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.

RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability. By Pong P. Chu

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_1164` 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

```
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;
   end process;
   y <= tmp(WIDTH-1);
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:

```
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

```
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='1') 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.

```
. . .
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:

```
. . .
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 `unconstrain_dff` 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

```
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);
begin
   aa <= 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;
   end process;
   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:

```
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:

```
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"`):

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

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

```
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:

```
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:

```
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:

```
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 <= '1' when (a="00000110") else ...
```

we can express the constant in the `natural` data type, as in

```
x <= '1' when (a=6) 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 <= to_unsigned(6,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 <= std_logic_vector(to_unsigned(6,WIDTH));
```

Similarly, the previous a=ZERO expression can also be written as follows without using the constant declaration

```
a=std_logic_vector(to_unsigned(0,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:

```
alias sign: std_logic := src(7);
```

Instead of a hard literal, we can use an attribute to refer to the MSB bit:

```
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

```
alias sign: std_logic := src(WIDTH-1);
```

Similarly, instead of expressing rotating right 1 bit as

```
dest <= src(0) & src(7 downto 1);
```

we can write

```
dest <= src(src'right) & 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'range=>'1')`. We use the `'range` attribute to obtain the range of the a signal and then construct an aggregate `(a'range=>'1')`, in which all elements are assigned to `'1'` (i.e., `"1...1"`). The `a=(a'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'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

```
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='1') 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

```
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 `"000000001"`. 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_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');
         elsif (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 to 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 n-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 ith 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.

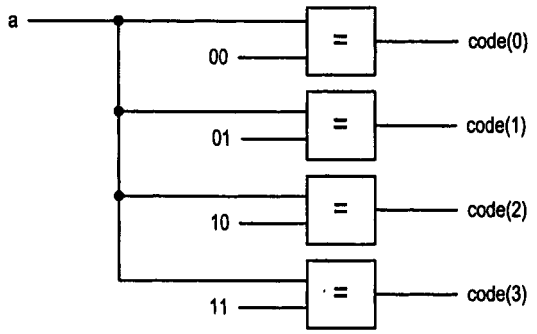


Figure 14.1 Block diagram of a 2-to-4 decoder.

Listing 14.11 Parameterized binary decoder using a for generate statement

```
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 ith 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 ith stage, the three signals can be expressed as

```
tmp(i+1) <= tmp(i) xor a(i+1);
```

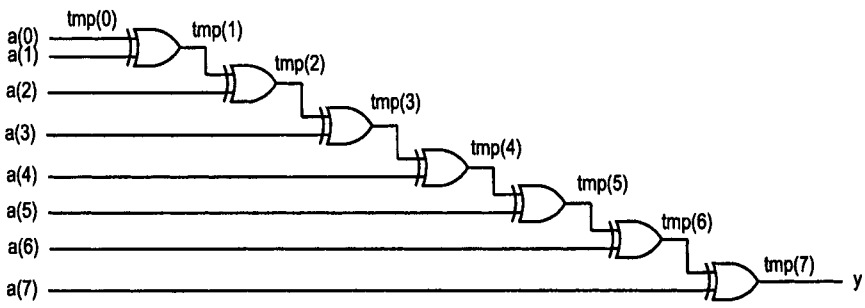


Figure 14.2 Block diagram of a reduced-xor circuit.

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

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

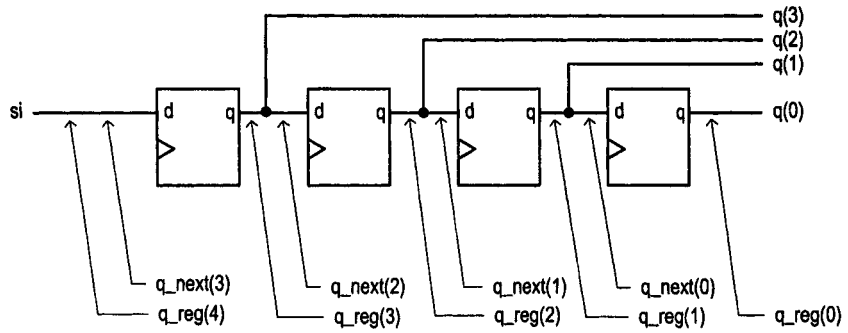


Figure 14.3 Block diagram of a serial-to-parallel converter.

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

```
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;
   dff_gen:
   for i in (WIDTH-1) downto 0 generate
      -- D FF
      process(clk)
      begin
         if (clk'event and clk='1') then
           q_reg(i) <= q_next(i);
         end if;
      end process;
      -- next-state logic
      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 D FF as an entity and use it through component instantiation. The VHDL codes for the D FF 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

```
-- D FF
library ieee;
use ieee.std_logic_1164.all;
entity dff is
   port(
      clk: in std_logic;
```

```
        d: in std_logic;
        q: out std_logic
    );
end dff;

architecture arch of dff is
begin
    process(clk)
    begin
        if (clk'event and clk='1') 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:

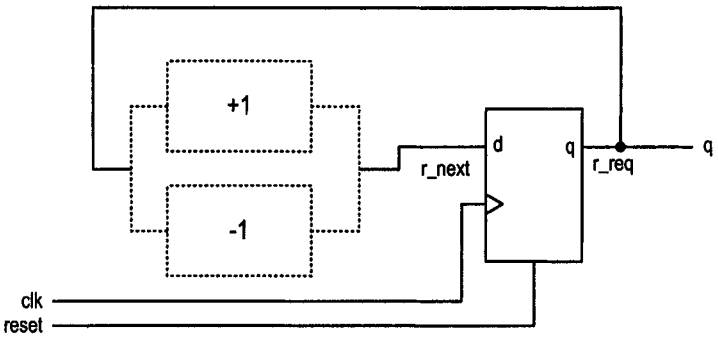
```
tmp(0) <= a(0);
y <= tmp(WIDTH-1);
```

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.

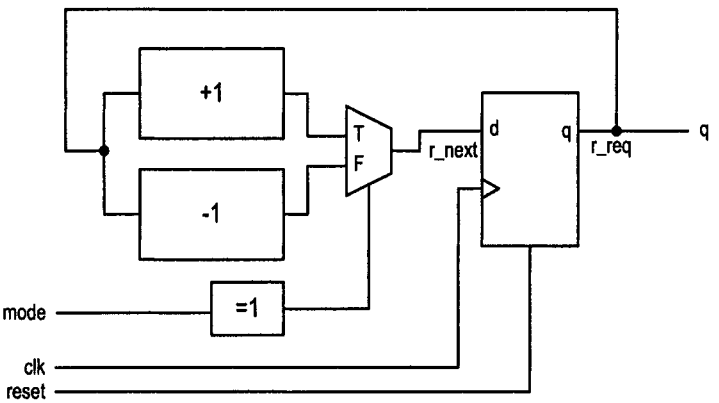
Listing 14.15 Parameterized reduced-xor circuit with a conditional generate statement

```
architecture gen_if_arch of reduced_xor is
   signal tmp: std_logic_vector(WIDTH-2 downto 1);
begin
   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 (1 < 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 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.



