CHAPTER 11

REGISTER TRANSFER METHODOLOGY: PRINCIPLE

To accomplish a complex task, we frequently describe the process by an *algorithm*, which is a sequence of steps or actions. Algorithms are generally implemented by programs written in a traditional programming language (i.e., by software) and executed in a general-purpose computer. However, to obtain better performance and efficiency, it is sometimes beneficial or even necessary to realize an algorithm in custom hardware. The *register transfer methodology* (*RT methodology*) is a design methodology that describes system operation by a sequence of data transfers and manipulations among the registers. This methodology can support the variables and sequential execution of an algorithm and provide a systematic way to convert an algorithm into hardware.

11.1 INTRODUCTION

11.1.1 Algorithm

An algorithm is a detailed sequence of actions or steps to accomplish a task or to solve a problem. Since the semantics of traditional programming languages is also based on sequential execution, an algorithm can easily be converted into a program using the constructs of these languages. The program is then compiled into the machine instructions and executed in a general-purpose computer. Let us consider a simple task that sums the four elements of an array, divides the sum by 8 and rounds the result to the closest integer. The pseudocode of one possible algorithm is

```
size = 4
sum = 0;
for i in (0 to size-1) do {
   sum = sum + a(i);}
q = sum / 8;
r = sum rem 8;
if (r > 3) {
   q = q + 1;}
outp = q;
```

The algorithm first adds individual elements and stores the result in a variable called sum. It then uses the division (/) and remainder (**rem**) operations to find the quotient and remainder. If the remainder is greater than 3, an extra 1 is added to quotient for rounding. The example demonstrates two basic characteristics of an algorithm:

- *Use of variables.* A variable in an algorithm or pseudocode can be interpreted as a "memory location with a symbolic address" (i.e., the name of the variable). It is used to store an intermediate computation result. For example, in the second statement, 0 is stored into the memory location with a symbolic address of sum. Inside the for loop, a(i) is added with the current content of sum, and then summation is stored back into the same memory location. In the fourth statement, the content of sum is divided by 8, and the result is stored into a memory location with a symbolic address of q.
- *Sequential execution.* The execution of an algorithm is performed sequentially and the order of the steps is important. For example, the summation of the elements must be obtained before the division operation can be performed. Note that the order of execution may rely on certain conditions, as in the for loop and if statements.

In VHDL, the variables and sequential execution are treated as a special case and encapsulated inside a process. Although a description with variables can be synthesized in some cases, the variables are mapped to signals and are not interpreted or realized as "memory locations with symbolic addresses."

11.1.2 Structural data flow implementation

To achieve better performance and efficiency, we frequently want to implement an algorithm in custom hardware. The variable and sequential semantics of algorithm are very different from the concurrent model of hardware. What we have learned so far is to transform "sequential execution" into "structural data flow" by mapping an algorithm into a system of cascading hardware blocks, in which each block represents a statement in the algorithm. For example, we can unroll the loop of the previous algorithm and convert the variables into internal connection signals. Assume that sum is an 8-bit signal. The corresponding VHDL code becomes

```
sum <= 0;
sum0 <= a(0);
sum1 <= sum0 + a(1);
sum2 <= sum1 + a(2);
sum3 <= sum2 + a(3);
q <= "000" & sum3(8 downto 3);
r <= "00000" & sum3(2 downto 0);
outp <= q + 1 when (r > 3) else
        q;
```

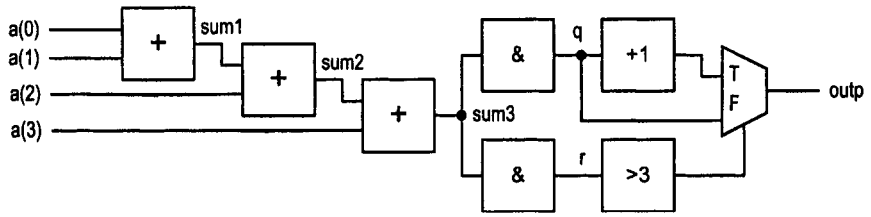


Figure 11.1 Structural data flow implementation.

Note that the `sum / 8` and `sum` **rem** `8` operations are implemented by concatenation (i.e., &) operations. The corresponding block diagram is shown in Figure 11.1.

Although the circuit can carry out the task, the operation of the hardware is very different from the sequential semantics of the original algorithm. In this construction, the circuit is a pure combinational logic, and the adders and dividers (concatenation operators) execute in parallel. The implementation does not use any concept of variable, and the sequential execution is implicitly embedded in the interconnection of components and the flow of data. To some degree, the synthesis essentially utilizes extra hardware to accelerate the operation. Instead of using a single arithmetic unit of a computer to perform these operations sequentially, the custom hardware utilizes multiple adders and division circuits to calculate the result in parallel.

The structural data flow implementation is not general and can be applied only for simple, trivial algorithms. The following two variations of the previous algorithm illustrate the limitation of this approach. First, let us consider an array with 10 elements. In the pseudocode, this can be done by replacing 4 with 10 in the first statement. This increases the number of loop iterations. We can unroll the loop and derive the structural data flow implementation, which needs nine adders. If the number of elements of the array continues to grow, the number of adders increases accordingly. Clearly, this approach needs excessive hardware resource and is not practical for a larger array. Second, let us assume that the size of the array is not fixed but is specified by an additional input, `n`. To accomplish this in the algorithm, we only need to substitute `n` into the first statement and make it `size = n`. This will be very difficult for structural data flow implementation. Since the hardware cannot expand or shrink dynamically, we have to construct a circuit that can calculate the results for *all* possible values of `n` and then use a multiplexer to route the desired value to output. The resulting hardware will be extremely complicated, and this approach is not practical in reality.

11.1.3 Register transfer methodology

The previous example shows the limitation and inflexibility of structural data flow implementation. To realize an algorithm in hardware, we need hardware constructs that resemble the variable and sequential execution model. The *register transfer methodology* (*RT methodology*) is aimed for this purpose. The key characteristics of this methodology are:

- Use registers to store the intermediate data and to imitate the variables used in an algorithm.
- Use a custom *data path* to realize all the required register operations.
- Use a custom *control path* to specify the order of the register operations.

We have utilized registers for regular sequential circuits and FSMs in previous chapters. They are usually dedicated to a specific circuit, as in a counter or an FSM. In the RT methodology, the registers are used as general storage that keeps the intermediate computed values, just as the variables of an algorithm. For example, consider a typical statement in pseudocode:

```
a = a + b
```

We can use two registers, `a_reg` and `b_reg`, to imitate the `a` and `b` variables. When this statement is executed, the content of the `a_reg` and `b_reg` registers will be added, and the result will be stored back into the `a_reg` register at the next rising edge of the clock.

When an algorithm is realized in RT methodology, the necessary data manipulation and data routing are performed by dedicated hardware. For example, an adder is required for the previous statement. The data manipulation circuit, routing network and the registers together are known as the *data path*.

Since an algorithm is described as a sequence of actions, we need a circuit to control *when* and *what* RT operations should take place. The circuit is known as the *control path*. A control path can be realized by an FSM, which can use states to enforce the order of the desired steps and use the decision boxes to imitate the branches and iterations (loops) in an algorithm.

We call this implementation methodology *register transfer methodology* since an algorithm is transformed into a sequence of actions that specifies how the data is manipulated and transferred among registers. A typical RT implementation includes a data path and a control path. We can use an extended FSM to describe the overall system operation. It is known as *FSM with a data path (FSMD)*.

As we mentioned in Section 1.4.3, use of the term *register transfer* is somewhat abused. It sometimes is used rather vaguely to represent a level of abstraction (i.e., the RT level) between the gate and processor levels. In this book, we use the term *RT methodology* for this specific design methodology and the term *RT level* for module-level abstraction.

11.2 OVERVIEW OF FSMD

An FSMD is the key to realizing the RT methodology. This section provides an overview of FSMD, including RT operation, data path, control path and extended ASM chart. The subsequent sections use examples to illustrate the detailed derivation and construction of an FSMD.

11.2.1 Basic RT operation

A basic action in RT methodology is a *register transfer operation*. We use the following notation for an RT operation:

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

In this notation, the register on the left-hand side (i.e., r_{dest}) is the destination register. The registers on the right-hand side (i.e., r_{src1}, r_{src2} and r_{srcn}) are the source registers and they represent the outputs (i.e., the contents) of these registers. The $f(\cdot)$ function is the operation to be performed. It is an expression composed of source registers and sometimes external inputs. The overall notation means that the new value of r_{dest} is calculated according to $f(r_{src1}, r_{src2}, \ldots, r_{srcn})$, and the result will be stored into r_{dest} at the next rising edge of

the clock. Note that the $\leftarrow$ notation is not defined in VHDL. It is only used in this book to denote the register transfer operation.

There is no specific restriction on the $f(\cdot)$ function. It can be any expression as long as it can be realized by a combinational circuit. A few representative RT operations are shown below.

- `r` $\leftarrow$ 1: A constant 1 is stored into the `r` register.
- `r` $\leftarrow$ `r`: The content of the `r` register is stored back into itself. The content, of course, remains unchanged.
- `r` $\leftarrow$ `r << 3`: The content of the `r` register is shifted left three positions and then stored back into itself.
- `r0` $\leftarrow$ `r1`: The content of the `r1` register is stored (or transferred) into the `r0` register.
- `n` $\leftarrow$ `n` $-$ 1: The content of the `n` register is decremented by 1 and the result is stored back into itself.
- `y` $\leftarrow$ `a` $\dot{\oplus}$ `b` $\oplus$ `c` $\oplus$ `d`: The contents of the `a`, `b`, `c` and `d` registers are xored and the result is stored into the `y` register.
- `s` $\leftarrow$ $\mathtt{a}^2 + \mathtt{b}^2$: The summation of `a` squared and `b` squared is stored into the `s` register. We can write this expression only if the predesigned combinational multiplier module is available.

The major difference between a variable of an algorithm and a register is that a *system clock* is embedded implicitly in an RT operation. Consider the register operation

$$r_{dest} \leftarrow f(r_{src1}, r_{src2}, \ldots, r_{srcn})$$

Its detailed actions are las follows:

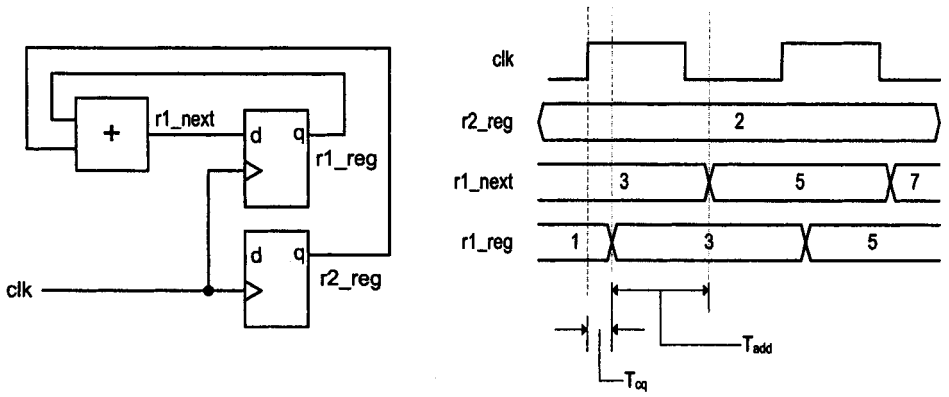
1. At the rising edge of the clock, new data from the source registers is available after the clock-to-q delay of the source registers.
2. The data is computed by a combinational circuit that realizes the $f(\cdot)$ function. We assume that the clock period is long enough to accommodate the propagation delay of the combinational circuit and the setup time of the r_{dest} register. The result is routed to the input of the r_{dest} register.
3. At the next rising edge of the clock, the result will be sampled and stored into the r_{dest} register.

In our discussion of sequential circuits, we use the suffixes `_reg` and `_next` for the current output and next input of a register. A more accurate description of an RT operation can be expressed using these suffixes. For example, consider the `r1` $\leftarrow$ `r1 + r2` operation. It actually means

- `r1_next <= r1_reg + r2_reg;`
- `r1_reg <= r1_next` at the rising edge of the clock;

Note that the `<=` notation is used for regular signal assignment.

Realizing an RT operation is straightforward. We basically construct the $f(\cdot)$ function using combinational components and then connect its output to the input of the destination register. Again, consider the `r1` $\leftarrow$ `r1 + r2` operation. It involves an addition in $f(\cdot)$. Its block diagram is shown in Figure 11.2(a) and the corresponding timing diagrams is shown in Figure 11.2(b). Note that the `r1` register will not be updated until the next rising edge of the clock.



(a) Block diagram (b) Timing diagram

Figure 11.2 Single RT operation.

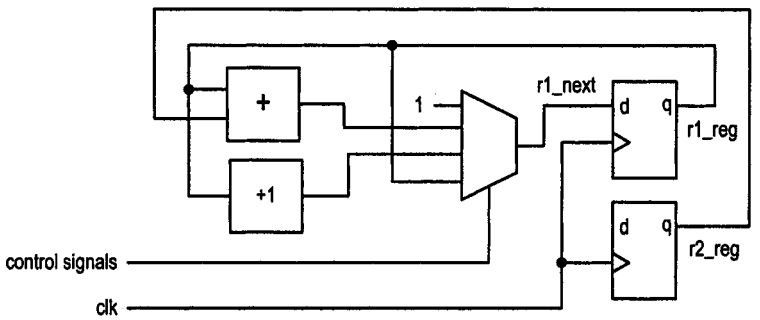


Figure 11.3 Block diagram of a set of RT operations with the same destination register.

11.2.2 Multiple RT operations and data path

An algorithm consists of many steps, and a destination register is not loaded with the same data in these steps. For example, the `r1` register may be set to 1 in the initialization step, added with the content of `r2` in a summation step, incremented in the two counting steps, and kept unchanged in the final step. Thus, four RT operations use `r1` as the destination register:

- `r1` ← 1;
- `r1` ← `r1` + `r2`;
- `r1` ← `r1` + 1;
- `r1` ← `r1`;

Because of the multiple possibilities, a multiplexing circuit is needed to route the desired value to the input of the `r1` register. The block diagram is shown in Figure 11.3. We can choose the desired RT operation by setting the proper selection signal in the multiplexing circuit.

A design with RT methodology normally involves many registers. We can repeat this procedure for every register. The resulting circuit constitutes the basic, unoptimized data path, which can perform every needed RT operation of an algorithm.

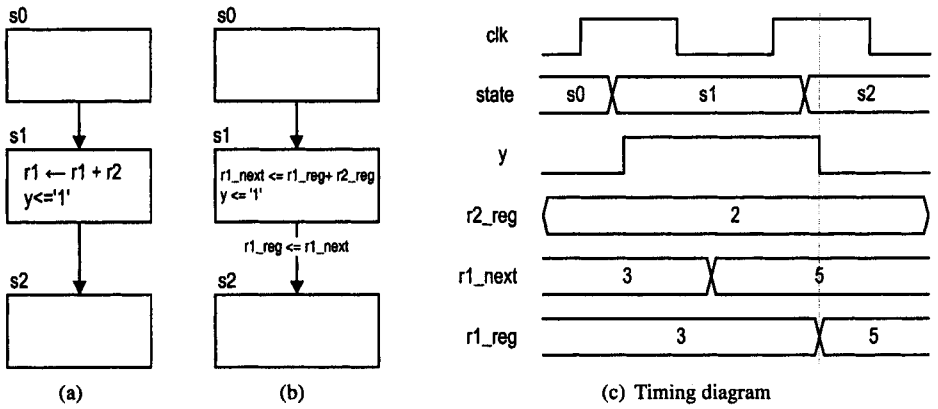


Figure 11.4 RT operation in a segment of an ASMD chart.

11.2.3 FSM as the control path

While a data path realizes all required RT operations in an algorithm, we need a mechanism to specify when and which RT operations should be performed. A control path is used to enforce the order of RT operations and to selectively perform certain RT operations based on the external commands or internal status. A control path can be realized by a custom FSM. An FSM is a natural match for this task for several reasons:

- The state transition of an FSM is performed on a clock-by-clock basis. Since an RT operation is also updated on a clock-by-clock basis, an RT operation can be specified in a state of the FSM.
- An FSM can enforce a specific sequence of actions.
- Upon an examination of input conditions, an FSM can branch to different paths and thus can alter the sequence of actions. This can be used to implement various branch constructs, such as the if and loop statements, in an algorithm.

11.2.4 ASMD chart

Since an RT operation is performed in a state of the FSM, we can extend the FSM to FSMD to indicate the desired RT operation in each state. The state representation and state transition of an FSMD are similar to those of an FSM. However, RT operations, in addition to output signals, are specified in states or transition arcs. We use an extended ASM chart, in which an RT operation can be specified either inside a state box or in a conditional output box, to describe the operation of an FSMD. It is known as an *ASM with a data path chart* (*ASMD chart*).

The construction and operation of an ASMD chart can best be explained by an example. A segment of an ASMD is shown in Figure 11.4(a). An RT operation, `r1` ← `r1 + r2`, is specified in the `s1` state. For comparison purposes, we also include a regular activated output, `y`, in the `s1` state. When the FSMD enters the `s1` state, the `r1 + r2` expression is calculated and its result becomes the next value of `r1`. At the next rising edge of the clock, the FSMD transits from `s1` to `s2` and `r1` is updated with the new value. Note that the `r1` register is not updated inside the state box but during the transition between the `s1` and `s2` states. The new value of `r1` is available only when the FSMD reaches the `s2` state.

A clumsy, but more accurate, notation is shown in Figure 11.4(b), in which the `r1_next` signal is calculated in the `s1` state, independent of the clock edge, and the `r1_reg` signal is updated at the transition. Note that the regular signal assignment notation, <=, is used in the diagram.

The timing diagram is shown in Figure 11.4(c). When the FSMD enters the `s1` state, the computation of `r1 + r2` starts but the output of `r1` remains unchanged. Note that the regular output, y, is activated after the clock-to-q delay in the `s1` state. At the next rising edge of the clock, the FSMD moves to the `s2` state and the new value is sampled and stored into `r1`. After the clock-to-q delay, the new value is propagated to the output of `r1`. Note that the y signal is deactivated in the `s2` state.

The `r1` register samples and stores the input data at every rising edge of the clock. Thus, `r1` is updated in the `s0` and `s2` states as well, even when no operation is needed. Since the system is synchronous, a register cannot be disabled or suspended. Instead, it just keeps its old value by sampling its own output; i.e., performing the `r1` ← `r1` operation. To reduce the clutter, we don't include this operation in an ASMD chart. If a destination register `r` is not associated with an RT operation in a state, we assume that it performs the default `r` ← `r` operation.

Although the appearances of an ASMD chart and a regular flowchart are somewhat similar, their operations are different. The operation of an FSMD is operated on a clock-by-clock basis. The register in an ASMD chart is not updated until the exit of the current state, and thus an RT operation exhibits some sort of *delayed-store* behavior. The most error-prone part of deriving an ASMD chart is this delayed-store operation. To obtain a correct and efficient ASMD chart, we need to have a clear understanding of the timing of an RT operation and know when a register is updated. Section 11.3.4 provides a comprehensive discussion of this issue.

11.2.5 Basic FSMD block diagram

The conceptual block diagram of an FSMD is shown in Figure 11.5. It is divided into a data path and a control path. The data path can perform all the required RT operations and is composed of three major parts:

- *Data registers.* The registers store the intermediate computation results.
- *Functional units.* The functional units perform the functions specified by RT operations. Typical functional units include an adder, subtractor, incrementor, decrementor and shifter.
- *Routing circuit.* The circuit routes the source registers' outputs to the proper functional units and routes the calculated results from the functional units to proper destination registers. It is normally constructed by customized multiplexers.

A data path normally includes the following input and output signals:

- `data input`: the external input data, which is to be processed by the FSMD.
- `data output`: the processed results of the FSMD.
- `control signal`: input signal used to specify which RT operations should be performed. It is generated by the control path.
- `internal status`: output signal indicating certain conditions of the data path, such as whether a specific register is 0. This signal is used by the control path to determine the future course of action.

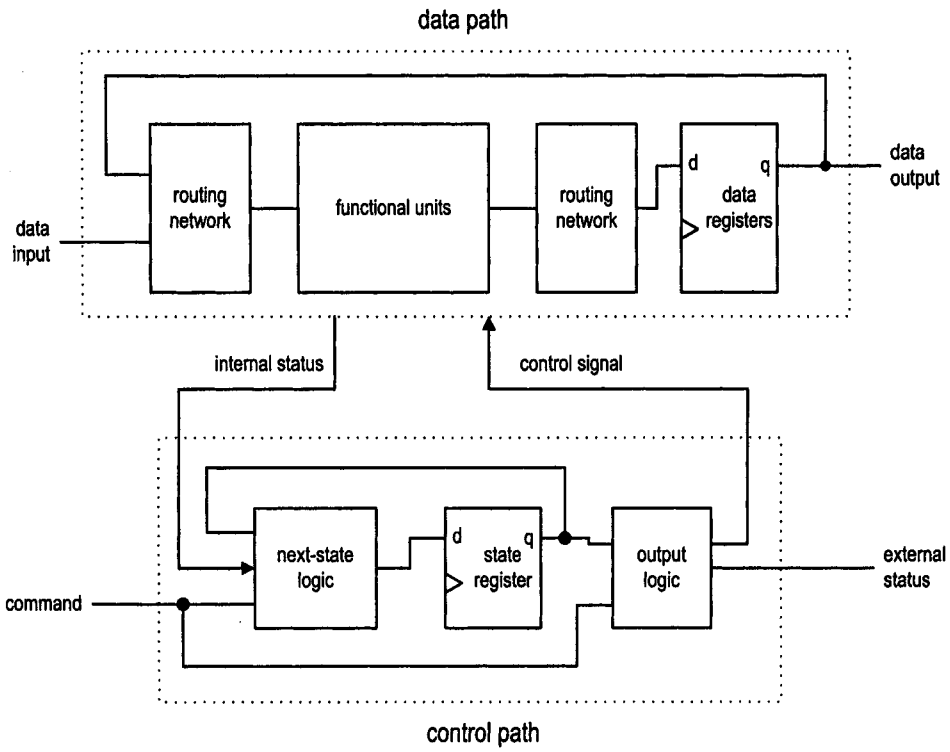


Figure 11.5 Basic block diagram of an FSMD.

The control path is an FSM. As a regular FSM, it contains a state register, next-state logic and output logic. A control path normally includes the following input and output signals:

- `command`: the external command signal to the FSMD, such as the start of the operation. It is an input to the FSM.
- `internal status`: signal from the data path, which is also an input to the FSM. The FSM uses it and the external command to determine the next state.
- `control signal`: output of the FSM used to control data path operation.
- `external status`: output of the FSM used to indicate the status of the FSMD operation, such as whether the system is busy.

In addition to these signals, the registers of the data path and control path are connected to the same clock signal and to an optional asynchronous reset signal.

Note that the data path resembles a regular sequential circuit, and the control path is an FSM and thus is a random sequential circuit. Therefore, an FSMD can be considered a combined sequential circuit, as discussed in Section 8.2.3. Although the FSMD consists of two types of sequential circuits, both circuits are synchronized by the same clock, and thus the FSMD still follows the same synchronous design methodology.

11.3 FSMD DESIGN OF A REPETITIVE-ADDITION MULTIPLIER

The derivation of an ASMD chart and the construction of an FSMD can best be explained by closely examining several examples. This section illustrates how to convert a simple repetitive-addition multiplication algorithm into an ASMD chart and realize it in hardware. Various alternatives are discussed in subsequent sections.

11.3.1 Converting an algorithm to an ASMD chart

We learned to implement a combinational multiplier in Section 7.5.4. The design utilized multiple adders and is somewhat like the data-flow implementation of Section 11.1.2. An alternative is to use one adder to perform the additions sequentially. Assume that the two operands of the multiplication are `a_in` and `b_in`. One simple sequential algorithm is to add `a_in` repetitively for `b_in` times. For example, 7*5 can be computed as $7+7+7+7+7$. While this method is not efficient, it is simple and we can concentrate on the derivation of the ASMD chart and hardware.

Consider a multiplier with input `a_in` and `b_in`, and with output `r_out`. All three signals are in unsigned integer format. The repetitive-addition algorithm can be formalized in the following pseudocode:

```
if (a_in=0 or b_in=0) then {
   r = 0;}
else{
   a = a_in;
   n = b_in;
   r = 0;
   while (n != 0 ){
      r = r + a;
      n = n - 1;}
}
r_out = r;
```

Note that the ASMD chart does not have a loop construct. Its decision box uses a Boolean condition to choose one of two possible exit paths and thus is somewhat like a combined if and goto statement. To make it closer to an ASMD chart, we convert the while loop using an if statement and two goto statements. The revised pseudocode becomes

```
        if (a_in=0 or b_in=0) then {
           r = 0;}
        else{
           a = a_in;
           n = b_in;
           r = 0;
op:        r = r + a;
           n = n - 1;
           if (n = 0) then{
              goto stop;}
           else{
              goto op;}
        }
stop:   r_out = r;
```

To realize this algorithm in hardware, we must first define its input and output signals. The input signals are:

- a_in and b_in: input operands. They are 8-bit signals with the std_logic_vector data type and interpreted as unsigned integers.
- start: command. The multiplier starts operation when the start signal is activated.
- clk: system clock.
- reset: asynchronous reset signal for system initialization.

The output signals are:

- r_out: the product. It is a 16-bit signal with the std_logic_vector data type and interpreted as an unsigned integer.
- ready: external status signal. It is asserted when the multiplication circuit is idle and ready to accept new inputs. It can also be interpreted that the previous operation has been completed and the result is ready.

Note that the start and ready signals are added to accommodate sequential operation. We can imagine that the sequential multiplier is part of a large system. When the main system wants to do a multiplication operation, it first checks the ready signal and then places the two operands on the two data inputs and asserts the start signal. When the start signal is activated, the sequential multiplier takes the two data inputs and begins computation. It activates the ready signal to inform the main system once the computation has been completed.

The ASMD chart is shown in Figure 11.6. It closely follows the pseudo algorithm. It uses n, a and r data registers to imitate the three variables, uses decision boxes to implement the two if statements, and uses RT operations to realize regular sequential statements.

Unlike the pseudocode, in which one statement is executed at a time, the ASMD chart allows some degree of parallelism. When the RT operations are scheduled in the same state, it means that they are performed in the same clock cycle and thus are done in parallel. For example, both r ← r + a and n ← n - 1 operations are scheduled in the op state. This implies that there are an adder and a decrementor in the physical circuit and that two calculations can be performed simultaneously. In general, we can schedule RT operations in the same state (i.e., the same clock cycle) as long as there is no data dependency, and enough hardware resources are available.

There are four states in the ASMD chart. The idle state indicates that the circuit is currently idle. The ready signal is asserted accordingly. If the start signal is asserted, the FSMD checks whether one of the inputs is zero and branches to the ab0 or load state. In the ab0 state, r is assigned to 0 and the FSMD returns to the idle state. Although not required, we assume that a and n are loaded with a_in and b_in. In the load state, r is initialized to 0, and a and n are loaded with the external input values. The FSMD then enters the loop and iterates through the op (for "operation") state b_in times. In each iteration, it adds the content of a to r and decrements n by 1. The n register is used to keep track of the number of operations. The loop stops when it reaches 0 and the FSMD returns to the idle state. We intentionally use a vague Boolean expression, count_0=1, inside the decision box. This is elaborated in Section 11.3.4. As in an ASM chart, we use a dashed box to represent an ASMD block and to emphasize that all operations inside the block are done in parallel at the same clock cycle.

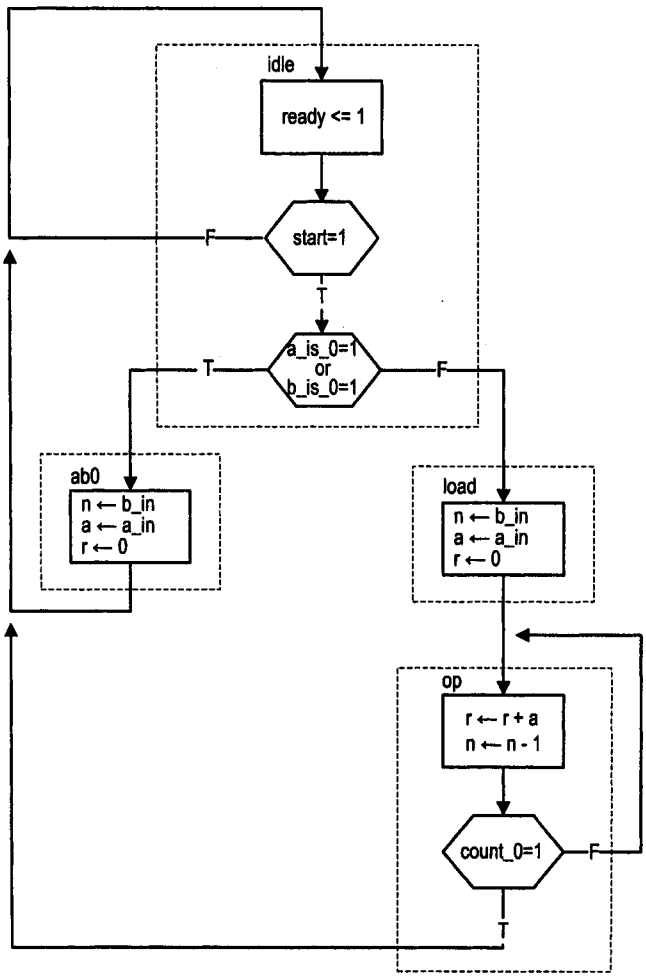


Figure 11.6 ASMD chart of a repetitive-addition multiplier.

11.3.2 Construction of the FSMD

Once the ASMD chart is constructed, more detailed information is available. We can refine the basic sketch of Figure 11.5 and derive a more detailed conceptual block diagram. We first divide the system into a control path and a data path.

The construction of the control path is the same as with the FSM. Recall that the signals inside the decision box constitute the input of the FSM. In the ASMD chart, the Boolean expressions use four signals: `start`, `a_is_0`, `b_is_0` and `count_0`. The `start` signal is the external command, and the other three are internal status signals from the data path. They are asserted when the corresponding conditions are met. The output of the control path includes the external `ready` status signal and the control signals that specify the RT operations of the data path. In this example, we use the output of the state register as the control signal. The block diagram is shown at the bottom of Figure 11.8.

At first glance, construction of the data path seems to be more involved. However, it can be derived systematically by following simple guidelines. The basic data path can be constructed as follows:

1. List all possible RT operations in the ASMD chart.
2. Group RT operations according to their destination registers.
3. For each group, derive the circuit following the process of Section 11.2.2:
   (a) Construct the destination register.
   (b) Construct the combinational circuits involved in each RT operation.
   (c) Add multiplexing and routing circuits if the destination register is associated with multiple RT operations.
4. Add the necessary circuits to generate the status signals.

The RT operations of the repetitive-addition multiplication ASMD are grouped as follows:

- RT operations with the `r` register:
  - `r` ← `r` (in the `idle` state)
  - `r` ← 0 (in the `load` and `ab0` states)
  - `r` ← `r + a` (in the `op` state)
- RT operations with the `n` register:
  - `n` ← `n` (in the `idle` state)
  - `n` ← `b_in` (in the `load` and `ab0` states)
  - `n` ← `n - 1` (in the `op` state)
- RT operations with the `a` register:
  - `a` ← `a` (in the `idle` and `op` states)
  - `a` ← `a_in` (in the `load` and `ab0` states)

Note that we must include the default RT operations for the three registers.

Let us consider the circuit associated with the `r` register. The conceptual diagram is shown in Figure 11.7. It has three possible sources for the input: 0, `r` and `r + a`. The routing of the next value is done by an abstract multiplexer, as the one discussed in Section 4.3.2. It uses the output of the state register as the select signal, and its input ports are labeled with the four possible symbolic values. The connection indicates that `r_reg` is routed to `r_next` if the `state_reg` signal is `idle`, 0 is routed to `r_next` if the `state_reg` signal is `ab0` or `load`, and `r_reg + a_reg` is routed to `r_next` if the `state_reg` signal is `op`.

We can repeat the process for two other registers and use three comparators to implement the three status signals. The complete data path, combined with the control path, is shown

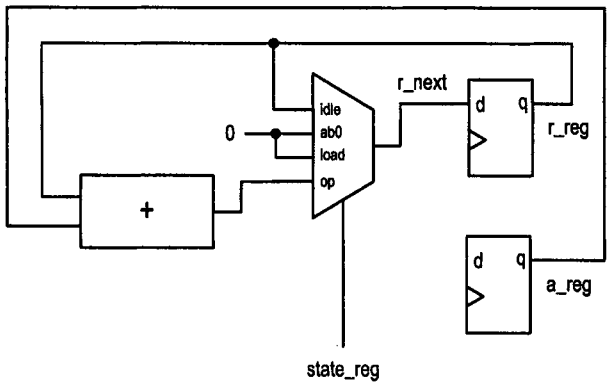


Figure 11.7 Data path associated with the r register.

in Figure 11.8. The clock and reset signals are connected to all registers. To reduce clutter, they are not shown on the diagram. The four major parts of the data path, functional units, routing circuit, data registers and status circuit, are grouped as shaded blocks.

Since this is a simple design, Figure 11.8 is somewhat unnecessarily complicated. For example, the multiplexing circuit for the a_next signal can be replaced by a register with an enable signal. The purpose of the diagram is to illustrate the derivation process. This process is very general and thus can be applied to any properly designed ASMD chart. Since the block diagram will eventually be described by VHDL code and synthesized, the multiplexing circuit will be optimized during logic synthesis.

11.3.3 Multi-segment VHDL description of an FSMD

After understanding the construction of an FSMD, we can derive the VHDL program accordingly. Our first VHDL description follows the detailed block diagram of Figure 11.8. The diagram is divided into seven blocks, which include the state register, next-state logic and output logic of the control path, and the data registers, functional units, routing network and status circuit of the data path. We use a VHDL segment for each block, and the code is shown in Listing 11.1.

Listing 11.1 Multi-segment description of a repetitive-addition multiplier

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity seq_mult is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(15 downto 0)
   );
end seq_mult;

architecture mult_seg_arch of seq_mult is
```

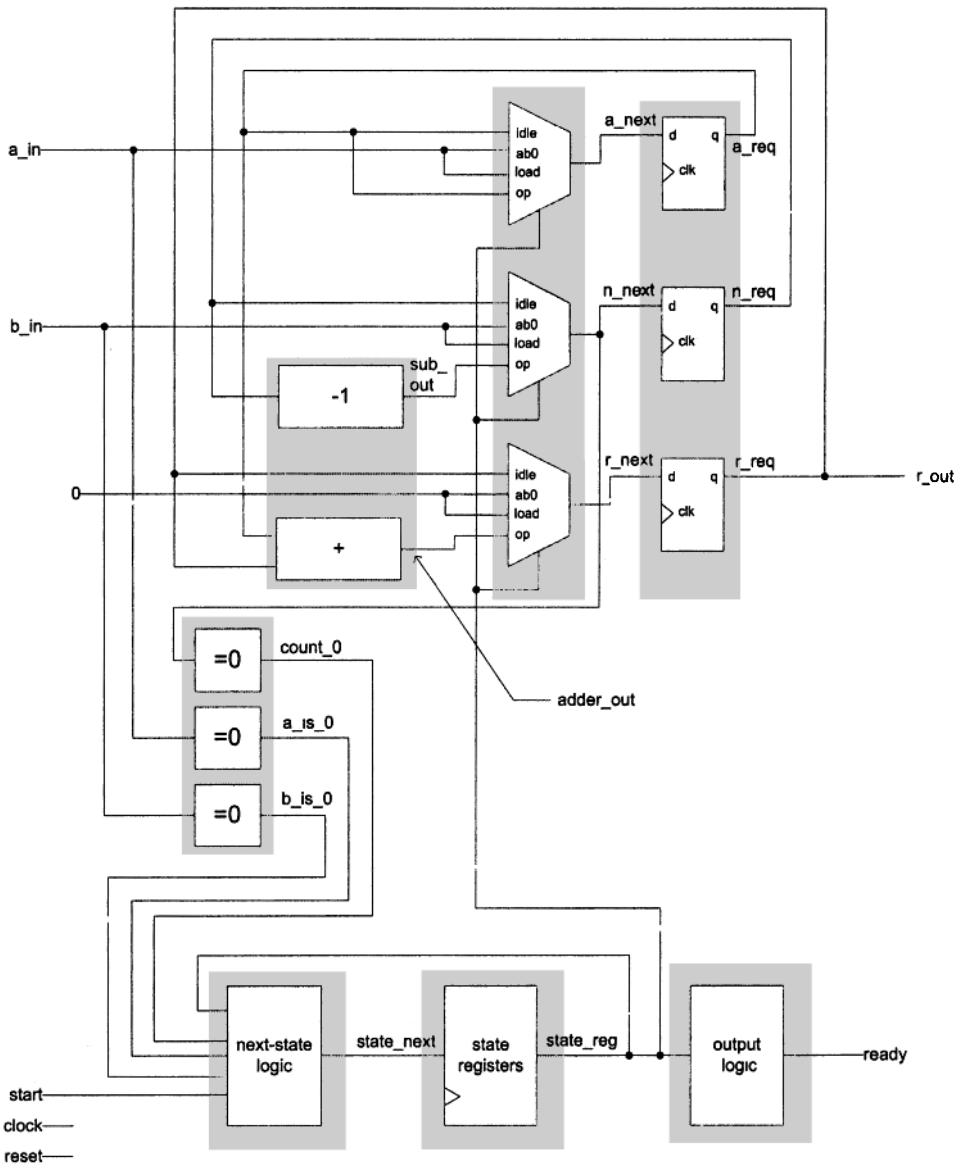


Figure 11.8 Complete block diagram of a repetitive-addition multiplier.

