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).

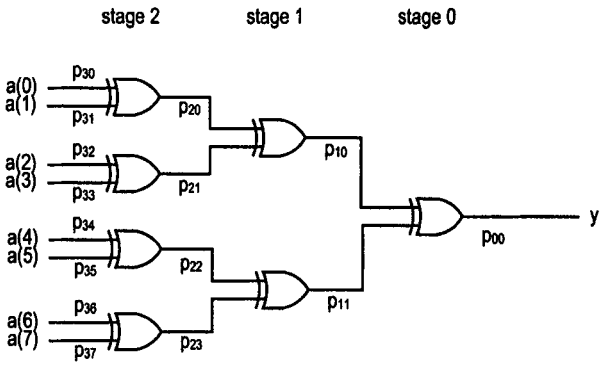


Figure 15.2 Tree-shaped reduced-xor circuit.

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;
```

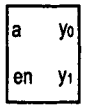
15.3.2 Binary decoder

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:

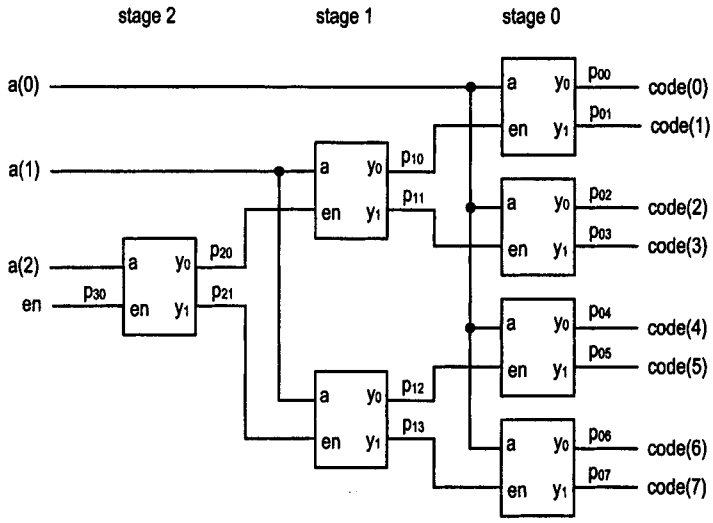
$$y_0 = en \cdot a'$$
$$y_1 = en \cdot a$$

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.

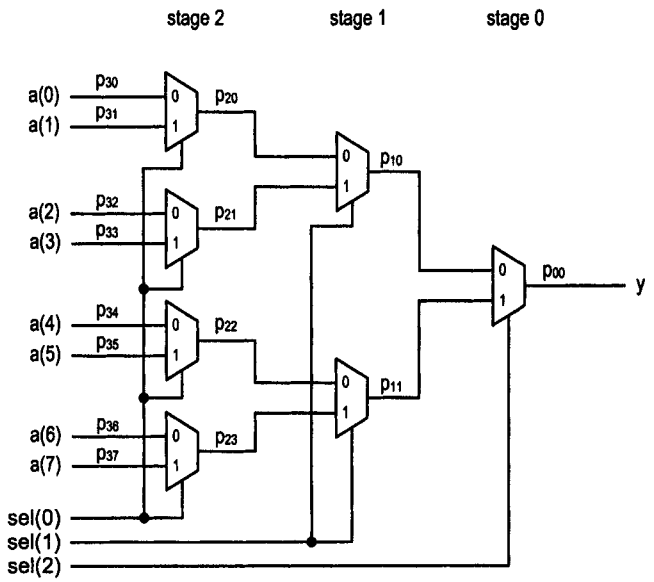


Figure 15.4 Tree-shaped 8-to-1 multiplexer.

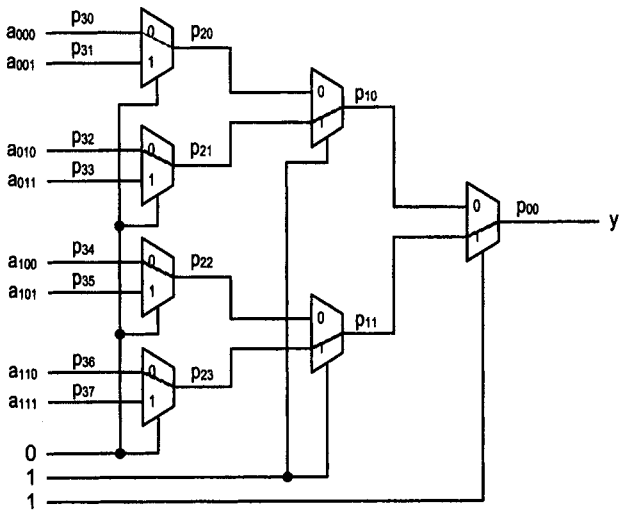