(a) Up-or-down counter



(b) Up-and-down counter

Figure 14.4 Block diagrams of up-or-down and up-and-down counters.

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

```
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(
```

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

15 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
20    process(clk,reset)
      begin
         if (reset='1') then
            r_reg <= (others=>'0');
         elsif (clk'event and clk='1') then
25          r_reg <= r_next;
         end if;
      end process;
      -- next-state logic
      inc_gen:  -- incrementor
30    if UP=1 generate
         r_next <= r_reg + 1;
      end generate;
      dec_gen:  --decrementor
      if UP/=1 generate
35       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
5    generic(WIDTH: natural);
     port(
        clk, reset: in std_logic;
        mode: in std_logic;
        q: out std_logic_vector(WIDTH-1 downto 0)
10   );
   end up_and_down_counter;
```

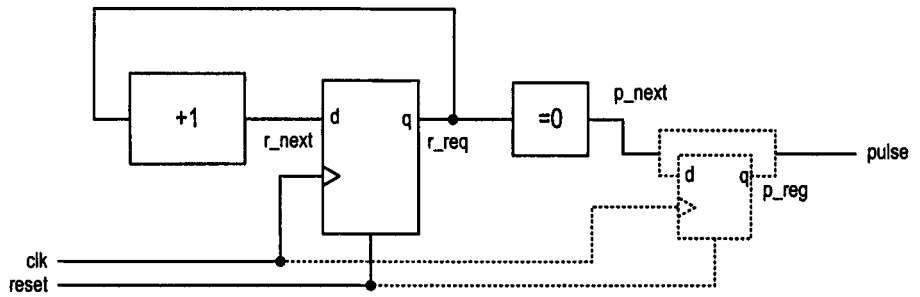


Figure 14.5 Block diagram of a counter with an optional output buffer.

```
architecture arch of up_and_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
   r_next <= r_reg + 1 when mode='1' else
             r_reg - 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end arch;
```

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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity op_buf_counter  is
   generic(
      WIDTH: natural;
      BUFF: natural
   );
   port(
```

```
10        clk, reset: in std_logic;
          pulse: out std_logic
      );
   end op_buf_counter;

15 architecture arch of op_buf_counter is
      signal r_reg: unsigned(WIDTH-1 downto 0);
      signal r_next: unsigned(WIDTH-1 downto 0);
      signal p_next, p_reg: std_logic;
   begin
20    -- register
      process(clk,reset)
      begin
         if (reset='1') then
            r_reg <= (others=>'0');
25       elsif (clk'event and clk='1') then
            r_reg <= r_next;
         end if;
      end process;
      -- next-state logic
30    r_next <= r_reg + 1;
      -- output logic
      p_next <= '1' when r_reg=0 else '0';
      buf_gen: -- with buffer
      if BUFF=1 generate
35       process(clk,reset)
         begin
            if (reset='1') then
               p_reg <= '0';
            elsif (clk'event and clk='1') then
40             p_reg <= p_next;
            end if;
         end process;
         pulse <= p_reg;
      end generate;
45    no_buf_gen: -- without buffer
      if BUFF/=1 generate
         pulse <= p_next;
      end generate;
   end arch;
```

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.

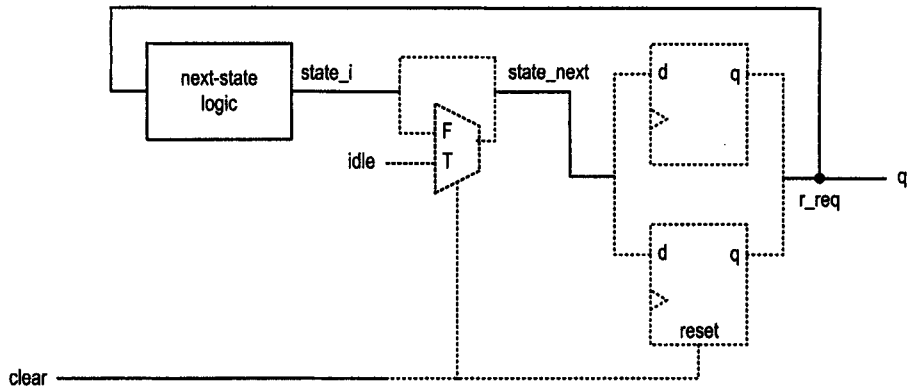


Figure 14.6 Block diagram of FSM with a selectable clear signal.

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;
begin
   -- state register
   reset_ff_gen: -- register with asynchronous clear
   if SYNC/=1 generate
      process(clk,clear)
      begin
         if (clear='1') then
            state_reg <= idle;
         elsif (clk'event and clk='1') then
            state_reg <= state_next;
         end if;
```

```
      end process;
   end generate;
   no_reset_ff_gen: -- register without asynchronous clear
   if SYNC=1 generate
      process(clk)
      begin
         if (clk'event and clk='1') 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='1' then
              if rw='1' 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='1') 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='1' else
                    state_i;
   end generate;
   -- Moore output logic
   process(state_reg)
   begin
```

```
        we <= '0';
        oe <= '0';
        case state_reg is
           when idle =>
           when write =>
              we <= '1';
           when read1 =>
              oe <= '1';
           when read2 =>
              oe <= '1';
           when read3 =>
              oe <= '1';
           when read4 =>
              oe <= '1';
        end case;
    end process;
end mult_seg_arch;
```

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:

```
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

```
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

```
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
```

```
    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:

```
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

```
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.

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

```
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='1') 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 the 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

```
architecture loop2_arch of s2p_converter is
   signal q_reg, q_next: std_logic_vector(WIDTH-1 downto 0);
begin
   -- registers
   process(clk)
   begin
      for i in WIDTH-1 downto 0 loop
         if (clk'event and clk='1') then
            q_reg(i) <= q_next(i);
         end if;
      end loop;
   end process;
   -- next-state logic
   process(si,q_reg)
   begin
     q_next(WIDTH-1) <= si;
```

```
      for i in WIDTH-2 downto 0 loop
         q_next(i) <= q_reg(i+1);
      end loop;
   end process;
   q <= q_reg;
end loop2_arch;
```

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(WIDTH-1 downto 0)`, and priority is given in descending order (i.e., the `r(WIDTH-1)` signal has the highest priority). In addition to the binary code, the `bcode` signal, the output includes the `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));
```

```
25          end if;
         end loop;
      end process;
      -- reduced-or circuit
      process(r,tmp)
30    begin
         tmp(0) <= r(0);
         for i in 1 to (WIDTH-1) loop
            tmp(i) <= r(i) or tmp(i-1);
         end loop;
35    end process;
      valid <= tmp(WIDTH-1);
   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 bcode signal, we can use the log2c function derived in Chapter 12. We assume that it is stored in the 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 bcode signal. Recall that the last signal assignment takes effect in a process. The 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 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 WIDTH is 4. The flattened code becomes

```
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.