```
    constant WIDTH: integer:=8;
    type state_type is (idle, ab0, load, op);
    signal state_reg, state_next: state_type;
    signal a_is_0, b_is_0, count_0: std_logic;
    signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
    signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
    signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
    signal adder_out: unsigned(2*WIDTH-1 downto 0);
    signal sub_out: unsigned(WIDTH-1 downto 0);
begin
   -- control path: state register
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- control path: next-state/output logic
   process(state_reg,start,a_is_0,b_is_0,count_0)
   begin
      case state_reg is
         when idle =>
            if start='1' then
               if (a_is_0='1' or b_is_0='1') then
                  state_next <= ab0;
               else
                  state_next <= load;
               end if;
            else
               state_next <= idle;
            end if;
         when ab0 =>
            state_next <= idle;
         when load =>
            state_next <= op;
         when op =>
            if count_0='1' then
               state_next <= idle;
            else
               state_next <= op;
            end if;
      end case;
   end process;
   -- control path: output logic
   ready <= '1' when state_reg=idle else '0';
   -- data path: data register
   process(clk,reset)
   begin
      if reset='1' then
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
```

```
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         a_reg <= a_next;
         n_reg <= n_next;
         r_reg <= r_next;
      end if;
   end process;
   -- data path: routing multiplexer
   process(state_reg,a_reg,n_reg,r_reg,
           a_in,b_in,adder_out,sub_out)
   begin
      case state_reg is
         when idle =>
            a_next <= a_reg;
            n_next <= n_reg;
            r_next <= r_reg;
         when ab0 =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when load =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when op =>
            a_next <= a_reg;
            n_next <= sub_out;
            r_next <= adder_out;
         end case;
   end process;
   -- data path: functional units
   adder_out <= ("00000000" & a_reg) + r_reg;
   sub_out <= n_reg - 1;
   -- data path: status
   a_is_0 <= '1' when a_in="00000000" else '0';
   b_is_0 <= '1' when b_in="00000000" else '0';
   count_0 <= '1' when n_next="00000000" else '0';
   -- data path: output
   r <= std_logic_vector(r_reg);
end mult_seg_arch;
```

11.3.4 Use of a register value in a decision box

The key to realizing RT methodology is to derive an efficient and correct ASMD description of an algorithm. Once this is accomplished, the VHDL derivation is more or less a mechanical procedure. The most subtle part of the ASMD derivation is using a register in Boolean expressions of the decision boxes. We intentionally avoided this issue in the ASMD of Figure 11.6 and used the somewhat vague a_is_0, b_is_0 and count_0 status signals inside the decision boxes. A more descriptive way is to express the Boolean conditions with registers or input signals.

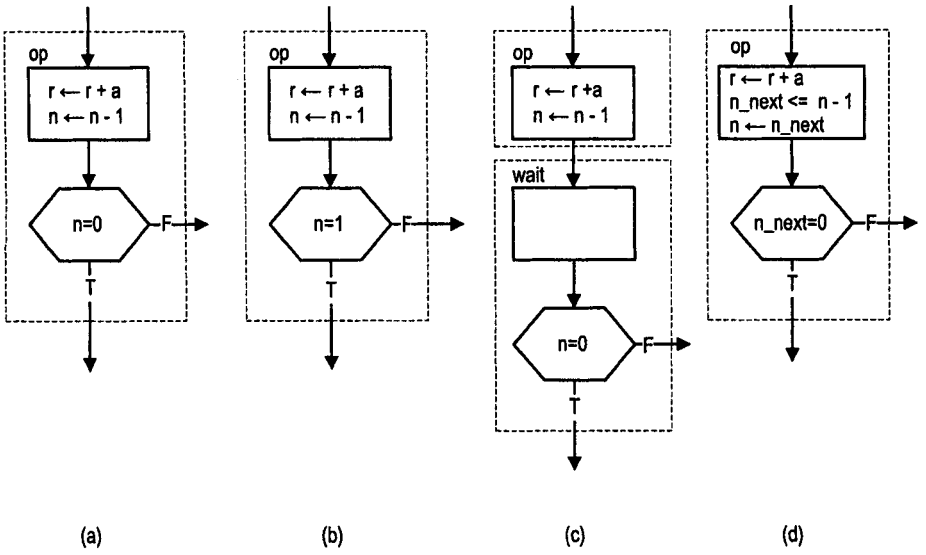


Figure 11.9 Register used in a decision box.

In the second decision box, the `a_is_0=1 or b_is_0=1` expression can easily be translated into `a_in=0 or b_in=0`. In the third decision box, the condition for the `count_0=1` expression is more subtle. The `n` register is used as a counter to keep track of the number of iterations. The iteration stops when `n` reaches 0. In pseudocode, it is expressed as

```
n = n - 1;
if (n = 0) then{
   goto stop;}
else{
   goto op;}
```

Since the execution is sequential, the `n` variable is updated in the `n = n - 1` statement, and then the new value is used in the `n = 0` expression of the if statement.

In the corresponding ASMD chart, the `n ← n - 1` operation and the decision box are in the same ASMD block. Since `n` is updated when the FSMD exits the block, the old value of `n` is used in decision box. If we write the condition as `n = 0` inside the decision box, as in Figure 11.9(a), one extra iteration is introduced and thus the result is not correct.

One way to fix the problem is to use the condition of the previous iteration, `n = 1`, to terminate the loop, as in Figure 11.9(b). This approach may not work for other algorithms when the condition of the previous iteration cannot be determined in advance. The discrepancy between the pseudocode and the ASMD chart also makes the ASMD chart less intuitive.

One clumsy way to solve the problem is to insert an artificial `wait` state so that the content of `n` can be updated before it is used in a decision box. This approach is shown in Figure 11.9(c). While this makes the ASMD looks like the original algorithm, the `wait` state introduces one extra clock cycle in the iteration and thus severely degrades the performance.

A better way is to use the *next value* of the `n` register in the Boolean expression of the decision box. Since the next value is calculated during the op state, it is available at the end

of the clock cycle and can be used in the decision box. Note that the previous VHDL code actually uses this value to generate the count_0 status signal:

```
count_0 <= '1' when n_next=0 else '0';
```

To express this idea in the ASMD chart, we have to split the RT operation r ← $f(\cdot)$ into two parts:

- r_next <= $f(\cdot)$
- r ← r_next;

The first part means that the next value of the r register is calculated and updated within the current clock cycle. We use the signal assignment notation, <=, to emphasize that the assignment is independent of the clock. The second part indicates that the r_next signal is then assigned to r at the exit of the current state, as a regular RT operation. We can use this notation to replace the count_0=0 expression of the ASMD chart, as shown in Figure 11.9(d). This approach is the preferred method since it does not use the condition of the previous iteration, maintains consistency with the original sequential algorithm and introduces no performance penalty.

11.3.5 Four- and two-segment VHDL descriptions of FSMD

The previous multi-segment description follows the detailed FSMD block diagram. For a simple design, some blocks are very straightforward, and partitioning the VHDL code into so many code segments is overkill. We can merge some blocks to make the code more compact.

For the FSMD block diagram in Figure 11.5, we can merge the combinational circuits of the data path and control path respectively, and divide the code into four segments: the data path registers, data path combinational circuit, control path register and control path combinational circuit. The detailed VHDL code is shown in Listing 11.2. Some duplicated segments are omitted. Note that we eliminate the a_is_0, b_is_0 and count_0 status signals, and use the a_in, b_in and n_next signals directly in the Boolean conditions of the control path.

Listing 11.2 Four-segment description of a repetitive-addition multiplier

```
architecture four_seg_arch of seq_mult is
    -- declarations same as mult_seg_arch , omitted
    . . .
begin
  -- control path: state register
    -- same as mult_seg_arch , omitted
    . . .
  -- control path: combinational logic
  process(start,state_reg,a_in,b_in,n_next)
  begin
    ready <='0';
    case state_reg is
      when idle =>
        if start='1' then
          if (a_in="00000000" or b_in="00000000") then
            state_next <= ab0;
          else
            state_next <= load;
```

```
                end if;
            else
                state_next <= idle;
            end if;
            ready <='1';
         when ab0 =>
            state_next <= idle;
         when load =>
            state_next <= op;
         when op =>
            if (n_next="00000000") then
               state_next <= idle;
            else
               state_next <= op;
            end if;
      end case;
   end process;
   -- data path: data register
     -- same as mult_seg_arch, omitted
     . . .
   -- data path: combinational circuit
   process(state_reg,a_reg,n_reg,r_reg,a_in,b_in)
   begin
      -- default value
      a_next <= a_reg;
      n_next <= n_reg;
      r_next <= r_reg;
      case state_reg is
         when idle =>
         when ab0 =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when load =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
         when op =>
            n_next <= n_reg - 1;
            r_next <= ("00000000" & a_reg) + r_reg;
      end case;
   end process;
   . . .
end four_seg_arch;
```

Since the data registers and the state register are synchronized by the same clock signal, we can merge them into a single code segment. Similarly, since the descriptions of both combinational circuits are based on the state of the FSM, we can merge them into one segment. The resulting code consists of only two segments, one for the registers and one for the combinational circuits. The VHDL code is shown in Listing 11.3.

Listing 11.3 Two-segment description of a repetitive-addition multiplier

```
architecture two_seg_arch of seq_mult is
    -- declarations same as mult_seg_arch , omitted
    . . .
begin
  -- state and data registers
  process(clk,reset)
  begin
    if reset='1' then
      state_reg <= idle;
      a_reg <= (others=>'0');
      n_reg <= (others=>'0');
      r_reg <= (others=>'0');
    elsif (clk'event and clk='1') then
      state_reg <= state_next;
      a_reg <= a_next;
      n_reg <= n_next;
      r_reg <= r_next;
    end if;
  end process;
  -- combinational circuit
  process(start,state_reg,a_reg,n_reg,r_reg,a_in,b_in,
          n_next)
  begin
    -- default value
    a_next <= a_reg;
    n_next <= n_reg;
    r_next <= r_reg;
    ready <='0';
    case state_reg is
      when idle =>
        if start='1' then
          if (a_in="00000000" or b_in="00000000") then
            state_next <= ab0;
          else
            state_next <= load;
          end if;
        else
            state_next <= idle;
        end if;
        ready <='1';
      when ab0 =>
        a_next <= unsigned(a_in);
        n_next <= unsigned(b_in);
        r_next <= (others=>'0');
        state_next <= idle;
      when load =>
        a_next <= unsigned(a_in);
        n_next <= unsigned(b_in);
        r_next <= (others=>'0');
        state_next <= op;
      when op =>
        n_next <= n_reg - 1;
```

```
            r_next <= ("00000000" & a_reg) + r_reg;
            if (n_next="00000000") then
               state_next <= idle;
            else
               state_next <= op;
            end if;
         end case;
      end process;
      r <= std_logic_vector(r_reg);
   end two_seg_arch;
```

The combinational segment basically follows the ASMD chart. It uses a case statement to list the states of the ASMD chart and specifies the actions needed in each state, which include the RT operations to be performed in the data path, the next state of the control path and the external status signal of the control path.

In the beginning of the process, we use the default signal assignment statements:

```
a_next <= a_reg;
n_next <= n_reg;
r_next <= r_reg;
ready <='0';
```

These imply that registers will keep their previous values and the output signal will be unasserted if they are not assigned in a branch of the case statement. Use of the default signal assignment statements is consistent with our notation of the ASMD chart, in which only the non-default RT operations and asserted output signals are listed inside a state box. Following the two-segment coding style, we can derive the VHDL code directly from an ASMD chart and quickly realize it in hardware.

The four- and two-segment coding styles are just some possible ways to merge the blocks of an FSMD. Since an FSMD is a sequential circuit, it is a good practice to separate the registers from the combinational circuit. Other than that, we can combine or isolate combinational blocks as needed and exercise different degrees of control over the underlying hardware configuration. In an FSMD design, the functional units of the data path are normally the most complex components and are the dominant factor in circuit size and system performance. We should pay more attention to these parts and may need to separate them from the remaining code to achieve the desired area or performance constraints. The other portion of the combinational circuit can be treated as "random logic" and will be optimized during logic synthesis.

11.3.6 One-segment coding style and its deficiency

In VHDL, it is possible to combine the registers and combinational circuit into a single segment. This style may introduce some subtle mistakes, as discussed in Section 8.7. We can use this style to code FSMD as well and it allows us to translate an ASMD chart directly into one-segment VHDL code. Although this approach seems to be quick and compact at first glance, it may again introduce many subtle problems and is not recommended. The following example illustrates some of the problems. The one-segment VHDL code of the repetitive-addition multiplier is shown in Listing 11.4.

Listing 11.4 One-segment description of a repetitive-addition multiplier

```
architecture one_seg_arch of seq_mult is
   constant WIDTH: integer:=8;
   type state_type is (idle, ab0, load, op);
   signal state_reg: state_type;
   signal a_reg, n_reg: unsigned(WIDTH-1 downto 0);
   signal r_reg: unsigned(2*WIDTH-1 downto 0);
begin
   process(clk,reset)
      variable n_next: unsigned(WIDTH-1 downto 0);
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         case state_reg is
            when idle =>
               if start='1' then
                  if (a_in="00000000" or b_in="00000000") then
                     state_reg <= ab0;
                  else
                     state_reg <= load;
                  end if;
               end if;
            when ab0 =>
               a_reg <= unsigned(a_in);
               n_reg <= unsigned(b_in);
               r_reg <= (others=>'0');
               state_reg <= idle;
            when load =>
               a_reg <= unsigned(a_in);
               n_reg <= unsigned(b_in);
               r_reg <= (others=>'0');
               state_reg <= op;
            when op =>
               n_next := n_reg - 1;
               n_reg <= n_next;
               r_reg <= ("00000000" & a_reg) + r_reg;
               if (n_next="00000000") then
                  state_reg <= idle;
               end if;
         end case;
      end if;
   end process;
   ready <='1' when (state_reg=idle) else '0';
   r <= std_logic_vector(r_reg);
end one_seg_arch;
```

There are several subtle problems in the code. First, since a register is inferred for any signal within the `clk'event` **and** `clk='1'` branch, the next value of a data register cannot be referred by a signal. To overcome this, we must define `n_next` as a variable for

immediate assignment. Note that the variable here is used to achieve the effect of immediate assignment and has nothing to do with the variables used in the pseudocode. Second, to avoid the unnecessary output buffer, the `ready` output signal has to be moved outside the process and be coded as a separate segment. The problems encountered in the one-segment coding style usually require more attention and offset the original hope for quick and clear coding. We avoid this coding style in this book.

11.4 ALTERNATIVE DESIGN OF A REPETITIVE-ADDITION MULTIPLIER

After studying the basic FSMD construction and VHDL coding of the repetitive-addition algorithm, we examine two variations in this section. The variations introduce the concept of sharing and Mealy-controlled RT operation.

11.4.1 Resource sharing via FSMD

We discussed combinational resource sharing in Section 7.2. It can be applied only to few restricted scenarios. Since an FSMD provides a mechanism to schedule RT operations, sharing can be achieved in a *time-multiplexed* fashion; i.e., we can assign the same functional unit in different states (i.e., different clock cycles) and use it repeatedly. For example, if an algorithm needs to perform three additions, instead of using three adders to perform the three additions at the same time, we can use one adder and schedule the additions in three states. The FSMD allows us to have another dimension of flexibility to obtain a good trade-off between the circuit size and performance.

When we convert an algorithm into an FSMD, the functional units of the data path are usually the most complex components. Since many RT operations perform the same or similar functions, some functional units can be shared as long as these operations are scheduled in different states. In the previous repetitive-addition multiplier implementation, the function units include a 16-bit adder and an 8-bit decrementor. In the original ASMD of Figure 11.6, both addition and decrementing RT operations are scheduled in the op state, and thus no sharing is possible. If we wish to reduce the circuit size, one possibility is to split the operations and schedule them into two states. This idea is shown in the revised ASMD chart in Figure 11.10, in which the original op state is split into the `op1` and `op2` states. Note that an iteration now travels through two states and thus requires two clock cycles. Since the main calculation of the algorithm is done through the iterations, it takes almost twice the number of clock cycles to complete the same task.

The block diagram of the revised data path is shown in Figure 11.11. Note that there is only one adder. Two 2-to-1 multiplexers route the desired inputs to the adder. The inputs can be either `a_reg` and `r_reg`, or `n_reg` and "11$\cdots$11", which is -1 in 2's-complement format. The former will be routed to the adder only if the current state of the control path is `op1`. Since there are already multiplexing circuits for the registers' inputs, no special routing circuit is needed for the output of the adder. Note that the output of the adder, `add_out`, is routed to the `op1` port of the `r_reg` register's input multiplexer and to the `op2` port of the `n_reg` register's input multiplexer.

As we discussed in Chapter 6, synthesis software is weak in performing RT-level optimization. If we use the two-segment coding style, the software may not be able to detect the intended sharing in the data path. To ensure the proper hardware construction, we can explicitly specify the desired functional unit sharing in VHDL code. The revised VHDL code is shown in Listing 11.5. It basically uses the two-segment coding style but isolates

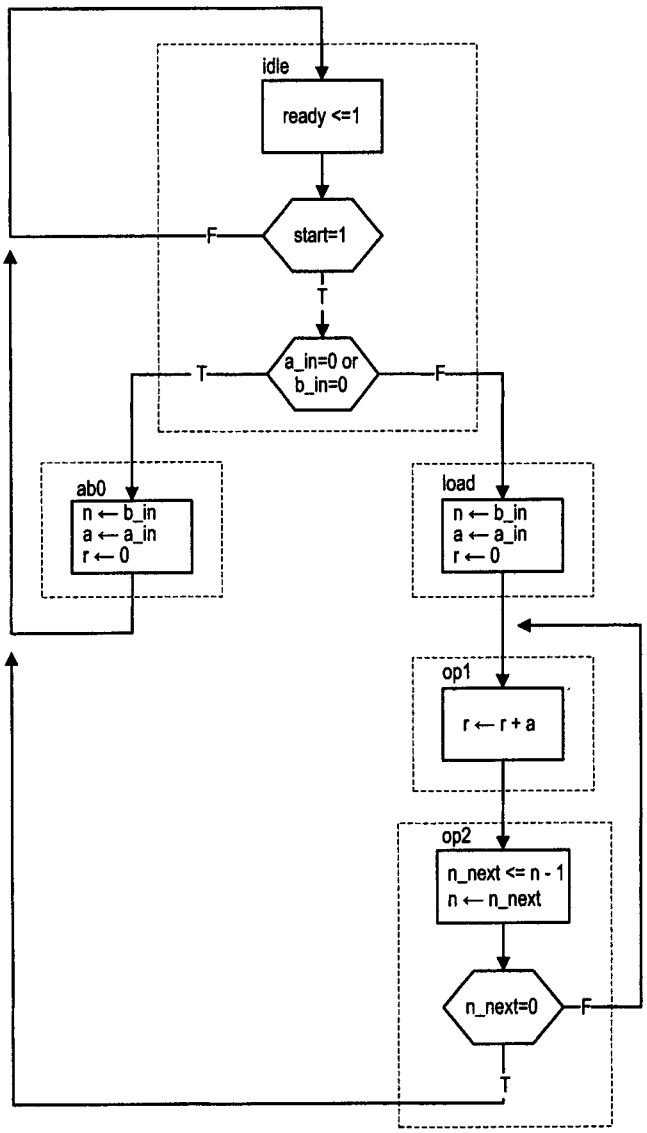


Figure 11.10 ASMD chart with sharing.

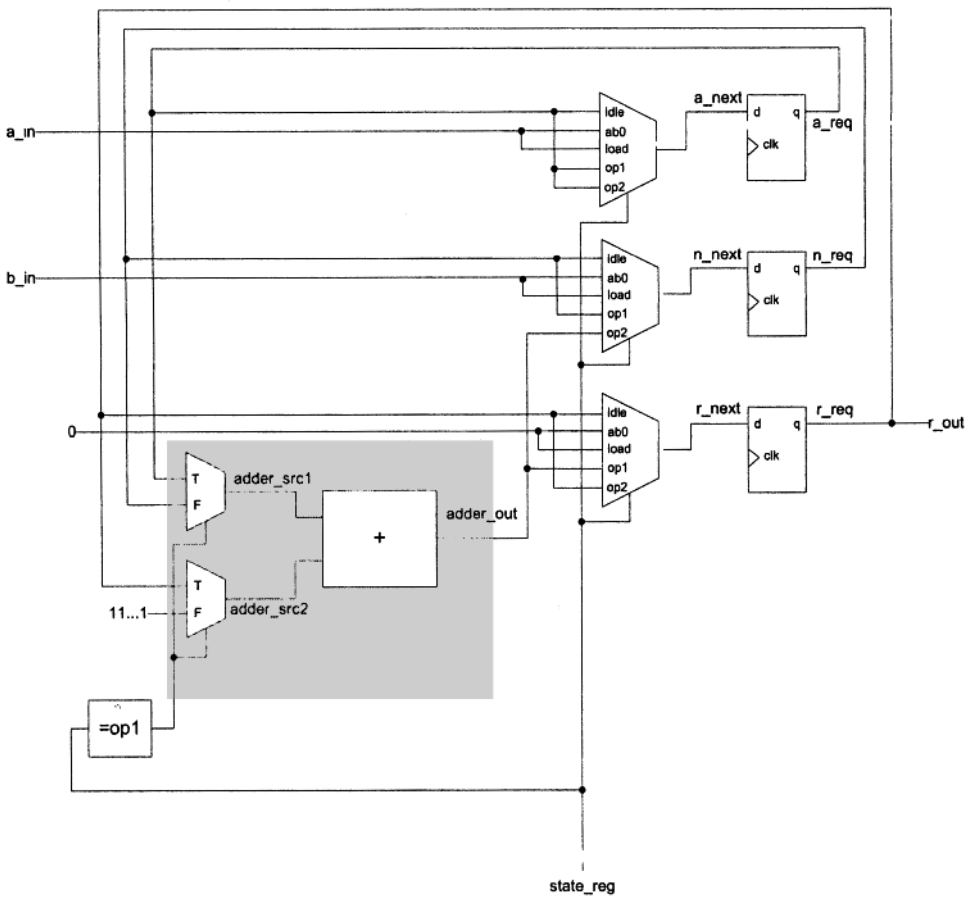


Figure 11.11 Conceptual block diagram of a sharing data path.

the functional unit from the remaining code. Note that the n register is only 8 bits wide and some adjustments are made in code to accommodate the 16-bit adder.

Listing 11.5 Sharing on a repetitive-addition multiplier

```
architecture sharing_arch of seq_mult is
   constant WIDTH: integer:=8;
   type state_type is (idle, ab0, load, op1, op2);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
   signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
   signal adder_src1,adder_src2: unsigned(2*WIDTH-1 downto 0);
   signal adder_out: unsigned(2*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');
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         a_reg <= a_next;
         n_reg <= n_next;
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic/ouput logic and data path routing
   process(start,state_reg,a_reg,n_reg,r_reg,a_in,b_in,
           adder_out,n_next)
   begin
      -- defaut value
      a_next <= a_reg;
      n_next <= n_reg;
      r_next <= r_reg;
      ready <='0';
      case state_reg is
         when idle =>
            if start='1' then
               if (a_in="00000000" or b_in="00000000") then
                  state_next <= ab0;
               else
                  state_next <= load;
               end if;
            else
                 state_next <= idle;
            end if;
            ready <='1';
         when ab0 =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
```

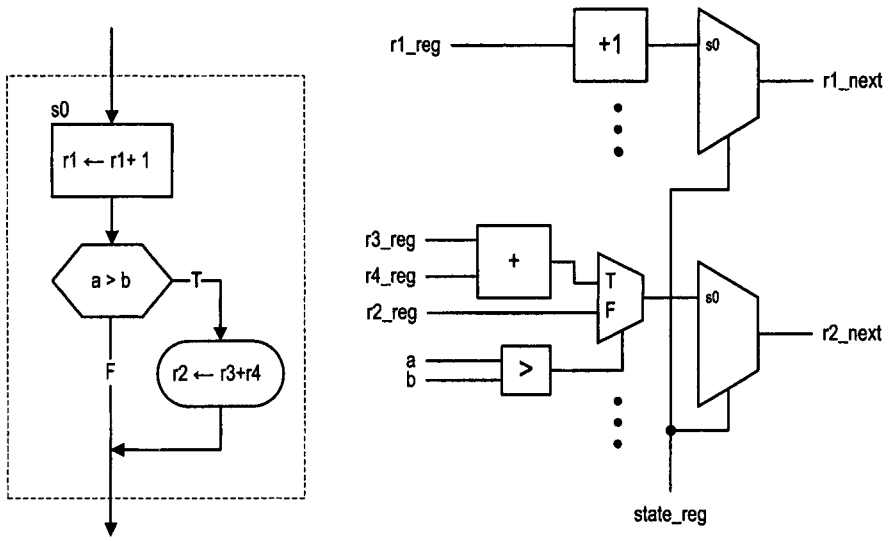
```
            r_next <= (others=>'0');
            state_next <= idle;
         when load =>
            a_next <= unsigned(a_in);
            n_next <= unsigned(b_in);
            r_next <= (others=>'0');
            state_next <= op1;
         when op1 =>
            r_next <= adder_out;
            state_next <= op2;
         when op2 =>
            n_next <= adder_out(WIDTH-1 downto 0);
            if (n_next="00000000") then
               state_next <= idle;
            else
               state_next <= op1;
            end if;
      end case;
   end process;
   -- data path input routing and functional units
   process(state_reg,r_reg, a_reg, n_reg)
   begin
      if (state_reg=op1) then
         adder_src1 <= r_reg;
         adder_src2 <= "00000000" & a_reg;
      else  -- for op2 state
         adder_src1 <= "00000000" & n_reg;
         adder_src2 <= (others=>'1');
      end if;
   end process;
   adder_out <= adder_src1 + adder_src2;
   -- output
   r <= std_logic_vector(r_reg);
end sharing_arch;
```

Because the 8-bit decrementor is a relatively simple functional unit, the new design will not reduce the circuit size significantly, and the sharing is probably overkill for this particular example. Clearly, the sharing will become more predominant if complex functional units, such as combinational multipliers, are involved.

11.4.2 Mealy-controlled RT operations

In Section 10.4, we discussed the difference between a Mealy output and a Moore output. A Mealy output is preferred for an edge-sensitive control signal because it responds faster and requires fewer states in an FSM. Since the control path and data path are synchronized by the same clock signal, the control signals connected to the data path are edge-sensitive, and thus the Mealy output can be used. In terms of the FSMD, this means that we can specify RT operations in a conditional output box of an ASMD chart.

A representative ASMD block with a conditional output box is shown in Figure 11.12(a). The conditional output box indicates that the `r2` ← `r3 + r4` operation will be performed if the `a > b` condition is `true`. If the condition is `false`, `r2` remains unchanged, which



(a) ASMD block

(b) Conceptual block diagram

Figure 11.12 ASMD block with a conditional output box.

means that the `r2` ← `r2` operation will be performed. For comparison purposes, a Moore output-controlled operation, `r1` ← `r1 + 1`, is included in the state box.

If this is a regular flowchart, the condition `a > b` is first evaluated and, if the condition is met, the `r2` ← `r3 + r4` operation will be performed accordingly. However, in an ASMD chart, all operations inside an ASMD block are evaluated in parallel. When the FSMD is in the `s0` state, evaluations of `a > b`, `r3 + r4` and `r1 + 1` are performed at the same time. At the end of the clock cycle, the FSMD checks the result of `a > b` and stores the value of `r2` or the result of `r3 + r4` to `r2` accordingly.

When an RT operation is specified inside a state box, as in `r1` ← `r1 + 1`, there is only one possible next value (i.e., `r1 + 1`) in the `s0` state. On the other hand, when a conditional output box exists, there are several possible next values (i.e., `r2` or `r3 + r4`). This implies that an additional multiplexing circuit is needed. The corresponding conceptual block diagram is shown in Figure 11.12(b). An additional 2-to-1 multiplexer is added to handle the conditional output box. The result of the `a > b` operation is used as a selection signal and routes the desired next value to the `s0` port of the abstract multiplexer.

We can apply this idea to the repetitive-addition multiplier. The original ASMD chart in Figure 11.6 is actually somewhat awkward. In the `idle` state, the `start`, `a_in` and `b_in` signals are used in the decision box, and thus they have to be available at the exit of the `idle` state. If the `start` signal is asserted, `a_in` and `b_in` will be loaded into the `a` and `n` registers in the `load` or `ab0` state. Because of the delayed store, the actual sampling of the `a_in` and `b_in` signals occurs when the FSMD exits the `load` or `ab0` state, and thus the `a_in` and `b_in` signals must be available at this clock edge again. For an external system that uses the multiplication circuit, this means that it has to place the two operands on `a_in` and `b_in` ports for two consecutive clocks. To release the external system from this artificial timing constraint, a better design should be able to sample the `start`, `a_in` and `b_in` signals at the same time and at only one clock edge.

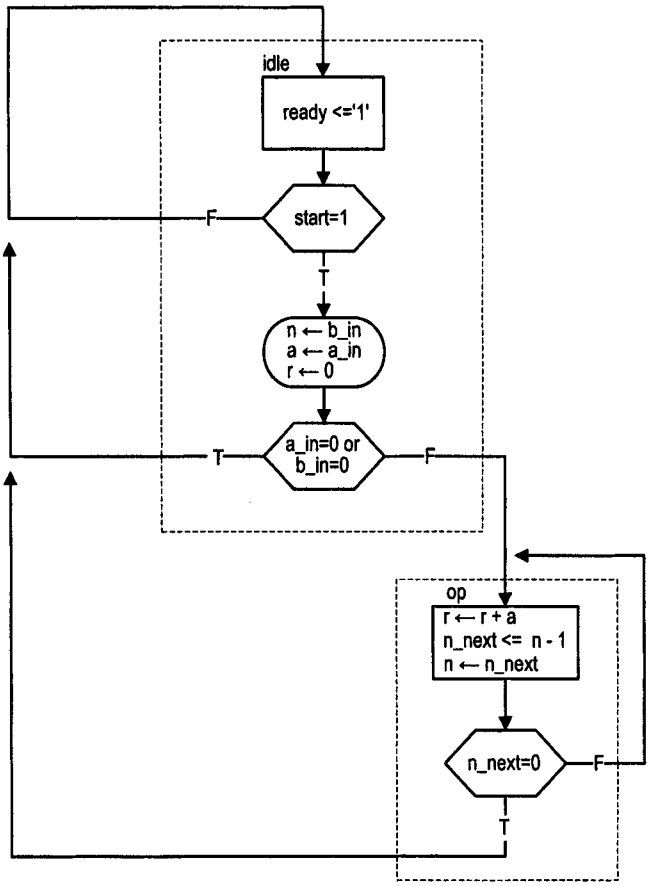


Figure 11.13 ASMD chart with Mealy-controlled RT operations.

This design can be achieved by using Mealy-controlled RT operations. The revised ASMD chart is shown in Figure 11.13. It merges the `ab0` and `load` states into the `idle` state and moves the corresponding RT operations into a conditional output box. In addition to relaxing the timing constraint on the external system, the revised design reduces the number of states from four to two and improves the overall performance. The VHDL code is shown in Listing 11.6. It uses the two-segment coding style. Note that some next-value statements, such as `a_next <= unsigned(a_in)`, are within the then branch of the if statement, which corresponds to the conditional output box of the ASMD chart.

Listing 11.6 Mealy-controlled RT operations for a repetitive-addition multiplier

```
architecture mealy_arch of seq_mult is
   constant WIDTH: integer:=8;
   type state_type is (idle, op);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
   signal r_reg, r_next: unsigned(2*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');
          r_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
          a_reg <= a_next;
          n_reg <= n_next;
          r_reg <= r_next;
       end if;
    end process;
    -- combinational circuit
    process(start,state_reg,a_reg,n_reg,r_reg,a_in,b_in,
            n_next)
    begin
       a_next <= a_reg;
       n_next <= n_reg;
       r_next <= r_reg;
       ready <='0';
       case state_reg is
          when idle =>
             if start='1' then
                a_next <= unsigned(a_in);
                n_next <= unsigned(b_in);
                r_next <= (others=>'0');
                if a_in="00000000" or b_in="00000000" then
                   state_next <= idle;
                else
                   state_next <= op;
                end if;
             else
                state_next <= idle;
             end if;
             ready <='1';
          when op =>
             n_next <= n_reg - 1;
             r_next <= ("00000000" & a_reg) + r_reg;
             if (n_next="00000000") then
                state_next <= idle;
             else
                state_next <= op;
             end if;
       end case;
    end process;
    r <= std_logic_vector(r_reg);
end mealy_arch;
```

11.5 TIMING AND PERFORMANCE ANALYSIS OF FSMD

An FSMD is a synchronous circuit and thus is subject to similar setup and hold time constraints. The setup time constraint, in turn, imposes the maximal clock rate. Unlike a regular sequential circuit, an algorithm described by an FSMD requires a sequence of RT operations to complete. Thus, in addition to the clock rate, the total computation time of an FSMD depends on the number of clock cycles needed to complete the computation as well. The following subsections discuss these issues.

11.5.1 Maximal clock rate

We analyzed timing of a regular sequential circuit and an FSM in Chapters 8 and 10. Both analyses are based on the basic block diagram shown in Figure 8.5. The basic diagram of an FSMD, shown in Figure 11.5, is somewhat different. It has two separate but interactive feedback loops, one for the control path and one for the data path. In theory, we can merge the two feedback loops, convert the FSMD block diagram into the standard diagram, and then analyze it as an ordinary sequential circuit. Because of the interaction between the two loops, it will be difficult to manually analyze the merged combinational circuit. We must rely on a software tool to do the timing analysis and determine the maximal clock rate.

Although the manual analysis cannot determine the exact maximal clock rate, it is possible to determine the boundaries of the rate. This analysis provides more insights into the FSMD operation and helps us to derive a more efficient design. The basic FSDM block diagram of Figure 11.5 has two feedback loops. The data path loop is based on the data register, and the control path loop is based on the state register. The two loops are not independent but interact via the control signals and status signals. For example, a function unit in the data path cannot operate until the control signals set the selection signal of the input multiplexer, and the next-state logic in the control path cannot proceed until the status signals are available. The exact maximal clock rate depends on where the control signals are needed and where the status signals are generated. This depends on the individual implementation and cannot be generalized. Our analysis considers the best- and worst-case scenarios and thus determines the boundaries of the maximal clock rate.

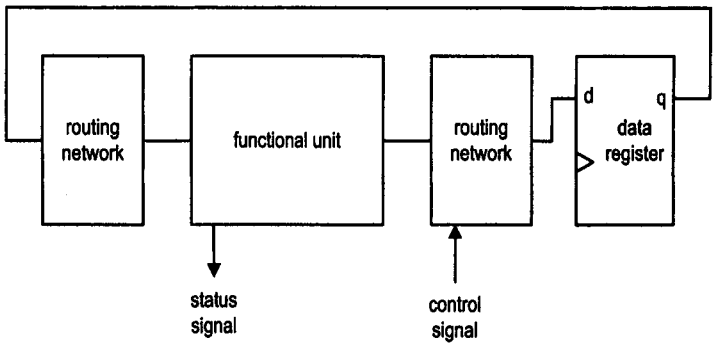
The control path is the bottom part of Figure 11.5. Its timing parameters are the same as those of an FSM. They are defined as follows:

- $T_{cq(state)}$: clock-to-q delay of the state register.
- $T_{setup(state)}$: setup time of the state register.
- T_{next}: maximal propagation delay of the next-state logic of the control path FSM.
- T_{output}: maximal propagation delay of the output-state logic of the control path FSM.

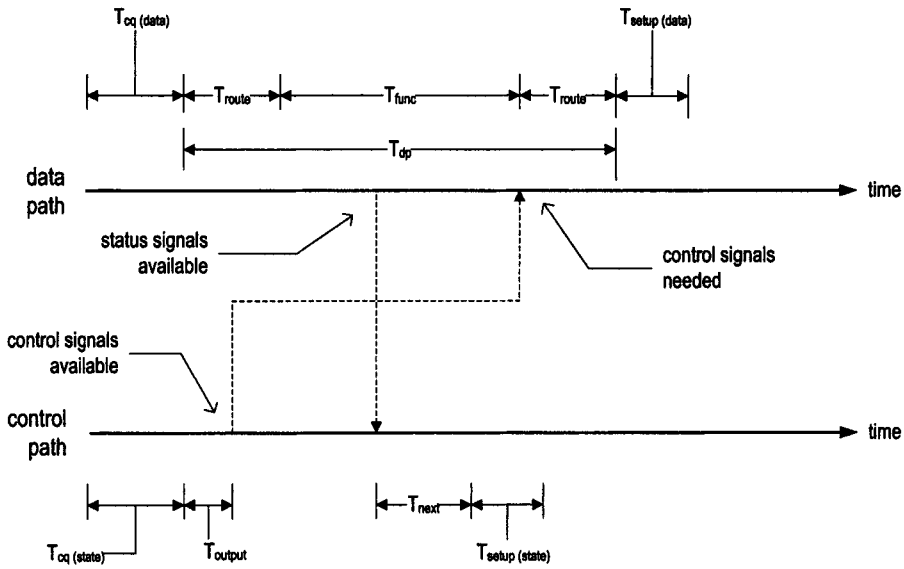
The conceptual diagram of the data path is shown at the top of Figure 11.5. The relevant timing parameters are:

- $T_{cq(data)}$: clock-to-q delay of the data register.
- $T_{setup(data)}$: setup time of the data register.
- T_{func}: maximal propagation delay of the functional units.
- T_{route}: maximal propagation delay of the routing multiplexing circuit.
- T_{dp}: maximal propagation delay of the combinational circuit of the data path, which is the sum of T_{func} and $2T_{route}$.

As before, we use T_c for the the clock period. In a normal design, T_{func} is likely to be the largest and the most dominant of all timing parameters. We use this assumption in the analysis.



(a) Conceptual data path diagram

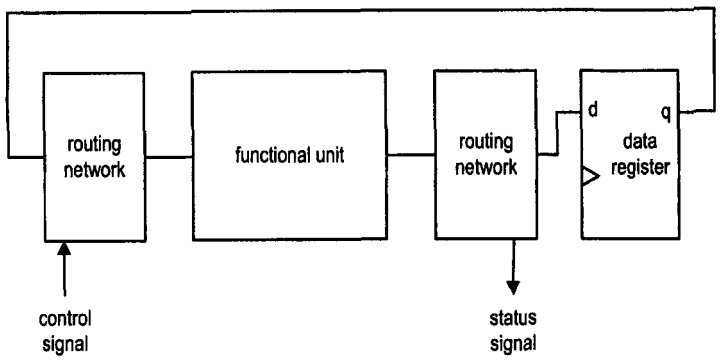


(b) Time lines

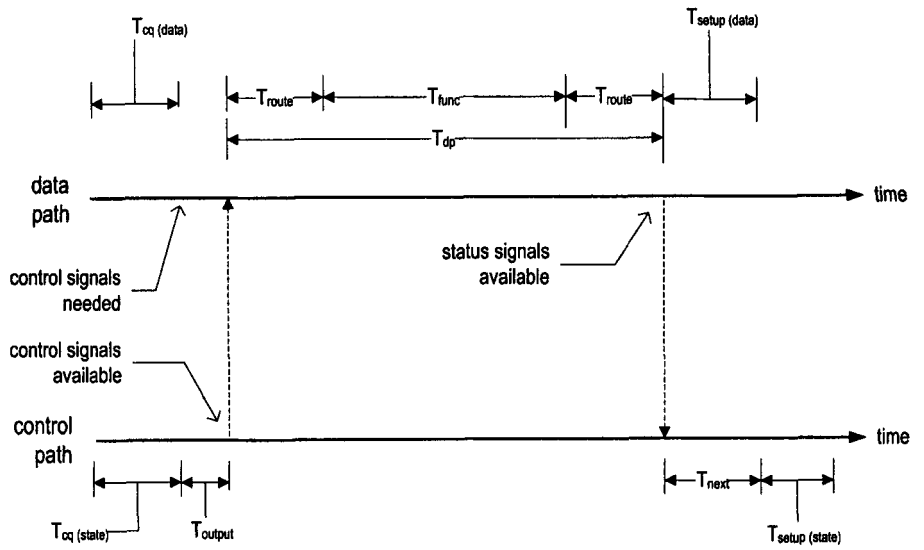
Figure 11.14 Time lines for the best-case scenario.

We first consider the best-case scenario. In this scenario, the control signals are required at a late stage of data path operation, and the status signals are generated in an early stage of data path operation, as shown in the conceptual data path diagram in Figure 11.14(a). The time lines of the data- and control-path operations are shown in Figure 11.14(b). They start with the rising edge of the clock signal. Since the data path uses control signals in the late stage, the operation of the output logic overlaps with the operation of data path and thus contributes no extra delay for the data path loop. Similarly, since the status signal is available at an early stage, the operation of the next-state logic of the control path and the computation of data path are done in parallel. When the data path computation is complete, the next-state value is also ready in the control path. The time lines show that the minimal clock period of the FSMD is the same as the clock period of the data path, which is

$$T_c = T_{cq(data)} + T_{dp} + T_{setup(data)}$$



(a) Conceptual data path diagram



(b) Time lines

Figure 11.15 Time lines for the worst-case scenario.

The worst-case scenario reverses the conditions of the best-case scenario. In this scenario, control signals are required at the beginning of data path operation, and the status signals are generated at the end of data path operation. The conceptual diagram of the data path and the time lines are shown in Figure 11.15. The data path must wait for the FSM to generate the output signals, and the control path must wait for status signals to generate the next-state value. Except for the register, there is no overlapped operation between the control path and the data path. The minimal clock period can be found by following the time lines, and it includes the propagation delays of all combinational components:

$$T_c = T_{cq(state)} + T_{output} + T_{dp} + T_{next} + T_{setup(state)}$$

Assume that the state register and data register have similar timing characteristics, and clock-to-q delay and setup time are T_{cq} and T_{setup} respectively. From the two extreme

scenarios, we can establish the boundaries of the minimal clock period:

$$T_{cq} + T_{dp} + T_{setup} \leq T_c \leq T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}$$

Consequently, the maximal clock rate is bound by

$$\frac{1}{T_{cq} + T_{output} + T_{dp} + T_{next} + T_{setup}} \leq f \leq \frac{1}{T_{cq} + T_{dp} + T_{setup}}$$

For a design with a wide, complex data path, T_{dp} will be much larger than T_{next} and T_{output}, and thus variation in the minimal clock period is relatively small. For a circuit with a complex control path, we may need to minimize T_{next} and T_{output} to obtain better performance. For this kind of design, we can isolate the control path FSM in VHDL code, as in the multi- or four-segment coding styles, and apply special FSM optimization software to obtain a more efficient FSM implementation.

11.5.2 Performance analysis

In an FSMD, computation is performed in a sequence of steps, and it usually takes many clock cycles to complete a task. Thus, the total required time becomes

$$T_{total} = K * T_c$$

where K is the number of clock cycles and T_c is the clock period. K is determined by the algorithm, the width of the input and the value of the input. The determination of K is an ad hoc process and can sometimes be very difficult. For certain algorithms, K and T_c may work against each other. For example, we can merge more computation steps into a single state. This will reduce the number of states (and thus the clock cycles) but increase the clock period due to the larger data path propagation delay (T_{dp}). On the other hand, we can sometimes divide an operation into several smaller steps and schedule them in multiple clock cycles. This will decrease T_{dp} and the clock period, but requires more clock cycles to complete the same computation.

Consider the original ASMD design in Figure 11.6. The width of input operands is 8 bits. The K of this algorithm is not a constant but depends on the value of the `b_in` input. In the best case, `b_in` is 0, and the FSMD goes through the `idle` and `ab0` states. The computation takes two clock cycles (i.e., $K=2$). In the worst case, `b_in` is 255 (i.e., 2^8-1) and `a_in` is not 0, the FSMD goes through the `idle` and `load` states once and loops the `op` state 255 times. K becomes 257. We can generalize this for n-bit input operands. In the worst case, the FSMD goes through the `idle` and `load` states once and loops the `op` state 2^n-1 times. Thus, it takes 2^n+1 clock cycles to complete the computation. We can apply the same analysis for the sharing ASMD design in Figure 11.10, in which the `op` state is split into two states. It requires two clock cycles for each loop iteration. For n-bit input operands, the worst-case K becomes $2+2(2^n-1)$, which is 2^{n+1}. Whereas the data path size is smaller in this design, the required computation time is nearly doubled.

11.6 SEQUENTIAL ADD-AND-SHIFT MULTIPLIER

Although simple, the previous repetitive-addition algorithm is not practical since the required computation time is on the order of $O(2^n)$. This section introduces a more efficient sequential multiplication algorithm. The algorithm is based on the add-and-shift method

					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 11.16 Multiplication as a summation of a_ib_j terms.

discussed in Section 7.5.4. The multiplication of two 4-bit numbers is illustrated in Figure 11.16. It includes three tasks:

1. Multiply the digits of the multiplier (b_3, b_2, b_1 and b_0) by the multiplicand (A) one at a time to obtain $b_3{*}A$, $b_2{*}A$, $b_1{*}A$ and $b_0{*}A$. The $b_i{*}A$ operation is bitwise and operation of b_i and the digits of A; that is,

$$b_i{*}A = (a_3{\cdot}b_i,\ a_2{\cdot}b_i,\ a_1{\cdot}b_i,\ a_0{\cdot}b_i)$$

2. Shift $b_i{*}A$ to left i positions.
3. Add the shifted $b_i{*}A$ terms to obtain the final product.

11.6.1 Initial design

The add-and-shift method can easily be converted into a sequential algorithm. We can process one digit of the multiplier (i.e., b_i) at a time and iterate through all digits of the multiplier (B). In each iteration, we calculate $b_i{*}A$, shift it to the left i positions, and then add it to the partial product. Since b_i is a binary digit, it can be either 0 or 1. Instead of computing $b_i{*}A$, we use an if statement to check the value of b_i and add the shifted A to the partial product when b_i is 1. Assume that the inputs are `a_in` and `b_in`. The pseudocode is