Figure 15.5 Routing with `sel="110"`.

- 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.

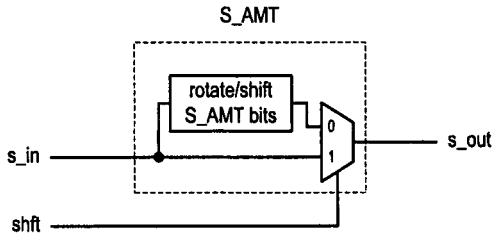
This is an efficient implementation for several reasons. First, as the number of inputs increases, the number of stages grows on the order of $O(\log_2 n)$. The length of the critical path grows in the same order, and thus its performance is much better than the cascading chain. Second, the circuit exhibits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

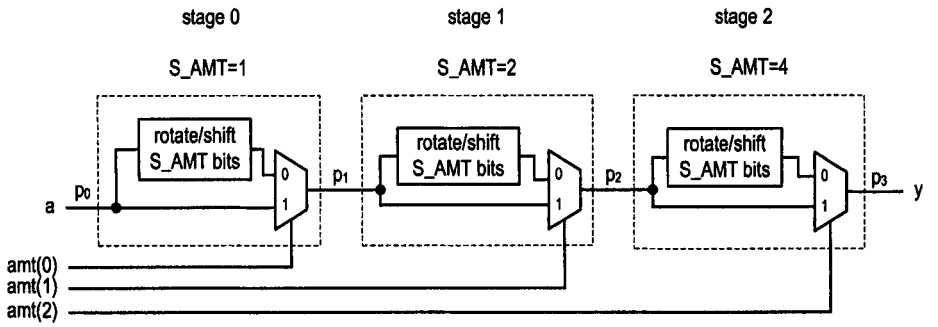
Listing 15.12 Parameterized fixed-size shifting–multiplexing module