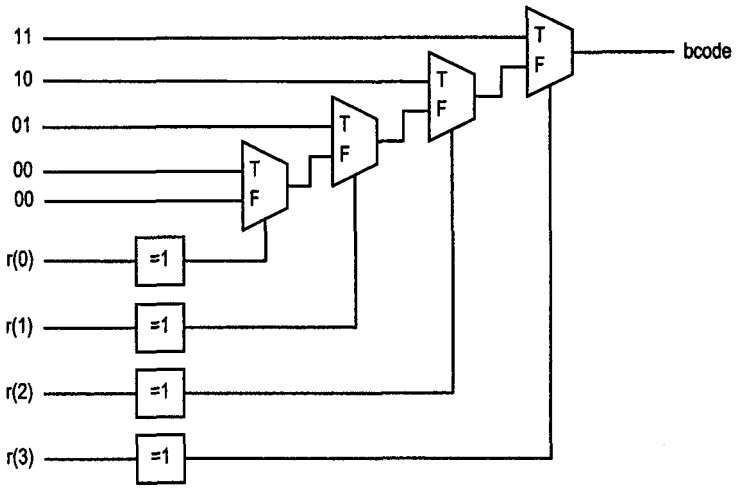


Figure 14.7 Block diagram of a priority encoder.

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.

Listing 14.25 Parameterized multiplexer using a for loop statement

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity mux1 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 mux1;

architecture loop_linear_arch of mux1 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 loop;
   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

```
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

```
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:

```
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

```
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.

Listing 14.27 Parameterized population counter

```
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:

```
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;
```

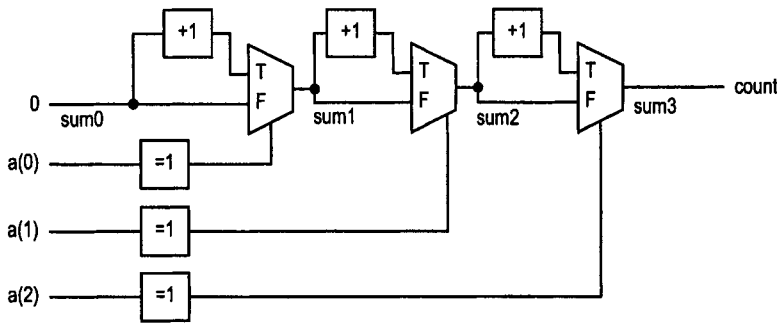


Figure 14.8 Block diagram of population counter.

```
end if;
-- 2nd stage
if a(1)= '1' then
   sum2 := sum1 + 1;
else
   sum2 := sum1;
end if;
-- 3rd stage
if a(2)= '1' then
   sum3 := sum2 + 1;
else
   sum3 := sum2;
end if;
count <= sum3;
```

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 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

```
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.

Listing 14.30 Parameterized leading-zero counting circuit using a bypass flag

```
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, `bypass(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 `bypass(3)` is set to '1', `bypass(2)`, `bypass(1)` and `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 `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 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.

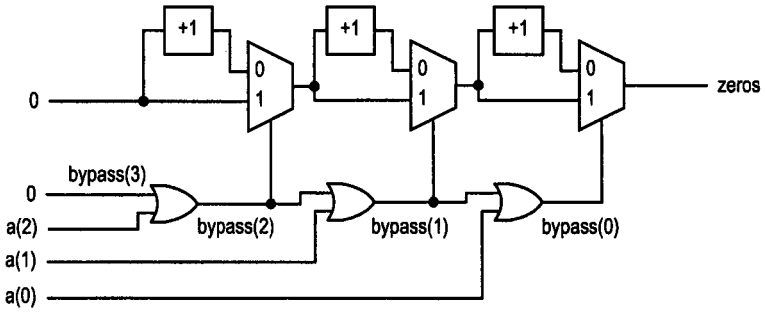


Figure 14.9 Block diagram of a leading-zero counting circuit.

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

```
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

```
next when boolean_expr;
```

When the `boolean_exp` 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)
      variable sum: unsigned(log2c(WIDTH)-1 downto 0);
   begin
      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 ... loop
   sequential statement 1;
   next when boolean_exp;
   sequential statement 2;
end loop;
```

It can be rewritten as

```
for ... loop
   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

CHAPTER 15

PARAMETERIZED DESIGN: PRACTICE

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:

```
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

```
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=>(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 v1: std_logic_vector(COL-1 downto 0);
. . .
process(...)
begin
   for i in COL-1 downto 0 loop
      v1(i) <= t5(1,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:

```
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_2(15 downto 0, 31 downto 0);
signal s4, s5: std_logic_2(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

```
. . .
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:

```
constant ROW natural:=4;
constant 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:

```
signal t1, t2, t3: sram_aoa_46_type;
. . .
-- positional association
t1 <= ("000000",
       "010101",
       "000111",
       "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

```
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:

```
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

```
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

```
. . .
entity . . .
   generic(
      ROW: natural;
      COL: natural
   );
. . .
architecture . . .
  type aoa_RC_type is
```

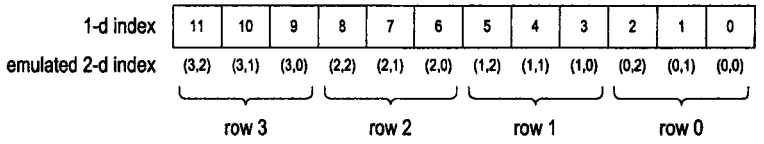


Figure 15.1 Emulation of a two-dimensional 4-by-3 array with a one-dimensional array.

```
  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:

$$n = i * 3 + j$$

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

```
constant ROW natural:=4;
constant COL natural:=6;
-- data type is std_logic_vector(ROW*COL-1 downto 0);
function ix(r,c: natural) return natural is
begin
   return (r*COL + c);
end ix;
```

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

```
signal t4: std_logic_vector(ROW*COL-1 downto 0);
signal e1, e2, e3: std_logic;
. . .
```

```
t4(ix(0,0)) <= '1';
t4(ix(1,2)) <= e1 and e2;
e3 <= t4(ix(3,5));
```

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 `ix(i,COL-1)` and the lower boundary is `ix(i,0)`, and thus we can use the range `ix(i,COL-1) downto 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

```
signal t5: std_logic_vector(ROW*COL-1 downto 0);
signal v1: std_logic_vector(COL-1 downto 0);
. . .
v1 <= t5(ix(1,COL-1) downto 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:

```
signal t1, t2, t3: std_logic_vector(ROW*COL-1 downto 0);
. . .
t1 <= "000000" &
      "010101" &
      "000111" &
      "111111";
-- "101010" to all rows
t2 <= "101010" & "101010" & "101010" & "101010";
-- all 0's
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.

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

```
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

```
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

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

can be replaced by

```
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

```
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 $\log_2 n$ 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 $2r$th row and $(2r+1)$th row of the left stage (i.e., the $(s+1)$th stage).