```
n = 0;
p = 0;
while (n!=8) {
   if (b_in(n)=1) then{
      p = p + (a_in << n);}
   n = n + 1;
}
r_out = p;
```

In hardware, it is expensive to do indexing (i.e., `b_in(n)`) and general shifting (i.e., `a_in << n`). To overcome the problem, we can "intelligently" shift `a_in` and `b_in` one position in each iteration. The pseudocode of this algorithm is

```
a = a_in;
b = b_in;
n = 8;
p = 0;
while (n!=0) {
```

```
        if (b(0)=1) then{
           p = p + a;}
        a = a << 1;
        b = b >> 1;
        n = n - 1;
     }
     r_out = p;
```

Four variables are used in the algorithm. The p variable is used to store the partial product, and the n variable is used to keep track of the number of iteration. Note that the counting direction of the n is reversed from the previous pseudocode to accommodate future hardware implementation. The a variable is used to store the shifted multiplicand (A), which is shifted left one position in each iteration. The b variable is used for the multiplier (B). It is shifted right one position in each iteration, and thus b_i of B becomes LSB (i.e., b(0)) of b in the ith iteration.

To facilitate development of an ASMD chart, we can convert the while loop into if and goto statements:

```
     a = a_in;
     b = b_in;
     n = 8;
     p = 0;
op:  if b(0)=1 then {
        p = p + a;}
     a = a << 1;
     b = b >> 1;
     n = n - 1
     if (n!=0) then{
        goto op;}
     r_out = p;
```

This pseudocode can easily be converted to an ASMD chart, as shown in Figure 11.17. The FSMD has three states. In the `idle` state, the FSMD checks the `start` signal. If it is asserted, the FSMD loads the initial values to registers and moves to either the add or `shift` state. If the corresponding bit of the multiplier is '1', the FSMD moves to the add state, in which the shifted multiplicand (A) is added to the partial product. Otherwise, the FSMD moves to the `shift` state, in which the multiplicand (A) is shifted left one position, the multiplier (B) is shifted right one position, and the counter is decremented by 1. The add-and-shift process continues to iterate until the counter reaches 0.

While the chart basically follows the pseudocode, there are two differences. First, since the two shift operations and the counter decrementing operation are independent, they are scheduled in the same state and performed in parallel. Second, due to the delayed store of RT operations, we use the next values of the registers in decision boxes. Note that `b_next(0)` and `n_next` are used in the decision boxes of the `shift` state, and `b_in(0)` is used in the decision box of the `idle` state.

After developing a correct, comprehensive ASMD chart, we can derive the VHDL description accordingly. The VHDL code is shown in Listing 11.7. Note that the two shifting operations are done by the concatenation operations (&).

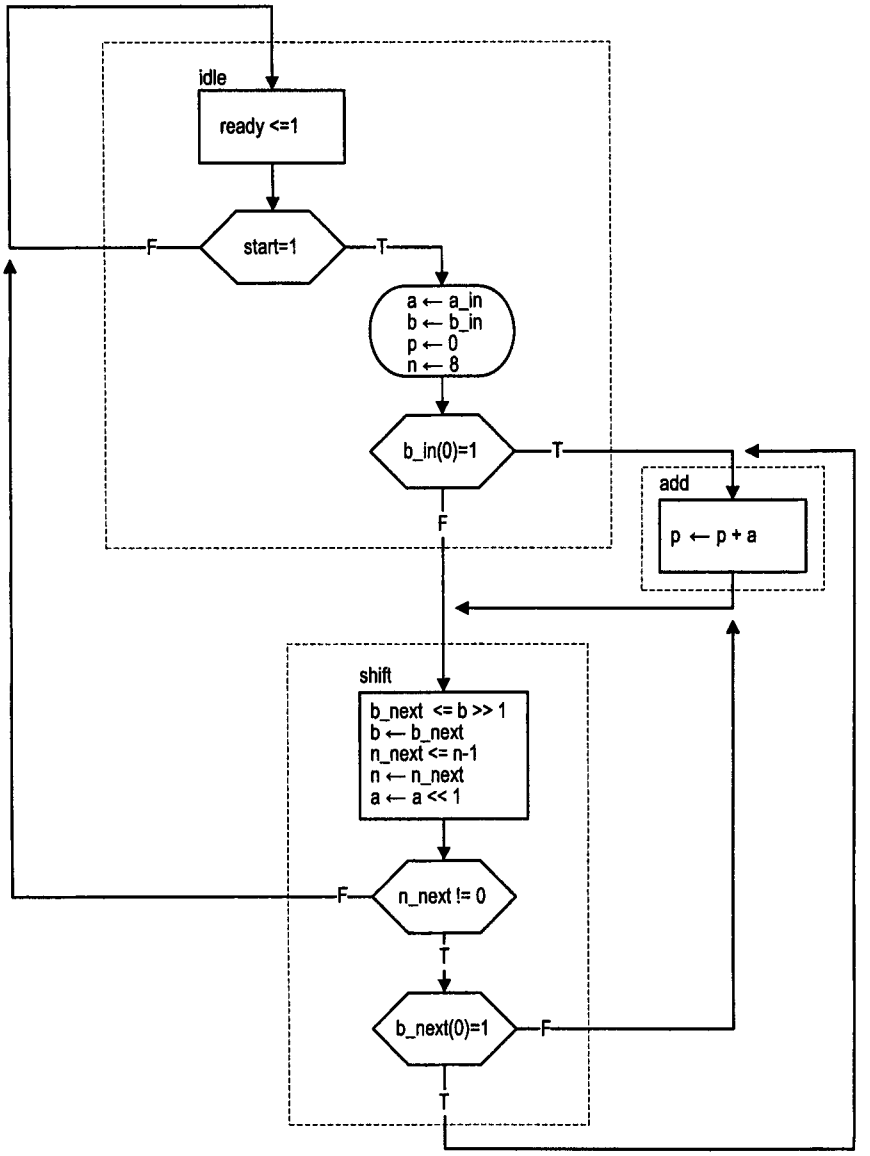


Figure 11.17 ASMD chart of the initial add-and-shift multiplier.

Listing 11.7 Initial description of an add-and-shift sequential multiplier

```
architecture shift_add_raw_arch of seq_mult is
   constant WIDTH: integer:=8;
   constant C_WIDTH: integer:=4; -- width of the counter
   constant C_INIT: unsigned(C_WIDTH-1 downto 0):="1000";
   type state_type is (idle, add, shift);
   signal state_reg, state_next: state_type;
   signal b_reg, b_next: unsigned(WIDTH-1 downto 0);
   signal a_reg, a_next: unsigned(2*WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(C_WIDTH-1 downto 0);
   signal p_reg, p_next: unsigned(2*WIDTH-1 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         b_reg <= (others=>'0');
         a_reg <= (others=>'0');
         n_reg <= (others=>'0');
         p_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         b_reg <= b_next;
         a_reg <= a_next;
         n_reg <= n_next;
         p_reg <= p_next;
      end if;
   end process;
   -- combinational circuit
   process(start,state_reg,b_reg,a_reg,n_reg,p_reg,
           b_in,a_in,n_next,a_next)
   begin
      b_next <= b_reg;
      a_next <= a_reg;
      n_next <= n_reg;
      p_next <= p_reg;
      ready <='0';
      case state_reg is
         when idle =>
            if start='1' then
               b_next <= unsigned(b_in);
               a_next <= "00000000" & unsigned(a_in);
               n_next <= C_INIT;
               p_next <= (others=>'0');
               if b_in(0)='1' then
                  state_next <= add;
               else
                  state_next <= shift;
               end if;
            else
               state_next <= idle;
            end if;
```

```
            ready <='1';
         when add =>
            p_next <= p_reg + a_reg;
            state_next <= shift;
         when shift =>
            n_next <= n_reg - 1;
            b_next <= '0' & b_reg (WIDTH-1 downto 1);
            a_next <= a_reg(2*WIDTH-2 downto 0) & '0';
            if (n_next /= "0000") then
               if a_next(0)='1' then
                  state_next <= add;
               else
                  state_next <= shift;
               end if;
            else
               state_next <= idle;
            end if;
      end case;
   end process;
   r <= std_logic_vector(p_reg);
end shift_add_raw_arch;
```

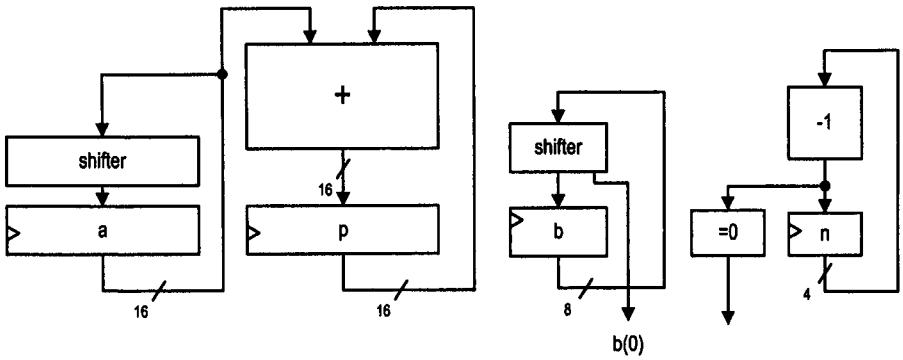
Recall that the functional units used in the data path are normally the most critical components in an FSMD, and understating their basic organization can help us to develop a more efficient design. The sketch of the data path is shown in Figure 11.18(a). To reduce the clutter, only functional units and major data flow are shown. Note that since the amount is fixed in two shift operations, the shifters require no real logic.

In the new algorithm, the number of iterations in the loop is equal to the width of the input operand. An iteration goes through the add and `shift` states if the corresponding multiplier bit is '1' and goes through only the `shift` state otherwise. For n-bit input operands, the computation requires $2n+1$ clock cycles in the worst case (i.e., `b_in` is "$1\cdots1$") and $n+1$ clock cycles in the best case (i.e., `b_in` is "$0\cdots0$"). It is far superior to the 2^n+1 clock cycles of the repetitive-addition algorithm.

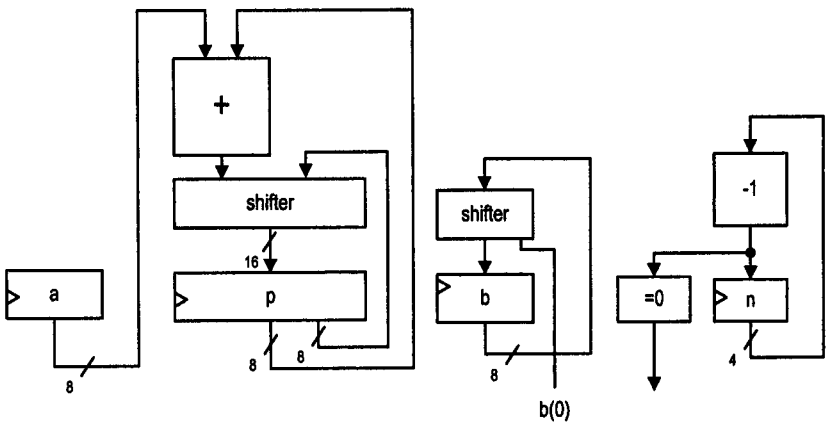
11.6.2 Refined design

Our initial implementation of the add-and-shift multiplier closely follows the sequential pseudocode, which is presumed to be executed in a general-purpose processor. However, hardware implementation provides more flexibility and gives us an opportunity to further streamline the design. There are several possible improvements. We can first improve the efficiency of the ASMD chart. The main computation is done by iterating the add-and-shift loop, and each iteration may go through up to two states. If we examine the add and `shift` states closely, the RT operations in these states are independent. It is possible to merge the two states and utilize a conditional output box for the p $\leftarrow$ p + a operation in the new state. The revised ASMD chart will require only one clock cycle for each iteration.

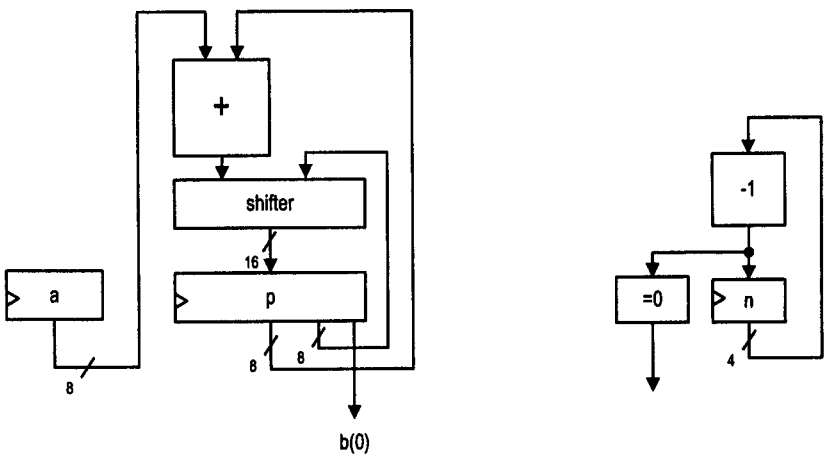
We can also improve the efficiency of the data path. Note that when a is added to the partial products, only the eight leftmost bits of the partial product are involved in the operation and the remaining trailing bits are kept unchanged. Instead of using a 16-bit adder, we can reduce the width of the adder to 9 bits (an 8-bit operand plus a 1-bit carry). This requires to selectively route a portion of the partial product to the adder. The "selective routing" involves complex multiplexing circuits and is not desirable. A better alternative



(a) Initial design



(b) "Shifting p register" design



(c) Final design

Figure 11.18 Sketches of the data path of add-and-shift multipliers.

is to shift the partial product to the right one position in each iteration, and thus the eight current leftmost bits are always connected to the input of the adder. This approach also eliminates the need of shifting multiplier (A) and reduces the width of the a register by half. The sketch of the revised data path is shown in Figure 11.18(b).

The circuit adds the upper half (the left half) of the p register and the a register and then combines the output of the adder with the original lower half (the right half) of the p register to form the new partial product. The p register is then shifted right one bit. Since we wish to merge the addition and shifting operations in the same state, there is no register between the adder and shifter and thus the two operations are performed in the same clock cycle. Because shifting right one bit involves only wiring, merging the two operations will not affect the critical path or increase the clock period.

Another minor improvement is to utilize the right unused portion of the p register. Note that initially only the left half of the p register contains the valid data. The valid portion expands to the right one position in each iteration when the shift-right operation is performed. On the other hand, the b register has eight valid bits initially. In each iteration, it shifts to the right one position and discards the LSB. Since the expansion of the left part of the p register matches the shrinkage of the b register, we can utilize the unused right part of the p register to function as the b register and eliminate the original b register. The sketch of the final data path is shown in Figure 11.18(c).

The refined and improved ASMD chart is shown in Figure 11.19. We use the notations pu and pl as the aliases for the upper half (left half) and lower half (right half) of the p register. Since the addition and shifting operations are performed in the same state, we must use the addition results for the following shift operation. To achieve the desired effect, we use the regular assignment notation (<=) instead of the RT notation (←) in the two conditional output boxes (i.e., pu_next <= pu and pu_next <= pu + a). The result (i.e., pu_next) is then used in a regular RT shift operation (i.e., p ← p_next >> 1). The VHDL code can be derived following the ASMD chart and is shown in Listing 11.8.

Listing 11.8 Refined description of an add-and-shift sequential multiplier

```
architecture shift_add_better_arch of seq_mult is
   constant WIDTH: integer:=8;
   constant C_WIDTH: integer:=4; -- width of the counter
   constant C_INIT: unsigned(C_WIDTH-1 downto 0):="1000";
   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 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;
```

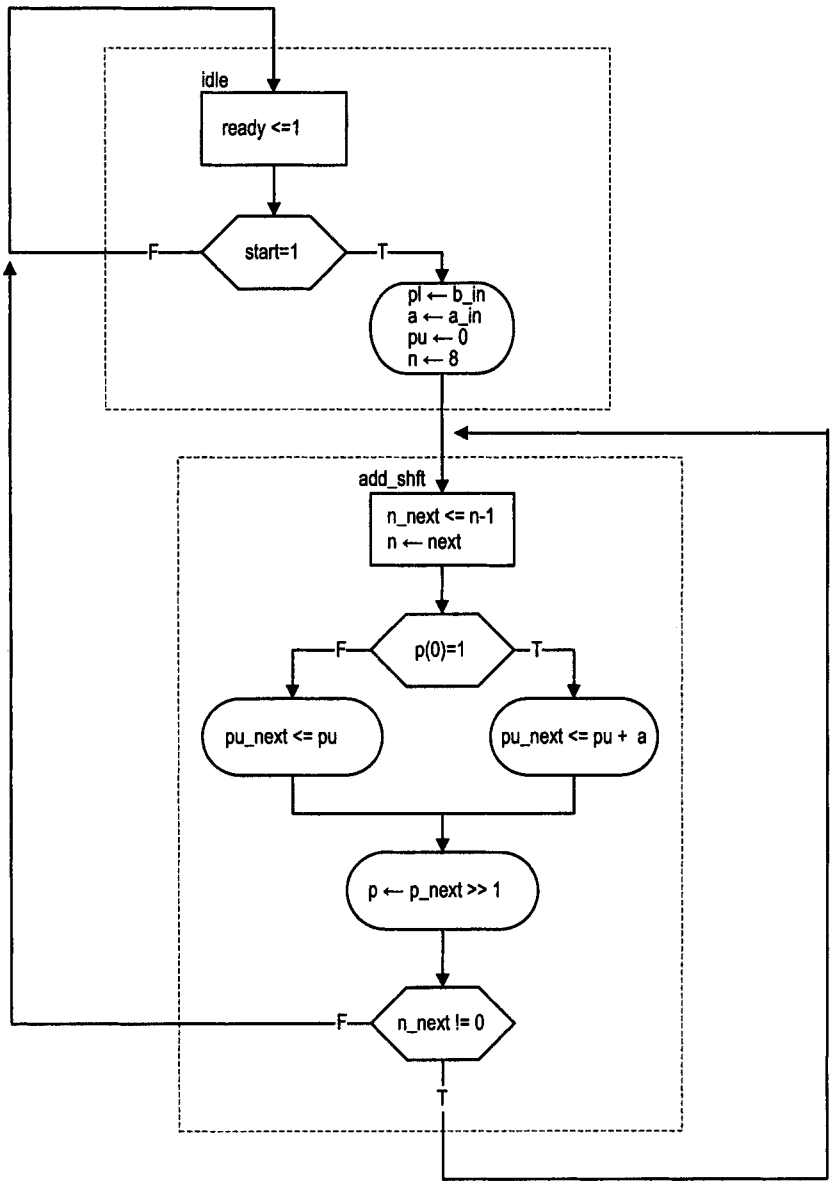


Figure 11.19 ASMD chart of the refined add-and-shift multiplier.

```
        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 <= "000000000" & unsigned(b_in);
              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 multiplier bit is '1'
           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 /= "0000") 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;
```

Table 11.1 Comparison of performance and circuit complexity of three multipliers

Design method	# Clock cycles	Size of functional units	# Register bits
Repetitive-addition	2 to $2^n + 1$	$2n$-bit adder, n-bit decrementor	$4n$
Add-and-shift (initial)	$n + 1$ to $2n + 1$	$2n$-bit adder, $\lceil \log_2(n + 1) \rceil$-bit dec	$5n + \lceil \log_2(n + 1) \rceil$
Add-and-shift (refined)	$n + 1$	n-bit adder, $\lceil \log_2(n + 1) \rceil$-bit dec	$3n + \lceil \log_2(n + 1) \rceil + 1$

11.6.3 Comparison of three ASMD designs

We have examined several designs of a sequential multiplier. Table 11.1 summarizes the key characteristics of these designs, including the Mealy-based repetitive-addition design of Section 11.4.2 and two add-and-shift designs in this section. We assume that the width of the input operands is n bits. The table lists the range of the number of clock cycles to complete the multiplication, the size of the functional units in the data path, and the total number of bits in the data registers. Note that in add-and-shift designs, the counter counts from n to 0 for n-bit operands. There are $n + 1$ patterns in the counting sequence, and thus the width of the counter is $\lceil \log_2(n + 1) \rceil$ bits.

The table shows that the hardware complexity of the repetitive-addition multiplier and the initial add-and-shift multiplier are comparable but that the latter significantly improves performance by reducing the worst-case clock cycles from about 2^n to $2n$. The refined add-and-shift design reduces the hardware complexity roughly by half and decreases the worst-case clock cycles from about $2n$ to n. The adder is the dominant part in the design and contributes most to the propagation delay of the data path. Because of the smaller adder, we can expect the refined add-and-shift design to have a smaller clock period as well.

After we become familiar with the process of converting an ASMD chart to VHDL code, it becomes more or less a mechanical procedure. The key is to find an efficient algorithm and researching an effective data path to support the RT operations in the algorithm. We then can derive the ASMD chart and VHDL code accordingly. As the sequential multiplier examples show, the effectiveness of a design ultimately relies on our understanding of the problem and hardware. No synthesis software can convert the repetitive-addition algorithm into an add-and-shift algorithm or convert the initial add-and-shift design to the refined design.

11.7 SYNTHESIS OF FSMD

The design methodology and VHDL coding style discussed in this chapter impose no new synthesis requirement. From the software's point of view, an FSMD code is just a code with both regular sequential circuits and an FSM and thus can be synthesized accordingly. We can separate the control path and data path in VHDL code if we want to use special FSM optimization software for the control path synthesis.

The synthesis of an algorithm can also be performed in a more abstract level, known as *high-level synthesis* or *behavioral synthesis*. The synthesis starts with abstract VHDL descriptions that are coded in pure sequential statements, similar to those used in the al-

gorithm's pseudocode. The behavioral synthesis software converts the initial description into RT operations and automatically derives a control path and a data path. This kind of synthesis is limited to certain specialized applications. We discuss this in an example in Chapter 12.

11.8 SYNTHESIS GUIDELINES

An FSMD is a synchronous circuit with a regular sequential circuit (the data path) and an FSM (the control path). We should follow the guidelines from Chapters 8, 9 and 10. Few additional guidelines are related primarily to RT operations and construction of the data path:

- As any sequential circuit, the registers of the FSMD should be separated from the combinational circuits.
- Be aware that an RT operation exhibits a delayed-store behavior. Use of a register in a decision box should be carefully examined.
- The variables used in Boolean expressions of a pseudo algorithm normally correspond to the next values of the registers used in an ASMD.
- The function units are normally the most dominant components in a FSMD design. To exercise more control, we may need to isolate them from the rest of the code.
- Separate the control path from the code if the FSM optimization is needed later.

11.9 BIBLIOGRAPHIC NOTES

FSMD and ASMD chart provide a powerful methodology to realize sequential algorithms in hardware. The text, *Principles of Digital Design* by D. D. Gajski, has a comprehensive chapter on the representation and synthesis of FSMD. The text, *Verilog Digital Computer Design* by M. G. Arnold, applies RT methodology for computer design. As its name shows, Verilog is used for the text.

Problems

11.1 The ASMD chart in Figure 11.6 uses the `n` register to keep track of the number of iterations. It is initialized with `b_in` and counts down to 0. Alternatively, it can be initialized with 0 and counts up to `b_in`. From the implementation point of view, which method is better? Explain.

11.2 The ASMD chart in Figure 11.6 must return to the `idle` state after completion even when the main system is ready with a new set of inputs. An alternative is to allow the circuit to perform back-to-back operation in which the FSMD jumps to the `ab0` or `load` state if the `start` signal is asserted while the current operation is completed.

(a) Modify the ASMD chart to reflect the change.

(b) Derive the VHDL code.

11.3 In the ASMD chart of Figure 11.13, what happens if we replace the `a_in=0 or b_in=0` expression of the `idle` state with the `n=0 or b=0` expression? Explain.

11.4 In Listing 11.4, what happens to the algorithm if `n_next` is declared as a signal?

11.5 Repetitive-subtraction division is an algorithm to implement division operation. Let `y` and `d` be the dividend and divisor respectively. This algorithm obtains the quotient (`q`) and the remainder (`r`) by subtracting `d` from `y` repeatedly until the remaining of `y` is smaller than `d`. Assume that all signals are 8 bits wide and interpreted as unsigned integers.

(a) Derive a pseudo algorithm.
(b) Convert the pseudo algorithm into an ASMD chart.
(c) Derive a detailed conceptual diagram.
(d) Derive the VHDL code according to the blocks of the conceptual diagram (i.e., in multi-segment style).
(e) Derive the VHDL code in two-segment style.
(f) Is this an efficient algorithm? Explain.

11.6 A leading-zero counting circuit counts the number of consecutive 0's from an input signal. We want to design a sequential version of this circuit. The design should check one bit of the input at a time and increment accordingly. The counting stops when the first '1' is encountered.

(a) Derive a pseudo algorithm.
(b) Convert the pseudo algorithm into an ASMD chart.
(c) Derive the VHDL code in two-segment style.
(d) Assume that the input is 16 bits wide. Synthesize both combinational and sequential versions in an ASIC technology. Compare the size and performance.

11.7 The Fibonacci function is defined as

$$fib(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

We want to implement this function in hardware. Assume that `n` is a 6-bit input and interpreted as an unsigned integer. Note that $fib(63)$ is 6557470319842.

(a) Derive an ASMD chart.
(b) Derive the VHDL code.

11.8 In the ASMD chart of Figure 11.13, we can express the condition in the bottom decision box as `n_next=0` or `n=1`. From the timing's point of view, which one can help to get a higher clock rate? Explain.

11.9 Synthesize the combinational multiplier in Section 7.5.4 and the sequential multiplier described by the ASMD chart of Figure 11.19 using an ASIC technology.

(a) Assume that the input operands are 8 bits wide. Compare the size and performance of the two circuits.
(b) Repeat part (a), but assume that the input operands are 16 bits wide.

11.10 For the sequential multiplier described by the ASMD chart of Figure 11.19, eight iterations of add-and-shift operation are needed. We can improve the design further by reducing the number of iterations to seven.

(a) Derive an ASMD chart.
(b) Derive the VHDL code.
(c) Assume that the width of the input operands is n. Calculate the relevant parameters of Table 11.1 for this improved design.

11.11 In the sequential add-and-shift multiplier, we can use a combinational circuit to process 2 bits at a time. Instead of adding 0 or shifted A to the partial product, we can add 0, shifted A, shifted $2A$, or shifted $3A$ to the partial product.

(a) Derive the revised ASMD chart for this circuit.

(b) Derive the VHDL code.

(c) Synthesize the design with an ASIC technology. Compare the size and performance of this design and the original design.

CHAPTER 12

REGISTER TRANSFER METHODOLOGY: PRACTICE

RT methodology is a powerful and versatile design technique. It can be applied to a wide variety of applications. In this chapter, we use several examples to illustrate how this methodology can be used in different types of problems and to highlight the design procedure and relevant issues.

12.1 INTRODUCTION

As discussed in Chapter 11, RT methodology can be thought of as a design technique that realizes an algorithm in hardware. The algorithm can be a complex process or just a simple sequential execution, and thus RT methodology is very flexible and versatile. We study five examples in this chapter, including a one-shot pulse generator, SRAM controller, universal asynchronous receiver and transmitter (UART), greatest common divisor (GCD) circuit, and square-root approximation circuit. The one-shot pulse generator is used to compare and contrast the differences among the regular sequential circuit, FSM and RT methodology. The SRAM controller illustrates the process of generating level-sensitive control signals to meet the timing requirement of a clockless device. The GCD circuit is another example of realizing a sequential algorithm in hardware. It shows how the hardware can be used to accelerate the performance. The UART receiver is a typical control-oriented application, which involves complex control structure and decision conditions. The square-root circuit, on the other hand, is a typical data-oriented application, which involves mainly arithmetic operations over data.

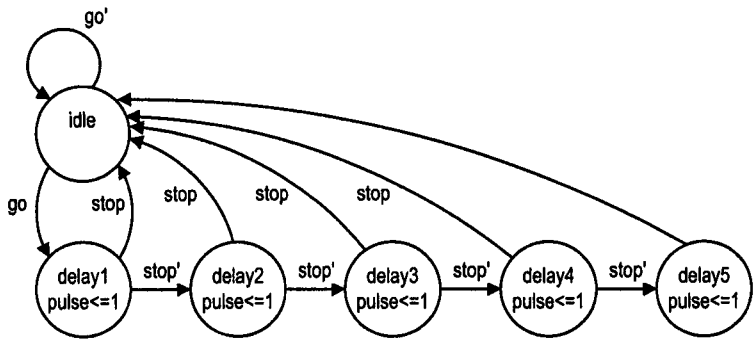


Figure 12.1 State diagram of a one-shot pulse generator.

12.2 ONE-SHOT PULSE GENERATOR

In Section 8.2.3, we divided sequential circuits into three categories based on the characteristics of the next-state logic:

- Regular sequential circuit. The next-state logic is regular.
- FSM. The next-state logic is random.
- RT methodology. The next-state logic consists of a regular part and a random part.

The RT methodology is the most flexible and capable scheme since it can accommodate both types of next-state logic.

The division is created to assist the circuit design and code development. There are no formal definitions of *regular* and *random*, and some applications can be designed as either type. In this section, we use a one-shot pulse generator as an example to illustrate the differences among the three types of circuits and to demonstrate the advantages and flexibility of the RT methodology.

A one-shot pulse generator is a circuit that generates a single fixed-width pulse upon activation of a trigger signal. We assume that the width of the pulse is five clock cycles. The detailed specifications are listed below.

- There are two input signals, go and `stop`, and one output signal, `pulse`.
- The go signal is the trigger signal that is usually asserted for only one clock cycle. During normal operation, assertion of the go signal activates the `pulse` signal for five clock cycles.
- If the go signal is asserted again during this interval, it will be ignored.
- If the `stop` signal is asserted during this interval, the `pulse` signal will be cut short and return to '0'.

Although the circuit is simple, it includes a regular part, which counts five clock cycles, and a random part, which keeps track of whether the circuit is idle or currently generating the pulse. Because of the simplicity, this circuit can be implemented as a pure regular sequential circuit, a pure FSM or a design using RT methodology.

12.2.1 FSM implementation

We first examine the FSM implementation. The state diagram is shown in Figure 12.1. The diagram consists of an idle state and five delay states, which activate the `pulse` signal for

five clock cycles. The five delay states essentially function as a regular sequential circuit that counts for five clock cycles. The identical transition patterns of these five states hints at the "regularity" of this part of the operation. The corresponding VHDL code is shown in Listing 12.1.

Listing 12.1 FSM implementation of a one-shot pulse generator

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pulse_5clk is
   port(
      clk, reset: in std_logic;
      go, stop: in std_logic;
      pulse: out std_logic
   );
end pulse_5clk;

architecture fsm_arch of pulse_5clk is
   type fsm_state_type is
      (idle, delay1, delay2, delay3, delay4, delay5);
   signal state_reg, state_next: fsm_state_type;
begin
   -- state register
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
      end if;
   end process;
   -- next-state logic & output logic
   process(state_reg,go,stop)
   begin
      pulse <= '0';
      case state_reg is
         when idle =>
            if go='1' then
               state_next <= delay1;
            else
               state_next <= idle;
            end if;
         when delay1 =>
            if stop='1' then
               state_next <=idle;
            else
               state_next <=delay2;
            end if;
            pulse <= '1';
         when delay2 =>
            if stop='1' then
               state_next <=idle;
            else
```

```
                state_next <=delay3;
            end if;
            pulse <= '1';
        when delay3 =>
            if stop='1' then
                state_next <=idle;
            else
                state_next <=delay4;
            end if;
            pulse <= '1';
        when delay4 =>
            if stop='1' then
                state_next <=idle;
            else
                state_next <=delay5;
            end if;
            pulse <= '1';
        when delay5 =>
            state_next <=idle;
            pulse <= '1';
    end case;
end process;
end fsm_arch;
```

12.2.2 Regular sequential circuit implementation

We can also implement the pulse generator as a regular sequential circuit. It can be considered a mod-5 counter with a special control circuit to enable or disable the counting. To accommodate the generation of a single pulse, an additional FF is needed to flag whether the counter is active or idle. The VHDL code is shown in Listing 12.2.