```
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.

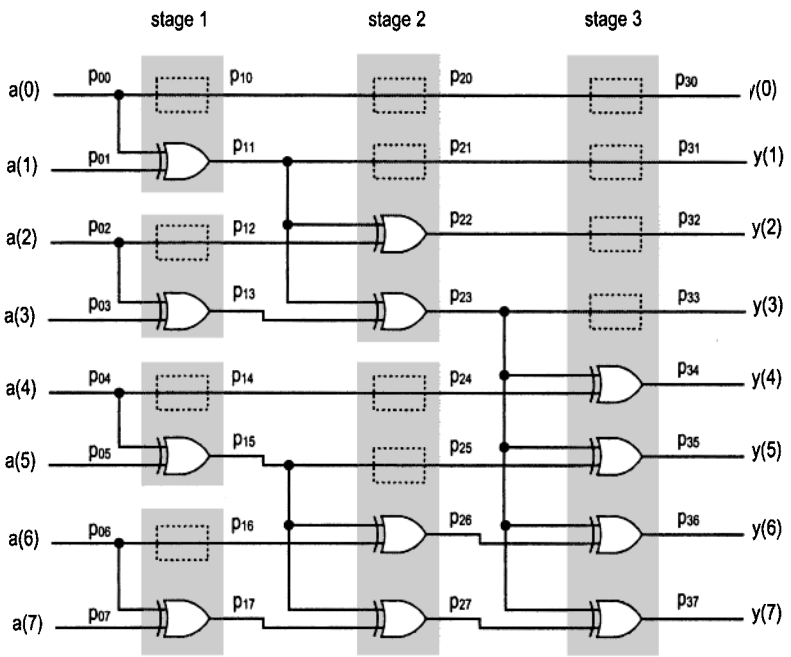


Figure 15.7 Parallel-prefix reduced-xor-vector circuit.

- 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

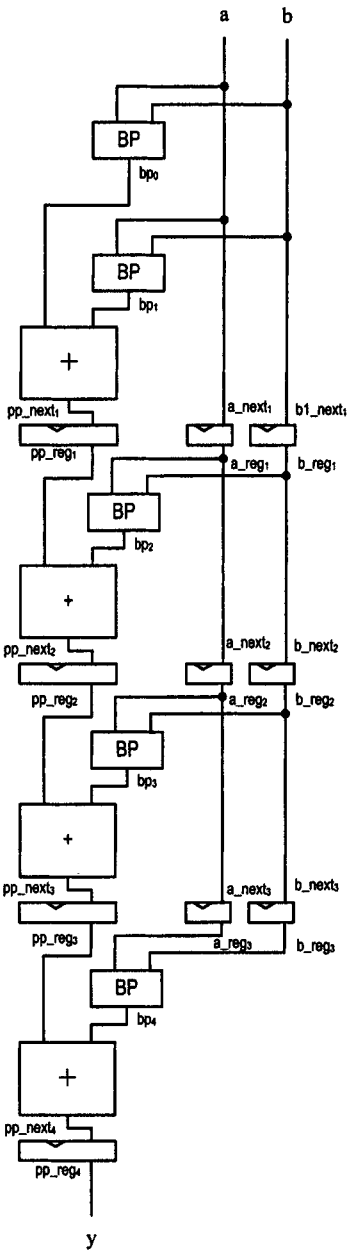


Figure 15.8 Adder-based combinational multiplier with new signal labels.

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.

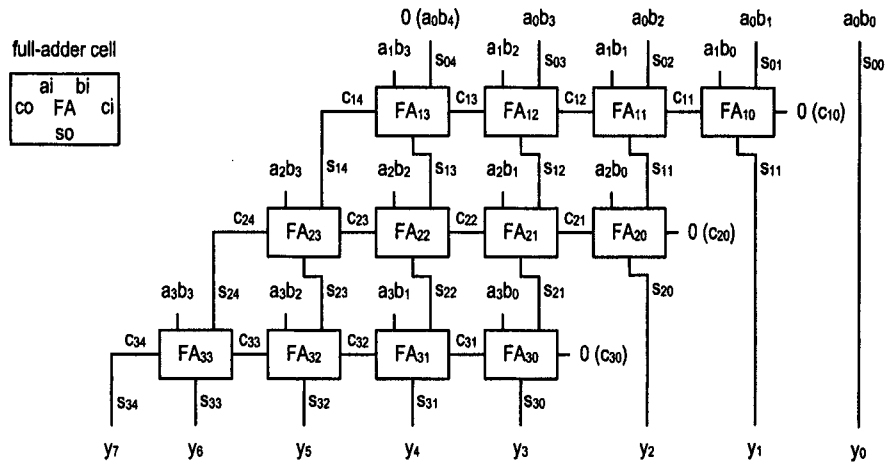


Figure 15.10 Cell-based carry-ripple combinational multiplier.

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.

