and empty status signals that are synchronized with their respective clock domains. To accomplish this, we must revise the design as follows:

- Add a synchronizer in the comparison circuit to synchronize the pointer from the other clock domain.
- Replace \((n + 1)\)-bit and \( n \)-bit binary counters with the \((n + 1)\)-bit and \( n \)-bit Gray counters.

In synchronous FIFO, the read and write pointers are implemented by binary counters or LFSRs. In these counters, there may be multiple bit changes in a transition. For example, consider a 4-bit binary counter. When the counter wraps around from "1111" to "0000", all four bits change. As discussed in Section 16.5.5, synchronizing multiple changing bits may lead to the capture of erroneous, intermediate transition values, and thus these counters can cause problems if the values are passed to a different clock domain. To prevent this, we must use a Gray counter for the pointer, in which only one bit is changed in a transition. The circulation pattern of a 4-bit counter is shown in the first column of Table 16.2.

In Section 9.3.2, we add an extra bit in the binary counter and use this bit (the MSB of the counter) to distinguish whether the FIFO is full or empty. In this approach, we use two \((n + 1)\)-bit binary counters as the read and write pointers, and use two \( n \)-bit binary counters for the write and read addresses of the memory array. Note that the MSB of the Gray counter is the same as the MSB of the binary counter, and thus it can be used to distinguish whether the FIFO is empty or full. As in the binary counter-based implementation, we need two \((n + 1)\)-bit Gray counters as the pointers and two \( n \)-bit Gray counters as the addresses.

It is straightforward to obtain the \( n \)-bit binary counting patterns since they are the same as the \( n \) LSBs of the \((n + 1)\)-bit binary counter. It is more difficult for the Gray counter. For example, the counting patterns of three LSBs of the 4-bit Gray counter and the counting patterns of a 3-bit Gray counter are shown in the second and third columns of Table 16.2. Their patterns are different in the bottom half. Although the patterns are not identical, there
is no need to construct a separate \( n \)-bit Gray counter from scratch. Closer observation shows that the \((n - 1)\) LSBs of the \((n + 1)\)-bit Gray counter and \( n \)-bit Gray counter are identical, and the MSB of the \( n \)-bit Gray counter can be obtained by performing an xor operation on the two MSBs of the \((n + 1)\)-bit Gray counter. In other words, let \( a_n, a_{n-1}, \ldots, a_0 \) be the bits of an \((n + 1)\)-bit Gray counter, and \( b_{n-1}, b_{n-2}, \ldots, b_0 \) be the bits of an \( n \)-bit Gray counter. We can derive the \( n \)-bit counting pattern by using

\[
b_i = \begin{cases} 
    a_{i+1} \oplus a_i & \text{if } i = n - 1 \\
    a_i & \text{otherwise}
\end{cases}
\]

The block diagram of an \( n \)-bit asynchronous FIFO control circuit is shown in Figure 16.26. In the write control part, an \((n + 1)\)-bit Gray counter is used as the write pointer and the derived \( n \)-bit Gray counter is used for the write address. The read pointer is obtained from the read control part. It is first synchronized by an \((n + 1)\)-bit synchronizer. The comparing circuit derives the \( n \)-bit read address and compares it to the write address. If the read and write addresses are the same and the MSBs of the read and write pointers are different, the FIFO is full and the `full` signal is asserted accordingly. Since all inputs of the comparing circuits are synchronized with the write controller’s clock, the `full` signal will not cause a timing violation when used. The read control part essentially mirrors the write control except for the minor difference in the comparing circuit. The `empty` signal will be asserted when the read and write addresses are the same and the MSBs of the read and write pointers are the same.

The VHDL codes of the write port control and the read port control are shown in Listings 16.13 and 16.14 respectively. The code of the Gray counter is similar to the code discussed in Section 7.5.1. We use a generic, \( N \), to express the number of bits of the FIFO control circuit. The code of a generic \( n \)-bit two-FF synchronizer is shown in Listing 16.15.