Listing 12.2 Regular sequential circuit implementation of a one-shot pulse generator

```
architecture regular_seq_arch of pulse_5clk is
   constant P_WIDTH: natural:= 5;
   signal c_reg, c_next: unsigned(3 downto 0);
   signal flag_reg, flag_next: std_logic;
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         c_reg <= (others=>'0');
         flag_reg <= '0';
      elsif (clk'event and clk='1') then
         c_reg <= c_next;
         flag_reg <= flag_next;
      end if;
   end process;
   -- next-state logic
   process(c_reg,flag_reg,go,stop)
   begin
      c_next <= c_reg;
```

```
      flag_next <= flag_reg;
      if (flag_reg='0') and (go='1') then
         flag_next <= '1';
         c_next <= (others=>'0');
      elsif (flag_reg='1') and
            ((c_reg=P_WIDTH-1) or (stop='1')) then
         flag_next <= '0';
      elsif (flag_reg='1') then
         c_next <= c_reg + 1;
      end if ;
   end process;
   -- output logic
   pulse <= '1' when flag_reg='1' else '0';
end regular_seq_arch;
```

There are two registers. The `c_reg` register is used for the counter, and the `flag_reg` register indicates whether the counter is active. The critical part of the description is the if statement of the next-state logic. The first condition, `(flag_reg='0') and (go='1')`, indicates that the counter is currently idle and the go signal is asserted. Under this condition, the flag is asserted and the counter enters the active counting state at the next rising edge of the clock. The second condition indicates that the counter reaches 5 or the `stop` signal is asserted and the counting should stop. The last condition indicates that the counter is in the active state and should keep on counting.

In this code, the `flag_reg` register functions as some sort of state register to keep track of the current condition of the circuit. The state transitions are implicitly embedded in the if statement of the next-state logic.

12.2.3 Implementation using RT methodology

The RT methodology can separate the regular and random logic, and the ASMD chart is shown in Figure 12.2. Two states in the chart indicate whether the counter is active, and the arcs show the transitions under various conditions. The RT operation in the `delay` state specifies the desired increment of the counter. Following the ASMD chart, we can easily derive the VHDL code, as shown in Listing 12.3.

Listing 12.3 FSMD implementation of a one-shot pulse generator

```
architecture fsmd_arch of pulse_5clk is
   constant P_WIDTH: natural:= 5;
   type fsmd_state_type is (idle, delay);
   signal state_reg, state_next: fsmd_state_type;
   signal c_reg, c_next: unsigned(3 downto 0);
begin
   -- state and data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         c_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         c_reg <= c_next;
      end if;
```

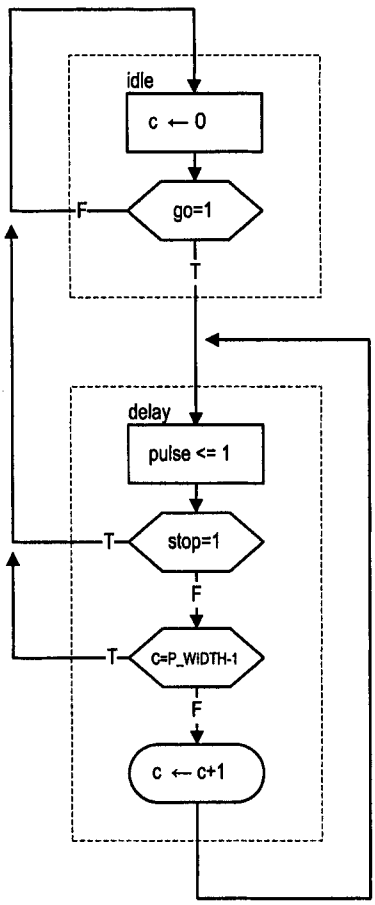


Figure 12.2 ASMD chart of a one-shot pulse generator.

```
    end process;
    -- next-state logic & data path functional units/routing
    process(state_reg,go,stop,c_reg)
    begin
       pulse <= '0';
       c_next <= c_reg;
       case state_reg is
          when idle =>
             if go='1' then
                state_next <= delay;
             else
                state_next <= idle;
             end if;
             c_next <= (others=>'0');
          when delay =>
             if stop='1' then
                state_next <=idle;
             else
                if (c_reg=P_WIDTH-1) then
                   state_next <=idle;
                else
                   state_next <=delay;
                   c_next <= c_reg + 1;
                end if;
             end if;
             pulse <= '1';
       end case;
    end process;
end fsmd_arch;
```

12.2.4 Comparison

The pulse generator example shows that we can use an FSM to emulate a regular sequential circuit, and vice versa. However, the emulation is cumbersome and convolved, and is only possible for a small design. On the other hand, the RT methodology can capture the essence of both regular and random logic, and the description is simple, flexible, clear and informative. That is why it is such a powerful methodology.

To further illustrate the capability of the RT methodology, let us consider an expanded programmable one-shot pulse generator. In this circuit, the width of the pulse can be programmed between 1 and 7. The "programming" is done as follows:

- The go and stop signals are asserted at the same time to indicate the beginning of the program mode.
- The desired value is shifted in via the go signal in the next three clock cycles.

With the RT methodology, we can easily incorporate the extension into the ASMD chart, as shown in Figure 12.3. The corresponding VHDL code is shown in Listing 12.4.

Listing 12.4 Programmable one-shot pulse generator

```
architecture prog_arch of pulse_5clk is
   type fsmd_state_type is (idle, delay, sh1, sh2, sh3);
   signal state_reg, state_next: fsmd_state_type;
   signal c_reg, c_next: unsigned(2 downto 0);
```

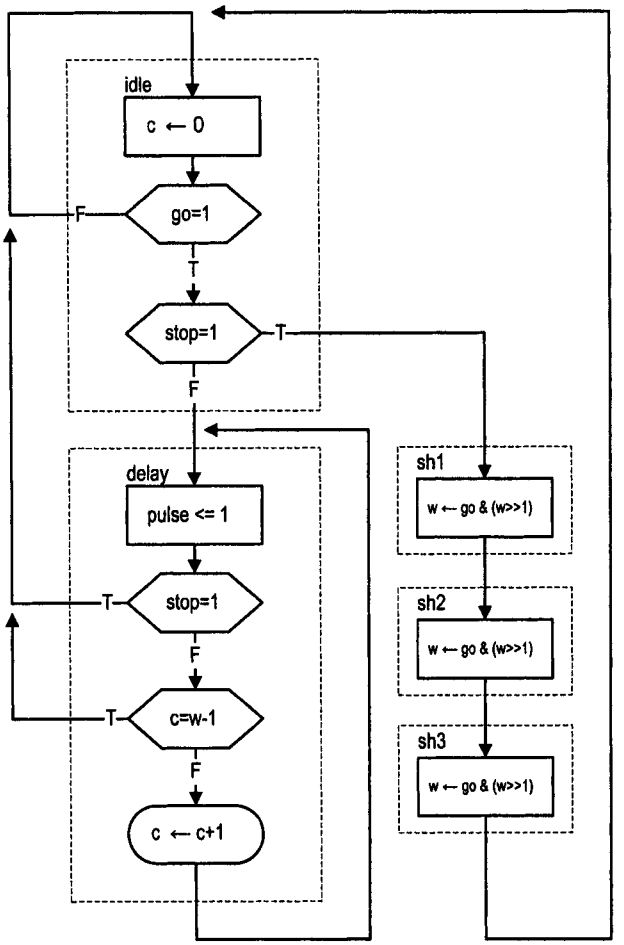


Figure 12.3 ASMD chart of a programmable one-shot pulse generator.

```
5    signal w_reg, w_next: unsigned(2 downto 0);
  begin
     -- state and data registers
     process(clk,reset)
     begin
10      if (reset='1') then
           state_reg <= idle;
           c_reg <= (others=>'0');
           w_reg <= "101";  -- default 5-cycle delay
        elsif (clk'event and clk='1') then
15         state_reg <= state_next;
           c_reg <= c_next;
           w_reg <= w_next;
        end if;
     end process;
20   -- next-state logic & data path functional units/routing
     process(state_reg,go,stop,c_reg,w_reg)
     begin
        pulse <= '0';
        c_next <= c_reg;
25      w_next <= w_reg;
        case state_reg is
           when idle =>
              if go='1' then
                 if stop='1' then
30                  state_next <= sh1;
                 else
                    state_next <= delay;
                 end if;
              else
35               state_next <= idle;
              end if;
              c_next <= (others=>'0');
           when delay =>
              if stop='1' then
40               state_next <=idle;
              else
                 if (c_reg=w_reg-1) then
                    state_next <=idle;
                 else
45                  c_next <= c_reg + 1;
                    state_next <=delay;
                 end if;
              end if;
              pulse <= '1';
50         when sh1 =>
              w_next <= go & w_reg(2 downto 1);
              state_next <= sh2;
           when sh2 =>
              w_next <= go & w_reg(2 downto 1);
55            state_next <= sh3;
           when sh3 =>
              w_next <= go & w_reg(2 downto 1);
```

```
            state_next <= idle;
        end case;
    end process;
end prog_arch;
```

While we can implement the extended pulse generator as a pure FSM circuit or a pure regular sequential circuit in theory, the emulation becomes very involved and error-prone. It will require lots of effort to derive the code.

12.3 SRAM CONTROLLER

Random access memory (RAM) provides massive storage for digital systems. It is constructed as a two-dimensional array of memory cells. A cell is designed and optimized at the transistor level to achieve maximal efficiency. Since the silicon real estate is the primary concern, a memory cell is kept as simple as possible. Its control is level sensitive and uses no clock signal. To incorporate a RAM device into a synchronous digital system, we need a special circuit, known as a *memory controller*, to act as an interface to the synchronous system. Design of the memory controller illustrates control of a clockless subsystem.

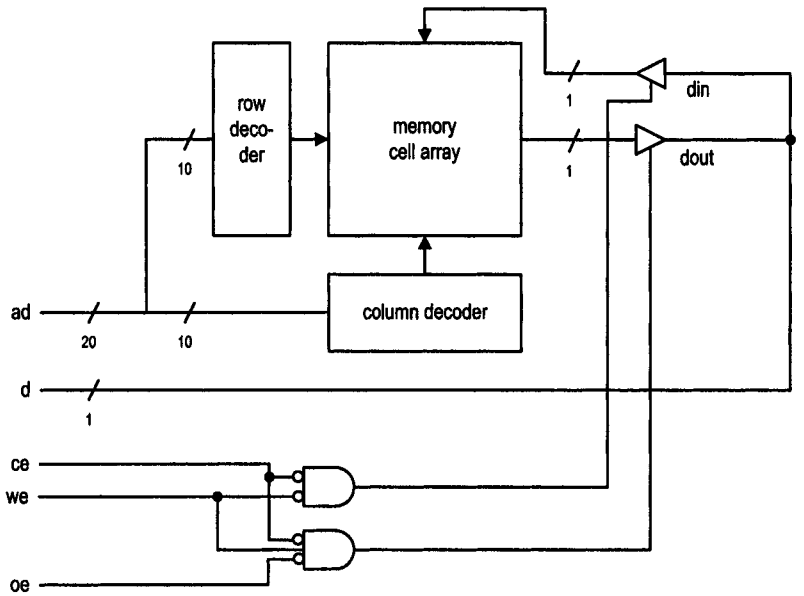
12.3.1 Overview of SRAM

RAM is organized as a two-dimensional array of memory cells with special decoding and multiplexing circuits. The block diagram of a typical 2^{20}-by-1 static RAM (SRAM) is shown in Figure 12.4(a). It contains a 2^{10}-by-2^{10} cell array, two 10-to-2^{10} decoders and an I/O control circuit. The I/O of the SRAM includes a 20-bit address signal, ad, a 1-bit bidirectional data signal, d, and three control signals, ce, we and oe. The ad signal is split and connected to two decoders, which, in turn, enable the cell of the specified location. The three control signals are used to control SRAM operation. The chip select signal, cs, specifies whether to enable the SRAM. The output enable signal, oe, and the write enable signal, we, choose between write and read modes and control the direction of data flow. The function table is shown in Figure 12.4(b). Note that these signals are active low.

Because of the lack of a clock signal, SRAM timing is quite involved. A set of minimum and maximum timing constraints has to be satisfied to ensure proper operation. We first examine the timing of a read operation. There are two methods to read data. In the first method, both the oe and cs signals are already activated (i.e., '0') and the address signal is used to access the desired data. It is known as an *address-controlled* read cycle and the timing diagram is shown in Figure 12.5(a). In the second method, the address signal is already stable and the cs signal already activated, and the oe signal is used to initiate the read operation. It is known as an *oe-controlled* read cycle, and the timing diagram is shown in Figure 12.5(b). Note the activities of the tri-state data bus when the oe signal is activated and deactivated.

The relevant timing parameters associated with a read cycle are:

- T_{aa}: address access time, the required time to obtain stable output data after an address change. It is somewhat like the propagation delay of the read operation and is used to characterize the speed of an SRAM, as in "50-ns SRAM."
- T_{oh}: output hold from address change time, the time that the output data remains valid after the address changes. This should not be confused with the hold time of an edge-triggered FF, which is a constraint for the d input.

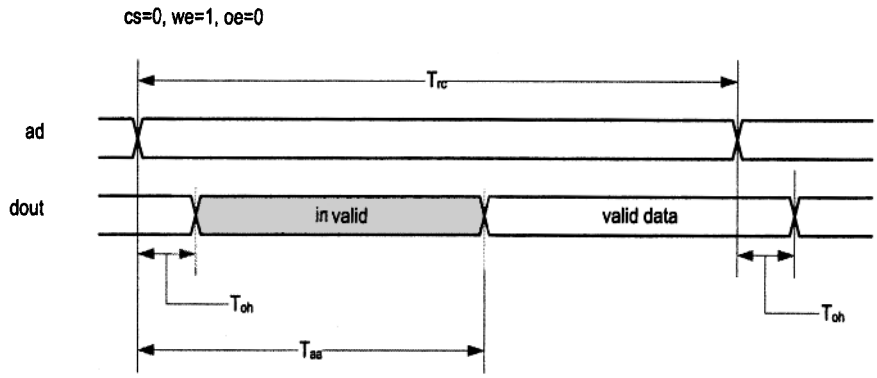


(a) Block diagram

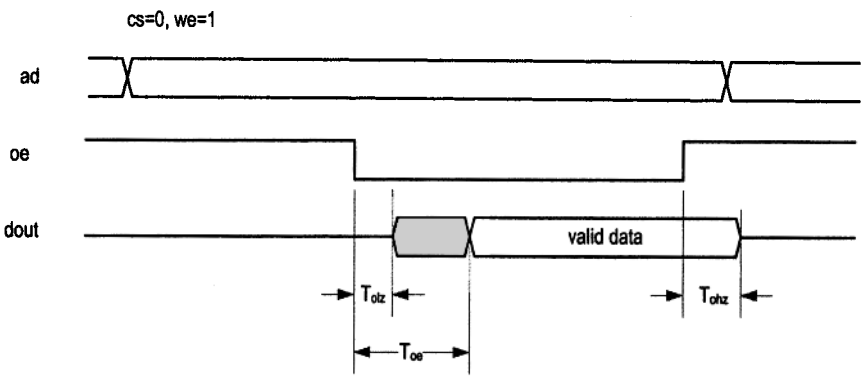
ce	we	oe	**Operation**	**Data pin** d
1	-	-	no operation	Z
0	0	-	write	data in
0	1	0	read	data out
0	1	1	no operation	Z

(b) Function table

Figure 12.4 Block diagram and functional table of a 2^{20}-by-1 SRAM.



(a) Address-controlled read cycle



(b) oe-controlled read cycle

Figure 12.5 Timing diagrams of an SRAM read cycle.

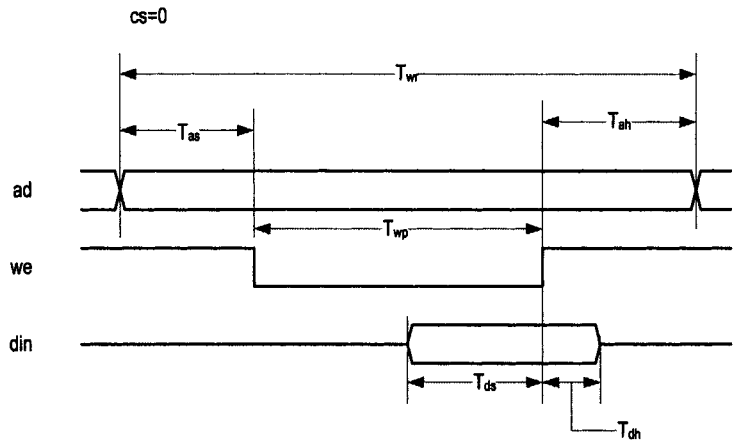


Figure 12.6 Timing diagram of an SRAM write cycle.

- T_{olz}: output enable to output in low-impedance time, the time for the tri-state buffer to leave from the high-impedance state after oe is activated. Note that even when the output is no longer in the high-impedance state, the data is still invalid.
- T_{oe}: output enable to output valid time, the time required to obtain valid data after oe is activated.
- T_{ohz}: output to Z time, the time for the tri-state buffer to enter the high-impedance state.
- T_{rc}: read cycle time, the minimal elapsed time between two read operations. It is about the same as T_{aa} for SRAM.

The write cycle is more complex. The timing diagram of a write cycle is shown in Figure 12.6. The key to understanding the write cycle timing is the assertion of the we signal, which latches the input data into the designated memory cell and plays a key role in the write operation. There are three major constraints:

- To latch data into the designated memory cell, the we signal must be activated (i.e., being '0') for a certain amount of time. This is specified by T_{wp}.
- The address needs to be stable for the entire write operation. Actually, it must be stable before we is activated and remain stable for a small amount of time after we is deactivated. The two time intervals are specified by T_{as} and T_{ah}.
- The input data must be stable in a small window when it is latched. The latch operation occurs at the edge when we transits from '0' to '1'. The input data has to be stable before and after the edge for a small amount of time. The two time intervals are specified by T_{ds} and T_{dh}. This constraint is somewhat like the constraint imposed on the d signal of a D FF at the rising edge of the clock.

These timing parameters are formally defined as follows:

- T_{wp}: write pulse width, the minimal time that the we signal must be activated.
- T_{as}: address setup time, the minimal time that the address must be stable before we is activated.
- T_{ah}: address hold time, the minimal time that the address must be stable after we is deactivated.

Table 12.1 Timing parameters of two SRAMs

Parameter	120-ns SRAM	20-ns SRAM
T_{aa} (max)	120 ns	20 ns
T_{oh} (min)	10 ns	3 ns
T_{olz} (min)	10 ns	0 ns
T_{oe} (max)	80 ns	9 ns
T_{ohz} (max)	40 ns	9 ns
T_{rc} (min)	120 ns	20 ns
T_{wp} (min)	70 ns	12 ns
T_{as} (min)	20 ns	0 ns
T_{ah} (min)	5 ns	0 ns
T_{ds} (min)	35 ns	1 ns
T_{dh} (min)	5 ns	0 ns
T_{wr} (min)	120 ns	20 ns

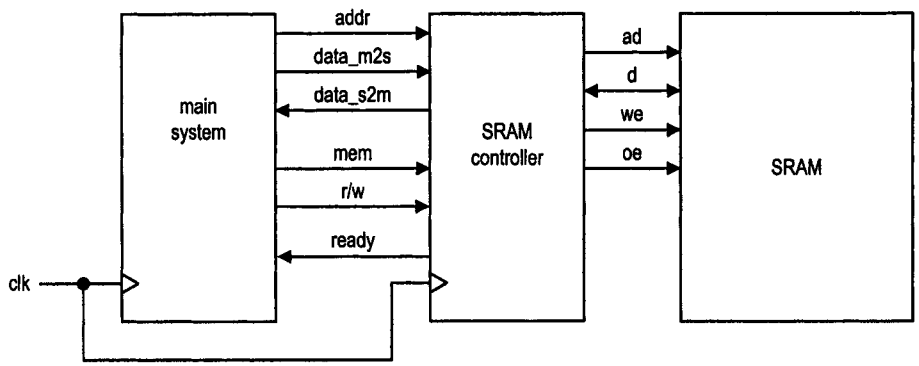


Figure 12.7 Role of an SRAM controller.

- T_{ds}: data setup time, the minimal time that data must be stable before the latching edge (the edge in which `we` moves from '0' to '1').
- T_{dh}: data hold time, the minimal time that data must be stable after the latching edge.
- T_{wr}: write cycle time, the minimal elapsed time between two write operations.

While there has been little change in the basic SRAM architecture over the years, its capacity and speed have improved significantly. The address access time (T_{aa}) can range from a few nanoseconds to several hundred nanoseconds. The typical timing parameters of an older, slow 120-ns SRAM and a more recent 20-ns SRAM are shown in Table 12.1.

12.3.2 Block diagram of an SRAM controller

The purpose of a memory controller is to interface the clockless memory and a synchronous system. The role of an SRAM controller is shown in Figure 12.7. It takes command from the main system and generates proper signals to store data into or retrieve data from the SRAM. The main system is a synchronous system. There are two command signals, `mem` and `rw`, and one status signal, `ready`. The main system activates the `mem` signal when a

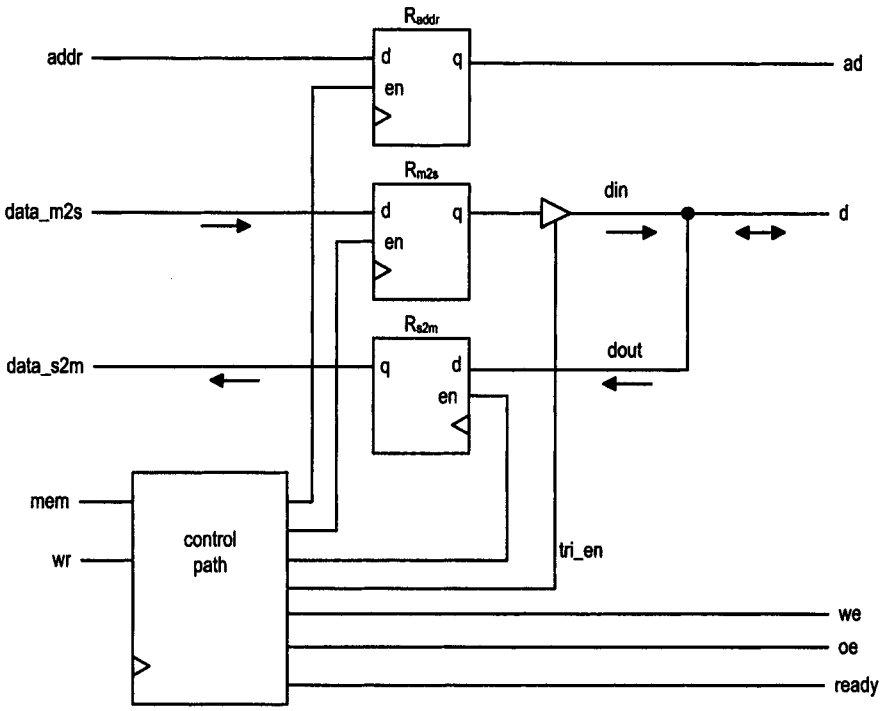


Figure 12.8 Block diagram of an SRAM controller.

memory operation is required and uses `rw` to specify the type of operation ('0' for write and '1' for read). The SRAM controller uses the `ready` signal to indicate whether it is ready for the operation. The `addr` signal is the address used to indicate the location of the memory. The `data_m2s` and `data_s2m` signals are the data transferred from the main system to the SRAM and from the SRAM to the main system respectively.

The main system treats the memory operation as a synchronous operation. For a write operation, it activates `mem`, makes `rw` '0', and places the address on `addr` and data on `data_m2s` for one clock cycle. At the rising edge of the clock, this information will be sampled by the SRAM controller, which, in turn, initiates an SRAM write cycle and generates proper control signals. It may take several clock cycles to complete an operation. For a read operation, the main system activates `mem`, makes `rw` '1', and places the address on `addr` for one clock cycle. Again, this information will be sampled by the SRAM controller at the rising edge of the clock, and an SRAM read cycle is initiated. After a predetermined number of clock cycles, the SRAM controller will put the data on `data_s2m` and make the data available to the main system.

Note that the main system and memory controller are controlled by the same clock. From the main system's point of view, the memory operation is completely synchronous. The combined memory controller and SRAM function somewhat like the register file of Section 9.3.1. However, whereas accessing a location in a register file can be done in one clock cycle, it takes many clock cycles to complete an SRAM read or write operation.

The block diagram of the SRAM controller is shown in Figure 12.8. The data path contains three registers, R_{addr}, R_{m2s} and R_{s2m}, which are used to store the address, the data from the main system to the SRAM, and the data from the SRAM to the main system

respectively. Since the data input of the SRAM is bidirectional, a tri-state buffer is used to avoid conflict. The output from the register R_{m2s} will be placed in the data line, d, when the tri-state buffer is enabled.

The control path coordinates the overall SRAM access and generates the control signals, which include the we and oe signals of the SRAM and the enable signals of tri-state buffer and registers in the data path. There are several requirements for these control signals. First, the signals must be activated in the order specified in the read and write cycles. Second, the signals must meet various timing constraints of the SRAM. Finally, the signals need to ensure that there is no conflict (i.e., fighting) on the bidirectional data line.

12.3.3 Control path of an SRAM controller

We design the control path in two steps:

- Derive a sketch of an FSM according to the activities in read and write cycles.
- Refine the FSM with the actual SRAM timing parameters and clock period.

In the first step, we derive a sketch of an FSM that can activate and deactivate various signals in the desired order. This can be done by dividing the read or write cycles into multiple parts according to the activities of the signals and assigning a state for each part. For example, the write cycle can be divided into five parts, as shown in Figure 12.9(a).

A segment of an FSM can be constructed accordingly, as shown in Figure 12.10(a). We assume that the address and data are stored into the registers before the FSM moves to the s1 state. The data can be placed on the bidirectional line by activating the tri_en signal. The task of the FSM is essentially to generate two output signals, we and tri_en, in the following order: "10", "00", "01", "11" and "10".

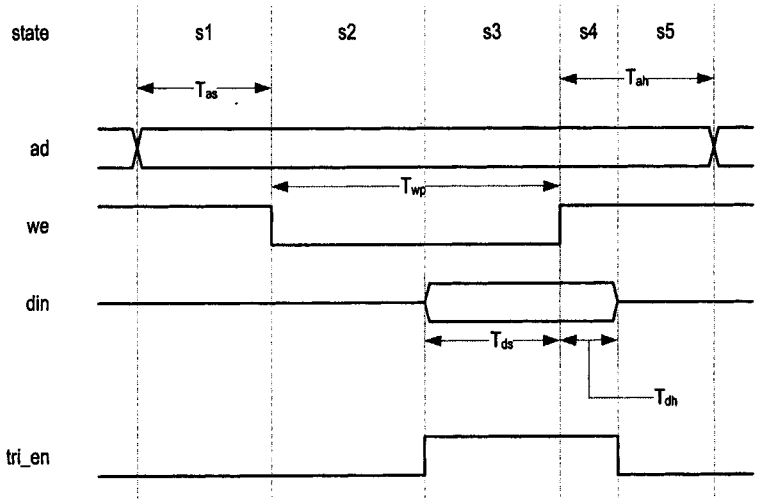
Closer examination of the SRAM's timing specifications can help us to simplify the FSM. For example, T_{wp} is normally much larger than T_{ds} in most SRAMs, and there is no harm in placing data on the din line earlier. Thus, we can merge the s2 and s3 states into a single state. Also, since there in no constraint specified between the order of deactivation of address and data, we can merge the s4 and s5 states. The revised division and FSM segment are shown in Figures 12.9(b) and 12.10(b) respectively.

There are two issues with the initial sketch. First, the length of a state in the FSM corresponds to the period of the clock signal. The period must be large enough to accommodate the most strenuous timing parameter. Since T_{wp} is much larger than other parameters, the time allocated to T_{as} and T_{dh} in the ss1 and ss3 states are unnecessarily inflated. Second, in a practical design, the memory controller is usually a part of a larger system, and the clock rate is determined by the main system. The memory controller cannot have a separate clock and must work with the system clock.

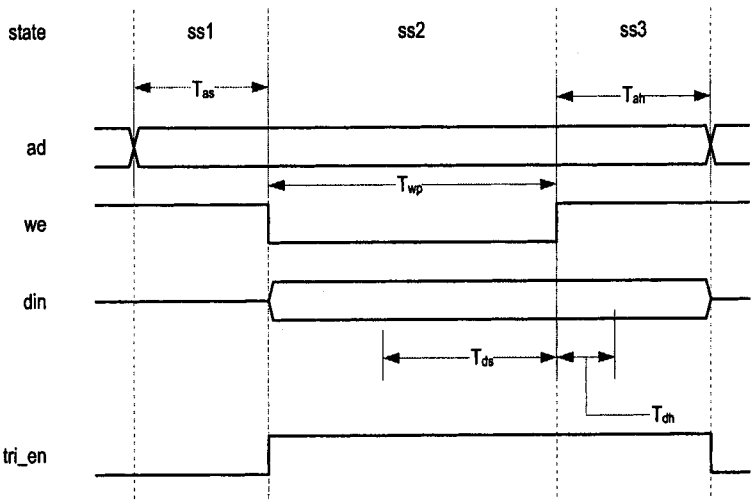
In the second step, we refine the FSM according to the system clock period and SRAM timing parameters. The SRAM's access time and the main system's clock rate are the two key factors in the final design of the control path. The following examples illustrate the design and relevant issues for a slow SRAM and a fast SRAM.

Control path for a slow SRAM The term *slow* here means that the SRAM's address access time (T_{aa}) is relatively large to the main system's clock period. For example, if we assume that the main system's clock period is 25 ns (i.e., the clock rate is 40 MHz), the 120-ns SRAM shown in Table 12.1 will be considered as a slow SRAM to this system since it takes about five clock cycles to complete a memory operation.

Because of the slow SRAM speed, it takes five (i.e., $\lceil \frac{120}{25} \rceil$) clock cycles to cover T_{aa} and three (i.e., $\lceil \frac{70}{25} \rceil$) cycles to cover T_{wp} We need to use multiple states in the FSM to



(a) Five-state division



(b) Three-state division

Figure 12.9 Divisions of a write cycle.

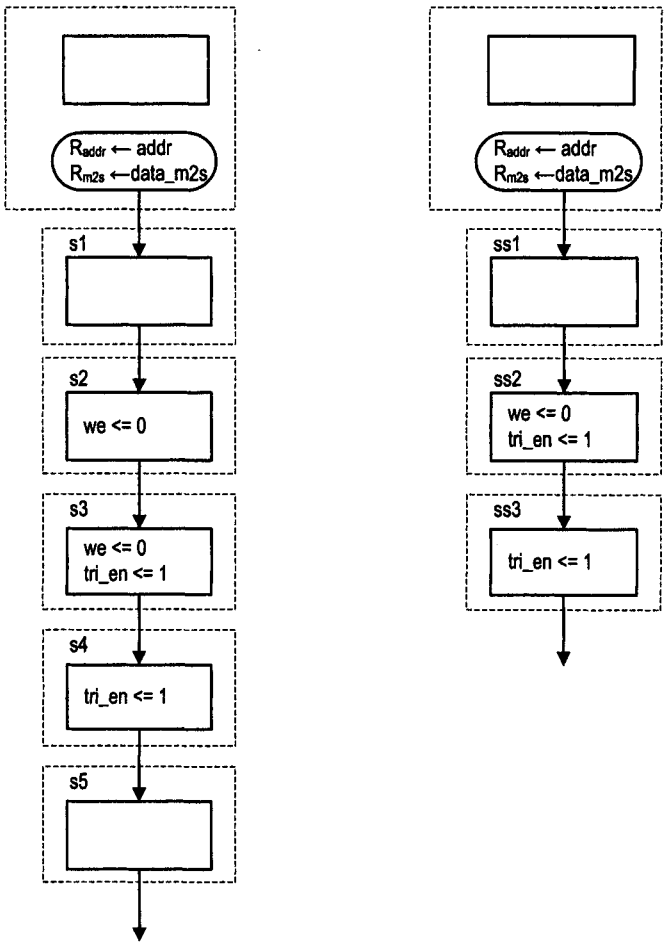


Figure 12.10 FSM segments for a write cycle.

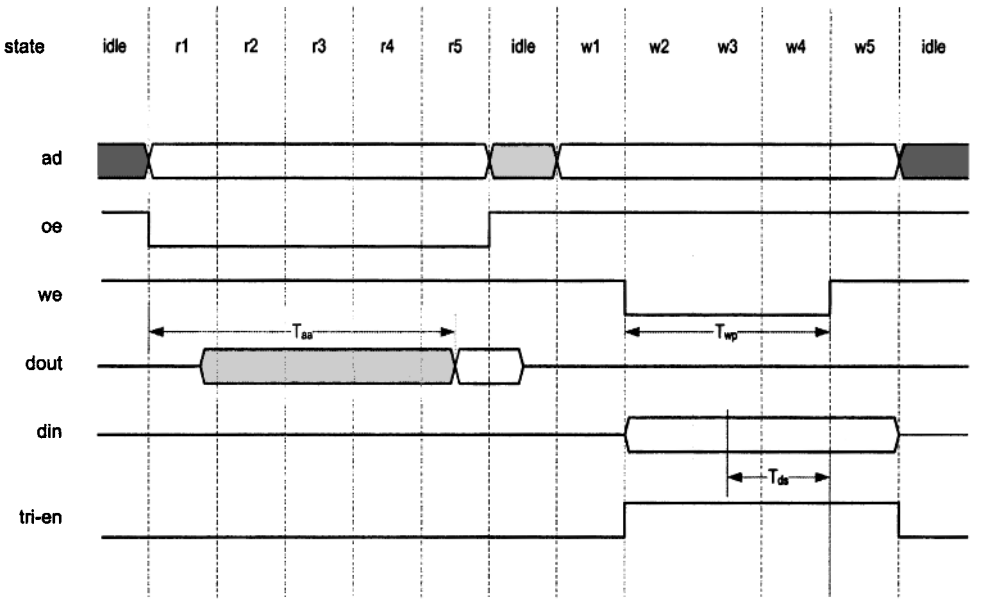


Figure 12.11 Division of read and write cycles of a slow SRAM.

accommodate the timing requirement. Figure 12.11 shows the division in the write and read cycles. An extra clock cycle, which represents the `idle` state, is inserted between the two operations. A read or write operation takes six clock cycles (i.e., 150 ns) to complete. The periods include one for the `idle` state and five for a read or write cycle. We can do a quick check on the SRAM timing parameters:

- T_{aa}: 120 ns < 125 ns (5*25 ns)
- T_{wp}: 70 ns < 75 ns (3*25 ns)
- T_{as}: 20 ns < 25 ns
- T_{ah}: 5 ns < 25 ns
- T_{ds}: 35 ns < 75 ns (3*25 ns)
- T_{dh}: 5 ns < 25 ns

It is clear that all timing parameters are satisfied and there is a margin of at least 5 ns.

The quick check is based on an ideal FSMD. To obtain more detailed timing information, we also need to consider the various propagation delays introduced by the data path and control path of the memory controller. For example, the `oe` signal is disabled in the end of the `r5` state and the data on the `d` line is sampled and stored when the FSM moves from the `r5` state to the `idle` state. We must perform a detailed timing analysis to ensure that there is no setup or hold time violation for the R_{s2m} register. The detailed timing diagram is shown in Figure 12.12. The read operation progresses as follows. At t_1, the FSM moves to the `r1` state. After the T_{ctrl} delay (at t_2), the `oe` signal is activated and the SRAM starts the read operation. After T_{aa} (at t_3), the data is available. At t_4, the FSM moves from the `r5` state to the `idle` state, and the memory controller samples and stores the data into the R_{s2m} register. After the T_{ctrl} delay (at t_5), the `oe` signal is deactivated. The data line (`dout`) of the SRAM returns to the high-impedance state after the T_{oz} delay (at t_6).

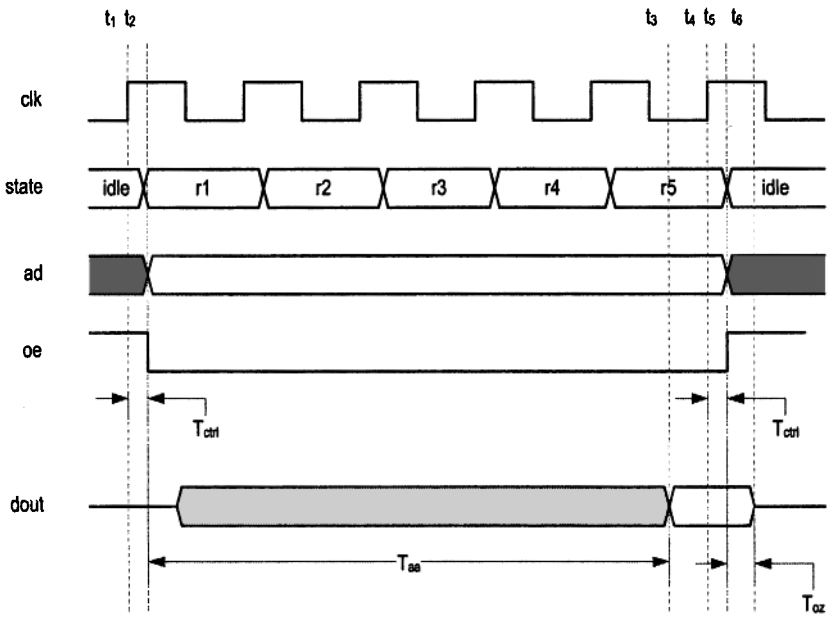


Figure 12.12 Detailed timing diagram of the read cycle.