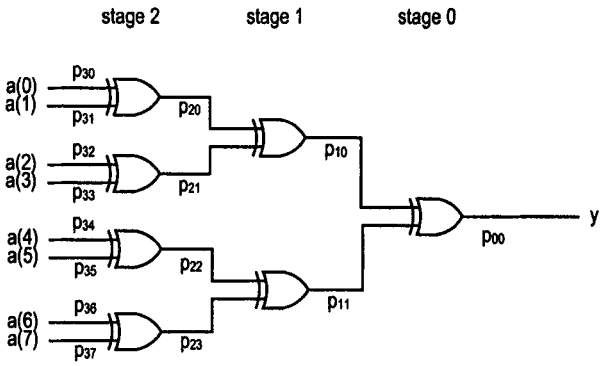


Figure 15.2 Tree-shaped reduced-xor circuit.

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^s rows in the sth 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

```
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_tree_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

```
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

```
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;
       -- replicated structure
      for s in (STAGE-1) downto 0 loop
         for r in 0 to (2**s-1) loop
            p(s,r) <= p(s+1,2*r) xor p(s+1, 2*r+1);
         end loop;
      end loop;
   end process;
   -- rename output signal
   y <= p(0,0);
end loop_tree_arch;
```

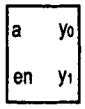
15.3.2 Binary decoder

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-2^1 decoders. The block diagram and the function table of the 1-to-2^1 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:

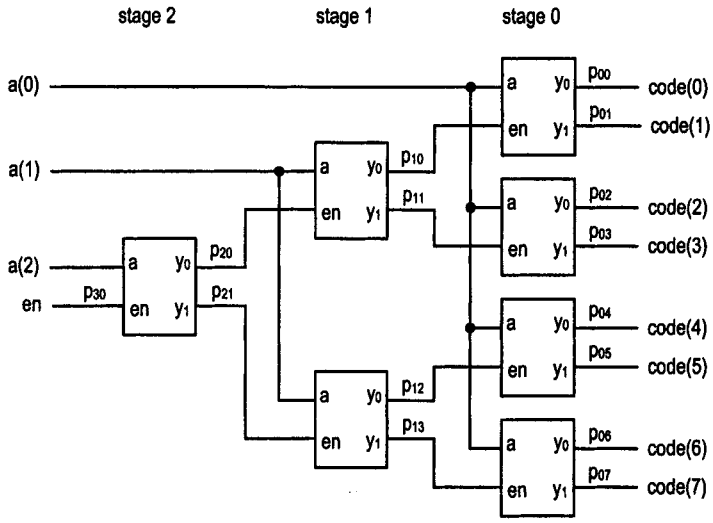
$$y_0 = en \cdot a'$$
$$y_1 = en \cdot a$$

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



input		output	
en	a	y_1	y_0
0	-	0	0
1	0	0	1
1	1	1	0

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



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

Figure 15.3 Tree-shaped binary decoder.

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

```
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)
   );
```

```
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.

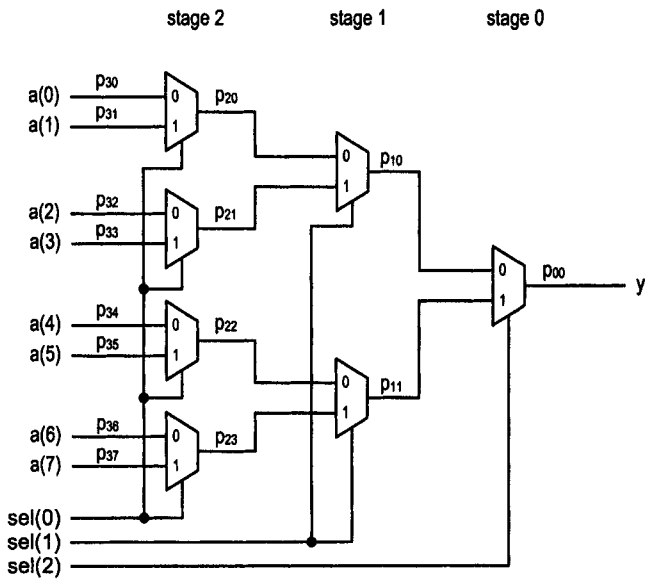


Figure 15.4 Tree-shaped 8-to-1 multiplexer.

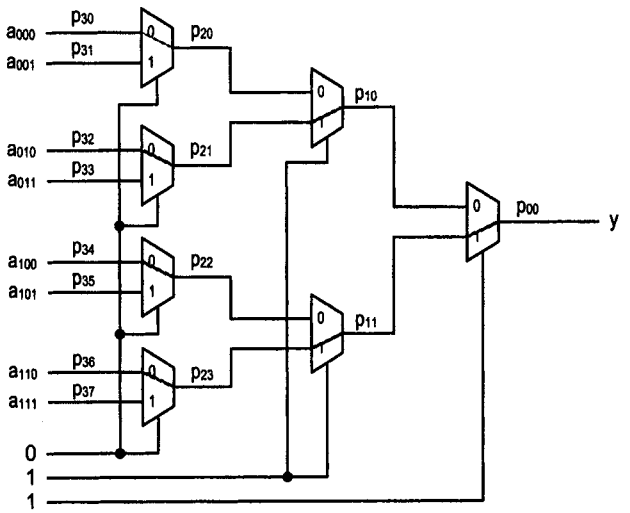


Figure 15.5 Routing with `sel="110"`.

- Stage 0 (the rightmost stage): The MSB of the `sel` signal is '1' and thus the signal with index "1xx", 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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity mux1 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 mux1;

architecture loop_tree_arch of mux1 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;
   end process;
   -- rename output signal
   y <= p(0,0);
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) xor p(s+1,2*r+1);
```

with a 2-to-1 multiplexer:

```
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;
```

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 `sel` signal as an index to select an element from the array. Based on this observation, we can derive the behaviorial VHDL code, as shown in Listing 15.9.

Listing 15.9 Behavioral description of a multiplexer

```
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 `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 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

```
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);
```

Table 15.1 Function table of an 8-to-3 binary encoder

Input $a_7a_6\cdots a_1a_0$	Encoded output $b_2b_1b_0$
0000 0001	000
0000 0010	001
0000 0100	010
0000 1000	011
0001 0000	100
0010 0000	101
0100 0000	110
1000 0000	111
others	don't-care

```
        sel: in std_logic_vector(log2c(WIDTH)-1 downto 0);
        y: out std_logic
    );
  end component;
begin
  -- convert to array-of-arrays data type
  process(a)
  begin
    for i in 0 to (B-1) loop
      for j in 0 to (P-1) loop
        aa(i)(j) <= a(j,i);
      end loop;
    end loop;
  end process;
  -- replicate 1-bit multiplexer B times
  gen_nbit: for i in 0 to (B-1) generate
    mux: mux1
      generic map(WIDTH=>P)
      port map(a=>aa(i), sel=>sel, y=>y(i));
  end generate;
end from_mux1d_arch;
```

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

$$b_2 = a_7 + a_6 + a_5 + a_4$$
$$b_1 = a_7 + a_6 + a_3 + a_2$$
$$b_0 = a_7 + a_5 + a_3 + a_1$$

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:

$$\frac{k}{2^i} \bmod 2 = 1$$

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

```
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)-1) 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.