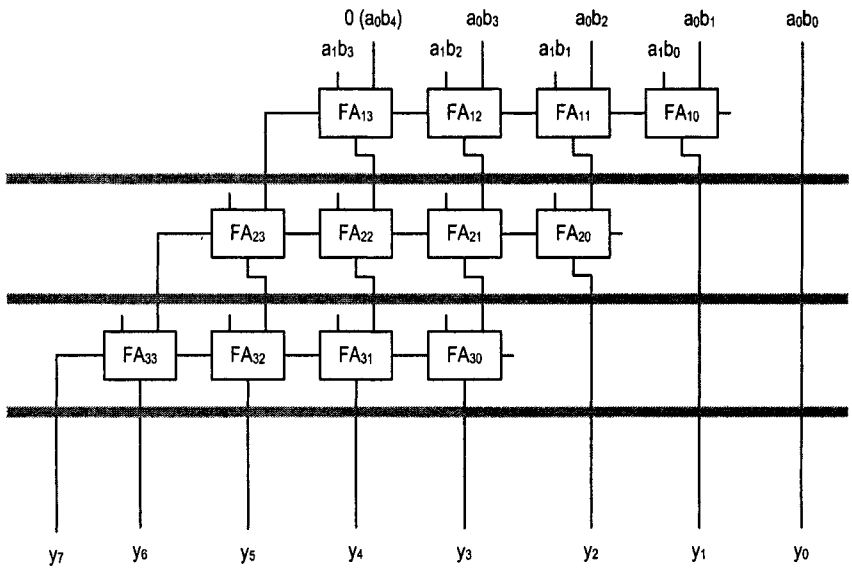


Figure 15.11 Non-optimal pipelined carry-ripple multiplier.

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,

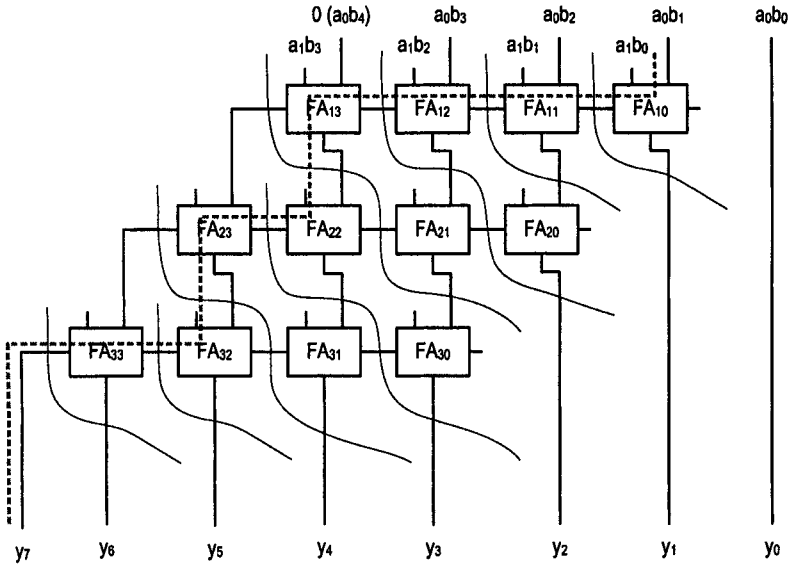


Figure 15.12 Propagation delay contour lines of a carry-ripple multiplier.