To avoid the setup time violation, the data has to be stable before T_{setup} of the rising edge of the clock; that is,

$$T_{setup} < 5T_c - T_{ctrl} - T_{aa}$$

We can use the look-ahead output buffer to minimize T_{ctrl} and reduce it to the clock-to-q delay (T_{cq}) of the FF. With a 25-ns clock and 120-ns SRAM, the inequality becomes

$$T_{setup} + T_{cq} < 5 \text{ ns}$$

This condition can be met by most of today's device technology.

To avoid the hold time violation, the data has to be stable after T_{hold} of the rising edge of the clock:

$$T_{hold} < T_{ctrl} + T_{oz}$$

Since T_{ctrl} is T_{cq}, this condition can be easily satisfied.

Other timing requirements, such as the data bus conflict, the exact timing on the SRAM's T_{ds} and T_{dh} requirement, can be analyzed in a similar way. Because of the relatively large safety margin of this design, the initial checking should still be valid.

Following the division and signal activation, we can derive the ASMD chart accordingly, as shown in Figure 12.13. The VHDL code of the complete memory controller is shown in Listing 12.5. It includes both the data path and control path. Note that we use the look-ahead output buffer scheme for the `we`, `oe` and `tri_en` signals to ensure that the signals are glitch-free and to minimize the clock-to-output delay.

Listing 12.5 Memory controller of a slow SRAM

```
library ieee;
use ieee.std_logic_1164.all;
entity sram_ctrl is
```

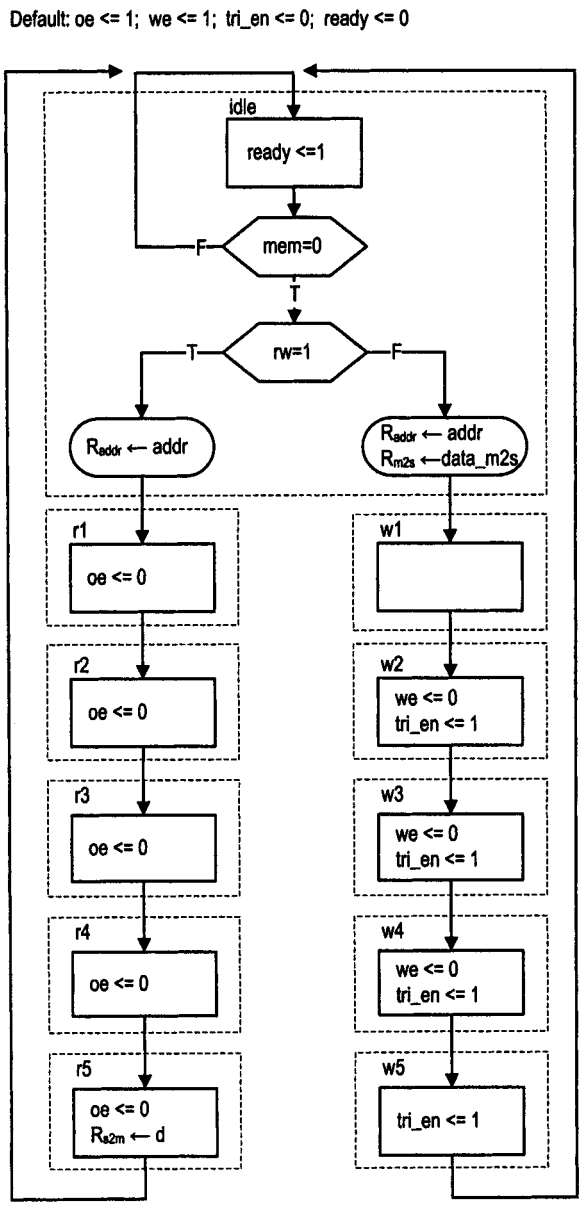


Figure 12.13 ASMD chart for a slow SRAM controller.

```
    port(
       clk, reset: in std_logic;
       mem: in std_logic;
       rw: in std_logic;
       addr: in std_logic_vector(19 downto 0);
       data_m2s: in std_logic;
       we, oe: out std_logic;
       ready: out std_logic;
       data_s2m: out std_logic;
       d: inout std_logic;
       ad: out std_logic_vector(19 downto 0)
  );
end sram_ctrl;

architecture arch of sram_ctrl is
   type state_type is
      (idle, r1, r2, r3, r4, r5, w1, w2, w3, w4, w5);
   signal state_reg, state_next: state_type;
   signal data_m2s_reg, data_m2s_next: std_logic;
   signal data_s2m_reg, data_s2m_next: std_logic;
   signal addr_reg, addr_next: std_logic_vector(19 downto 0);
   signal tri_en_buf, we_buf, oe_buf: std_logic;
   signal tri_en_reg, we_reg, oe_reg: std_logic;
begin
   -- state & data registers
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= idle;
         addr_reg <= (others=>'0');
         data_m2s_reg <= '0';
         data_s2m_reg <= '0';
         tri_en_reg <= '0';
         we_reg <= '1';
         oe_reg <='1';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         addr_reg <= addr_next;
         data_m2s_reg <= data_m2s_next;
         data_s2m_reg <= data_s2m_next;
         tri_en_reg <= tri_en_buf;
         we_reg <= we_buf;
         oe_reg <= oe_buf;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
   process(state_reg,mem,rw,d,addr,data_m2s,
           data_m2s_reg,data_s2m_reg,addr_reg)
   begin
      addr_next <= addr_reg;
      data_m2s_next <= data_m2s_reg;
      data_s2m_next <= data_s2m_reg;
      ready <= '0';
```

```
      case state_reg is
         when idle =>
            if mem='0' then
               state_next <= idle;
            else
               if rw='0' then --write
                  state_next <= w1;
                  addr_next <= addr;
                  data_m2s_next <= data_m2s;
               else -- read
                  state_next <= r1;
                  addr_next <= addr;
               end if;
            end if;
            ready <= '1';
         when w1 =>
            state_next <= w2;
         when w2 =>
            state_next <= w3;
         when w3 =>
            state_next <= w4;
         when w4 =>
            state_next <= w5;
         when w5 =>
            state_next <= idle;
         when r1 =>
            state_next <= r2;
         when r2 =>
            state_next <= r3;
         when r3 =>
            state_next <= r4;
         when r4 =>
            state_next <= r5;
         when r5 =>
            state_next <= idle;
            data_s2m_next <= d;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      tri_en_buf <='0';
      oe_buf <= '1';
      we_buf<= '1';
      case state_next is
         when idle =>
         when w1 =>
         when w2 =>
            we_buf <= '0';
            tri_en_buf <= '1';
         when w3 =>
            we_buf <= '0';
            tri_en_buf <= '1';
```

```
110          when w4 =>
                we_buf <= '0';
                tri_en_buf <= '1';
             when w5 =>
                tri_en_buf <= '1';
115          when r1 =>
                oe_buf <= '0';
             when r2 =>
                oe_buf <= '0';
             when r3 =>
120             oe_buf <= '0';
             when r4 =>
                oe_buf <= '0';
             when r5 =>
                oe_buf <= '0';
125       end case;
      end process;
      -- output
      we <= we_reg;
      oe <= oe_reg;
130   ad <= addr_reg;
      d <= data_m2s_reg when tri_en_reg ='1' else 'Z';
      data_s2m <= data_s2m_reg;
   end arch;
```

Control path for a fast SRAM The major problem with the previous memory system is its speed. Since it takes six clock cycles to read or write a data item from the SRAM, it can be used only if the main system accesses the memory sporadically. One way to improve the memory performance is to use a faster SRAM. For example, we can use the 20-ns SRAM of Table 12.1, whose address access time (T_{aa}) is smaller than the 25-ns clock period of the main system. Figure 12.14 shows the timing of one possible design, in which a read cycle and a write cycle are done in one clock cycle. We can again check the division against the SRAM timing parameters:

- T_{aa}: 20 ns < 25 ns
- T_{wp}: 12 ns < 25 ns
- T_{as}: 0 ns $\leq$ 0 ns
- T_{ah}: 0 ns $\leq$ 0 ns
- T_{ds}: 12 ns < 25 ns
- T_{dh}: 0 ns $\leq$ 0 ns

Although all constraints are satisfied, the timing is very tight. The timing of T_{as}, T_{ah} and T_{dh} just meet the specification and there is no safety margin. The propagation delays of the control path and data path may cause timing violations. We may need to manually fine-tune the propagation delays of various signals to ensure correct operation.

In this design, a read or write operation requires two clock cycles because the FSM must return to the `idle` state after each operation. Since performance is the goal of a fast SRAM controller, it is desirable to perform back-to-back memory operations without returning to the `idle` state. This requires an ad hoc circuit to generate a `we` pulse whose activation time is only a fraction of a clock period and more manual fine-tuning on propagation delays to avoid data bus fighting.

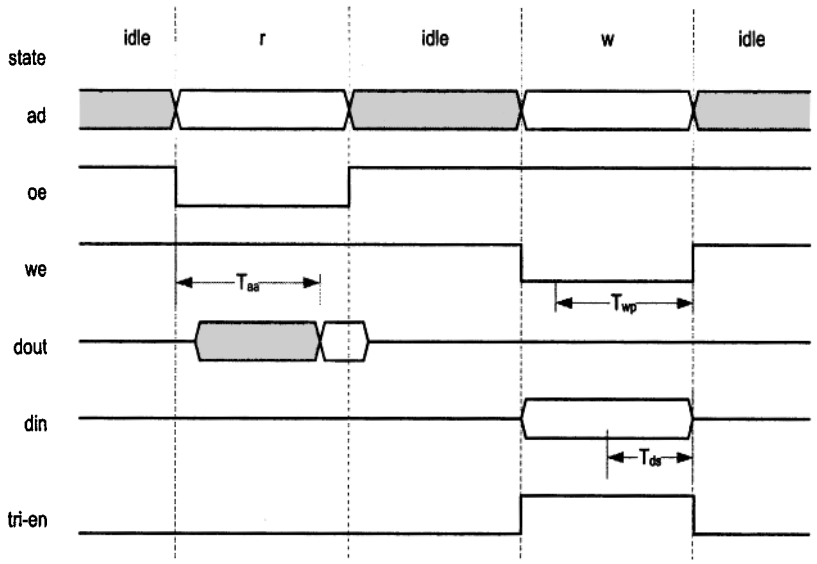


Figure 12.14 Division of read and write cycles of a fast SRAM.

In summary, while it is possible to perform single-clock back-to-back memory operations, the design imposes very strict and tricky timing requirements on the control signals. These requirements are delay-sensitive and cannot be expressed or implemented by a regular FSM. This kind of circuit is not suitable for RT-level synthesis. To implement the controller, we need to manually derive the schematic using cells from the device library and even to manually do the placement and routing. Many device manufacturers have recognized the design difficulty and incorporated the memory controller into a memory chip. This kind of device is known as *synchronous memory*. Since the main system only needs to issue commands, place address and data, or retrieve data at rising edges of the clock, this type of device greatly simplifies the memory interface to a synchronous system.

12.4 GCD CIRCUIT

The $\gcd(a, b)$ function returns the greatest common divisor (GCD) of two positive integers, a and b. For example, $\gcd(1, 10)$ is 1 and $\gcd(12, 9)$ is 3. The gcd function can be obtained by using subtraction, which is based on the equation

$$\gcd(a, b) = \begin{cases} a & \text{if } a = b \\ \gcd(a - b, b) & \text{if } a > b \\ \gcd(a, b - a) & \text{if } a < b \end{cases}$$

Assume that `a_in` and `b_in` are positive nonzero integers and their GCD is `r`. The equation can easily be converted into the following pseudocode:

```
a = a_in;
b = b_in;
while (a /= b) {
   if (a > b) then
      a = a - b;
```

```
            else
               b = b - a;
            end if
         }
         r = a;
```

To make the pseudocode more compatible with the ASMD chart, we convert the while loop into a goto statement and use a swap operation to reduce the number of required subtractions. The revised pseudocode becomes

```
         a = a_in;
         b = b_in;
swap:    if (a = b) then
            goto stop;
         else
            if (a < b) then — swap a and b
               a = b;  — assume the two operations
               b = a;  — can be done in parallel
            end if;
            a = a - b;
            goto swap;
         end if;
stop:    r = a;
```

The code first moves the larger value into a and then performs a single subtraction of a - b. This code can easily be converted into an ASMD chart, as shown in Figure 12.15. As the sequential multiplier circuit of Chapter 11, the `start` and `ready` signals are added to interface external systems. The corresponding VHDL code is shown in Listing 12.6.

Listing 12.6 Initial implementation of a GCD circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gcd is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(7 downto 0)
   );
end gcd ;

architecture slow_arch of gcd is
   type state_type is (idle, swap, sub);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
begin
   — state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
```

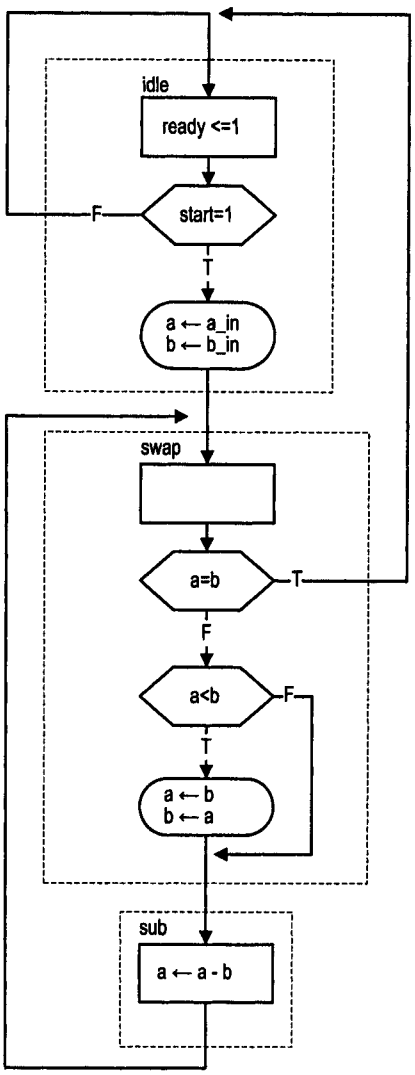


Figure 12.15 ASMD chart of the initial GCD circuit.

```
25          b_reg <= (others=>'0');
         elsif (clk'event and clk='1') then
            state_reg <= state_next;
            a_reg <= a_next;
            b_reg <= b_next;
30       end if;
      end process;
      -- next-state logic & data path functional units/routing
      process(state_reg,a_reg,b_reg,start,a_in,b_in)
      begin
35       a_next <= a_reg;
         b_next <= b_reg;
         case state_reg is
            when idle =>
               if start='1' then
40                a_next <= unsigned(a_in);
                  b_next <= unsigned(b_in);
                  state_next <= swap;
               else
                  state_next <= idle;
45             end if;
            when swap =>
               if (a_reg=b_reg) then
                  state_next <= idle;
               else
50                if (a_reg < b_reg) then
                     a_next <= b_reg;
                     b_next <= a_reg;
                  end if;
                  state_next <= sub;
55             end if;
            when sub =>
               a_next <= a_reg - b_reg;
               state_next <= swap;
            end case;
60    end process;
      -- output
      ready <= '1' when state_reg=idle else '0';
      r <= std_logic_vector(a_reg);
   end slow_arch;
```

As discussed in Section 11.5, one factor in the performance of an FSMD is the number of clock cycles required to complete the computation. In this design, the input values are subtracted successively until the a_reg=b_reg condition is reached. The number of clock cycles required to complete computation of this GCD circuit depends on the input values. It requires more time if only a small value is subtracted each time. The calculation of $\gcd(1, 2^8 - 1)$ represents the worst-case scenario. The loop has to be repeated $2^8 - 1$ times until the two values are equal. For a circuit with an N-bit input, the computation time is on the order of $O(2^N)$, and thus this is not an effective design.

One way to improve the design is to take advantage of the binary number system. For a binary number, we can tell whether it is odd or even by checking the LSB. Based on the

LSBs of two inputs, several simplification rules can be applied in the derivation of the GCD function:

- If both a and b are even, $\gcd(a, b) = 2\gcd(\frac{a}{2}, \frac{b}{2})$.
- If a is odd and b is even, $\gcd(a, b) = \gcd(a, \frac{b}{2})$.
- If a is even and b is odd, $\gcd(a, b) = \gcd(\frac{a}{2}, b)$.

Since the divided-by-2 operation corresponds to shifting right one position, it can be implemented easily in hardware. The previous equation can be extended:

$$\gcd(a,b) = \begin{cases} a & \text{if } a = b \\ 2\gcd(\frac{a}{2}, \frac{b}{2}) & \text{if } a \neq b \text{ and } a, b \text{ even} \\ \gcd(a, \frac{b}{2}) & \text{if } a \neq b \text{ and } a \text{ odd}, b \text{ even} \\ \gcd(\frac{a}{2}, b) & \text{if } a \neq b \text{ and } a \text{ even}, b \text{ odd} \\ \gcd(a-b, b) & \text{if } a > b \text{ and } a, b \text{ odd} \\ \gcd(a, b-a) & \text{if } a < b \text{ and } a, b \text{ odd} \end{cases}$$

To incorporate the new rules into the algorithm, the main issue is how to handle computation of $2\gcd(\frac{a}{2}, \frac{b}{2})$. One way is ignoring the factor 2 in initial iterations and using an additional register, n, to keep track of the number of occurrences in which both operands are even. The final GCD value can be restored by multiplying the initial result by 2^n, which corresponds to shifting the initial result left n positions.

The expanded ASMD chart is shown in Figure 12.16. It has several modifications. In the `swap` state, the LSBs of the `a` and `b` registers are checked. The register is shifted right one position (i.e., divided by 2) if it is even. Furthermore, the `n` register is incremented if both are even. If the `a` and `b` registers are odd, they are compared and, if necessary, swapped, and the FSM moves to the `sub` state. An extra state, labeled `res` (for "restore"), is added to restore the final GCD value. The initial result in `a` is shifted left repeatedly until the `n` counter reaches 0. The corresponding VHDL code is shown in Listing 12.7.

Listing 12.7 More efficient implementation of a GCD circuit

```
architecture fast_arch of gcd is
   type state_type is (idle, swap, sub, res);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
   signal n_reg, n_next: unsigned(2 downto 0);
begin
   -- state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         a_reg <= (others=>'0');
         b_reg <= (others=>'0');
         n_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         a_reg <= a_next;
         b_reg <= b_next;
         n_reg   <= n_next;
      end if;
   end process;
   -- next-state logic & data path functional units/routing
```

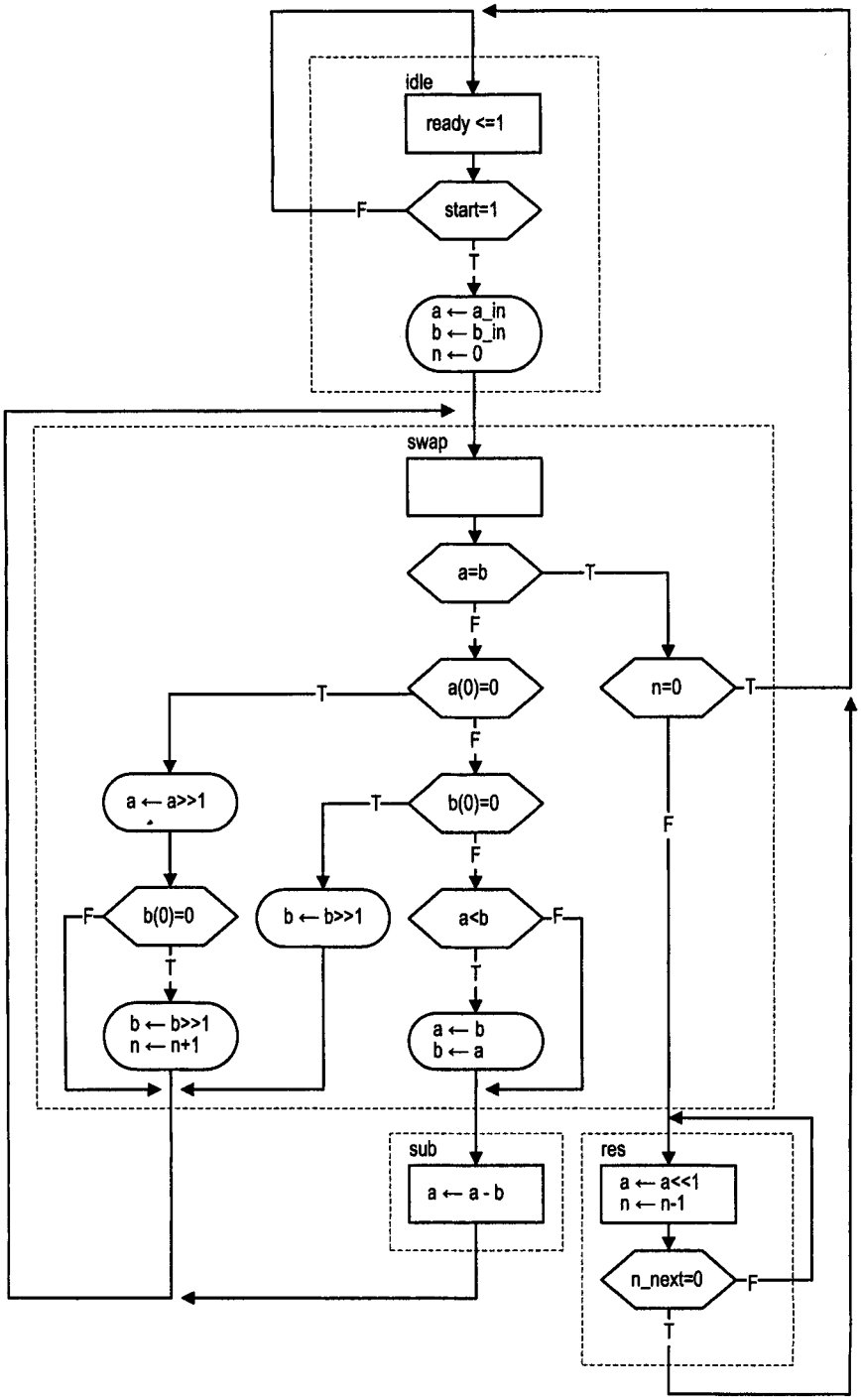


Figure 12.16 ASMD chart of the revised GCD circuit.

```
process(state_reg,a_reg,b_reg,n_reg,start,a_in,b_in,n_next)
begin
   a_next <= a_reg;
   b_next <= b_reg;
   n_next <= n_reg;
   case state_reg is
      when idle =>
         if start='1' then
            a_next <= unsigned(a_in);
            b_next <= unsigned(b_in);
            n_next <= (others=>'0');
            state_next <= swap;
         else
            state_next <= idle;
         end if;
      when swap =>
         if (a_reg=b_reg) then
            if (n_reg=0) then
               state_next <= idle;
            else
               state_next <= res;
            end if;
         else
            if (a_reg(0)='0') then -- a_reg even
               a_next <= '0' & a_reg(7 downto 1);
               if (b_reg(0)='0') then  -- both even
                  b_next <= '0' & b_reg(7 downto 1);
                  n_next <= n_reg + 1;
               end if;
               state_next <= swap;
            else -- a_reg odd
               if (b_reg(0)='0') then  -- b_reg even
                  b_next <= '0' & b_reg(7 downto 1);
                  state_next <= swap;
               else -- both a_reg and b_reg odd
                  if (a_reg < b_reg) then
                     a_next <= b_reg;
                     b_next <= a_reg;
                  end if;
                  state_next <= sub;
               end if;
            end if;
         end if;
      when sub =>
         a_next <= a_reg - b_reg;
         state_next <= swap;
      when res =>
         a_next <= a_reg(6 downto 0) & '0';
         n_next <= n_reg - 1;
         if (n_next=0) then
            state_next <= idle;
         else
            state_next <= res;
```

```
            end if;
        end case;
    end process;
    --output
    ready <= '1' when state_reg=idle else '0';
    r <= std_logic_vector(a_reg);
end fast_arch;
```

Now let us consider the number of clock cycles needed to complete one computation. Assume that the width of the input operand is N bits. The algorithm gradually reduces the values in the a_reg and b_reg until they are equal. In the worst case, there are $2N$ bits to be processed initially. If a value is even, the LSB is shifted out and thus the number of bits is reduced by 1. If both values are odd, a subtraction is performed and the difference is even, and the number of bits can be reduced by 1 in the next iteration. In the most pessimistic scenario, the $2N$ bits can be processed in $2 * 2N$ iterations, and the required computation time is on the order of $O(N)$, which is much better than the $O(2^N)$ of the original algorithm.

Because of the flexibility of hardware implementation, it is possible to invest extra hardware resources to improve the performance. For example, instead of handling the data bit by bit in the swap and res states, we can use more sophisticated combinational circuits to process the data in parallel. In the swap state, the circuit checks and shifts out the trailing 0's of a and b. In the res state, a shift-left barrel shifter restores the final result in a single step. The revised VHDL code is shown in Listing 12.8.

Listing 12.8 Performance-oriented implementation of a GCD circuit

```
architecture fastest_arch of gcd is
    type state_type is (idle, swap, sub, res);
    signal state_reg, state_next: state_type;
    signal a_reg, a_next, b_reg, b_next: unsigned(7 downto 0);
    signal n_reg, n_next, a_zero, b_zero: unsigned(2 downto 0);
begin
    -- state & data registers
    process(clk,reset)
    begin
        if reset='1' then
            state_reg <= idle;
            a_reg <= (others=>'0');
            b_reg <= (others=>'0');
            n_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            state_reg <= state_next;
            a_reg <= a_next;
            b_reg <= b_next;
            n_reg   <= n_next;
        end if;
    end process;
    -- next-state logic & data path functional units/routing
    process(state_reg,a_reg,b_reg,n_reg,start,
            a_in,b_in,a_zero,b_zero)
    begin
        a_next <= a_reg;
        b_next <= b_reg;
        n_next <= n_reg;
```

```
        a_zero <= (others=>'0');
30      b_zero <= (others=>'0');
        case state_reg is
           when idle =>
              if start='1' then
                 a_next <= unsigned(a_in);
35               b_next <= unsigned(b_in);
                 n_next <= (others=>'0');
                 state_next <= swap;
              else
                 state_next <= idle;
40            end if;
           when swap =>
              if (a_reg=b_reg) then
                 if (n_reg=0) then
                    state_next <= idle;
45               else
                    state_next <= res;
                 end if;
              else
                 if (a_reg(0)='1' and b_reg(0)='1') then -- swap
50                  if (a_reg < b_reg) then
                       a_next <= b_reg;
                       b_next <= a_reg;
                    end if;
                    state_next <= sub;
55               else
                    -- shift out 0s of a_reg
                    if (a_reg(0)='1') then
                         a_zero <="000";
                    elsif (a_reg(1)='1') then
60                     a_next <= "0" & a_reg(7 downto 1);
                         a_zero <="001";
                    elsif (a_reg(2)='1') then
                       a_next <= "00" & a_reg(7 downto 2);
                         a_zero <="010";
65                  elsif (a_reg(3)='1') then
                       a_next <= "000" & a_reg(7 downto 3);
                         a_zero <="011";
                    elsif (a_reg(4)='1') then
                       a_next <= "0000" & a_reg(7 downto 4);
70                       a_zero <="100";
                    elsif (a_reg(5)='1') then
                       a_next <= "00000" & a_reg(7 downto 5);
                         a_zero <="101";
                    elsif (a_reg(6)='1') then
75                     a_next <= "000000" & a_reg(7 downto 6);
                         a_zero <="110";
                    else  -- a_reg(7)='1'
                       a_next <= "0000000" & a_reg(7);
                         a_zero <="111";
80                  end if;
                    -- shift out 0s of b_reg
```

```
                if (b_reg(0)='1') then
                    b_zero <="000";
                elsif (b_reg(1)='1') then
                   b_next <= "0" & b_reg(7 downto 1);
                    a_zero <="001";
                elsif (b_reg(2)='1') then
                   b_next <= "00" & b_reg(7 downto 2);
                    b_zero <="010";
                elsif (b_reg(3)='1') then
                   b_next <= "000" & b_reg(7 downto 3);
                    b_zero <="011";
                elsif (b_reg(4)='1') then
                   b_next <= "0000" & b_reg(7 downto 4);
                    b_zero <="100";
                elsif (b_reg(5)='1') then
                   b_next <= "00000" & b_reg(7 downto 5);
                    b_zero <="101";
                elsif (b_reg(6)='1') then
                   b_next <= "000000" & b_reg(7 downto 6);
                    b_zero <="110";
                else  -- b_reg(7)='1'
                   b_next <= "0000000" & b_reg(7);
                    b_zero <="111";
                end if;
                -- find common number of 0s
                if (a_zero > b_zero) then
                   n_next <= n_reg + b_zero;
                else
                   n_next <= n_reg + a_zero;
                end if;
                state_next <= swap;
            end if;
        end if;
    when sub =>
        a_next <= a_reg - b_reg;
        state_next <= swap;
    when res =>
        case n_reg is
            when "000" =>
                a_next <= a_reg;
            when "001" => a_next <=
                a_reg(6 downto 0) & '0';
            when "010" =>
                a_next <= a_reg(5 downto 0) & "00";
            when "011" =>
                a_next <= a_reg(4 downto 0) & "000";
            when "100" => a_next <=
                a_reg(3 downto 0) & "0000";
            when "101" =>
                a_next <= a_reg(2 downto 0) & "00000";
            when "110" =>
                a_next <= a_reg(1 downto 0) & "000000";
            when others =>
```

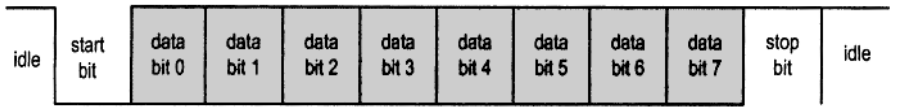


Figure 12.17 Transmission of a byte.

```
                a_next <= a_reg(0) & "0000000";
            end case;
            state_next <= idle;
        end case;
    end process;
    -- output
    ready <= '1' when state_reg=idle else '0';
    r <= std_logic_vector(a_reg);
end fastest_arch;
```

12.5 UART RECEIVER

Universal asynchronous receiver and transmitter (UART) is a scheme that sends bytes of data through a serial line. The transmission of a single byte is shown in Figure 12.17. The serial line is in the '1' state when it is idle. The transmission is started with a *start bit*, which is '0', followed by eight data bits and ended with a *stop bit*, which is '1'. It is also possible to insert an optional parity bit in the end of the data bits to perform error detection. Before the transmission starts, the transmitter and receiver must agree on a set of parameters in advance, which include the baud rate (i.e., number of bits per second), the number of data bits, and use of the parity bit.

The UART transmitter is essentially a shift register that shifts out data bits at a specific rate. Construction of a UART receiver is more involved since no clock information is conveyed through the serial line. The receiver can retrieve the data bits only by using the predetermined parameters. It uses an oversampling scheme to ensure that the data bits are retrieved at the correct point. This scheme utilizes a high-frequency sampling signal to estimate the middle point of a data bit and then retrieve data bits at these points. For example, assume that the sampling rate is 16 times the baud rate (i.e., there are 16 sampling pulses for each bit). The incoming stream can be recovered as follows:

1. When the incoming line becomes '0' (i.e., the beginning of the start bit), initiate the sampling pulse counter.
2. When the counter reaches 7, clear it to 0 and restart. At this point, the incoming signal reaches about a half of the start bit (i.e., the middle point of the start bit).
3. When the counter reaches 15, clear it to 0 and restart. At this point, the incoming signal progresses for one bit and reaches the middle of the first data bit. The data in the serial line should be retrieved and shifted into a register.
4. Repeat Step 3 seven times to retrieve the remaining seven data bits.
5. Repeat Step 3 one more time but without shifting. The incoming signal should reach the middle of the stop bit at this point, and its value should be '1'.

The idea behind this scheme is to use oversampling to overcome the uncertainty of the initiation of the start bit. Even when we don't know the exact onset point of the start bit, it

can be off by at most $\frac{1}{16}$. The subsequent data bit retrievals are off by at most $\frac{1}{16}$ from the middle point as well.

With understanding of the oversampling procedure, we can derive the ASMD chart accordingly. One issue is the creation of sampling pulses. The easiest way is to treat the UART as a separate subsystem that utilizes a clock signal whose frequency is just 16 times that of the baud rate. This approach violates the synchronous design principle and should be avoided. A better alternative is to use a single-clock enable pulse that is synchronized with the system clock, as discussed in Section 9.1.3. Assume that the system clock is 1 MHz and the baud rate is 1200 baud. The frequency of the sampling enable pulse should be $16 * 1200$, which can be obtained by a mod-52 counter (note that $\frac{1{,}000{,}000}{16*2000} = 52$). It can easily be coded in VHDL:

```
process(clk,reset)
begin
   if reset='1' then
      clk16_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      clk16_reg <= clk16_next;
   end if;
end process;
-- next-state/output logic
clk16_next <= (others=>'0') when clk16_reg=51 else
              clk16_reg + 1 ;
s_pulse <= '1' when clk16_reg=0 else '0';
```

The ASMD chart of a simplified UART receiver is shown in Figure 12.18. The chart follows the previous steps and includes three major states, `start`, `data` and `stop`, which represent the processing of the start bit, data bits and stop bit respectively. The `s_pulse` signal is the enable pulse whose frequency is 16 times that of the baud rate. Note that the FSMD stays in the same state unless the `s_pulse` signal is activated. There are two counters, represented by the `s` and `n` registers. The `s` register keeps track of the number of sampling pulses and counts to 7 in the `start` state and to 15 in the `data` and `stop` states. The `n` register keeps track of the number of data bits received in the `data` state. The retrieved bits are shifted into and reassembled in the `b` register. The corresponding VHDL code is shown in Listing 12.9. We assume that the system clock is 1 MHz and the baud rate is 1200 baud.

Listing 12.9 Simplified UART receiver

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity uart_receiver is
   port(
      clk, reset: in std_logic;
      rx: in std_logic;
      ready: out std_logic;
      pout: out std_logic_vector(7 downto 0)
   );
end uart_receiver ;

architecture arch of uart_receiver is
   type state_type is (idle, start, data, stop);
```

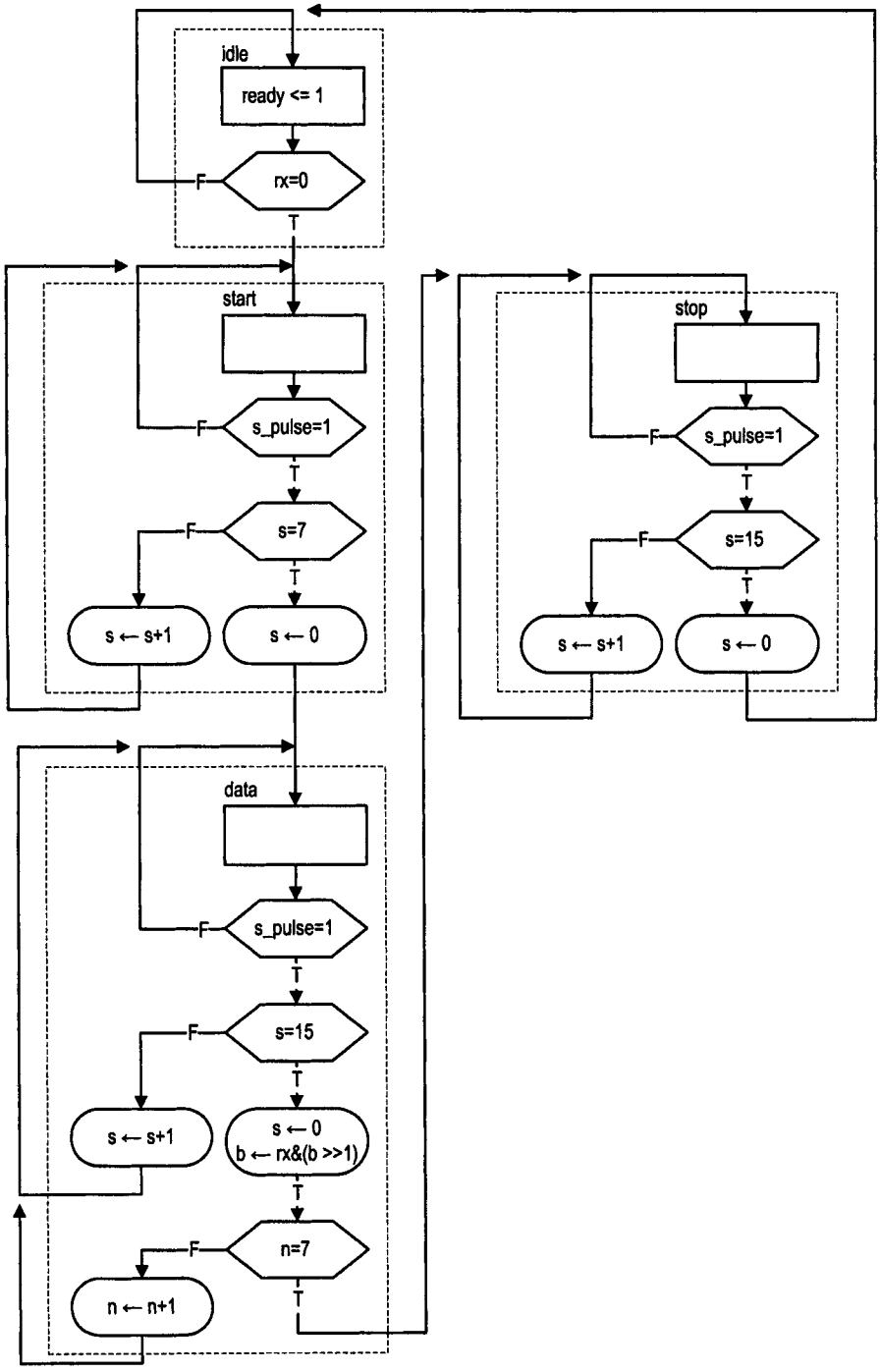


Figure 12.18 ASMD chart of a UART receiver.

```
    signal state_reg, state_next: state_type;
    signal clk16_next, clk16_reg: unsigned(5 downto 0);
    signal s_reg, s_next: unsigned(3 downto 0);
    signal n_reg, n_next: unsigned(2 downto 0);
    signal b_reg, b_next: std_logic_vector(7 downto 0);
    signal s_pulse: std_logic;
    constant DVSR: integer := 52;
begin
    -- free-running mod-52 counter, independent of FSMD
    process(clk,reset)
    begin
       if reset='1' then
          clk16_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          clk16_reg <= clk16_next;
       end if;
    end process;
    -- next-state/output logic
    clk16_next <= (others=>'0') when clk16_reg=(DVSR-1) else
                  clk16_reg + 1 ;
    s_pulse <= '1' when clk16_reg=0 else '0';

    -- FSMD state & data registers
    process(clk,reset)
    begin
       if reset='1' then
          state_reg <= idle;
          s_reg <= (others=>'0');
          n_reg <= (others=>'0');
          b_reg <= (others=>'0');
       elsif (clk'event and clk='1') then
          state_reg <= state_next;
          s_reg <= s_next;
          n_reg <= n_next;
          b_reg <= b_next;
       end if;
    end process;
    -- next-state logic & data path functional units/routing
    process(state_reg,s_reg,n_reg,b_reg,s_pulse,rx)
    begin
       s_next <= s_reg;
       n_next <= n_reg;
       b_next <= b_reg;
       ready <='0';
       case state_reg is
          when idle =>
             if rx='0' then
                state_next <= start;
             else
                state_next <= idle;
             end if;
             ready <='1';
          when start =>
```

```
            if (s_pulse = '0') then
               state_next <= start;
            else
               if s_reg=7 then
                  state_next <= data;
                  s_next <= (others=>'0');
               else
                  state_next <= start;
                  s_next <= s_reg + 1;
               end if;
            end if;
         when data =>
            if (s_pulse = '0') then
               state_next <= data;
            else
               if s_reg=15 then
                  s_next <= (others=>'0');
                  b_next <= rx & b_reg(7 downto 1);
                  if n_reg=7 then
                     state_next <= stop ;
                     n_next <= (others=>'0');
                  else
                     state_next <= data;
                     n_next <= n_reg + 1;
                  end if;
               else
                  state_next <= data;
                  s_next <= s_reg + 1;
               end if;
            end if;
         when stop =>
            if (s_pulse = '0') then
               state_next <= stop;
            else
               if s_reg=15 then
                  state_next <= idle;
                  s_next <= (others=>'0');
               else
                  state_next <= stop;
                  s_next <= s_reg + 1;
               end if;
            end if;
      end case;
   end process;
   pout <= b_reg;
end arch;
```

Several extensions are possible for this UART receiver, including adding a parity bit to detect the transmission error, checking the stop bit for the framing error, and making the baud rate adjustable. The main problem with the UART scheme is its performance. Because of the oversampling, the baud rate can be only a small fraction of the system clock rate, and thus this scheme can be used only for a low data rate.