---

**Table 16.2** Circulation pattern of 4-bit and 3-bit Gray counters

<table>
<thead>
<tr>
<th>4-bit Gray counter</th>
<th>3 LSBs of 4-bit Gray counter</th>
<th>3-bit Gray counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>0001</td>
<td>001</td>
<td>001</td>
</tr>
<tr>
<td>0011</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>0010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>0110</td>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>0111</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>0101</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>0100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>1100</td>
<td>100</td>
<td>000</td>
</tr>
<tr>
<td>1101</td>
<td>101</td>
<td>001</td>
</tr>
<tr>
<td>1111</td>
<td>111</td>
<td>011</td>
</tr>
<tr>
<td>1110</td>
<td>110</td>
<td>010</td>
</tr>
<tr>
<td>1010</td>
<td>010</td>
<td>110</td>
</tr>
<tr>
<td>1011</td>
<td>011</td>
<td>111</td>
</tr>
<tr>
<td>1001</td>
<td>001</td>
<td>101</td>
</tr>
<tr>
<td>1000</td>
<td>000</td>
<td>100</td>
</tr>
</tbody>
</table>
The complete asynchronous FIFO control circuit follows the basic block diagram, and its VHDL code is shown in Listing 16.16.

Listing 16.13  Write port control of an asynchronous FIFO

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_write_ctrl is
  generic(N: natural);
  port(
    clkw, resetw: in std_logic;
    wr: in std_logic;
    r_ptr_in: in std_logic_vector(N downto 0);
    full: out std_logic;
    w_ptr_out: out std_logic_vector(N downto 0);
    w_addr: out std_logic_vector(N-1 downto 0)
  );
end fifo_write_ctrl;