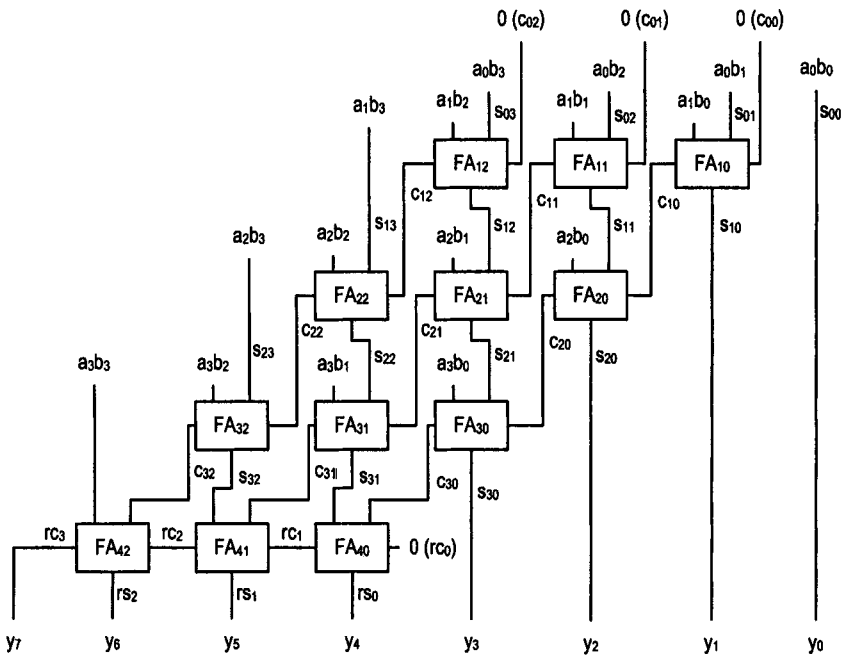


Figure 15.13 Cell-based carry-save multiplier.

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;
```

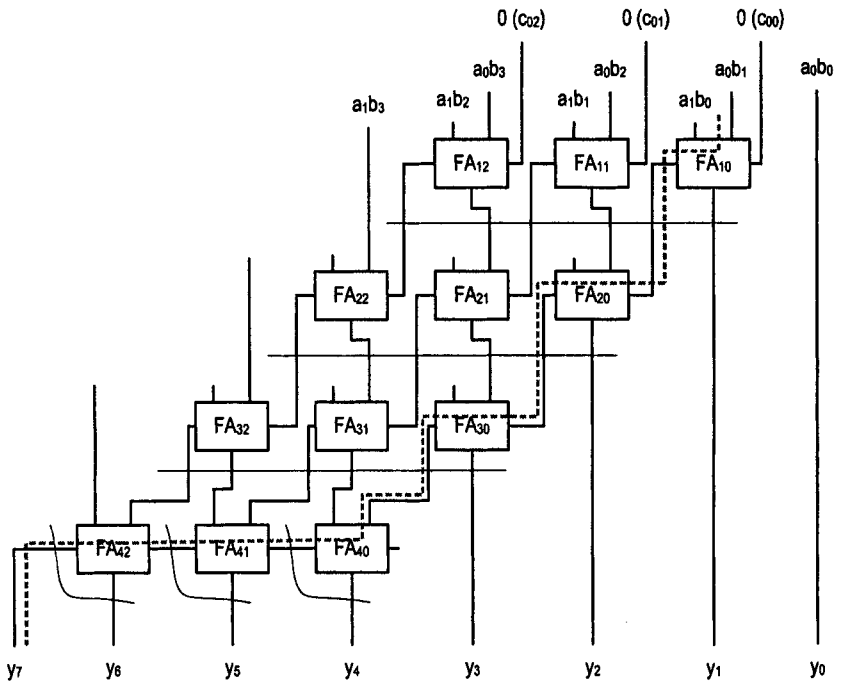


Figure 15.14 Propagation delay contour lines of a carry-save multiplier.

```
    -- 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.