12.6 SQUARE-ROOT APPROXIMATION CIRCUIT

The previous UART example is a typical *control-oriented* application, which is characterized by the dominance of the sophisticated decision conditions and branching structures in the algorithm. The opposite type is a *data-oriented* application, which involves mainly data manipulation and arithmetic operations. It is also known as a *computation-intensive* application.

Although a data-oriented application can be implemented by a combinational circuit in theory, the approach uses a large number of functional units and thus requires a significant amount of hardware resources. RT methodology allows us to share functional units in a time-multiplexed fashion, and we can schedule the operations sequentially to achieve the desired trade-off between performance and circuit complexity. A square-root approximation circuit in this section illustrates the design procedure and relevant issues of data-oriented applications.

The square-root approximation circuit uses simple adder-type components to obtain the approximate value of $\sqrt{a^2+b^2}$, where a and b are signed integers. The approximation is obtained by the following formula:

$$\begin{aligned}\sqrt{a^2+b^2} \quad &\approx \quad \max(((x-0.125x)+0.5y), x) \\ &\qquad \text{where } x = \max(|a|,|b|) \text{ and } y = \min(|a|,|b|)\end{aligned}$$

Note that the $0.125x$ and $0.5y$ operations correspond to shift x right three positions and shift y right one position, and that no actual multiplication circuit is needed. The equation can be coded in a traditional programming language. Let the two input operands be `a_in` and `b_in` and the output be `r`. One possible pseudocode is

```
a = a_in;
b = b_in;
t1 = abs(a);
t2 = abs(b);
x = max(t1, t2);
y = min(t1, t2);
t3 = x*0.125;
t4 = y*0.5;
t5 = x - t3;
t6 = t4 + t5;
t7 = max(t6, x)
r = t7;
```

To help VHDL conversion, we intentionally avoid reuse of the same variable name on the left-hand side of the statements. Because of the lack of control structure, the pseudocode can be translated to synthesizable VHDL code directly. The corresponding code is shown in Listing 12.10.

Listing 12.10 Square-root approximation circuit using direct dataflow description

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sqrt is
  port(
    a_in, b_in: in std_logic_vector(7 downto 0);
    r: out std_logic_vector(8 downto 0)
```

```
    );
end sqrt;

architecture comb_arch of sqrt is
    constant WIDTH: natural:=8;
    signal a, b, x, y: signed(WIDTH downto 0);
    signal t1, t2, t3, t4, t5, t6, t7: signed(WIDTH downto 0);
begin
    a <= signed(a_in(WIDTH-1) & a_in); -- signed extension
    b <= signed(b_in(WIDTH-1) & b_in);
    t1 <= a when a > 0 else
          0 - a;
    t2 <= b when b > 0 else
          0 - b;
    x <= t1 when t1 - t2 > 0 else
         t2;
    y <= t2 when t1 - t2 > 0 else
         t1;
    t3 <= "000" & x(WIDTH downto 3);
    t4 <= "0" & y(WIDTH downto 1);
    t5 <= x - t3;
    t6 <= t4 + t5;
    t7 <= t6 when t6 - x > 0 else
          x;
    r <= std_logic_vector(t7);
end comb_arch;
```

Note that the code consists only of concurrent statements, and thus their order does not matter. The original sequential execution is embedded in the interconnection of components and the flow of data. The VHDL code consists of seven arithmetic components, including one adder and six subtractors. Since the addition and subtractions are not mutually exclusive, sharing is not possible.

For a data-oriented application, it will be helpful to examine the dependency and movement of the data. This information can be visualized by a *dataflow graph*, in which an operation is represented by a node (a circle), and its input and output variables are represented by the incoming and outgoing arcs. The dataflow graph of the square-root approximation algorithm is shown in Figure 12.19.

The graph shows that the algorithm has only a limited degree of parallelism since at most only two operations can be executed concurrently. The seven arithmetic components of the previous VHDL code cannot significantly increase the performance, and most hardware resources are wasted. Thus, while the code is simple, it is not very efficient. RT methodology is a better alternative.

To transform a dataflow chart to an ASMD chart, we need to specify when and how operations in the dataflow graph are executed. The transformation include two major tasks: scheduling and binding. *Scheduling* specifies *when* a function (i.e., a circle) can start execution, and *binding* specifies *which* functional unit is assigned to perform the execution. One important design constraint is the number of functional units allowed to be used in a design. We can allocate a minimal number of functional units to reduce the circuit size, allocate a maximal number of units to exploit full potential parallelism, or find a specific number to achieve the desired trade-off between performance and circuit size. Obtaining

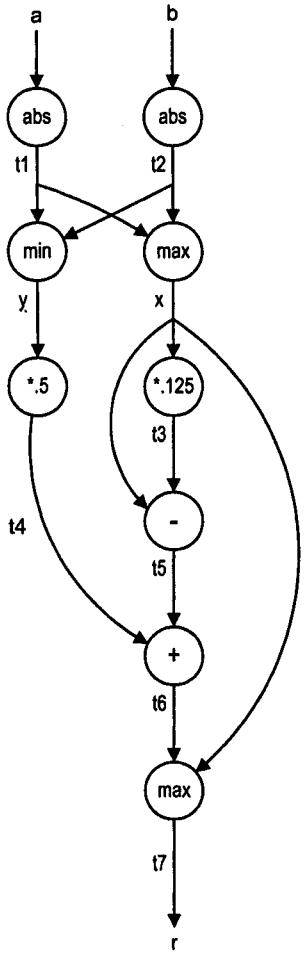


Figure 12.19 Dataflow graph.

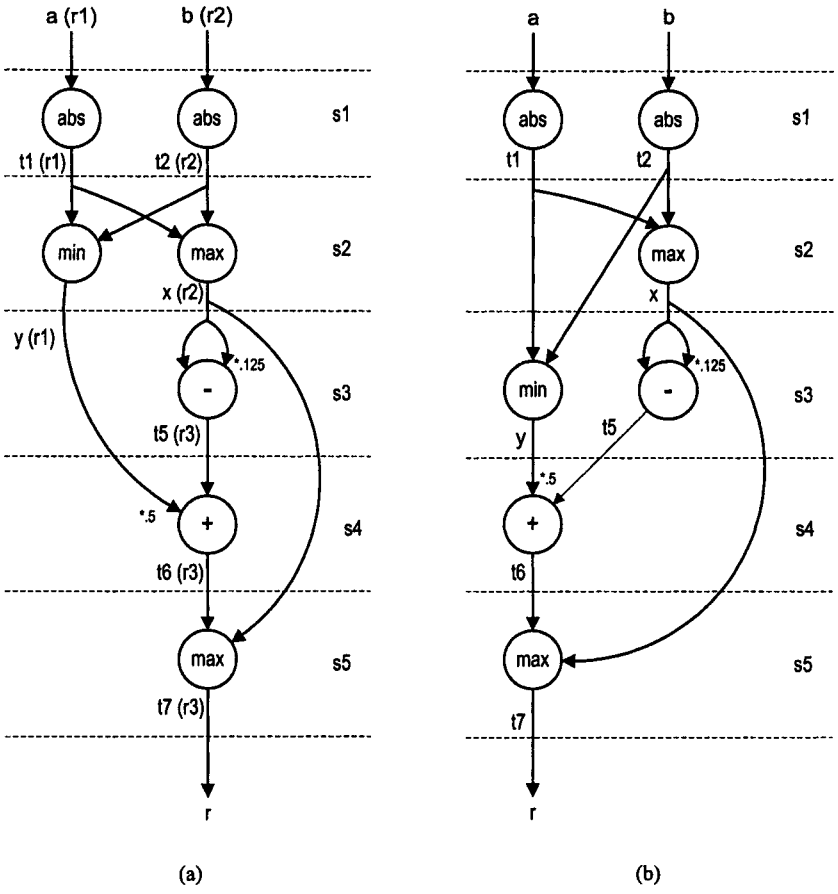


Figure 12.20 Schedules with two functional units.

an optimal schedule involves sophisticated algorithms and is a difficult task. Specialized EDA software tools are needed for a complex dataflow graph.

The dataflow graph of the square-root approximation algorithm involves a variety of operations. The $*.125$ and $*.5$ operations can be implemented by fixed-amount shifting circuits, which require no physical logic and thus should not be considered in the scheduling process. The other operations can be constructed by adders with some "glue" and routing logic. Thus, we can assume that the adder/subtractor is the only functional unit type required for the algorithm. Because at most two operations can be executed in parallel, the ASMD design can only utilize up to two functional units.

One possible schedule is shown in Figure 12.20(a). Note that the $*.125$ and $*.5$ operations are removed from the graph. The parentheses associated with the variables will be explained later. The dataflow graph is divided into five time intervals, which are later mapped into five states of an ASMD chart. It utilizes two units. One possible binding is to assign the two operations in the left column to one unit and the five operations in the right column to another unit. An alternative schedule and binding is shown in Figure 12.20(b), which requires the same amount of time to complete the computation. A schedule that uses only

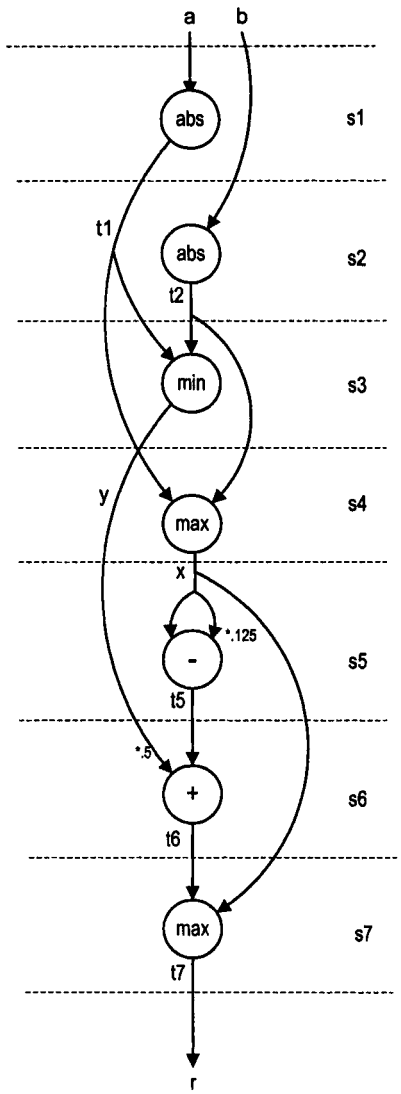


Figure 12.21 Schedule with one functional unit.

one functional unit is shown in Figure 12.21. It needs two extra time intervals to complete the operation.

Once the scheduling and binding are done, the dataflow graph can be transformed into an ASMD chart. Since each time interval represents a state in the chart, a register is needed when a signal is passed through the state boundary. The corresponding ASMD chart of Figure 12.20(a) is shown in Figure 12.22(a). The variables in the graph are mapped into the registers of the ASMD chart. There are two operations in the `s1` and `s2` states and one operation in the `s3`, `s4` and `s5` states. The `start` and `ready` signals and an additional `idle` state are included to interface the circuit with an external system.

Additional optimization schemes can be applied to reduce the number of registers and to simplify the routing structure. For example, instead of creating a new register for each

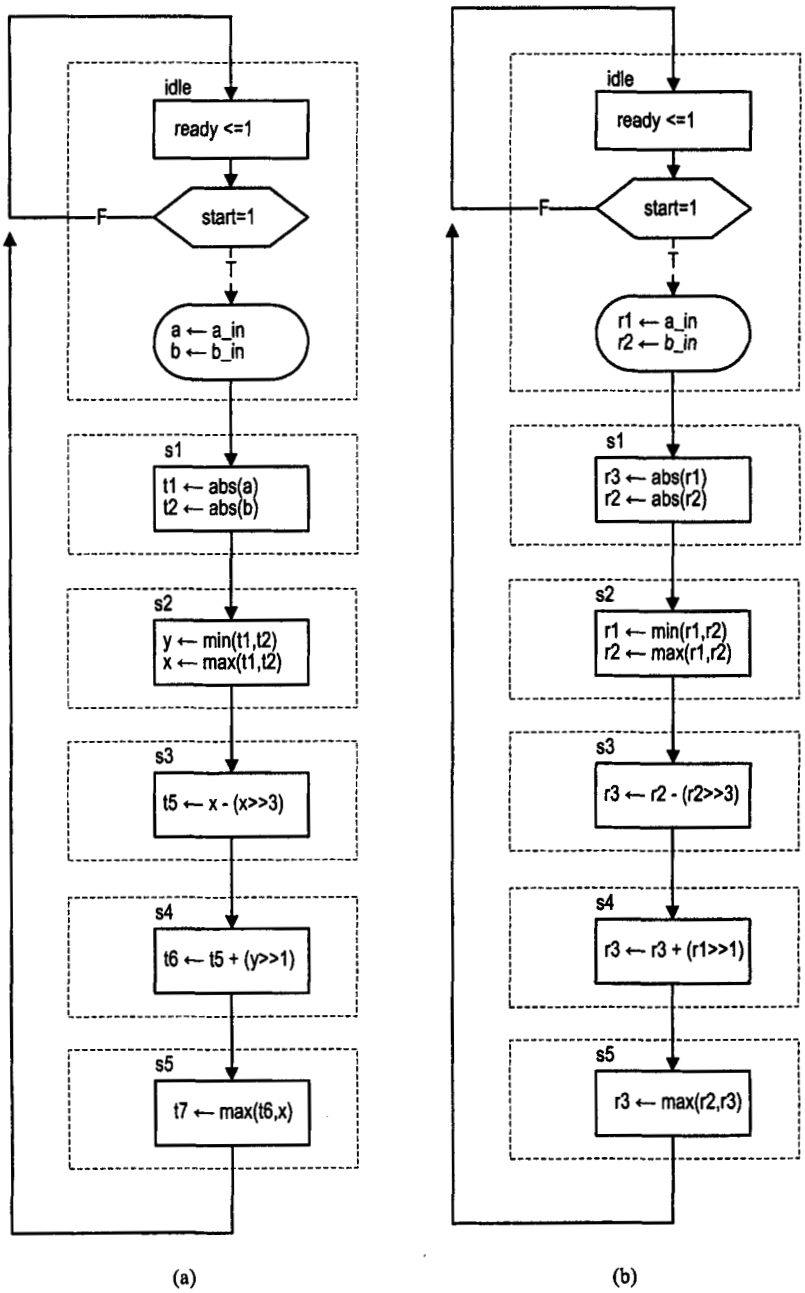


Figure 12.22 ASMD charts of a square-root approximation circuit.

variable, we can reuse an existing register if its value is no longer needed. This corresponds to properly renaming the variables in the dataflow graph. Close examination of Figure 12.20(a) shows that we can use three variables to cover the entire operation. The relationship between the new registers and the original registers is:

- Use `r1` to replace `a`, `t1` and `y`.
- Use `r2` to replace `b`, `t2` and `x`.
- Use `r3` to replace `t5`, `t6` and `t7`.

The replacement variables are shown in parentheses in Figure 12.20(a). The revised ASMD chart is shown in Figure 12.22(b). The number of the registers is reduced from seven to three.

The VHDL code can be derived according to the ASMD chart and is shown in Listing 12.11. To ensure proper sharing, the two functional units are isolated from the other description and coded as two separated segments. The first unit uses a single subtractor to perform the `max` and `abs` functions. The second unit uses a single adder to perform the `abs` and `max` functions as well as addition and subtraction. For clarity, we use the + operator for the carry-in signal. The synthesis software should be able to map it to the carry-in port of the adder rather than inferring another adder.

Listing 12.11 Square-root approximation circuit using RT methodology

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity sqrt is
   port(
      clk, reset: in std_logic;
      start: in std_logic;
      a_in, b_in: in std_logic_vector(7 downto 0);
      ready: out std_logic;
      r: out std_logic_vector(8 downto 0)
   );
end sqrt;

architecture seq_arch of sqrt is
   constant WIDTH: integer:=8;
   type state_type is (idle, s1, s2, s3, s4, s5);
   signal state_reg, state_next: state_type;
   signal r1_reg, r2_reg, r3_reg: signed(WIDTH downto 0);
   signal r1_next, r2_next, r3_next: signed(WIDTH downto 0);
   signal sub_op0, sub_op1, diff, au1_out:
      signed(WIDTH downto 0);
   signal add_op0, add_op1, sum, au2_out:
      signed(WIDTH downto 0);
   signal add_carry: integer ;
begin
   -- state & data registers
   process(clk,reset)
   begin
      if reset='1' then
         state_reg <= idle;
         r1_reg <= (others=>'0');
         r2_reg <= (others=>'0');
```

```
        r3_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
        state_reg <= state_next;
        r1_reg <= r1_next;
        r2_reg <= r2_next;
        r3_reg <= r3_next;
      end if;
  end process;
  -- next-state logic and data path routing
  process(start,state_reg,r1_reg,r2_reg,r3_reg,
          a_in,b_in,au1_out,au2_out)
  begin
    r1_next <= r1_reg;
    r2_next <= r2_reg;
    r3_next <= r3_reg;
    ready <='0';
    case state_reg is
      when idle =>
        if start='1' then
          r1_next <= signed(a_in(WIDTH-1) & a_in);
          r2_next <= signed(b_in(WIDTH-1) & b_in);
          state_next <= s1;
        else
          state_next <= idle;
        end if;
        ready <='1';
      when s1 =>
        r1_next <= au1_out; -- t1=|a|
        r2_next <= au2_out; -- t2=|b|
        state_next <= s2;
      when s2 =>
        r1_next <= au1_out; -- y=min(t1,t2)
        r2_next <= au2_out; -- x=max(t1,t2)
        state_next <= s3;
      when s3 =>
        r3_next <= au2_out; -- t5=x-0.125x
        state_next <= s4;
      when s4 =>
        r3_next <= au2_out; -- t6=0.5y+t5
        state_next <= s5;
      when s5 =>
        r3_next <= au2_out; -- t7=max(t6,x)
        state_next <= idle;
    end case;
  end process;
  -- arithmetic unit 1
  -- subtractor
  diff <= sub_op0 - sub_op1;
  -- input routing
  process(state_reg,r1_reg,r2_reg)
  begin
    case state_reg is
      when s1 => -- 0-a
```

```
            sub_op0 <= (others=>'0');
            sub_op1 <= r1_reg; -- a
         when others =>  -- s2: t2-t1
            sub_op0 <= r2_reg; -- t2
            sub_op1 <= r1_reg; -- t1
      end case;
   end process;
   -- output routing
   process(state_reg,r1_reg,r2_reg,diff)
   begin
      case state_reg is
         when s1 => --|a|
            if diff(WIDTH)='0' then  --(0-a)>0
               au1_out <= diff; -- -a
            else
               au1_out <= r1_reg; -- a
            end if;
         when others =>  -- s2: min(a,b)
            if diff(WIDTH)='0' then --(t2-t1)>0
               au1_out <= r1_reg; -- t1
            else
               au1_out <= r2_reg; -- t2
            end if;
      end case;
   end process;
   -- arithmetic unit 2
   -- adder
   sum <= add_op0 + add_op1 + add_carry;
   -- input routing
   process(state_reg,r1_reg,r2_reg,r3_reg)
   begin
      case state_reg is
         when s1 => -- 0-b
            add_op0 <= (others=>'0'); --0
            add_op1 <= not r2_reg;  -- not b
            add_carry <= 1;
         when s2 => -- t1-t2
            add_op0 <= r1_reg; --t1
            add_op1 <= not r2_reg; --not t2
            add_carry <= 1;
         when s3 => -- -- x-0.125x
            add_op0 <= r2_reg; --x
            add_op1 <= not ("000" & r2_reg(WIDTH downto 3));
            add_carry <= 1;
         when s4 => -- 0.5*y + t5
            add_op0 <= "0" & r1_reg(WIDTH downto 1);
            add_op1 <= r3_reg;
            add_carry <= 0;
         when others => -- t6 - x
            add_op0 <= r3_reg; --t1
            add_op1 <= not r2_reg; --not x
            add_carry <= 1;
      end case;
```

```
    end process;
140 -- output routing
    process(state_reg,r1_reg,r2_reg,r3_reg,sum)
    begin
       case state_reg is
          when s1 => -- |b|
145          if sum(WIDTH)='0' then  -- (0-b)>0
                au2_out <= sum; -- -b
             else
                au2_out <= r2_reg; -- b
             end if;
150       when s2 =>
             if sum(WIDTH)='0' then
                au2_out <= r1_reg;
             else
                au2_out <= r2_reg;
155          end if;
          when s3|s4 => -- +,-
             au2_out <= sum;
          when others => -- s5
             if sum(WIDTH)='0' then
160             au2_out <= r3_reg;
             else
                au2_out <= r2_reg;
             end if;
       end case;
165 end process;
    -- output
    r <= std_logic_vector(r3_reg);
end seq_arch;
```

12.7 HIGH-LEVEL SYNTHESIS

The square-root approximation circuit of Section 12.6 shows that deriving the optimal RT design for data-oriented applications is by no means a simple task. The procedure is complex and involves many sophisticated algorithms. Derivation of this type of circuit belongs to a specific class of design, known as *high-level synthesis* or as somewhat misleading *behavioral synthesis*.

The synthesis starts with a set of constraints and an abstract VHDL description similar to the algorithm's pseudocode. The high-level synthesis software converts the initial description into an FSMD and automatically derives code for the control path and data path. In other words, the high-level synthesis software basically transforms from code in the form of Listing 12.10 to code in the form of Listing 12.11. The main task of the synthesis is to find an optimal schedule and binding to minimize the required hardware resources, to maximize performance or to obtain the best trade-off within a given constraint.

High-level synthesis is best for data-oriented, computation-intensive applications, such as those encountered in signal processing. It requires a separate software package, and its output is fed to regular synthesis software.

12.8 BIBLIOGRAPHIC NOTES

High-level synthesis covers primarily algorithms to perform the *binding* and *scheduling* of hardware resources, with emphasis on functional units. The treatment is normally very theoretical. The texts, *Synthesis and Optimization of Digital Circuits* by G. De Micheli, and *High-Level Synthesis: Introduction to Chip and System Design* by D. D. Gajski et al., provide good coverage on this topic. The square-root approximation circuit is adopted from the text, *Principles of Digital Design* by D. D. Gajski, which uses the circuit to demonstrate the procedures and various optimization algorithms of high-level synthesis.

Problems

12.1 In the ASMD chart of the programmable one-shot pulse generator of Section 12.2, shifting the desired values requires three states. This operation can be done by using a single state and a counter.

(a) Revise the ASMD chart to accommodate the change.

(b) Derive the VHDL code of the revised chart.

12.2 Redesign the programmable one-shot pulse generator of Section 12.2 as a pure regular sequential circuit. Derive the VHDL code.

12.3 Redesign the programmable one-shot pulse generator of Section 12.2 as a pure FSM.

(a) Derive the state diagram.

(b) Derive the VHDL code.

12.4 For the memory controller in Section 12.3, assume that the period of the system clock is 50 ns. Redesign the circuit for the 120-ns SRAM. The design should use a minimal number of states in the FSMD.

(a) Derive the revised ASMD chart.

(b) Derive the VHDL code.

(c) Determine the required time to perform a read operation.

12.5 Repeat the Problem 12.4 with a system clock of 15 ns.

12.6 The memory controller of Listing 12.5 must return to the `idle` state after each operation. We can improve performance by skipping this state when back-to-back memory operations are issued.

(a) Derive the revised ASMD chart.

(b) Derive the VHDL code.

(c) When a read operation follows immediately after a write operation, the direction of data flow in the bidirectional `d` line changes. Do a detailed timing analysis to examine whether a conflict can occur. We can assume that the timing parameters of the tri-state buffer in the data path are similar to those of the SRAM.

(d) Repeat part (c) for a write operation immediately following a read operation.

12.7 The FIFO buffer of Section 9.3.2 uses a register file as temporary storage. Revise the design to use an SRAM device for storage. Assume that the 120-ns SRAM is used and the system clock is 25 ns. We wish to design a FIFO controller for this system. Since it takes several clock cycles to complete a memory operation, the controller should have an additional status signal, `ready`, to indicate whether the SRAM is currently in operation.

(a) Derive the ASMD chart for the FIFO controller.

(b) Derive the VHDL code.

12.8 Repeat Problem 12.7 for a stack controller.

12.9 Consider the GCD circuit in Section 12.4. Assume that inputs are N-bit unsigned integers. For each architecture:

(a) Determine the input pattern that leads to the maximal number of clock cycles.

(b) Calculate the exact number of clock cycles for the pattern.

12.10 For the `fast_arch` architecture of the GCD circuit, we can combine the subtraction with the comparison and merge the `sub` state into the `swap` state.

(a) Derive the revised ASMD chart.

(b) Derive the VHDL code.

(c) Assume that the clock period is doubled because of the merge. Discuss whether the merge actually increases the performance (i.e., completes the computation in less time).

(d) Repeat part (c), but assume that the clock period is increased by only 50%.

12.11 In the `fastest_arch` architecture of the GCD circuit, up to two shifting operations are performed in the `swap` state and one is performed in the `res` state. Since a barrel shifter is a complex circuit, we want the three operations to share one unit.

(a) Derive the VHDL code of a barrel shifter that can perform both shift-right and shift-left operations.

(b) Derive the revised ASMD chart.

(c) Derive the VHDL code.

12.12 Design a transmitter for the UART discussed in Section 12.5.

(a) Derive the ASMD chart.

(b) Derive the VHDL code.

12.13 Expand the UART receiver of Section 12.5 to make the baud rate adjustable. Assume that there is an additional 2-bit control signal, `baud_sel`, which specifies the desired baud rate, which can be 1200, 2400, 4800 or 9600 baud.

12.14 Revise the UART of Section 12.5 to include an even-parity bit. The length of the data is now 7 bits, and the eighth bit is the parity bit. The parity bit is asserted when there is an odd number of 1's in data bits (and thus makes the 8 received bits always have an even number of 1's). Design a transmitter and a receiver for the modified UART.

(a) Derive the ASMD chart for the transmitter.

(b) Derive the VHDL code for the transmitter.

(c) Derive the ASMD chart for the receiver. The receiver should include an extra output signal to indicate the occurrence of a parity error.

(d) Derive the VHDL code for the receiver.

12.15 Expand the UART receiver of Section 12.5 to accommodate different parity schemes. Assume that there is an extra 2-bit control signal, `pairty_sel`, which selects the desired parity scheme, which can be odd parity, even parity or no parity.

12.16 Consider a UART that can communicate at four baud rates: 1200, 2400, 4800 and 9600 baud. Assume that the actual baud rate is unknown but the transmitter always sends a "11111111" data byte at the beginning of the session. Design a circuit that can automatically determine the baud rate and derive the VHDL code.

12.17 Consider the schedule in Figure 12.20(b).

(a) Map the variables into a minimum number of registers.

(b) Derive the ASMD chart for the schedule. Recall that two arithmetic units are used in this schedule.

(c) Derive the VHDL code.

12.18 Repeat Problem 12.17 for the schedule in Figure 12.21. Note that only one arithmetic unit is used in this schedule.

12.19 Multiplication can be implemented by performing additions of shifted bit-products, as discussed in Section 7.5.4. Let $p_7, p_6, \ldots, p_1, p_0$ be the shifted bit-products of an 8-bit multiplier. The final product can be expressed as

$$y = p_7 + p_6 + \cdots + p_1 + p_0$$

(a) Derive the dataflow graph for the expression. Arrange the additions as a tree to exploit parallelism.

(b) Assume that only one adder is provided. Derive a schedule.

(c) Derive the ASMD chart for the schedule. Use a minimal number of registers in the chart.

(d) Derive the VHDL code.

(e) Discuss the difference between this design and the sequential multiplier discussed in Section 11.6.

12.20 Repeat parts (b), (c) and (d) of Problem 12.19, but use two adders to accelerate the operation.

12.21 Repeat parts (b), (c) and (d) of Problem 12.19, but use three adders to accelerate the operation.

CHAPTER 13

HIERARCHICAL DESIGN IN VHDL

As the size of a digital system increases, its complexity grows accordingly. One method of managing the complexity is to describe the system in a hierarchical structure, in which the system is gradually divided into smaller parts. With a hierarchy, we only need to focus on a small, manageable part at a time. One of the goals of VHDL is to facilitate the development and modeling of large digital systems. It consists of versatile mechanisms and language constructs to specify and configure a design hierarchy and to organize design information and files. This chapter provides an overview of constructs relevant to the RT-level design and synthesis.

13.1 INTRODUCTION

Hierarchical design is a methodology that divides a system recursively into small modules and then constructs each module independently. The term *recursively* means that the division process can be applied repeatedly and the modules can be further decomposed. For example, consider the sequential multiplier of Section 11.3.3. One possible design hierarchy is shown in Figure 13.1. The system is first divided into a control path and a data path. The control path is then divided into the next-state logic and the state register, and the data path is divided into a routing circuit, functional units and a data register. The functional units are then further decomposed into an adder and a decrementor. If needed, we can continue the process and further refine the leaf modules. The sequential multiplier can also be part of a larger system. For example, it can be a module of an arithmetic unit, which in turn, can be a module of a processor.

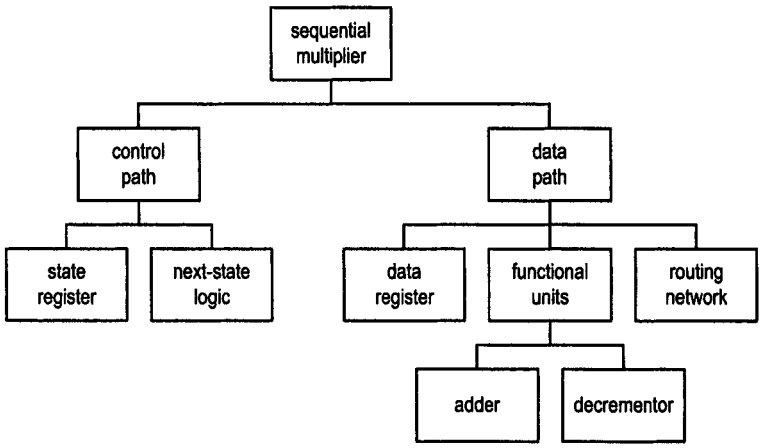


Figure 13.1 Hierarchical description of a sequential multiplier.

13.1.1 Benefits of hierarchical design

There are two major benefits of using the hierarchy: *complexity management* and *design reuse*. As the size of transistor continues to decrease, more functionality can be included in a device and the digital system grows larger and more complex. Managing the complexity becomes a key challenge in today's design. Hierarchical design methodology allows us to apply the divide-and-conquer strategy and break a system into smaller modules. This approach helps us to manage a large design in several ways:

- Instead of looking at the entire system, we can focus on a manageable portion of the system, and analyze, design and verify each module in isolation.
- Once the hierarchy and modules are specified, a large system can be constructed in stages by a designer or concurrently by a team of designers.
- The synthesis software may require a significant amount of memory space and computation time to synthesize a large system. Breaking the system into smaller modules and synthesizing them independently can make the process more effective.

Hierarchical design methodology also helps to facilitate design reuse:

- Some predesigned modules or third-party cores (i.e., IPs) may exist and can be used in the system. Therefore, we don't need to construct every system from scratch.
- Many systems contain some common or similar functionalities. After we design and verify a module in a system, the same module can be used in future design.
- Some design may contain certain device-dependent components, such as an SRAM module. To achieve portability, we can isolate these components in the top level of the hierarchy and substitute them according to the target technology.

13.1.2 VHDL constructs for hierarchical design

One objective of VHDL is to facilitate the modeling and developments of complex digital systems. Many language constructs are designed for this purpose. These include the following:

- Component
- Generic

- Configuration
- Library
- Package
- Subprogram

The *component*, *generic* and *configuration* constructs provide flexible and versatile mechanism to describe a hierarchical design. These constructs are discussed in Sections 13.2, 13.3 and 13.4. The *library*, *package*, and *subprogram* help the management of complicated code and are briefly reviewed in Section 13.5. To take full advantage of the hierarchical design methodology, we have to develop general and flexible modules. This issue is discussed in Chapters 14 and 15.

13.2 COMPONENTS

Hierarchical design methodology basically divides a system into smaller modules and then constructs the modules accordingly. Although not stated explicitly, our previous derivations generally followed this approach. We usually started with a top-level diagram with several major parts and then derived the VHDL code according to the diagram, with a VHDL segment for each part. The VHDL component construct provides a formal and explicit way to describe a hierarchical design.

We examined the VHDL component construct briefly in Section 2.2.2. It is the mechanism used to describe a digital system in a structural view. Recall that a structural view is essentially a block diagram, in which we specify the types of parts used and the interconnections among these parts. While the component construct is supported in both VHDL 87 and VHDL 93, the syntax of VHDL 93 is much simpler. However, since the IEEE RTL synthesis standard is based on VHDL 87, it follows the old syntax. To obtain maximal portability, our discussion mainly follows the IEEE RTL synthesis standard (i.e., VHDL 87). In Section 13.4.4, we briefly examine the newer version.

In VHDL 87, using a component involves two steps. The first step is *component declaration*, in which a component is "make known" to an architecture. The second step is *component instantiation*, in which an instance of the component is created and its external I/O interface is specified.

13.2.1 Component declaration

Component declaration provides information about the external interface of a component, which includes the input and output ports and relevant parameters. The information is similar to that provided in an entity declaration. The simplified syntax of component declaration is as follows:

```
component component_name
   generic(
      generic_declaration;
      generic_declaration;
      . . .
   );
   port(
      port_declaration;
      port_declaration;
      . . .
   );
```

```
end component;
```

The generic portion is optional and consists of relevant parameters to be passed into the component. It is discussed in the next section. The port portion consists of port declarations, which are similar to those in an entity declaration. Note that the **is** keyword is in the entity declaration but not in the component declaration (the **is** keyword is allowed in VHDL 93). As in entity declaration, no information about internal implementation is specified in component declaration.

Assume that we have already designed a decade (i.e., mod-10) counter and that its entity declaration is

```
entity dec_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
   );
end dec_counter;
```

If we want to use it as a component in other designs, the easiest way is to declare a component that has the same name and ports:

```
component dec_counter
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
   );
end component;
```

Note that the same information is used in the entity declaration and the corresponding component declaration. Graphically, a component can be thought of as a circuit part with properly named input and output ports. The conceptual diagram of the `dec_counter` component is shown in Figure 13.2(a).

During the elaboration process, the component eventually has to be bound with an architecture body. The complete VHDL code of the decade counter is shown in Listing 13.1. The `en` input functions as an enable signal. The `pulse` output is a status signal and is asserted when the counter reaches 9 and is ready to warp around.