architecture gray_arch of fifo_write_ctrl is
  signal w_ptr_reg, w_ptr_next:
    std_logic_vector(N downto 0);
  signal gray1, bin, bin1: std_logic_vector(N downto 0);
```

Figure 16.26  Block diagram of an asynchronous FIFO controller.
signal waddr_all: std_logic_vector(N-1 downto 0);
signal waddr_msb, raddr_msb: std_logic;
signal full_flag: std_logic;
begin
  register
process(clkw,resetw)
begin
  if (resetw='1') then
    w_ptr_reg <= (others=>'0');
  elsif (clkw'event and clkw='1') then
    w_ptr_reg <= w_ptr_next;
  end if;
end process;
— (N+1)-bit Gray counter
bin <= w_ptr_reg xor ('0' & bin(N downto 1));
binl <= std_logic_vector(unsigned(bin) + 1);
grayl <= binl xor ('0' & binl(N downto 1));
— update write pointer
w_ptr_next <= grayl when wr='1' and full_flag='0' else
  w_ptr_reg;
— N-bit Gray counter
waddr_msb <= w_ptr_reg(N) xor w_ptr_reg(N-1);
waddr_all <= waddr_msb & w_ptr_reg(N-2 downto 0);
— check for FIFO full
raddr_msb <= r_ptr_in(N) xor r_ptr_in(N-1);
full_flag <=
  '1' when r_ptr_in(N) /= w_ptr_reg(N) and
  r_ptr_in(N-2 downto 0)=w_ptr_reg(N-2 downto 0) and
  raddr_msb = waddr_msb else
  '0';
— output
w_addr <= waddr_all;
w_ptr_out <= w_ptr_reg;
full <= full_flag;
end gray_arch;

Listing 16.14  Read port control of an asynchronous FIFO

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_read_ctrl is
  generic(N: natural);
  port(
    clkr, resetr: in std_logic;
    w_ptr_in: in std_logic_vector(N downto 0);
    rd: in std_logic;
    empty: out std_logic;
    r_ptr_out: out std_logic_vector(N downto 0);
    r_addr: out std_logic_vector(N-1 downto 0)
  );
end fifo_read_ctrl;
architecture gray_arch of fifo_read_ctrl is
  signal r_ptr_reg, r_ptr_next: std_logic_vector(N downto 0);
signal gray1, bin, bin1: std_logic_vector(N downto 0);
signal raddr_all: std_logic_vector(N-1 downto 0);
signal raddr_msb, waddr_msb: std_logic;
signal empty_flag: std_logic;
begin
  register
  process(clkr, reset)
  begin
    if (reset='1') then
      r_ptr_reg <= (others=>'0');
    elsif (clkr'event and clkr='l') then
      r_ptr_reg <= r_ptr_next;
    end if;
  end process;
  -- (N+1)-bit Gray counter
  bin <= r_ptr_reg xor ('0' & bin(N downto 1));
  bin1 <= std_logic_vector(unsigned(bin1) + 1);
  gray1 <= bin1 xor ('0' & bin(N downto 1));
  -- update read pointer
  r_ptr_next <= gray1 when rd='1' and empty_flag='0' else
    r_ptr_reg;
  -- N-bit Gray counter
  raddr_msb <= r_ptr_reg(N) xor r_ptr_reg(N-1);
  raddr_all <= raddr_msb & r_ptr_reg(N-2 downto 0);
  waddr_msb <= w_ptr_in(N) xor w_ptr_in(N-1);
  -- check for FIFO empty
  empty_flag <=
    '1' when w_ptr_in(N)=r_ptr_reg(N) and
    w_ptr_in(N-2 downto 0)=r_ptr_reg(N-2 downto 0) and
    raddr_msb = waddr_msb else
    '0';
  -- output
  r_addr <= raddr_all;
  r_ptr_out <= r_ptr_reg;
  empty <= empty_flag;
end gray_arch;

Listing 16.15  n-bit synchronizer

library ieee;
use ieee.std_logic_1164.all;
entity synchronizer_g is
  generic(N: natural);
  port(
    clk, reset: in std_logic;
in_async: in std_logic_vector(N-1 downto 0);
out_sync: out std_logic_vector(N-1 downto 0)
  );
end synchronizer_g;

architecture two_ff_arch of synchronizer_g is
signal meta_reg, sync_reg: std_logic_vector(N-1 downto 0);
signal meta_next, sync_next:
    std_logic_vector(N-1 downto 0);

begin
    -- two registers
    process(clk, reset)
    begin
        if (reset='1') then
            meta_reg <= (others=>'0');
            sync_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            meta_reg <= meta_next;
            sync_reg <= sync_next;
        end if;
    end process;
    -- next-state logic
    meta_next <= in_async;
    sync_next <= meta_reg;
    -- output
    out_sync <= sync_reg;
end two_ff_arch;

Listing 16.16 Top-level structural description of an asynchronous FIFO control circuit

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_async_ctrl is
    generic(DEPTH: natural);
port(
    clkw: in std_logic;
    resetw: in std_logic;
    wr: in std_logic;
    full: out std_logic;
    w_addr: out std_logic_vector (DEPTH-1 downto 0);
    clkr: in std_logic;
    resetr: in std_logic;
    rd: in std_logic;
    empty: out std_logic;
    r_addr: out std_logic_vector (DEPTH-1 downto 0)
);
end fifo_async_ctrl;

architecture str_arch of fifo_async_ctrl is
signal r_ptr_in: std_logic_vector(DEPTH downto 0);
signal r_ptr_out: std_logic_vector(DEPTH downto 0);
signal w_ptr_in: std_logic_vector(DEPTH downto 0);
signal w_ptr_out: std_logic_vector(DEPTH downto 0);
    -- component declarations
component fifo_read_ctrl
generic(N: natural);
port(
    clkr: in std_logic;
Because of the synchronizer, the onset of the empty and full signals may be delayed. For example, assume that the FIFO is originally empty. After a write operation is performed, the write pointer changes and the FIFO contains one data item. It takes two clock cycles