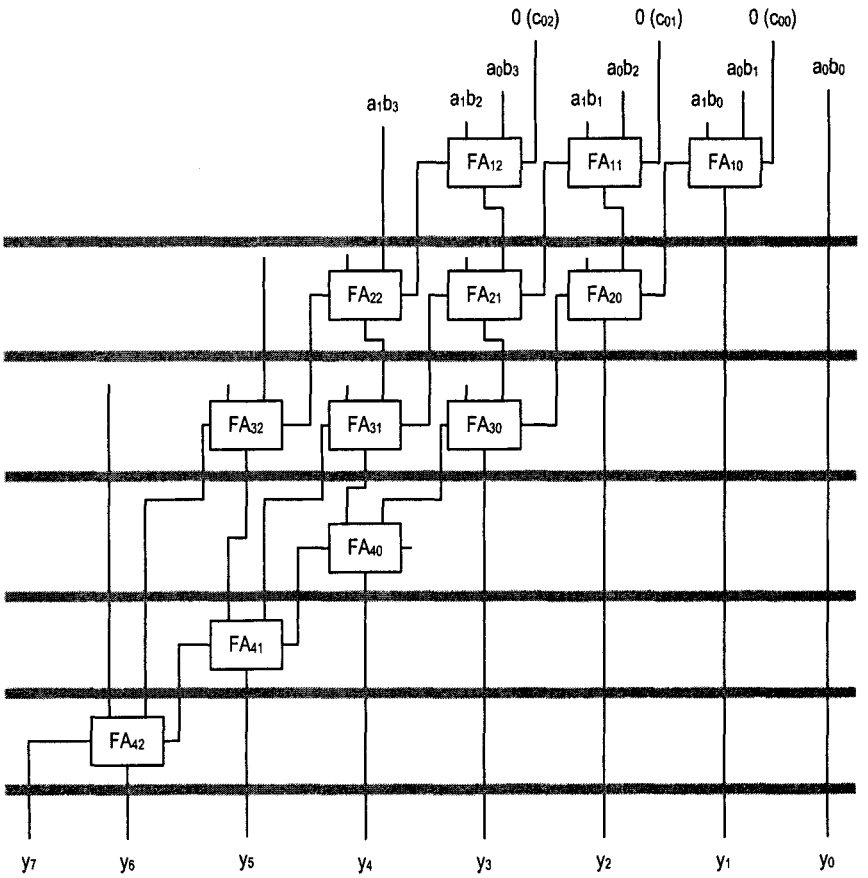


Figure 15.15 Pipelined carry-save multiplier.

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
```

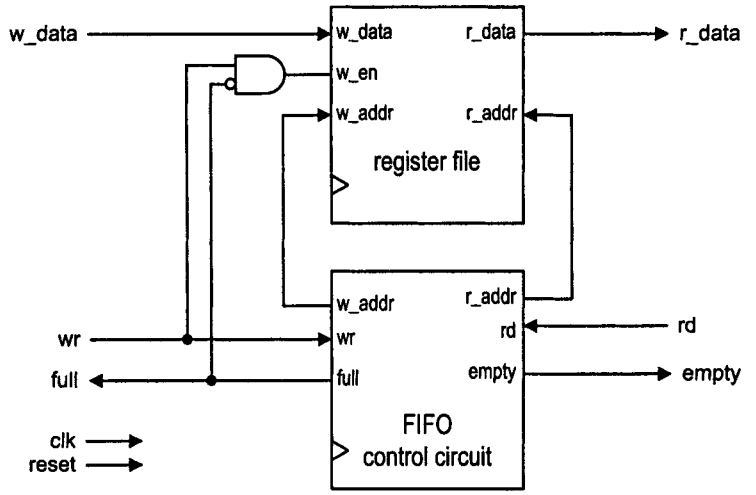


Figure 15.16 Block diagram of a FIFO buffer.

```
        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.

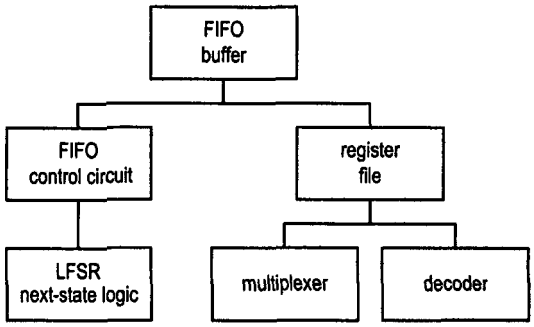


Figure 15.17 Design hierarchy of a FIFO buffer.

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;
```