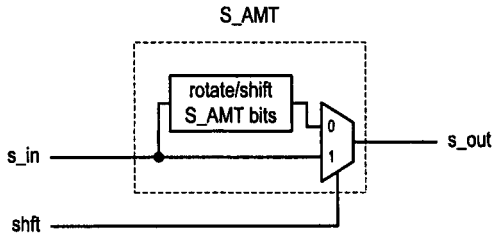
This is an efficient implementation for several reasons. First, as the number of inputs increases, the number of stages grows on the order of $O(\log_2 n)$. The length of the critical path grows in the same order, and thus its performance is much better than the cascading chain. Second, the circuit exhibits 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.

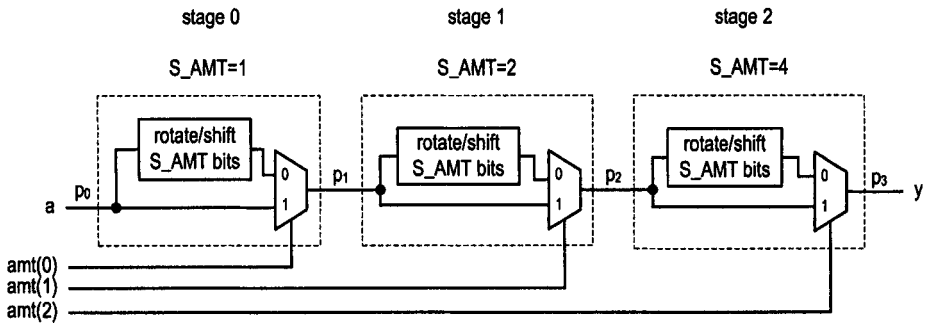
Listing 15.12 Parameterized fixed-size shifting–multiplexing module

```
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);
      shft: 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) &
```



(a) Block diagram of a shifting–multiplexing module



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

Figure 15.6 Parameterized 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 shft='1' else
             s_in;
end para_arch;
```

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 ith bit of the `amt` signal is connected to the `shft` signal of the ith stage. The amount of shifting is determined by the stage and is 2^i for the ith 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
```

```
    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;
begin
    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.

Listing 15.14 Parameterized cascading-chain reduced-xor-vector circuit

```
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 ith row in the sth 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 sth 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.

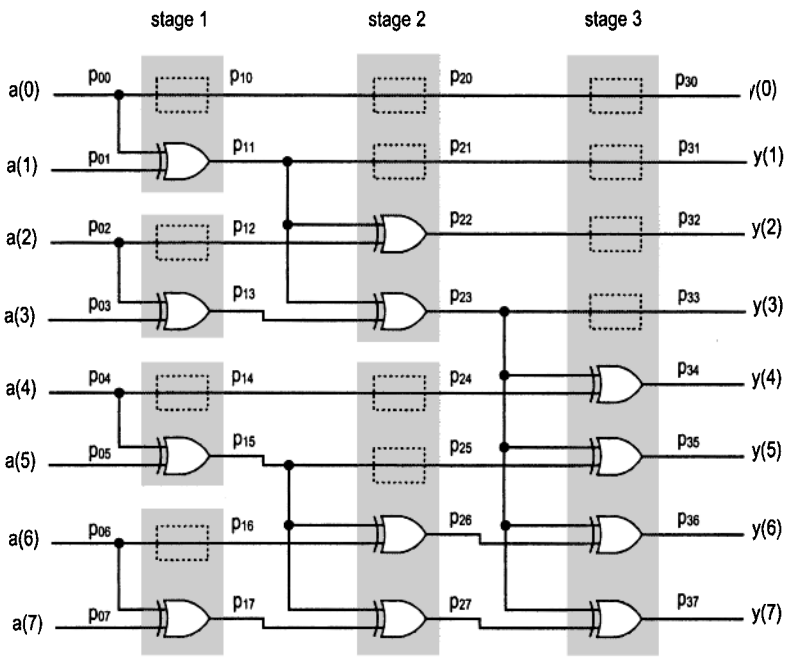


Figure 15.7 Parallel-prefix reduced-xor-vector circuit.

- 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

```
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);
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: 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-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;
```

The main structure is described by a nested three-level for loop statement. The outer loop specifies the iterations over ST stages:

```
for s in 1 to ST loop
```

The middle loop iterates over the modules:

```
for k in 0 to (2**(ST-s)-1) loop
```

The two inner loops iterate over the blocks inside a module:

```
for i in 0 to (2**(s-1)-1) loop
. . .
for i in 2**(s-1) to (2**s-1) loop
. . .
```

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.

Listing 15.16 Parameterized sequential multiplier

```
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_shft);
   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 for the upper part and lower parts of p_reg
   alias pu_next: unsigned(WIDTH 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;
```

```
        n_next <= n_reg;
        p_next <= p_reg;
        ready <='0';
        case state_reg is
           when idle =>
              if start='1' then
                 p_next(WIDTH-1 downto 0) <= unsigned(b_in);
                 p_next(2*WIDTH downto WIDTH) <= (others=>'0');
                 a_next <= unsigned(a_in);
                 n_next <= C_INIT;
                 state_next <= add_shft;
              else
                 state_next <= idle;
              end if;
              ready <='1';
           when add_shft =>
              n_next <= n_reg - 1;
              -- add
              if (p_reg(0)='1') then
                 pu_next <= pu_reg + ('0' & a_reg);
              else
                 pu_next <= pu_reg;
              end if;
              --shift
              p_next <= '0' & pu_next & pl_reg(WIDTH-1 downto 1);
              if (n_next /= 0) then
                 state_next <= add_shft;
              else
                 state_next <= idle;
              end if;
        end case;
     end process;
     r <= std_logic_vector(p_reg(2*WIDTH-1 downto 0));
  end shift_add_better_arch;
```

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 to 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_aoa_n_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