to propagate the change through the two cascading FFs of the synchronizer, and thus the onset of the empty signal is delayed by two clock cycles. During this interval, the reading subsystem will falsely assume that there is no data to retrieve and will stay idle. While the delay causes late data retrieval, the functionality of the FIFO remains intact and no invalid data item is retrieved through the buffer. The same situation happens to the full signal. The deassertion of the full signal is delayed by two clock cycles. During this interval, the sending subsystem falsely assumes that the FIFO is full and suspends the write operation.

The delayed empty and full signals force the subsystems to be idle unnecessarily and thus penalize the performance. The penalty is essentially due to the overhead associated with the synchronization of two clock domains and cannot be avoided. However, the idle situation occurs only when the FIFO is almost empty or almost full. There is no overhead or extra delay when the FIFO is partially full. In comparison, the handshaking scheme involves the overhead in every data transaction, and thus the FIFO buffer is more efficient.

16.9.2 Shared memory

Another frequently used buffering scheme is shared memory. The basic idea is to allow multiple subsystems to access a common memory. The sending subsystem can first write the data into the memory, and the receiving subsystem then obtains the data by reading the same memory location. This scheme is best suited when a large chunk of data, such as a high-resolution image, has to be transferred.

The shared memory scheme can be implemented by using a regular single-port memory or a special dual-port memory. For a single-port memory configuration, we can treat the memory as the shared resource and use an arbiter to resolve the conflicting requests and coordinate the memory usage. The basic design of the arbiter is similar to that in Section 10.8.2. Because the interactions are between different clock domains, synchronization circuits are needed for all request and grant signals.

The request-grant process is somewhat like the handshaking procedure and has a similar overhead. However, the overhead is associated with each resource arbitration, not each data transaction. Since one round of arbitration allows any amount of data to be transferred (up to the size of the shared memory), the average overhead of a single data transaction becomes very small.

A better alternative is to use dual-port memory. A dual-port memory has two independent access ports, each containing its own address line, data line and control signals. The multiplexing and decoding circuits are duplicated inside the memory module. Two memory accesses can be performed simultaneously as long as the memory addresses are different. A conflict occurs when two memory operations access the same memory location (i.e., two operations have the same memory addresses). An arbiter is used to resolve the condition. When there is no clock in the regular dual-port memory, the internal arbitrator is an asynchronous sequential circuit. It may enter a metastable state if the two request signals are asserted too closely. As in the synchronizer, the arbitration circuit needs to provide some time to resolve the metastable condition and thus introduces similar overhead.

As with other memory modules, dual-port memory cannot be synthesized from scratch in RT-level code. We must instantiate the predesigned module from the target device technology.
16.10 SYNTHESIS OF A MULTIPLE-CLOCK SYSTEM

The synchronous design is the most important methodology and the cornerstone of the entire design and fabrication process. The synthesis, timing analysis, verification and testing of synchronous systems are well understood, and many EDA software tools have been developed to automate the tasks.

One major motivation behind the synchronous methodology is to provide a systematic way to satisfy the timing constraints. The objective of synthesis and verification is to identify and prevent timing violations. The EDA software tools are developed to assist designers in achieving this objective. On the other hand, the metastability analysis and synchronization circuit are to deal with the scenario that a timing violation has already occurred. This is essentially a transistor-level phenomenon, and its behavior cannot easily be modeled at the gate or RT level. The EDA tools are not able to handle metastability, and the analysis and synthesis process cannot be automated. Most software can only detect and warn the onset of a time violation but cannot model or analyze what happens afterward. For example, in the std_logic data type, the 'X' value is used to model the output after a timing violation. After a timing violation occurs, the output 'X' value will be permanent and propagated to the downstream circuit. This is very different from the actual timing violation, in which the output signal may become '0' or '1', or be resolved to a stable value after a random period of time.