Listing 13.1 Decade counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity dec_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0);
      pulse: out std_logic
   );
end dec_counter;
```

```
architecture up_arch of dec_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   constant TEN: integer:= 10;
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
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
          if r_reg=(TEN-1) then
             r_next <= (others=>'0');
          else
             r_next <= r_reg + 1;
          end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
   pulse <= '1' when r_reg=(TEN-1) else
            '0';
end up_arch;
```

In a VHDL code, a component declaration is included in the declaration part of an architecture body, as shown in Listing 13.2. If it is used in multiple architecture bodies, the declaration may be placed in a package. Use of packages is discussed in Section 13.5.3.

13.2.2 Component Instantiation

Once a component is declared, its instance can be created inside the architecture body. The simplified syntax of component instantiation is

```
instance_label: component_name
   generic map(
      generic_association;
      generic_association;
      . . .
   )
   port map(
      port_association;
      port_association;
      . . .
   );
```

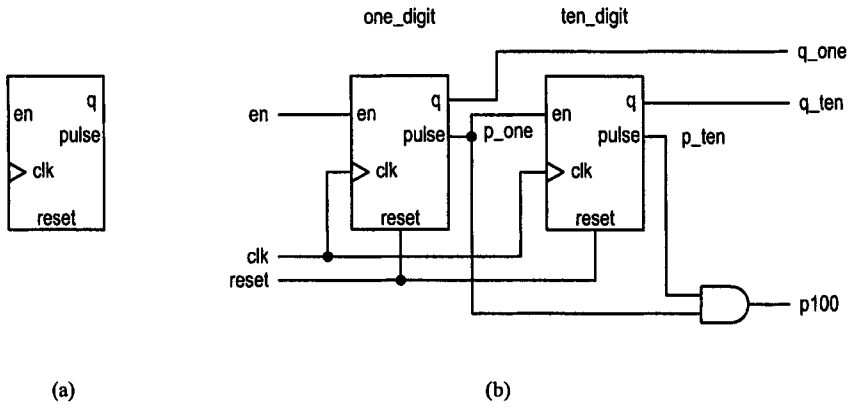


Figure 13.2 Block diagram of a two-digit decimal counter.

In the first line, `component_name` specifies the component to be used, and `instance_label` assigns the instance with a unique label for identification. The **generic map** portion assigns the actual values to the generics. This portion, which is optional, is discussed in Section 13.3. Note that there is no semicolon after the generic map portion. The **port map** portion specifies the connections (i.e., "association") between the component's ports (known as *formal signals*) and the external signals (known as *actual signals*). The `port_association` term has the general format

```
port_name => signal_name
```

This is known as the *named association.*

The use of component instantiation can best be explained by an example. Assume that we want to implement a two-digit decimal counter, which counts up in BCD format (i.e., from 00 to 99) and wraps around. One possible implementation is to cascade two decade counters. The diagram is shown in Figure 13.2(b). The left decade counter represents the digit in the one's place. Its `pulse` port is connected to an external wire, which is labeled as the p_one signal, which in turn is connected to the en port of the right decade counter, which represents the digit in the ten's place. If the two-digit decimal counter is enabled, the left decade counter asserts p_one signal every 10 clock cycles and wraps around. The right decade counter is controlled by the p_one signal and thus counts only once for every 10 clock cycles. The `p100` output is a pulse to indicate that the two-digit decimal counter reaches 99 and is ready to wrap around.

To describe this diagram in VHDL, we need to create two instances of the `dec_counter` component and specify the relevant I/O connections. The main task in component instantiation is to specify the mapping between the formal signals and the actual signals. This is a tedious and error-prone task. The best way to do it is to draw a properly labeled block diagram and then derive the VHDL code following the connections of the diagram. The diagram should contain necessary information, which includes the component names, instance labels, and properly labeled ports and connection signals. The block diagram in Figure 13.2(b) is created for this purpose. In our convention, the information from the component declaration, which includes the component name and port names (i.e., the formal signals), is placed inside the block. The external signal names (i.e., actual signals) and instance names, on the other hand, are placed outside the block. Note that a formal signal name and an actual signal name can be the same.

Following the diagram, we can derive the code segments for the two instances:

```
one_digit: dec_counter
   port map (clk=>clk, reset=>reset, en=>en,
             pulse=>p_one, q=>q_one);
ten_digit: dec_counter
   port map (clk=>clk, reset=>reset, en=>p_one,
             pulse=>p_ten, q=>q_ten);
```

The port mapping used here is known as the *named association* because both the name of the formal signal and the name of the actual signal are listed in each port association. The order of the port associations does not matter. For example, the first instance can also be written as

```
one_digit: dec_counter
   port map (pulse=>p_one, reset=>reset, en=>en,
             q=>q_one, clk=>clk);
```

The complete VHDL code is shown in Listing 13.2.

Listing 13.2 Two-digit decimal counter using decade counters

```
library ieee;
use ieee.std_logic_1164.all;
entity hundred_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q_ten, q_one: out std_logic_vector(3 downto 0);
      p100: out std_logic
   );
end hundred_counter;

architecture vhdl_87_arch of hundred_counter is
   component dec_counter
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(3 downto 0);
         pulse: out std_logic
      );
   end component;
   signal p_one, p_ten: std_logic;
begin
   one_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>en,
                pulse=>p_one, q=>q_one);
   ten_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>p_ten, q=>q_ten);
   p100 <= p_one and p_ten;
end vhdl_87_arch;
```

In VHDL, component instantiation is just another concurrent statement and thus can be mixed with other statements. For example, a simple signal assignment statement is used to derive the p100 signal.

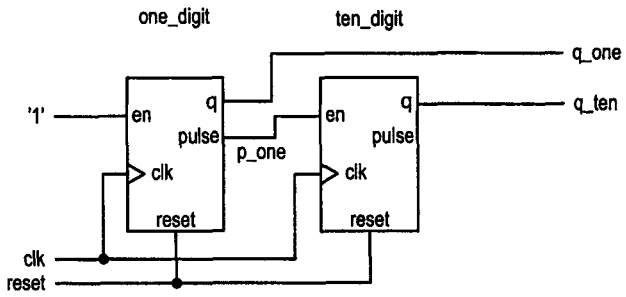


Figure 13.3 Block diagram of a free-running two-digit decimal counter.

13.2.3 Caveats in component instantiation

As long as we derive a proper block diagram, the use of components is straightforward. There are two caveats about component instantiation. One is the use of position association in port mapping, and the other is the handling of unused ports.

So far, we have used the *named association* method for port mapping. Alternatively, we can omit the formal signal names and place the actual names according to the positions of the formal signals. This is known as *positional association*. For example, the component declaration of `dec_counter` shows that the order of the ports is

```
clk, reset, en, q, pulse
```

In the previous `vhdl_87_arch` architecture, we can put the actual signals in this order in component instantiation. The VHDL code becomes

```
one_digit: dec_counter
   port map (clk, reset, en, q_one, p_one);
ten_digit: dec_counter
   port map (clk, reset, p_one, q_ten, p_ten);
```

At first glance this method may seem to be more compact, but it can cause problems in the long run, especially for a component with many I/O ports. For example, we may revise the port declaration of `dec_counter` later and switch the order of the `clk` and `reset` signals:

```
. . .
port(
  reset, clk: in std_logic;
  . . .
```

The modification has no effect for the `dec_counter` code in Listing 13.1 but introduces a serious problem for code that instantiates `dec_counter` with positional association. To make the code more reliable, it is good practice to use named association in port and generic mapping.

When we instantiate a component, some ports may not be needed to connect to actual signals. For example, assume that we wish to design a free-running two-digit decimal counter, in which the `en` and `p100` signals are removed. The modified block diagram is shown in Figure 13.3, in which the `en` signal of the `one_digit` instance is tied to logic '1', and the `pulse` signal of the `ten_digit` instance is left unconnected. To describe the mapping of `en`, we can simply use a constant expression to replace the actual signal and the association becomes `en=>'1'`. Some synthesis software may not accept the constant

expression. To overcome this, we can create a signal, assign it with the desired constant and then use it as the actual signal in port mapping.

To specify the unused port, we can associate the port with the **open** keyword and the association becomes `pulse=>open`. Good synthesis software should know that the port is not used, backtrack the corresponding circuit and remove the unneeded circuit from implementation. The **open** keyword can also be associated with an input port, and the association means that the initial value in port declaration will be used for the port. Since it is not good practice to assign an initial value to a signal or port in synthesis, this should be avoided.

The VHDL code for the free-running two-digit decimal counter is shown in Listing 13.3.

Listing 13.3 Free-running two-digit decimal counter

```
library ieee;
use ieee.std_logic_1164.all;
entity free_run_hundred_counter is
   port(
      clk, reset: in std_logic;
      q_ten, q_one: out std_logic_vector(3 downto 0)
   );
end free_run_hundred_counter;

architecture vhdl_87_arch of free_run_hundred_counter is
   component dec_counter
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(3 downto 0);
         pulse: out std_logic
      );
   end component;
   signal p_one: std_logic;
begin
   one_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>'1',
                pulse=>p_one, q=>q_one);
   ten_digit: dec_counter
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>open, q=>q_ten);
end vhdl_87_arch;
```

In named association, a formal port can be omitted in the list and VHDL assumes that it is mapped to **open** by default. To make the code reliable, it is good practice to list all ports in port map and explicitly associate the unused output ports with **open**.

13.3 GENERICS

The *generic* construct of VHDL is a mechanism to pass information into an entity and a component. Generics are like parameters. They are first declared in entity and component declaration and later assigned a value during component instantiation.

The use of generics starts with the entity declaration by adding a generic declaration section. The simplified syntax is

```
entity entity_name is
   generic(
      generic_names: data_type;
      generic_names: data_type;
      . . .
   );
   port(
      port_names: mode data_type;
      ...
   );
end entity_name;
```

Once a generic is declared, it can be used in subsequent port declarations and associated architecture bodies. For example, consider the free-running binary counter of Section 8.5.4, which has a fixed width of 4 bits. We can modify it to a more versatile parameterized free-running binary counter by defining a WIDTH generic to specify the desired width (i.e., number of bits). The modified entity declaration becomes

```
entity para_binary_counter  is
   generic(WIDTH: natural);
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(WIDTH-1 downto 0)
   );
end para_binary_counter;
```

Note that the range of the q output is not fixed, but is expressed in terms of the WIDTH generic, as in std_lgic_vector(WIDTH-1 **downto** 0).

After the declaration, the generic can be used in the associated architecture bodies. A generic cannot be modified inside the architecture body and thus functions like a constant. It is sometimes referred to as a *generic constant.* As a constant, we use uppercase letters for the generics in the book.

The corresponding architecture body of the binary counter is

```
architecture arch of para_binary_counter is
   signal r_reg, r_next: unsigned(WIDTH-1 downto 0);
begin
   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;
   r_next <= r_reg + 1;
   q <= std_logic_vector(r_reg);
end arch;
```

Again, note that the WIDTH generic is used to specify the range of internal signals.

To use the parameterized free-running binary counter in a hierarchical design, a similar component declaration should be included in the architecture declaration. The generic can then be assigned a value in the generic mapping section when a component instance is instantiated. An example code is shown in Listing 13.4. The code creates a 4-bit counter and a 12-bit counter.

Listing 13.4 Example of the use of generics

```
library ieee;
use ieee.std_logic_1164.all;
entity generic_demo is
   port(
      clk, reset: in std_logic;
      q_4: out std_logic_vector(3 downto 0);
      q_12: out std_logic_vector(11 downto 0)
   );
end generic_demo;

architecture vhdl_87_arch of generic_demo is
   component para_binary_counter
      generic(WIDTH: natural);
      port(
         clk, reset: in std_logic;
         q: out std_logic_vector(WIDTH-1 downto 0)
      );
   end component;
begin
   four_bit: para_binary_counter
      generic map (WIDTH=>4)
      port map (clk=>clk, reset=>reset, q=>q_4);
   twe_bit: para_binary_counter
      generic map (WIDTH=>12)
      port map (clk=>clk, reset=>reset, q=>q_12);
end vhdl_87_arch;
```

In the second example, we consider the design of a parameterized mod-n counter, in which n can be specified as a parameter. The counter counts from 0 to $n-1$ and then wraps around. To count to n patterns, the counter needs at least $\lceil \log_2 n \rceil$ bits. Our first design uses two generics, the `N` generic for n and the `WIDTH` generic for the number of bits in the counter. The VHDL code is shown in Listing 13.5. It is patterned after the decade counter of Listing 13.1 and includes an `en` control signal and a `pulse` output signal, which is asserted when the counter reaches $n-1$.

Listing 13.5 Parameterized mod-n counter

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

```
architecture arch of mod_n_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
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
          if r_reg=(N-1) then
             r_next <= (others=>'0');
          else
             r_next <= r_reg + 1;
          end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
   pulse <= '1' when r_reg=(N-1) else
            '0';
end arch;
```

Note that WIDTH is not an independent parameter. It can be derived from N and thus is not actually required. Section 13.5 shows how to achieve this.

We can redesign the two-digit decimal counter of Section 13.2.2 by replacing the two decade counters with two parameterized mod-n counters. To do this, we simply assign 10 to N and 4 to WIDTH in component instantiation and thus customize the mod-n counter as a mod-10 counter. The corresponding VHDL code is shown in Listing 13.6.

Listing 13.6 Two-digit decimal counter using parameterized mod-n counters

```
architecture generic_arch of hundred_counter is
   component mod_n_counter
      generic(
         N: natural;
         WIDTH: natural
      );
      port(
         clk, reset: in std_logic;
         en: in std_logic;
         q: out std_logic_vector(WIDTH-1 downto 0);
         pulse: out std_logic
      );
   end component;
```

```
    signal p_one, p_ten: std_logic;
begin
    one_digit: mod_n_counter
       generic map (N=>10, WIDTH=>4)
       port map (clk=>clk, reset=>reset, en=>en,
                 pulse=>p_one, q=>q_one);
    ten_digit: mod_n_counter
       generic map (N=>10, WIDTH=>4)
       port map (clk=>clk, reset=>reset, en=>p_one,
                 pulse=>p_ten, q=>q_ten);
    p100 <= p_one and p_ten;
end generic_arch;
```

The two examples show the potential of combining the generics and component. Instead of creating an array of counters with different widths, we can use a single parameterized module and customize it to the desired width. This makes the module more flexible and more versatile, and greatly enhances its chance to be reused. The next two chapters provide comprehensive coverage of the design of parameterized modules.

Another major application of generics is to pass delay information in modeling and simulation. For example, we can define the Tpd generic for the propagation delay. It can then be used in a statement like

```
y <= a + b after Tpd ns;
```

This allows us to pass delay information into the model when it becomes available.

13.4 CONFIGURATION

13.4.1 Introduction

When a component is declared and instantiated, as the example in Listing 13.2, only basic generic and port information is provided. *Configuration* of VHDL is the process of *binding* a component with a design entity and architecture. The process includes two parts:

1. Bind a component with a design entity.
2. Bind the design entity with an architecture body.

The configuration of VHDL is very flexible, and thus its detailed syntax and use are quite complex. However, most features are not needed in RT-level design and synthesis. The IEEE RTL synthesis standard supports only the second part of the process, the binding of entity and architecture. We discuss only default binding, and top-level entity and architectural body binding in the book.

Explicit configuration is not always required for a component. For example, the codes in previous sections use no configuration constructs. In this case, the component is bound by the *default binding*, which is processed as follows:

- The component is bound to an entity with the identical name.
- Component ports are bound to entity ports of the same names.
- The most recently analyzed architecture body is bound to the entity declaration.

In RT-level design, the hierarchy is normally simple, and only one architecture body exists. The default binding should be satisfactory most of the time, and no explicit configuration statement is needed.

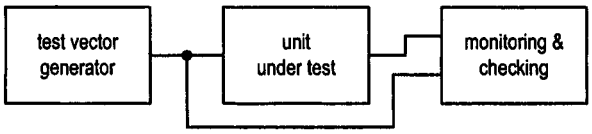


Figure 13.4 Block diagram of a testbench.

In synthesis, multiple architectures may be needed for several reasons. First, there is frequently a trade-off between area and performance in a digital circuit. A complex circuit, such as a multiplier, may have several implementations, each with a unique area–delay characteristics. Each implementation represents an architecture body. Second, we sometimes need to adjust certain circuit characteristics to fit a specific application. For example, we may need to force a counter to circulate different patterns. One way to accomplish this is to use a separate architecture body for each pattern.

In modeling and simulation, having multiple architectures is more common. For example, we discussed the concept of testbench in Section 2.2.4. The diagram of a basic testbench is shown in Figure 13.4. A complex design is normally first specified in an abstract behavioral description, converted to an RT-level description, and then synthesized to a cell-level structural description. Each description represents an architecture body. As the design progresses, a more detailed architecture body becomes available. We can use configuration to bind the new description for verification.

There are two ways to specify the configuration. It can be described in an independent design unit, which is known as *configuration declaration*, or included in the declaration section of the architecture body, which is known as *configuration specification*. The two methods are discussed in the following subsections. The IEEE RTL standard supports only the configuration declaration method.

13.4.2 Configuration declaration

In the configuration declaration method, we create a new kind of design unit, known as *configuration*, to specify the binding of a component. A configuration unit is an independent design unit in VHDL, just like an entity declaration and an architecture body. It is analyzed and stored independently when the VHDL code is processed. The simplified syntax of a configuration unit is

```
configuration conf_name of entity_name is
  for archiecture_name
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
    . . .
  end for;
end;
```

The `conf_name` term is the unique identifier for this configuration unit. The `entity_name` and `architecture_name` terms identify the entity and the architecture for which the configuration is intended. The `instance_label` term specifies a specific component instance,

and the following "**use** ..." clause indicates the entity and architecture to be bound to the instance. The `lib_name` term is the name of the library in which the entity and architecture reside. The library is discussed in Section 13.5. In the place of `instance_label`, we can use the **all** keyword to represent all instances of this particular component, or use **others** in the end to represent all the unbound instances of the component.

To demonstrate the use of configuration, we create a second architecture, `down_arch`, for the `dec_counter` entity of Section 13.2.1. The VHDL code is shown in Listing 13.7. It counts down from 9 to 0 and then wraps around. The `pulse` output is asserted when the counter reaches 0 and is ready to wrap around. If we use this architecture in the `vhdl_87_arch` architecture of the two-digit decimal counter of Section 13.2.2, the two-digit counter counts down from 99 to 00 and then wraps around.

Listing 13.7 Decade counter with a count-down sequence

```
architecture down_arch of dec_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   constant TEN: integer := 10;
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
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
          if r_reg=0 then
             r_next <= to_unsigned(TEN-1,4);
           else
             r_next <= r_reg - 1;
          end if;
      end if;
   end process;
   -- output logic
   q <= std_logic_vector(r_reg);
   pulse <= '1' when r_reg=0 else
            '0';
end down_arch;
```

Depending on the requirement of the direction of a two-digit decimal counter, we can create a configuration unit to specify the desired binding. The VHDL in Listing 13.8 binds the two instances with the `down_arch` architecture.

Listing 13.8 Configuration for a two-digit decimal counter

```
configuration count_down_config of hundred_counter is
   for vhdl_87_arch
      for one_digit: dec_counter
         use entity work.dec_counter(down_arch);
      end for;
      for ten_digit: dec_counter
         use entity work.dec_counter(down_arch);
      end for;
   end for;
end;
```

Note that the work library is the default library used in VHDL. It represents the current working library.

13.4.3 Configuration specification

The configuration declaration is general and flexible. However, for a simple design, creating a new design unit for this purpose is somewhat cumbersome. An alternative is to specify the relevant configuration in the declaration section of the architecture body. This is known as a *configuration specification*. The simplified syntax is

```
for instance_label: component_name
   use entity lib_name.bound_entity_name(bound_arch_name);
for instance_label: component_name
   use entity lib_name.bound_entity_name(bound_arch_name);
. . .
```

For example, the configuration declaration in Listing 13.8 can also be specified by a configuration specification. We simply revise the vhdl_87_arch by adding the relevant configuration information to the declaration section:

```
architecture vhdl_87_config_arch of hundred_counter is
   component dec_counter
      port(
        clk, reset: in std_logic;
        en: in std_logic;
        q: out std_logic_vector(3 downto 0);
        pulse: out std_logic
      );
   end component;
   for one_digit: dec_counter
      use entity work.dec_counter(down_arch);
   for ten_digit: dec_counter
      use entity work.dec_counter(down_arch);
   signal p_one, p_ten: std_logic;
begin
   . . .
```

13.4.4 Component instantiation and configuration in VHDL 93

Components and configuration are flexible in VHDL, but its syntax is involved and tedious. Since RT-level design uses relatively simple component instantiation and bind-

ing, the syntactic constructs becomes cumbersome. For example, consider the previous vhdl_87_config_arch architecture. We need a relatively lengthy declaration to use and bind the two component instances in design. VHDL 93 provides a much simpler mechanism. It allows a component to be bound directly to an entity and an architecture in component instantiation. The simplified syntax is

```
instance_label:
   entity lib_name.bound_entity_name(bound_arch_name)
      generic map (. . .)
      port map (. . .);
```

The **entity** `lib_name.bound_entity_name(bound_arch_name)` clause specifies the associated entity and architecture, and no component declaration or any additional configuration construct is needed. The `(bound_arch_name)` term is optional. If it is omitted, the most recently analyzed architecture will be bound to the entity.

Consider the two-digit decimal counter of Section 13.2.2. With this mechanism, a more compact code can be derived, as shown in Listing 13.9.

Listing 13.9 Two-digit decimal counter with direct entity binding

```
architecture vhdl_93_arch of hundred_counter is
   signal p_one, p_ten: std_logic;
begin
   one_digit: entity work.dec_counter(up_arch)
      port map (clk=>clk, reset=>reset, en=>en,
                pulse=>p_one, q=>q_one);
   ten_digit: entity work.dec_counter(up_arch)
      port map (clk=>clk, reset=>reset, en=>p_one,
                pulse=>p_ten, q=>q_ten);
   p100 <= p_one and p_ten;
end vhdl_93_arch;
```

Since this kind of instantiation is valid only in VHDL 93, it is not supported by the IEEE RTL synthesis standard. However, some software does accept this type of component instantiation.

13.5 OTHER SUPPORTING CONSTRUCTS FOR A LARGE SYSTEM

13.5.1 Library

As we discussed in Section 3.2.5, a VHDL program is analyzed and stored as individual design units, which include entity declaration, architecture body, configuration declaration, and so on. A VHDL *library* is the virtual repository that stores the analyzed design units. VHDL does not define the physical location of a library. Most software maps a library to a physical directory in a hard disk. By default, the current design units are stored in a working library named `work`. For example, the `work` library is used in the previous component instantiation:

```
. . .
one_digit: entity work.dec_counter(up_arch)
. . .
```

For a complex design, there may exist a large number of design units. It is desirable to organize these units and store them in separate places. Also, we may have a collection of

commonly used design units that are shared by many different designs. It is more effective to save these units in a common library rather than duplicating them in every design directory.

To access the content of a library, we must first make it known by using a library statement. The syntax is

```
library lib_name, lib_name, ... , lib_name;
```

For example, assume that we create a library named `c_lib` and save the previous `dec_counter` entity and relevant architectures in the library. The `count_down_config` configuration discussed in Section 13.4.2 must be revised accordingly:

```
library c_lib;  -- make c_lib visible
configuration clib_config of hundred_counter is
   for vhdl_87_arch
      for one_digit: dec_counter
         use entity c_lib.dec_counter(down_arch); -- c_lib
      end for;
      for ten_digit: dec_counter
         use entity c_lib.dec_counter(down_arch); -- c_lib
      end for;
   end for;
end;
```

Note that the `work` library of the original code is replaced with `c_lib`.

If a design unit is accessed frequently, we can make it visible by adding a **use** clause. The syntax is

```
use lib_name.unit_name;
```

The unit can then be accessed directly without referring to the library. The **all** keyword can be used in place of `unit_name` to make all units of the library visible. For example, the previous code can be revised as:

```
library c_lab;
use c_lib.dec_counter; -- make dec_counter visible
configuration clib_config of hundred_counter is
   for vhdl_87_arch
      for one_digit: dec_counter
         use entity dec_counter(down_arch); -- lib dropped
      end for;
      for ten_digit: dec_counter
         use entity dec_counter(down_arch);
      end for;
   end for;
end;
```

Note that the library name is dropped from the "**use entity** `dec_counter(down_arch);`" statement.

The `work` library is declared implicitly by VHDL definition, and thus there is no need for the "**library** `work;`" statement.

13.5.2 Subprogram

Subprograms in VHDL include *functions* and *procedures*. Their bodies are made of sequential statements, and their behaviors are similar to those in traditional programming languages. Unlike entity and architecture, procedures and functions are not design units and thus cannot be processed independently. For example, we cannot isolate a function from the code and synthesize it separately. Therefore, while the functions and procedures are basic building blocks of software hierarchy, they are not adequate to describe the hardware hierarchy.

VHDL functions are more versatile and useful than procedures, and thus our discussion focuses mainly on functions. In synthesis, functions should not be used to specify the design hierarchy, but should be treated as a shorthand for simple, repeatedly used operations. Functions are also needed to perform certain house-keeping tasks, such as data type conversion or operator overloading in IEEE packages.

A VHDL function takes several parameters and returns a single value. It must first be declared in the declaration section and then can be called later. A function can be thought of as an extension of the expression and can be "called" wherever an expression is used. The simplified syntax of a function is

```
function func_name(parameter_list) return data_type is
   declarations;
begin
   sequential statement;
   sequential statement;
   . . .
   return(expression);
end;
```

The following examples illustrate the construction of a function. The first example is a function that performs a majority function. It returns '1' if two or more input parameters, a, b and c, are '1'. The function can be treated as a shorthand for the $a \cdot b + a \cdot c + b \cdot c$ expression.

```
function maj(a, b, c: std_logic) return std_logic is
   variable result: std_logic;
begin
   result := (a and b) or (a and c) or (b and c);
   return result;
end maj;
```

The `maj` function must be declared and then can be invoked. The following code segment illustrates its use:

```
architecture arch of . . .
   -- declaration
   function maj(a, b, c: std_logic) return std_logic is
      variable result: std_logic;
   begin
      result := (a and b) or (a and c) or (b and c);
      return result;
   end maj;
   signal i1, i2, i3, i4, x, y: std_logic;
begin
   . . .
```

```
x <= maj(i1, i2, i3) or i4;
y <= i1 when maj(i2, i3, i4)='1' else
. . .
```

Note that the entire function definition is included in the declaration section of the architecture body. This may become cumbersome. An alternative is to declare the function in a package, which is discussed in the next subsection.

The second example is a function that performs data type conversion. It converts the `std_logic` data type to the `boolean` data type.

```
function to_boolean(a: std_logic) return boolean is
   variable result: boolean;
begin
   if a='1' then
      result := true;
   else
      result := false;
   end if;
   return result;
end to_boolean;
```

If this function is declared, we can use `to_boolean(a)` to replace the `a='1'` expression.

The last example is a function that performs $\lceil \log_2 n \rceil$, which is frequently needed in calculating the width of data signals.

```
function log2c(n: integer) return integer is
   variable m, p: integer;
begin
   m := 0;
   p := 1;
   while p < n loop
      m := m + 1;
      p := p * 2;
   end loop;
   return m;
end log2c;
```

13.5.3 Package

As a system becomes complex, more information is included in the declaration section. The declaration section of an architecture body may consist of the declarations of constants, data types, components, functions and so on. When a system is divided into several smaller subsystems, some declarations must be duplicated in many different design units. The VHDL *package* construct is a method of organizing declarations. We can gather the commonly used declarations in a design, group them together and store them in a package. A design unit just needs to include a use clause to access these declarations.

A VHDL package is divided into *package declaration* and *package body*. The declaration items are placed in a package declaration. If an item is a subprogram, only the declaration of the subprogram is included. The body (i.e., the implementation) of the subprogram is placed in the associated package body. The package body is optional and is needed only when subprograms exist.

Package declaration and package body are design units of VHDL. They are analyzed independently and stored in a library. To make a declaration item visible, a use clause is needed. Its syntax is

```
use lib_name.package_name.item_name;
```

Most of the time, we use the **all** keyword in place of `item_name` to make all items of the named package visible.

Many extensions to VHDL are done by defining additional packages, such as the IEEE `std_logic_1164` and `numeric_std` packages. Almost all of our VHDL codes include the statement

```
use ieee.std_logic_1164.all;
```

It makes all of the declaration items of the predefined `std_logic_1164` package visible, and thus we can use the `std_logic` and `std_logic_vector` data types in VHDL code.

We can also define our own package. The syntax of a package declaration is very simple:

```
package package_name is
   declaration item;
   declaration item;
   . . .
end package_name;
```

If the declaration items include subprograms, an associated package body is needed. Its syntax is

```
package body package_name is
   subprogram;
   subprogram;
   . . .
end package_name;
```

An example of package declaration is shown in the first part of Listing 13.10. It consists of the definition of the `std_logic_2d` data type, which is a two-dimensional array with element of `std_logic` data type, and the declaration of the `log2c` function. Note that the package also invokes the IEEE `std_logic_1164` package so that the `std_logic` data type can be used. The corresponding package body is shown in the second part of Listing 13.10, which is the implementation of the `log2c` function.

Listing 13.10 Example of a package

```
-- package declaration
library ieee;
use ieee.std_logic_1164.all;
package util_pkg is
   type std_logic_2d is
      array(integer range <>, integer range <>) of std_logic;
   function log2c (n: integer) return integer;
end util_pkg ;

--package body
package body util_pkg is
   function log2c(n: integer) return integer is
      variable m, p: integer;
   begin
```

```
        m := 0;
        p := 1;
        while p < n loop
           m := m + 1;
           p := p * 2;
        end loop;
        return m;
     end log2c;
  end util_pkg;
```

For the parameterized mod-n counter of Section 13.3, one drawback is that we must use a redundant `WIDTH` generic to specify the width of the output signal. To overcome the problem, we need a previously defined function to calculate `WIDTH` from `N`. The `log2c` function of the `util_pkg` package can be used for this purpose. We can invoke this package before the entity declaration. Assume that the package is saved in the same working directory. The improved code is shown in Listing 13.11.

Listing 13.11 Improved parameterized mod-n counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.util_pkg.all;
entity better_mod_n_counter is
   generic(N: natural);
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(log2c(N)-1 downto 0);
      pulse: out std_logic
   );
end better_mod_n_counter;

architecture arch of better_mod_n_counter is
   constant WIDTH: natural := log2c(N);
   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
   process(en,r_reg)
   begin
      r_next <= r_reg;
      if (en='1') then
          if r_reg=(N-1) then
             r_next <= (others=>'0');
```

```
            else
               r_next <= r_reg + 1;
            end if;
         end if;
40    end process;
      -- output logic
      q <= std_logic_vector(r_reg);
      pulse <= '1' when r_reg=(N-1) else
               '0';
45 end arch;
```

We add the statement

```
use work.util_pkg.all;
```

to make the `log2c` function visible. The function can then be used in the range specification for the `q` output, as in `std_logic_vector(log2c(N)-1 downto 0)`, as well as the calculation of the internal `WIDTH` constant.

13.6 PARTITION

VHDL provides powerful mechanisms and versatile language constructs to support hierarchical design methodology and to manage the design of large systems. To apply these features, we must first determine the design hierarchy and divide the system into smaller parts. The process is sometimes known as *design partition.*

For synthesis, the design partition can be viewed from two perspectives: physical partition and logical partition. *Physical partition* is the division of the physical implementation. It specifies how the circuit is divided during synthesis. *Logical partition* is imposed by human designers. The goal is to make the design, development and verification of a system manageable. The two kinds of partitions are correlated but not necessarily identical.

13.6.1 Physical partition

A digital system can be described by a hierarchy of an arbitrary number of levels. The circuit parts becomes simpler as we traverse down the hierarchy. In VHDL code, the circuit parts are described as component instances. When the code is processed, the components are replaced by the actual architecture bodies level by level, and the hierarchy is gradually converted to a flattened description. One way to perform synthesis is to collapse the entire hierarchy into a one big, flattened circuit and then to synthesize the circuit accordingly. More sophisticated software provides mechanisms to selectively flatten the hierarchy and preserve some high-level components. The software will synthesize and optimize these components separately and then merge the resulting netlists (i.e., the cell level descriptions) to form the final circuit.

An important issue is the size of the preserved components. Since synthesis involves many sophisticated algorithms, the required computation time and memory space are normally much worse than the linear order, $O(n)$, where n is the size of the circuit. This implies that synthesizing a large circuit will take much more computation time and memory space than that required by several smaller circuits. For example, assume that an algorithm is on the order of $O(n^3)$. If it requires 1 second to synthesize a 1000-gate circuit, it will take 125 seconds (i.e., 5^3 seconds) to synthesize a 5000-gate circuit and 35 hours (i.e., 50^3 seconds) to synthesize a 50,000-gate circuit. However, if we break the 50,000-gate circuit into

ten 5000-gate circuits, it requires only about 21 minutes (i.e., $10 * 5^3$ seconds) to synthesize the ten small parts.

On the other hand, when we preserve a component, it implicitly forms a "synthesis boundary," and optimization can only be performed within the boundary. This prevents synthesis software from exploring optimization opportunities that exist between components. Thus, small component size hinders the optimization process and leads to less efficient overall implementation. Many software tools suggest that the maximal gate count for synthesis is between 5000 and 50,000 gates.

13.6.2 Logical partition

The logical partition is determined by human designers. A good partition simplifies and streamlines the development and verification process, and makes the code reliable and portable. To facilitate the synthesis, a "logical circuit part" in the hierarchy should be within the range of the maximal gate count recommended by the synthesis software. On the other hand, since synthesis software can flatten and collapse a part of the hierarchy, smaller "logical parts" can be used in the design hierarchy.

In addition to synthesis concerns, partition should also be used to help develop reliable design, and portable and reusable code. We should pay particular attention to the circuits that may hinder portability or introduce problems in development flow. It is a good idea to separate these circuits from ordinary logic and instantiate them as components in a design hierarchy. Two types of circuits of concern are:

- Device-dependent circuits
- Non-Boolean circuits

Device-dependent circuits Device-dependent circuits are those not synthesized by generic logic gates. They are predesigned or even prefabricated for a specific device technology. For example, most device technology has various types of prefabricated memory modules. These circuits are inferred by component instantiation and require no synthesis. Once a device-dependent circuit is used, the VHDL code becomes device dependent. To maintain portability, one way is to isolate these circuits in the top-level hierarchy and instantiate them as individual components. If the VHDL code is used later for a different device technology, we need only substitute these components with equivalent circuits of the new technology and keep the remaining code intact.

Non-Boolean circuits Digital system design is primarily based on a mathematical model of Boolean algebra and its derivations. The algorithms in analysis, synthesis, verification and testing are developed within this framework. If a circuit does not follow the basic mathematical model, we call it a *non-Boolean* circuit. Some examples are listed below.

- *Tri-state buffer.* It has a third possible value, high impedance, in its output. The high impedance cannot be optimized or propagated as regular logic values.
- *Delay-sensitive circuit.* It uses logic gates to introduce a specific amount of propagation delay. The function of the circuit relies on the delay characteristics, not on Boolean algebra manipulation.
- *Clock distribution circuit.* It distributes the clock signal to the connected FFs. The circuit functions as a current amplifier and performs no logic operation.
- *Synchronization circuit.* It uses FFs to resolve the metastable condition, not for regular storage. This is discussed in Chapter 16.

These circuits should be isolated in the hierarchy so that later they can be processed independently.

13.7 SYNTHESIS GUIDELINES

- Use components, not subprograms, to specify the design hierarchy.
- Use the `std_logic` and `std_logic_vector` data types in the ports of components to maintain portability.
- Use named association, not positional association, in port mapping and generic mapping.
- List all ports of a component in port mapping and use **open** for unused output ports.
- For synthesis, partition the system into 5000- to 50,000-gate modules. Collapse and flatten low-level hierarchy if the components are too small.
- Separate device-dependent parts from ordinary logic and instantiate them as components in a hierarchy.
- Separate non-Boolean circuits from ordinary logic and instantiate them as components in a hierarchy.

13.8 BIBLIOGRAPHIC NOTES

This chapter provides a detailed discussion of VHDL components and gives a brief review of many other language constructs. The review is aimed primarily at synthesis of the RT-level system. Comprehensive coverage and the syntax of these constructs can be found in VHDL texts, such as *The Designer's Guide to VHDL, 2nd edition*, by P. J. Ashenden.

Problems

13.1 Consider a three-digit decimal counter that counts from 000 to 999 and wraps around. Use the `dec_counter` of Section 13.2.1 as a component to design this circuit.

(a) Derive the block diagram and properly label the formal and actual signals.
(b) Follow the block diagram to derive the VHDL code.
(c) Use a configuration specification for configuration.
(d) Same as part (c), but use a configuration declaration for configuration.

13.2 Redesign the three-digit decimal counter of Problem 13.1 using the mod-n counter of Section 13.3.

(a) Derive the block diagram and properly label the formal and actual signals.
(b) Follow the block diagram to derive the VHDL code.

13.3 We want to design a timer that counts from 00 to 59 seconds and then wraps around. Assume that the system clock is 1 MHz. Use the mod-n counter of Section 13.3 as a component to design this circuit.

(a) Derive the block diagram and properly label the formal and actual signals.
(b) Follow the block diagram to derive the VHDL code.

13.4 Consider a counter that counts from m to n and then wraps around. Derive VHDL code for the counter. Use generics, `M` and `N`, for m and n of the counter.

13.5 Divide the FIFO control circuit in Figure 9.14 into a hierarchy of two counters and two comparison circuits.

(a) Derive VHDL entities and architectures for the counter and comparison circuits.

(b) Follow the diagram in Figure 9.14 to derive the VHDL code.

13.6 Some synthesis software does not accept the `**` operator. Derive a VHDL function, `power2`, that implements the function $f(n) = 2^n$.

13.7 Derive a function that converts the `boolean` data type into the `std_logic` data type. The `true` and `false` values of the `boolean` type are converted to `'1'` and `'0'` of the `std_logic` data type respectively.

CHAPTER 14

PARAMETERIZED DESIGN: PRINCIPLE

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

14.1 INTRODUCTION

As the size of digital systems continues to grow, designing every system from scratch requires a tremendous amount of time and effort. One way to increase productivity and efficiency is design reuse. Many applications use parts of common functionalities. We can design and verify these parts once, store them in a library and then reuse them in other applications. As we discussed in Chapter 13, VHDL provides a versatile and powerful framework to facilitate the hierarchical design methodology and to accommodate predesigned components. Thus, once the commonly used parts are developed, design reuse can readily be incorporated into the VHDL environment.

While circuits share some parts of common functionalities, the exact specification of the part differs. For example, many applications need a binary counter. The basic construction of counters is similar, but the numbers of bits and the direction of the counting sequence depend on the need of a specific application. The chance that a fixed-size counter, say, an 11-bit up counter, will be reused is very small. On the other hand, if we develop a counter module that can be customized with different numbers of bits and counting directions, it can be utilized by many applications. The customization is normally done by describing certain circuit aspects with external parameters, and thus we call this *parameterized design*.

VHDL supports parameterized design in several ways. First, it provides mechanisms to pass parameters into an entity and to extract information from objects inside the entity. Second, most operators of VHDL and overloaded operators of the `std_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.

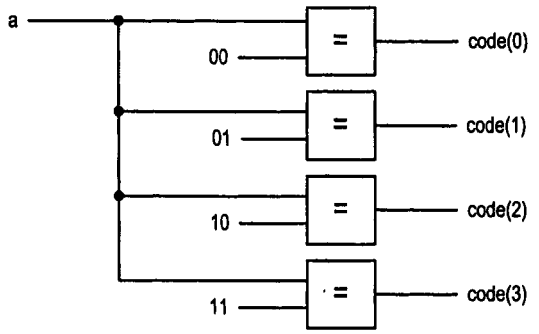


Figure 14.1 Block diagram of a 2-to-4 decoder.