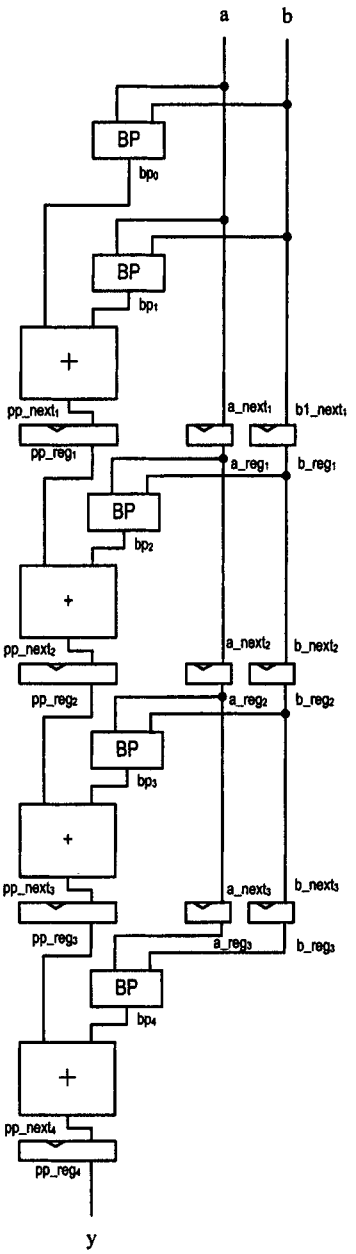


Figure 15.8 Adder-based combinational multiplier with new signal labels.

stage is represented by the bp(i) signal, which is in the form of $0\cdots0\ a_{n-1}b_i\cdots a_0b_i\ 0\cdots0$. 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

```
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/=1) 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=1) 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 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
   g1:
   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);
   -- rename 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, `ai` and `bi`, and a 1-bit carry-in, `ci`, and the output includes a sum bit, `so`, and a carry-out, `co`. The gate-level VHDL description is shown in Listing 15.18. For

					a_3	a_2	a_1	a_0	multiplicand
$\times$					b_3	b_2	b_1	b_0	multiplier
					a_3b_0	a_2b_0	a_1b_0	a_0b_0	
				a_3b_1	a_2b_1	a_1b_1	a_0b_1		
			a_3b_2	a_2b_2	a_1b_2	a_0b_2			
$+$		a_3b_3	a_2b_3	a_1b_3	a_0b_3				
	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	product

Figure 15.9 Multiplication as a summation of a_ib_j terms.

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

```
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_ib_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 ith row and the jth 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 $\text{FA}_{i+1,j}$ cell.
- The `so` port is connected to the $s_{i,j}$ signal, which is connected to the `bi` port of the $\text{FA}_{i+1,j-1}$ cell.
- The `ai` port is connected to the a_ib_j term.
- The `bi` port is connected to the $s_{i-1,j+1}$ signal, which is the `so` signal of the $\text{FA}_{i-1,j+1}$ cell.

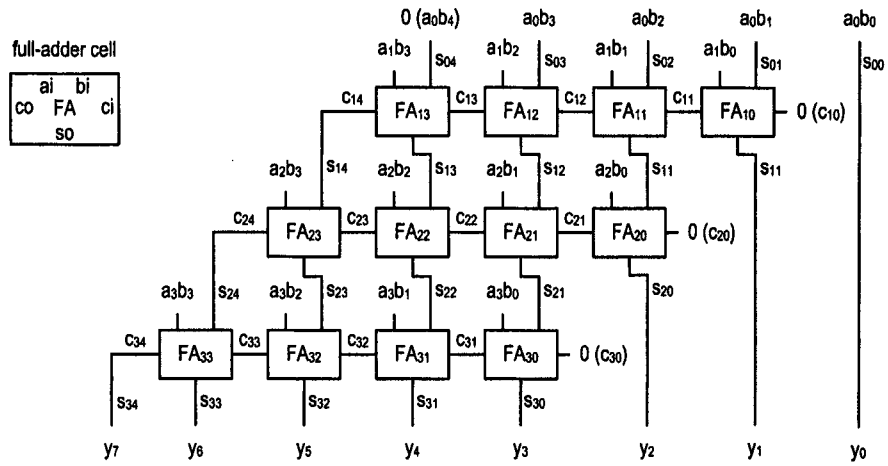


Figure 15.10 Cell-based carry-ripple combinational multiplier.

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_{1j} cell is connected to the a_0b_{j+1} term. Note that the b_4 bit does not exist and the leftmost term (i.e., a_0b_4 in the diagram) is used for the naming convention. The a_0b_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_{i0} 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_{i4} 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 $a_i \cdot b_j$ 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.

Listing 15.19 Parameterized cell-based carry-ripple combinational multiplier

```
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)
```

```
    );
end mult_array;

architecture ripple_carry_arch of mult_array is
    type two_d_type is
        array(N-1 downto 0) of std_logic_vector(N downto 0);
    signal ab, c, s: two_d_type;
    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 and rightmost columns
    g_0_N_col:
    for i in 1 to (N-1) generate
        c(i)(0) <= '0';
        s(i)(N) <= c(i)(N); -- leftmost column
    end generate;
    -- top row
    s(0) <= ab(0);
    ab(0)(N) <= '0';
    -- middle rows
    g_fa_row:
    for i in 1 to (N-1) generate
        g_fa_col:
        for j in 0 to (N-1) generate
            u_middle: fa
                port map
                    (ai=>ab(i)(j), bi=>s(i-1)(j+1), ci=> c(i)(j),
                     so=>s(i)(j), co=>c(i)(j+1));
        end generate;
    end generate;
    -- bottom row and output
    g_out:
    for i in 0 to (N-2) generate
        y(i) <= s(i)(0);
    end generate;
    y(2*N-1 downto N-1) <= s(N-1);
end ripple_carry_arch;
```

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.

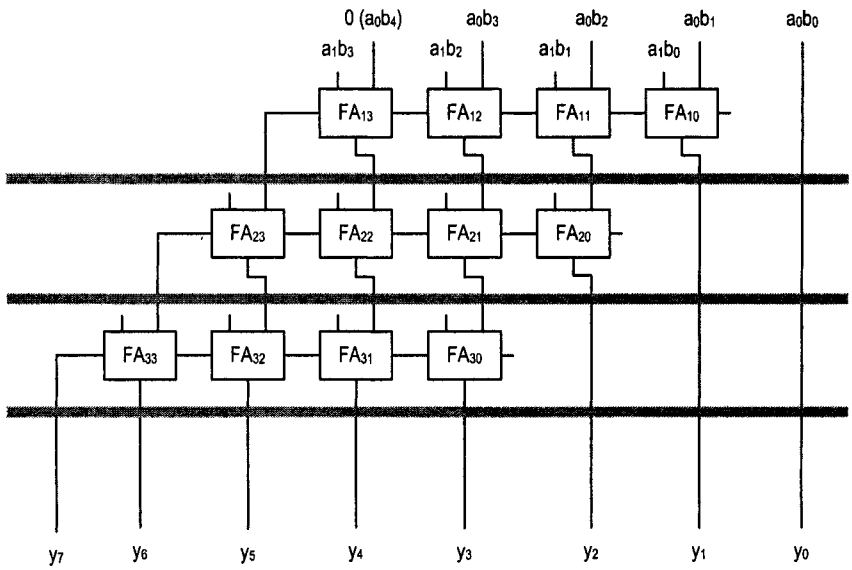


Figure 15.11 Non-optimal pipelined carry-ripple multiplier.