While the design considerations for a multiple-clock system are different from those of a synchronous system, it is not wise to abandon the synchronous design methodology and start from scratch. Instead, we want to incorporate the methodology into the new design flow and utilize the previous techniques and EDA tools as much as possible.

To achieve this goal, a multiple-clock system should be divided into synchronous subsystems and crossing-domain interfaces. A synchronous subsystem is within the same clock domain, and thus we can design it just as a regular synchronous system. On the other hand, the crossing domain interface involves the synchronization and data transfer protocol. Its analysis and design are very different from those of the synchronous system, and very few EDA tools are available for these tasks. We usually have to manually analyze, design and verify the interface circuit and protocols. Since the synchronization circuit depends on the device characteristics and the data transfer protocol sometimes depends on the clock rates of the domains, the interface is usually device dependent and is not portable.

The general design approach for a multiple-clock system can be summarized as follows:

1. Partition the system into locally synchronous subsystems.
2. Design and verify these subsystems following the synchronous methodology.
3. Develop protocol to pass data and exchange information between clock domains.
4. Manually analyze and design the necessary synchronization circuits between clock domains.
5. Verify operation of the entire system.

A representative top-level partition of a system with two clock domains is shown in Figure 16.27. It is a good idea to treat the synchronization circuit and data transfer interface as separate modules and instantiate them individually in VHDL code. These modules are normally device dependent and may need to be reanalyzed and redesigned when the system is ported to a different device technology or operation environment (e.g., a different clock rate).
16.11 SYNTHESIS GUIDELINES

16.11.1 Guidelines for general use of a clock

- Do not manipulate the clock signal in regular RT-level design and synthesis.
- Minimize the number of clock signals in a system.
- Minimize the number of clock domains (i.e., the number of independent clock signals). Use a derived clock signal when possible.
- If a derived clock signal is needed, manually derive and instantiate the circuit and separate it from the regular synthesis.

16.11.2 Guidelines for a synchronizer

- Synchronize a signal in a single place.
- Avoid synchronizing related signals.
- Use a glitch-free signal for synchronization.
- Reanalyze and examine the synchronizer and MTBF when the device is changed or the clock rate is revised.

16.11.3 Guidelines for an interface between clock domains

- Clearly identify the boundary of the clock domain and the signals that cross the domain.
- Separate the synchronization circuits and asynchronous interface from the synchronous subsystems and instantiate them as individual modules.
- Use a reliable synchronizer design to provide sufficient metastability resolution time.
- Analyze the data transfer protocol over a wide range of scenarios, including faster and slower clock frequencies and different data rates.
The construction of the clock network involves the transmission and distribution of electronic signals. It is normally covered under the subjects of signal integration and high-speed design. Two texts by Howard Johnson, *High-Speed Digital Design: A Handbook of Black Magic* and *High-Speed Signal Propagation: Advanced Black Magic*, provide comprehensive coverage of this topic.

The study of metastability and synchronization failure relies on transistor-level model and analysis. The text, *Digital Systems Engineering* by William J. Dally and John W. Poulton, covers the theoretical foundation of this subject. The book also includes a discussion of the design of asynchronous circuit.

The more practical design materials on the synchronizer and asynchronous interface can be found in manufacturers' application notes or articles from technical conferences. Articles by Clifford E. Cummings, *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*, *Simulation and Synthesis Techniques for Asynchronous FIFO Design*, and *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*, provide many practical design examples and good advice.

**Problems**

16.1 Assume that a sequential system with an ideal clock signal can operate at a maximal clock rate of 100 MHz. If the physical clock distribution network introduces a 1.5-ns clock skew, what is the new maximal clock rate?

16.2 Consider a D FF with $w$ and $\tau$.

(a) If we improve the D FF by reducing $w$ by 10%, discuss the effect on MTBF.

(b) If we improve the D FF by reducing $\tau$ by 10%, discuss the effect on MTBF.

16.3 At the end of Section 16.4.2, we discuss the difference between $T_r$ and $T_{r2}$. Assume that $w$ and $\tau$ are identical for the calculation of MTBF($T_r$) and MTBF($T_{r2}$). Derive MTBF($T_r$) in terms of MTBF($T_{r2}$), $w$ and $\tau$.

16.4 For the two-FF synchronizer discussed in Section 16.5.3, determine the new MTBF for the following:

(a) The placement and routing process adds a 2.5-ns wiring delay.

(b) The system clock rate is decreased by 10%.

(c) The setup time of the D FF is reduced by 10%.

(d) The setup time of the D FF is reduced by 25%.

(e) The $\tau$ of the D FF is reduced by 10%.

(f) The $\tau$ of the D FF is reduced by 25%.

16.5 We want to regenerate the enable pulse in the listener's clock domain using the four-phase handshaking protocol. In this scheme, the listener has an output signal that is asserted once during the handshaking process.

(a) Revise the listener ASM chart of Figure 16.16 to add a Mealy output signal.

(b) Modify the VHDL code to reflect the revised ASM chart.

16.6 Repeat Problem 16.5, but add a Moore output signal.

16.7 Repeat Problem 16.5 for the two-phase handshaking protocol of Figure 16.19.
16.8 Repeat Problem 16.5, but add a Moore output signal for the two-phase handshaking protocol of Figure 16.19.

16.9 Revise the talker ASM chart of the two-phase handshaking protocol of Figure 16.19 to eliminate the idle state.

16.10 In a handshaking protocol, we like to include a ready signal in talker to indicate that the system is idle and ready to accept another operation. Revise the talker ASM chart of the two-phase handshaking protocol of Figure 16.19 to include the ready output signal.

16.11 We want to design a four-phase handshaking asynchronous interface for the sequential multiplier in Section 11.6. The operand width is 8 bits and the data is passed by a 16-bit bidirectional bus. After sensing the start signal, the talker of the sending subsystem places the data on the data bus and activates the handshaking operation. Once the receiving subsystem detects the request, it retrieves the data and performs the multiplication operation. When the operation is completed, the listener of the receiving subsystem places the result on the data bus and asserts the acknowledge signal, and the talker retrieves the result accordingly. Draw the block diagram and derive VHDL code for this system.

16.12 Repeat Problem 16.11, but use an 8-bit data bus. Since the operation involves two data transfers, we need a master control FSM to coordinate the operation. Derive VHDL code for this system.

16.13 Modify the push-and-pull system of Section 16.8.1 using the two-phase handshaking protocol (additional data lines are needed). Derive the revised block diagram and VHDL code.

16.14 Repeat Problem 16.11, but use the two-phase handshaking protocol and two 16-bit unidirectional data buses. Derive the VHDL code.

16.15 Consider the one-phase push operation in Section 16.8.3. Derive VHDL code for the sending and receiving subsystems with the following clock rates.
   
   (a) \( f_{\text{send}} = 10 \text{ MHz} \) and \( f_{\text{receive}} = 10 \text{ MHz} \)
   
   (b) \( f_{\text{send}} = 10 \text{ MHz} \) and \( f_{\text{receive}} = 40 \text{ MHz} \)
   
   (c) \( f_{\text{send}} = 10 \text{ MHz} \) and \( f_{\text{receive}} = 2.5 \text{ MHz} \)

16.16 Repeat Problem 16.15 for the one-phase pull operation.

16.17 Consider the FIFO buffer of Section 16.9.1, and let the clock periods of the writing and reading subsystems be \( T_w \) and \( T_r \) respectively. Assume that the sending subsystem has 10 words to pass through the FIFO buffer. Determine the total time to complete the operation with the following buffer sizes:

   (a) One word.
   
   (b) Two words.
   
   (c) Four words.
   
   (d) Eight words.
   
   (e) 16 words.
REFERENCES


INDEX

abstraction, 9
  gate-level, 10
  processor-level, 12
  register-transfer (RT) level, 11
  transistor-level, 10
actual signal, 478
adder, 171, 201, 510
alias, 52
ALU, 75, 87, 104, 112
arbiter, 353
architecture body, 28, 46
array
  aggregate, 59
  array-of-arrays, 548
  constrained, 546
  emulated two-dimensional, 550
  two-dimensional, 546
  unconstrained, 503, 547
ASIC, 3
  full custom, 3
  gate array, 3
  standard-cell, 3, 143
ASM chart, 317
ASMD chart, 379
association
  named, 479
  positional, 480
asynchronous circuit, 216, 219
attribute
  'event, 222
  'high, 502
  'left, 502
  'length, 502
  'low, 502
  'range, 502
  'reverse_range, 502
  'right, 502
  array, 502
  enum_encoding, 339
  user, 339
barrel shifter, 178, 192, 566
bidirectional I/O, 134
big-O notation, 126
binary decoder, 73, 86, 104, 112, 513, 528, 558
binary encoder, 564
binding, 461
CAM (content addressable memory), 287
case statement, 112
clock
  derived, 262, 611
  distribution network, 603
gated, 260
  skew, 605