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., so) and the carry-out bit (i.e., co) 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,

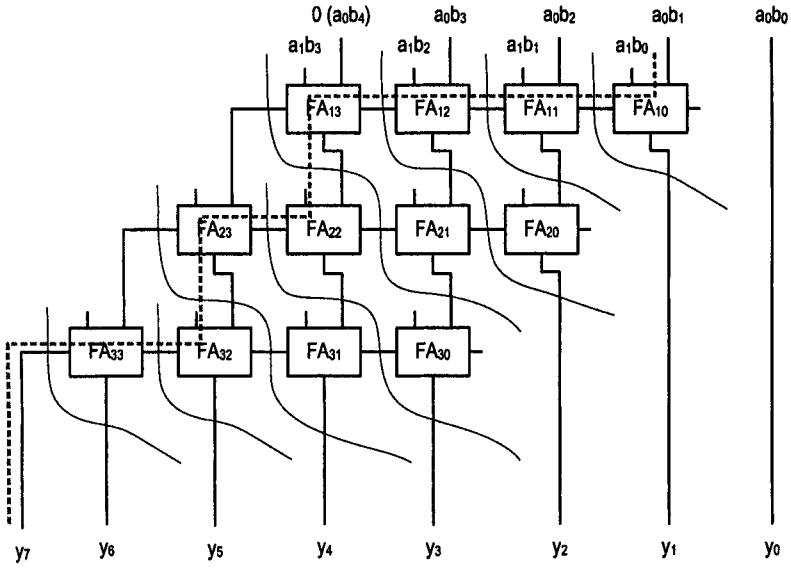


Figure 15.12 Propagation delay contour lines of a carry-ripple multiplier.

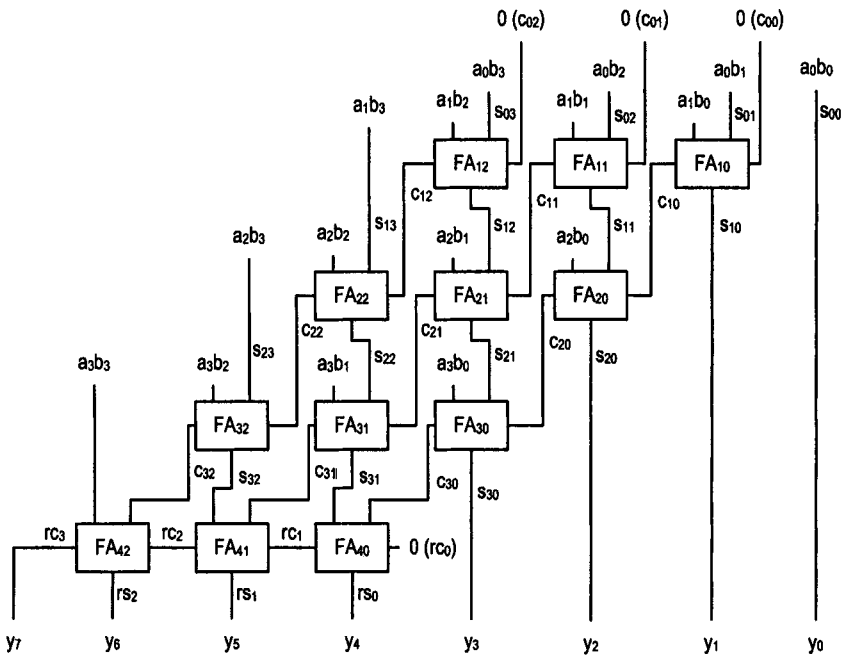


Figure 15.13 Cell-based carry-save multiplier.

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

```
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;
```

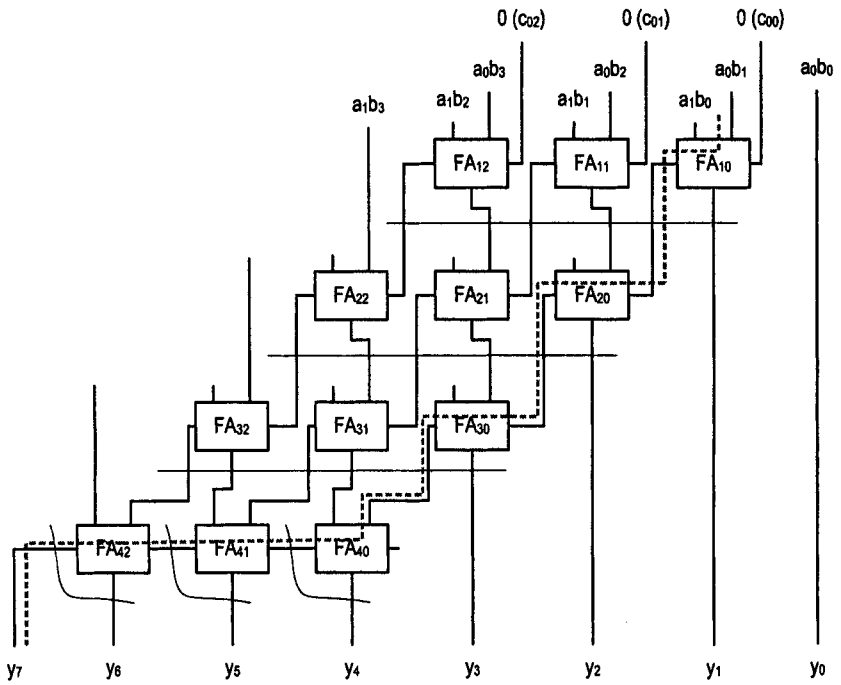


Figure 15.14 Propagation delay contour lines of a carry-save multiplier.

```
    -- output signal
    g_out:
    for i in 0 to (N-1) generate
       y(i) <= s(i)(0);
50  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

$$(N-1)T_{fa} + (N-1)T_{fa} = (2N-2)T_{fa}$$

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.

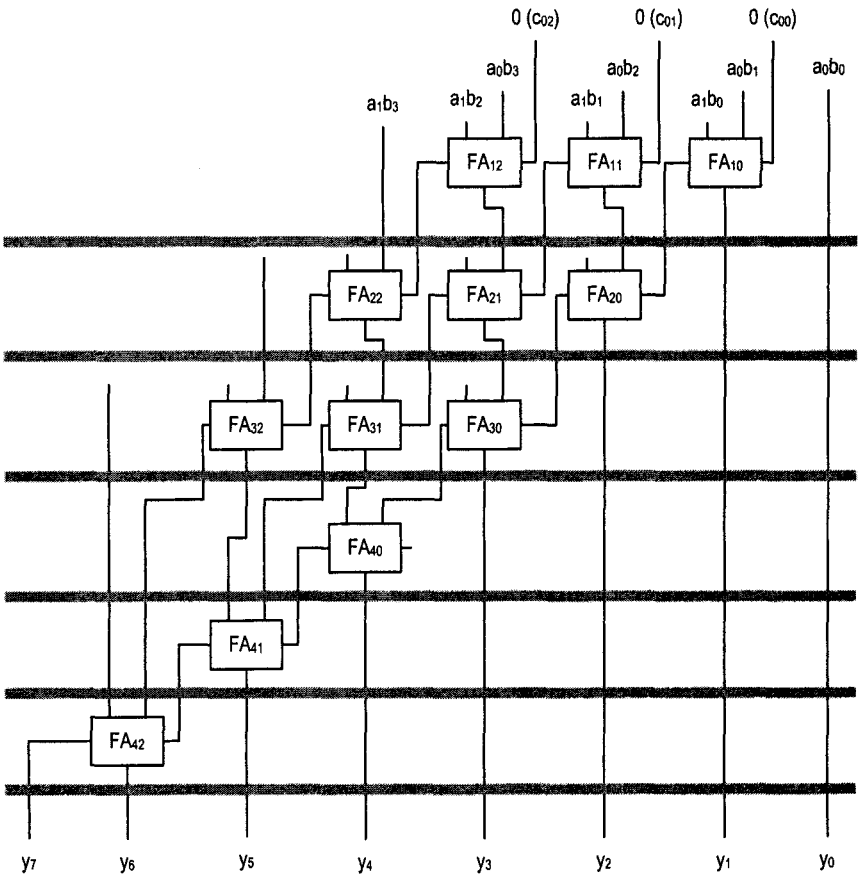


Figure 15.15 Pipelined carry-save multiplier.

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 pervious 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:

```
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 WITH_ZERO, 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

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity lfsr is
   generic(
      N: natural;
      WITH_ZERO: 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)
      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 (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) ;
```

```
    -- output logic
    q <= r_reg;
end para_arch;
```

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 "0010000" 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:

$$\begin{aligned}
t_7 &= a_7 \\
t_6 &= a_6 \cdot a_7' \\
t_5 &= a_5 \cdot a_7' \cdot a_6' \\
t_4 &= a_4 \cdot a_7' \cdot a_6' \cdot a_5' \\
t_3 &= a_3 \cdot a_7' \cdot a_6' \cdot a_5' \cdot a_4' \\
t_2 &= a_2 \cdot a_7' \cdot a_6' \cdot a_5' \cdot a_4' \cdot a_3' \\
t_1 &= a_1 \cdot a_7' \cdot a_6' \cdot a_5' \cdot a_4' \cdot a_3' \cdot a_2' \\
t_0 &= a_0 \cdot a_7' \cdot a_6' \cdot a_5' \cdot a_4' \cdot a_3' \cdot a_2' \cdot a_1'
\end{aligned}$$

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

$$\begin{aligned}
&a_7' \\
&a_7' \cdot a_6' \\
&a_7' \cdot a_6' \cdot a_5' \\
&\cdots \\
&a_7' \cdot a_6' \cdot a_5' \cdot a_4' \cdot a_3' \cdot a_2' \\
&a_7' \cdot a_6' \cdot a_5' \cdot a_4' \cdot a_3' \cdot a_2' \cdot a_1'
\end{aligned}$$

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

```
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.

Listing 15.23 Parameterized priority encoder

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity prio_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 prio_encoder;

architecture para_arch of prio_encoder is
   component 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 component;
   component 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 component;
   signal a_not_rev: std_logic_vector(N-1 downto 0);
   signal a_vec, a_vec_rev, t: std_logic_vector(N-1 downto 0);
begin
   -- reverse a
   gen_reverse_a:
   for i in 0 to (N-1) generate
      a_not_rev(i) <= not a(N-1-i);
   end generate;
   -- reduced and operation
   unit_token: reduced_and_vector
      generic map(N=>N)
      port map(a=>a_not_rev, y =>a_vec_rev);
   -- reverse the result
   gen_reverse_t:
   for i in 0 to (N-1) generate
      a_vec(i) <= a_vec_rev(N-1-i);
   end generate;
   -- form one-hot code
   t <= a and ('1' & a_vec(N-1 downto 1));
   -- regular binary encoder
   unit_bin_code: bin_encoder
```

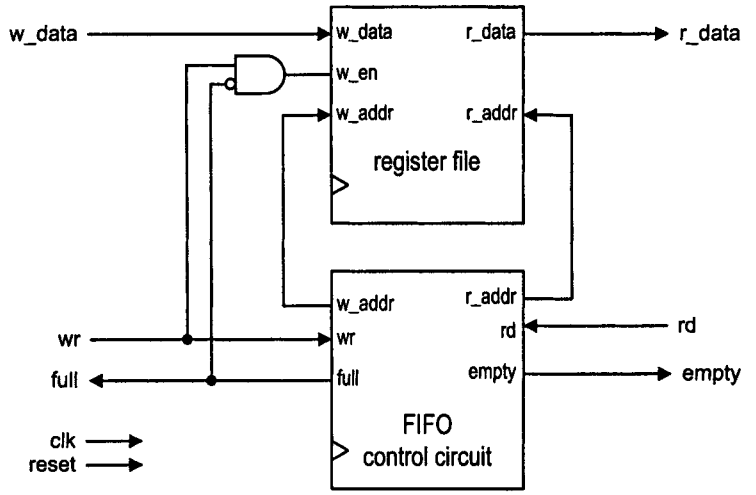


Figure 15.16 Block diagram of a FIFO buffer.

```
        generic map(N=>N)
50      port map(a=>t, bcode=>bcode);
end para_arch;
```

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.

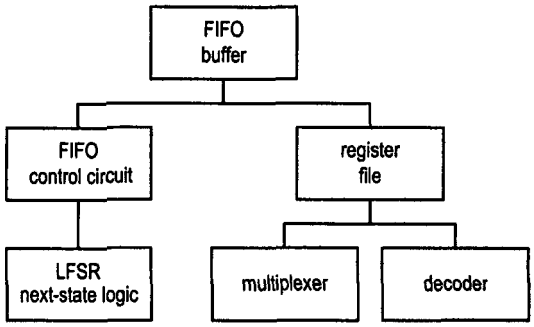


Figure 15.17 Design hierarchy of a FIFO buffer.

Listing 15.24 Parameterized FIFO buffer top-level instantiation

```
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(
```

```
            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(N=>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;
```

Register file 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

```
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 reg_file_para;

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);
begin
  -- 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;
   -- convert to std_logic_2d
   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;
   -- 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='1') 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
   r_data <= array_reg(to_integer(unsigned(r_addr)));
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 15.21 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 15.21. The VHDL code is shown in Listing 15.27.

Listing 15.27 Parameterized LFSR next-state logic

```
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|3|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$\cdots$11" pattern, instead of the "00$\cdots$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

```
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;
   -- statue 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;
```

```
        end case;
      end process;
      -- output
105   w_addr <= w_ptr_reg;
      full <= full_reg;
      r_addr <= r_ptr_reg;
      empty <= empty_reg;
  end lookahead_arch;
```

15.5 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.

15.6 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-2^1 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-2^1 decoder in the leftmost stage if the input of the parameterized decode 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.