combinational circuit, 69, 213
comment, 47
comparator, 173, 177
component, 30, 475
declaration, 31, 475
  instantiation, 32, 477
computability, 126
computation complexity, 126
concurrent statement, 46
conditional signal assignment statement, 72, 105
derivation, 486
specification, 488
constant, 52
cost
development, 7
non-recurring engineering (NRE), 6
part, 6
time-to-market, 7
counter
arbitrary-sequence, 232
binary, 233, 247, 367, 482, 518, 521
decade, 236, 476
decimal, 272, 481
Gray, 265
mod-n, 483
programmable, 237, 248, 252
ring, 266, 511
critical path, 152
D FF, 214, 222, 226, 245
D latch, 214, 219, 221
data type, 53
bit, 53
bit-vector, 53
boolean, 53
realization, 133
signed, 60
std_logic, 56
std_logic_vector, 56
unsigned, 60
dataflow graph, 461
delay-sensitive, 159
delta delay (&delta), 30, 70
design reuse, 21, 218, 474
design unit, 44, 46
design-for-test, 16
development flow, 17, 38
difference circuit, 175
don’t-care, 137
EDA (Electronic Design Automation), 16, 125, 148, 218
detect circuit, 326, 348, 623
entity declaration, 28, 44
equivalence check, 126
exit statement, 537
false path, 152
field programmable device, 4
CPLD, 4
FPGA, 4, 146
simple, 4
FIFO buffer, 279, 591, 652
floor planning, 14
for loop statement, 118, 528
formal signal, 478
FSM, 219, 313, 379, 422
output buffering, 342
safe, 342
FSMD, 376, 385
function, 491
gate count, 11, 132
generate statement
conditional, 517
for, 512
generic, 481, 501
flicker, 156
globally asynchronous locally synchronous (GALS), 216, 612
Gray code, 196, 338
greatest common divisor (GCD), 445
Hamming distance circuit, 206
handshaking, 630
four-phase, 630, 641
two-phase, 637, 650
hardware emulation, 16, 218
HDL (hardware description language), 23
hold time, 216, 243, 609
identifier, 48
IEEE standard, 26
1076 VHDL standard, 26
1076.3 Synthesis Packages, 26
1076.6 RTL Synthesis, 26
1164 Multivalue Logic, 26
if statement, 103
intractable, 128
IP (Intellectual Property), 12
leading-zero counting circuit, 538
LFSR (linear feedback shift register), 269, 586
library, 46, 489
iee, 47
work, 46
LUT (look-up table), 146
Manchester encoding circuit, 363
mask, 2
maximal clock rate, 240
Mealy output, 314, 327, 400
metastability, 613
micron, 2
mode, 45
buffer, 45
in, 45
inout, 45, 135
out, 45
Moore output, 314, 326
look-ahead output buffer, 344
MTBF (mean time between synchronization failure), 614
multiplexer, 72, 78, 85, 103, 112, 532, 552, 560
abstract, 77, 88, 90
multiplier
cell-based carry-save, 581
cell-based ripple-carry, 577
combinational, 203
pipelined, 297, 303, 305, 574
repetitive-addition, 382
sequential add-and-shift, 407, 572
netlist, 9
next statement, 540
number, 49
object, 51
INDEX 669

one-shot pulse generator, 422
operator, 54
  overloaded, 57, 61, 65
  precedence, 55
  realization, 129
  sharing, 164, 396
  synthesis support, 130
package, 46, 492
parallel-prefix structure, 187, 570
parameter, 500
  feature, 501
  width, 500
partition, 495
physical design, 14
pipeline, 293
placement and routing, 14
population counter, 206, 534, 540
priority encoder, 74, 86, 104, 112, 187, 199, 530, 588
process, 33, 97
propagation delay, 132, 150
PWM (pulse width modulation), 275
RAM, 225
  controller, 430
  reduced-and, 509, 537
  reduced-xor, 181, 501, 505, 509, 514, 518, 528, 533, 555
  reduced-xor-vector, 183, 570
register file, 276, 593
register transfer, 12
  RT methodology, 12, 219, 373, 375, 425
  RT-level abstraction, 12
register, 225
reserved words, 48
resolution time, 614
scheduling, 461
selected signal assignment statement, 85, 114
sensitivity list, 33, 98
sequential circuit, 69, 213
  combined, 219
  random, 219
  regular, 219, 424
sequential signal assignment statement, 100
sequential statement, 97
serial-to-parallel converter, 510, 515, 529
setup time, 216, 240, 606
shift register, 229
signal, 51
square-root approximation circuit, 460
state assignment, 338
state diagram, 315
string, 49
strobe generation circuit, 358
strongly typed language, 53
synchronizer, 617
synchronous circuit, 216–217
synthesis, 13
  behavioral, 469
  clock, 604
  flow, 139
  FSM, 337
  FSMD, 417
  gate-level, 13
  high-level, 13, 469
  iterative structure, 541
  logic, 142
  module generator, 141
  multilevel, 142
  multiple clock, 661
  pipelined circuit, 307
  retiming, 308
  RT-level, 13, 139
  sequential circuit, 253
  technology mapping, 14, 143
two-level, 142
TFF, 228, 246
testbench, 35
testing, 16, 218
timing analysis, 15, 218
  clock skew, 606
  FSM, 324
  FSMD, 404
  synchronous sequential circuit, 239
timing hazards, 156
dynamic hazards, 156
  static hazards, 156
tractable, 128
tri-state buffer, 133
tri-state bus, 135
type casting, 63
type conversion, 53, 57, 62
UART, 455
variable assignment statement, 101
variable, 51, 250
verification, 14
  formal, 15
  functional, 14
  timing, 15
Verilog, 25
VHDL, 25
  analysis, 47
  elaboration, 47
view
  behavioral, 8, 33
  physical, 9
  structural, 9, 30
wait statement, 99