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);
```

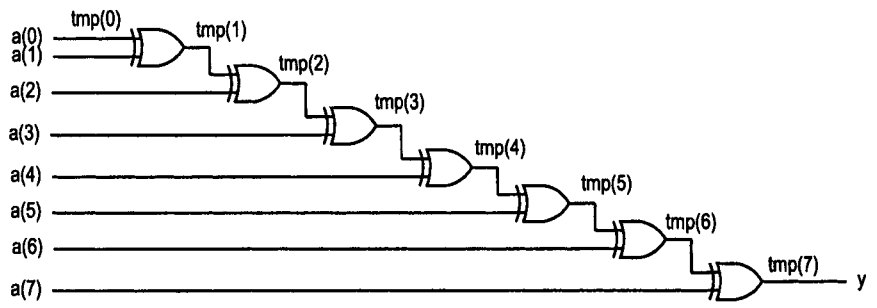


Figure 14.2 Block diagram of a reduced-xor circuit.

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

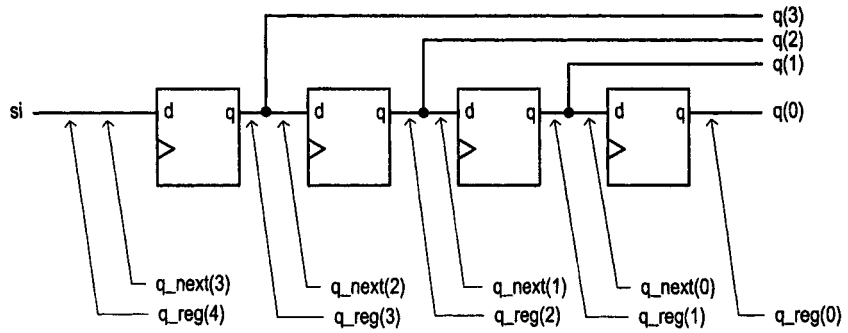


Figure 14.3 Block diagram of a serial-to-parallel converter.

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:

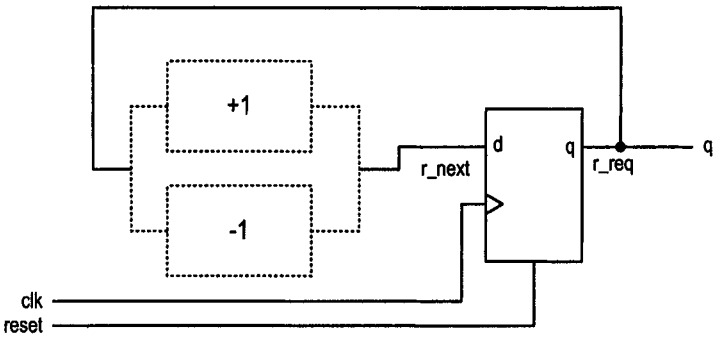
```
tmp(0) <= a(0);
y <= tmp(WIDTH-1);
```

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.

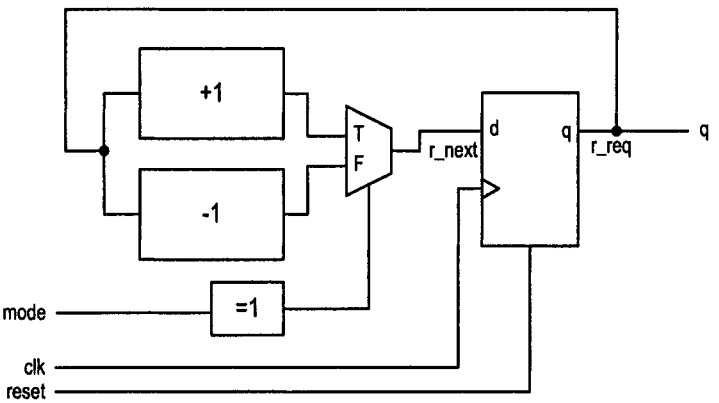
Listing 14.15 Parameterized reduced-xor circuit with a conditional generate statement

```
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;
```

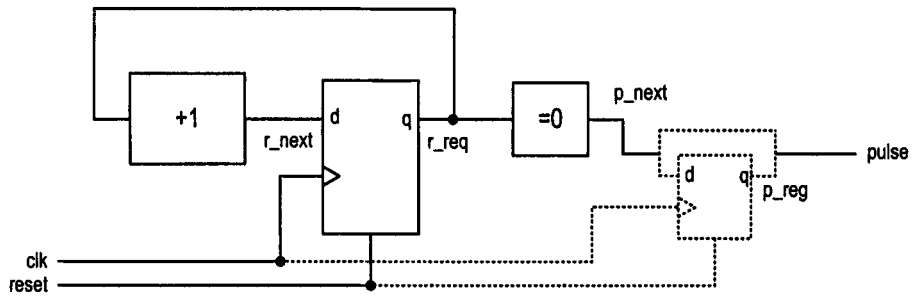


Figure 14.5 Block diagram of a counter with an optional output buffer.

```
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.

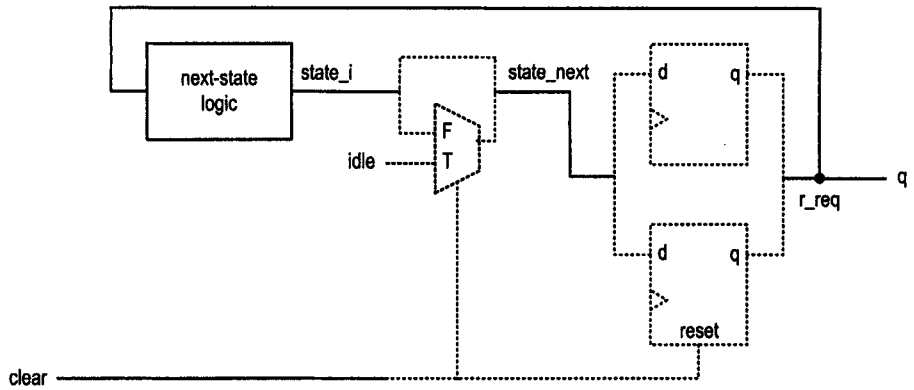


Figure 14.6 Block diagram of FSM with a selectable clear signal.

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.

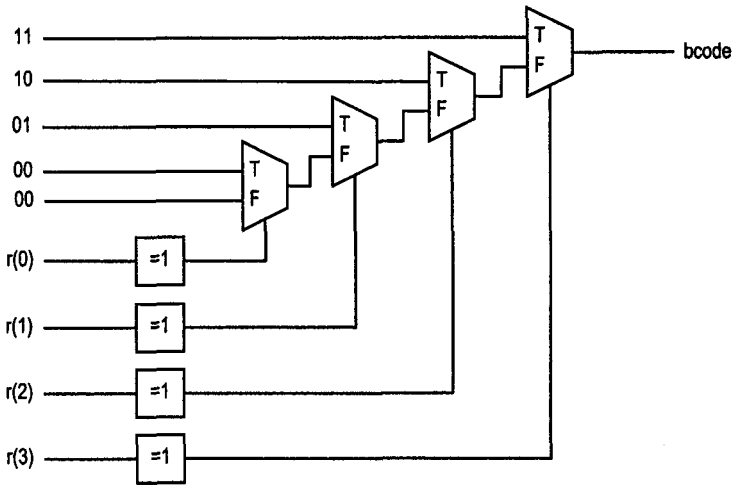


Figure 14.7 Block diagram of a priority encoder.

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;
```

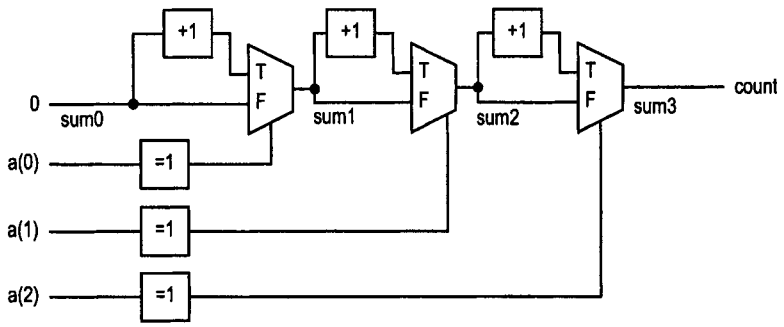


Figure 14.8 Block diagram of population counter.

```
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.

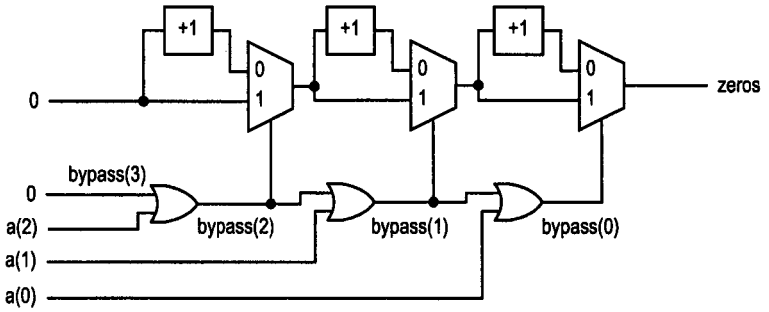


Figure 14.9 Block diagram of a leading-zero counting circuit.

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
```

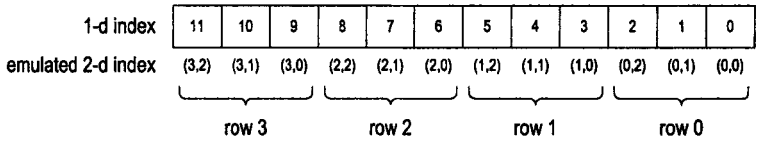


Figure 15.1 Emulation of a two-dimensional 4-by-3 array with a one-dimensional array.

```
  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).

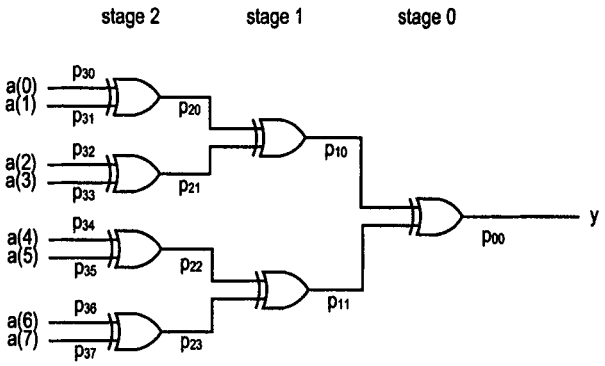


Figure 15.2 Tree-shaped reduced-xor circuit.

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;
```

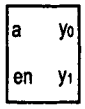
15.3.2 Binary decoder

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:

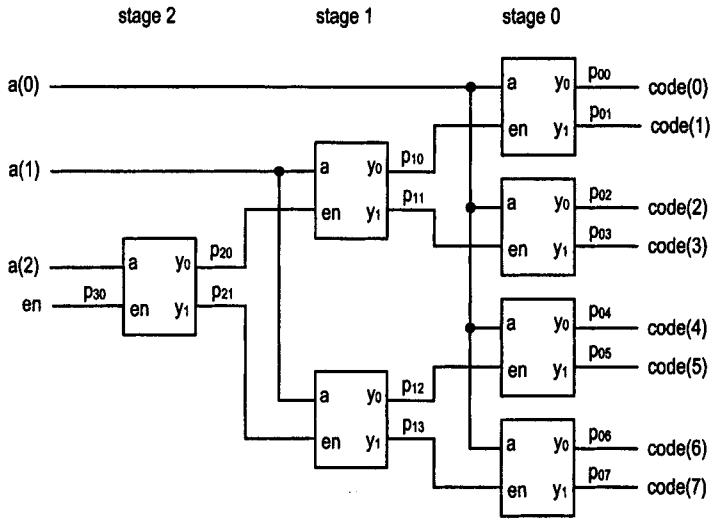
$$y_0 = en \cdot a'$$
$$y_1 = en \cdot a$$

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.

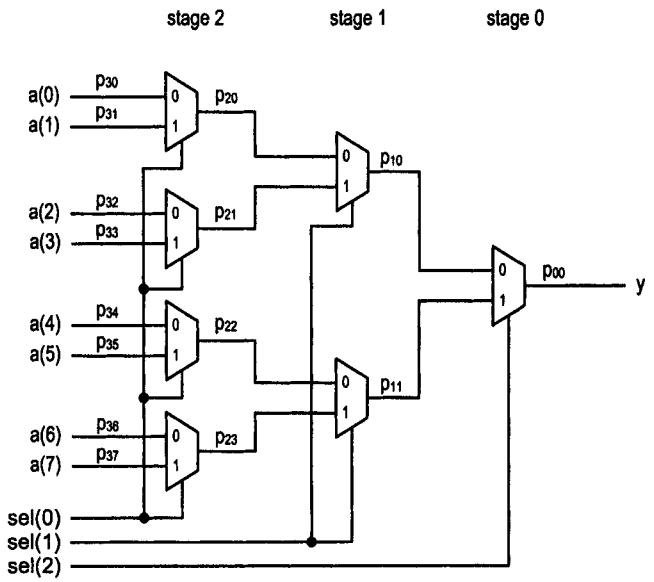


Figure 15.4 Tree-shaped 8-to-1 multiplexer.

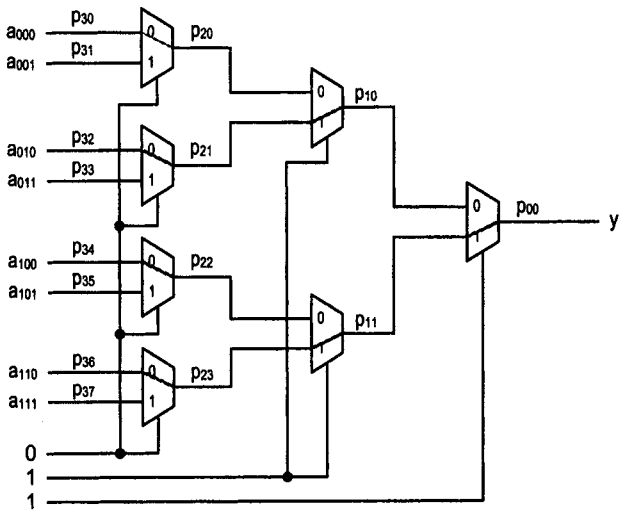


Figure 15.5 Routing with `sel="110"`.

- 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.

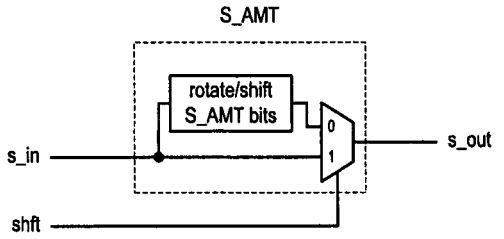
This is an efficient implementation for several reasons. First, as the number of inputs increases, the number of stages grows on the order of $O(\log_2 n)$. The length of the critical path grows in the same order, and thus its performance is much better than the cascading chain. Second, the circuit exhibits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

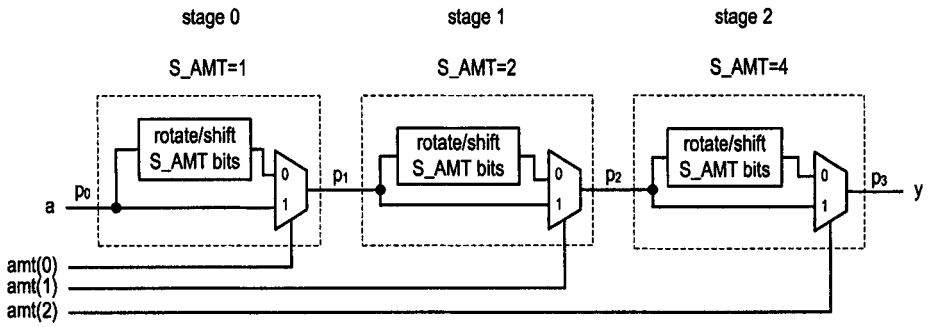
Listing 15.12 Parameterized fixed-size shifting–multiplexing module

```
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.

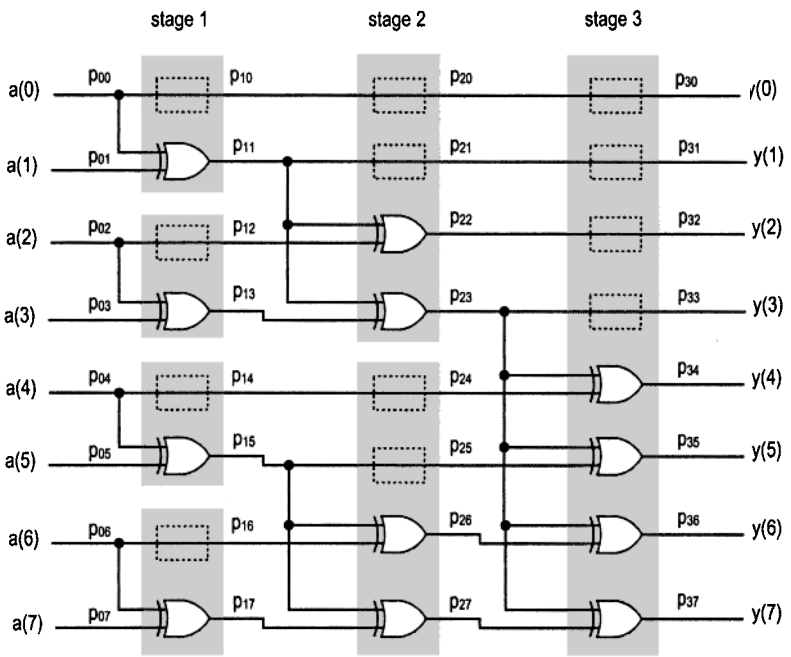


Figure 15.7 Parallel-prefix reduced-xor-vector circuit.

- 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

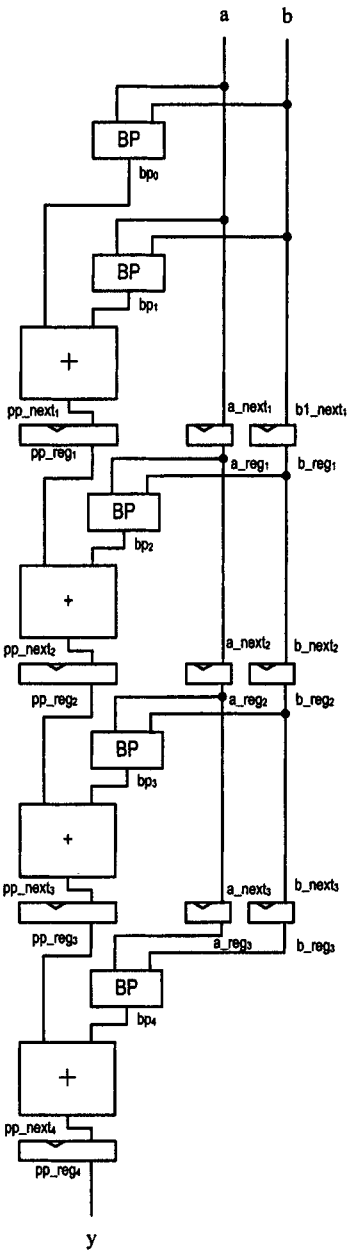


Figure 15.8 Adder-based combinational multiplier with new signal labels.

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.

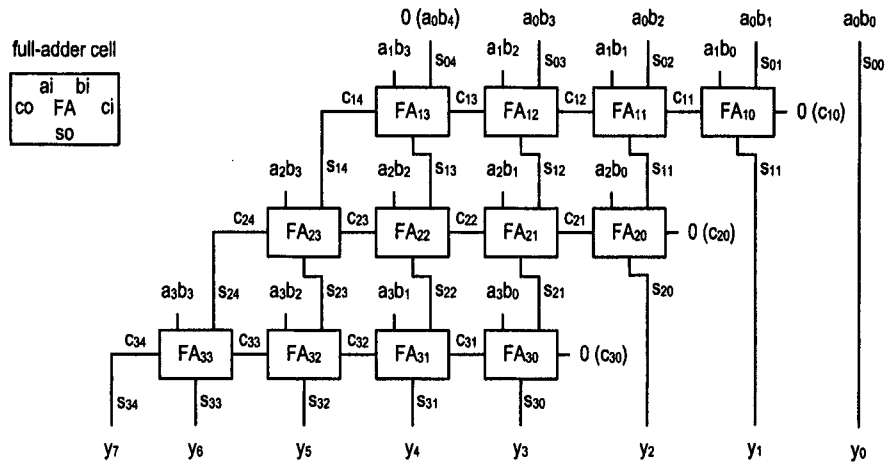


Figure 15.10 Cell-based carry-ripple combinational multiplier.

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.

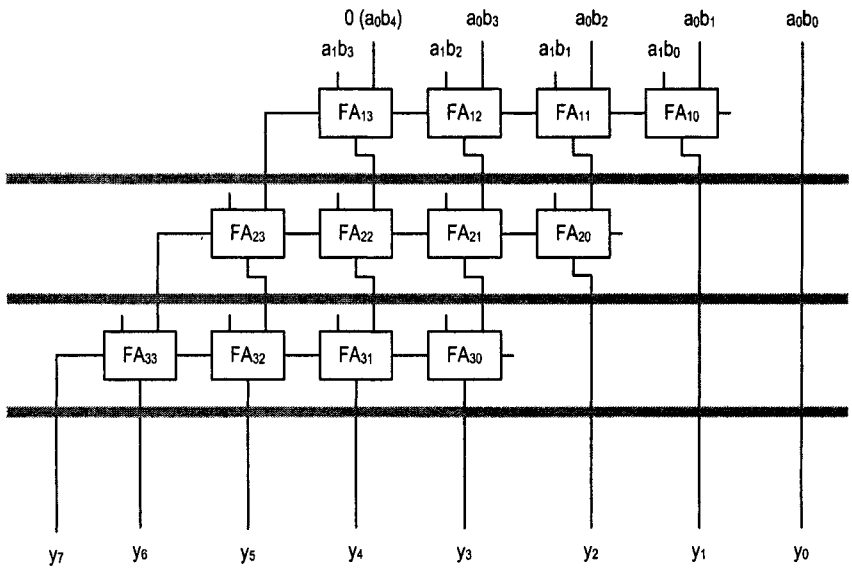


Figure 15.11 Non-optimal pipelined carry-ripple multiplier.

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,

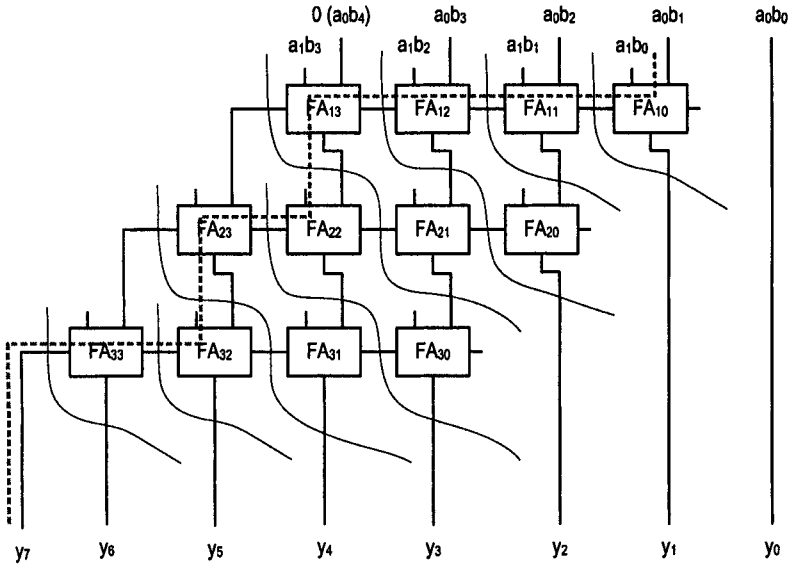


Figure 15.12 Propagation delay contour lines of a carry-ripple multiplier.

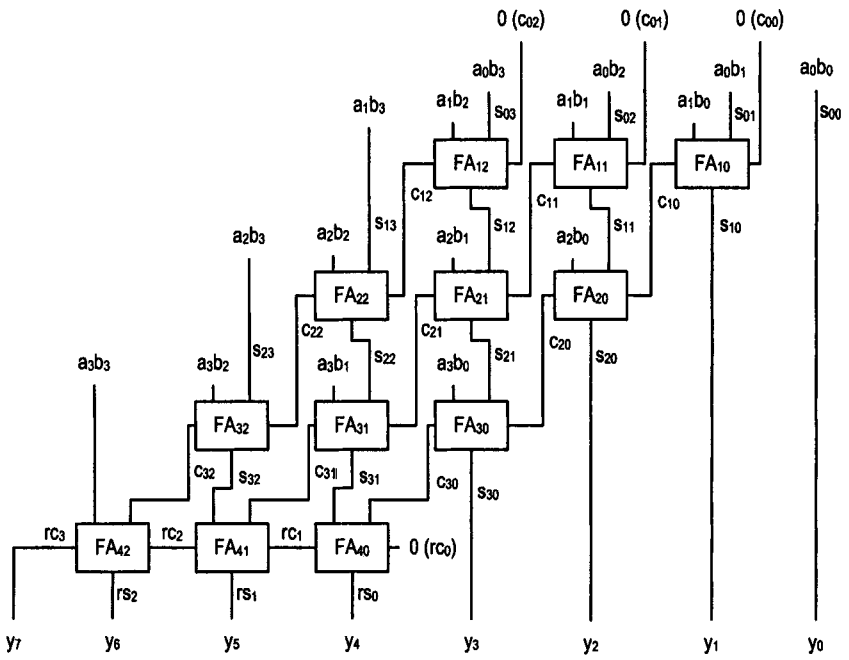


Figure 15.13 Cell-based carry-save multiplier.

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;
```

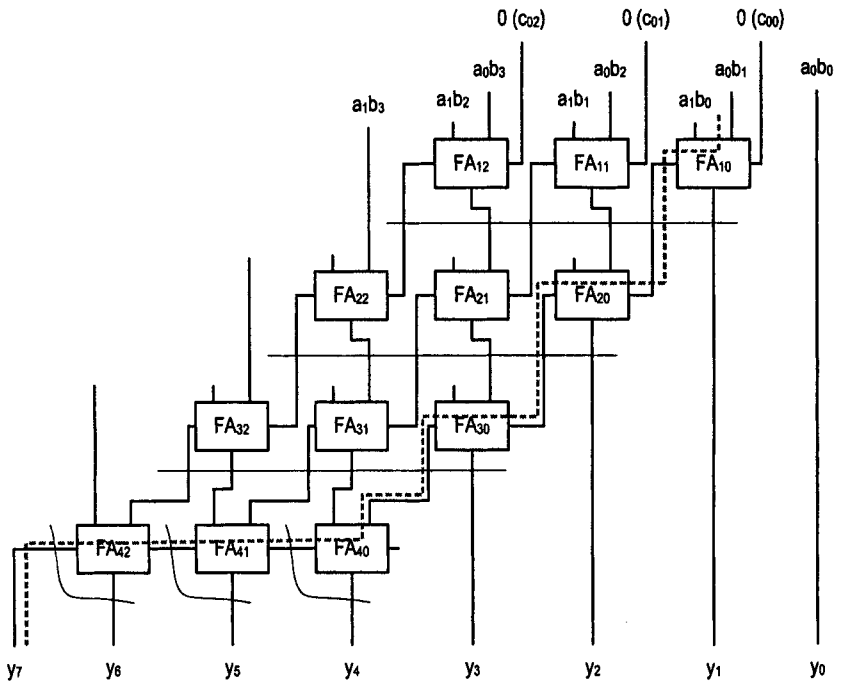


Figure 15.14 Propagation delay contour lines of a carry-save multiplier.

```
    -- 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.

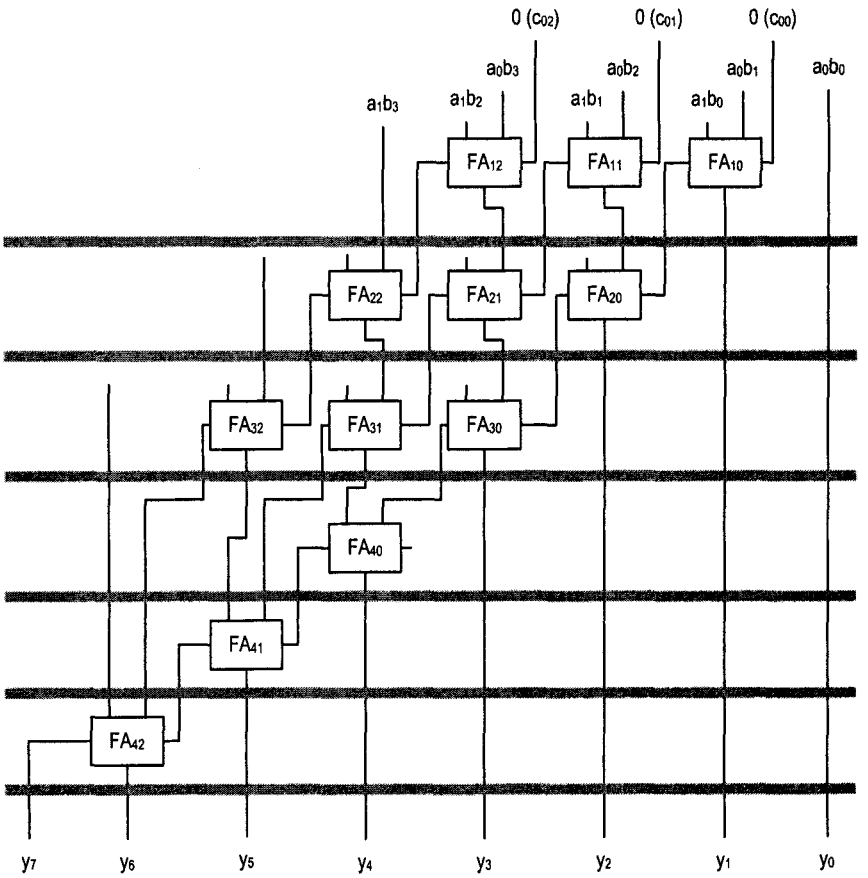


Figure 15.15 Pipelined carry-save multiplier.

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
```

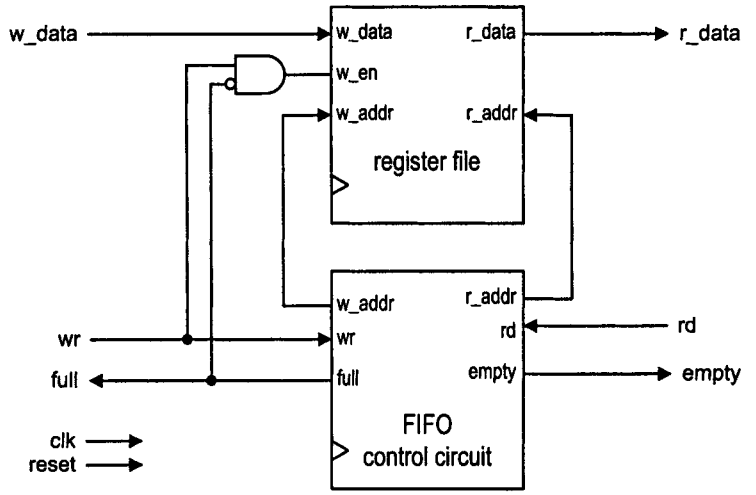


Figure 15.16 Block diagram of a FIFO buffer.

```
        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.

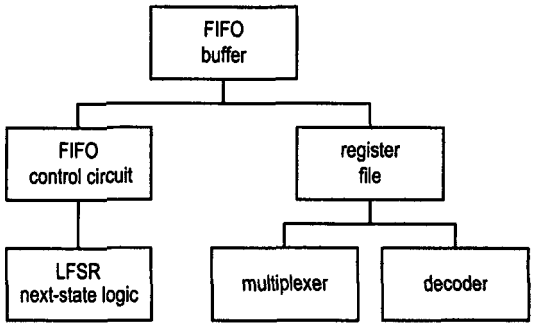


Figure 15.17 Design hierarchy of a FIFO buffer.

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.

This Page Intentionally Left Blank

CHAPTER 16

CLOCK AND SYNCHRONIZATION: PRINCIPLE AND PRACTICE

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

16.1 OVERVIEW OF A CLOCK DISTRIBUTION NETWORK

16.1.1 Physical implementation of a clock distribution network

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

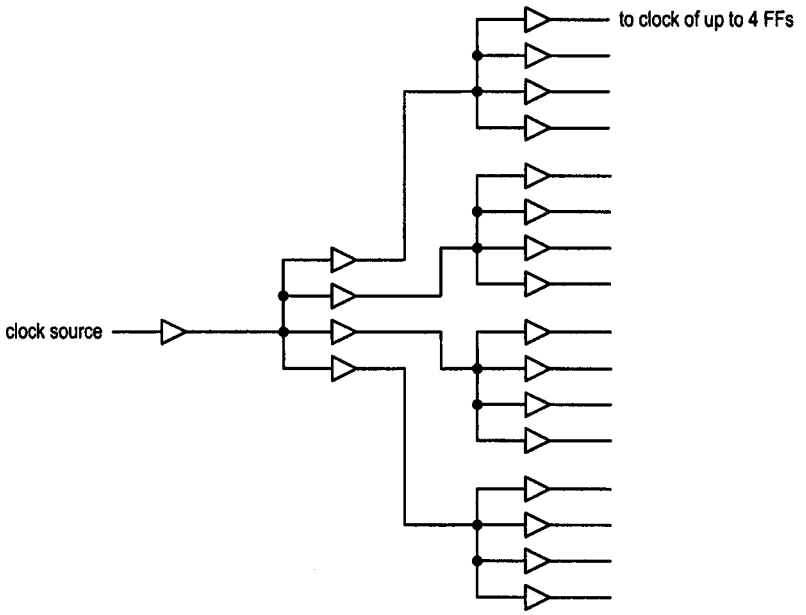


Figure 16.1 Conceptual clock distribution network.

a cell depends on the electrical characteristics of the internal transistors. A typical cell can normally drive up to half a dozen cells.

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

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

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

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

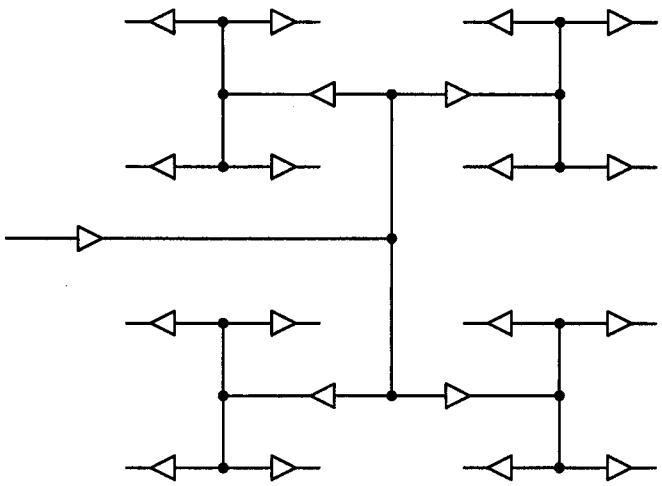


Figure 16.2 Idealized routing of a clock distribution network.

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

16.1.2 Clock skew and its impact on synchronous design

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

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

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

For a moderately-sized system, the clock skew is normally a small fraction (a few percent) of the clock period. We can treat it as an ideal synchronous system and design it accordingly. However, during the analysis of setup time and hold time constraints, the

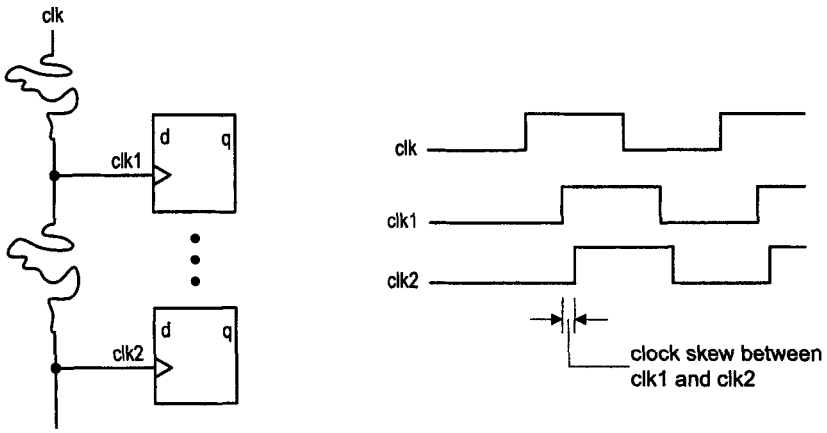


Figure 16.3 Clock skew.

clock skew must be taken into consideration. The skew usually introduces tighter timing requirements and reduces system performance. Current technology can support a clock distribution network with up to several tens of thousands of FFs with an acceptable clock skew. Recall that the proper partition size for synthesis is between 5000 and 50,000 gates. There are no problems applying synchronous design methodology inside each partition block. Section 16.2 discusses the effect of small clock skew in timing analysis.

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

16.2 TIMING ANALYSIS WITH CLOCK SKEW

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

16.2.1 Effect on setup time and maximal clock rate

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

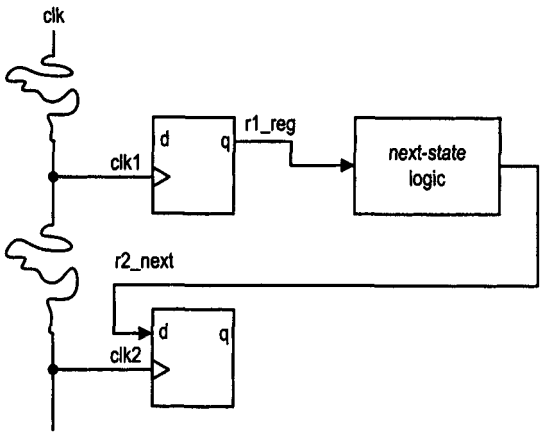


Figure 16.4 Next-state logic feedback path with positive clock skew.

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

The timing diagram is shown in Figure 16.5. We follow the procedure used in Section 8.6.2 to analyze the new circuit. To satisfy the setup time constraint, the `r2_next` signal must be stabilized before t_4. This requirement translates into the condition

$$t_3 < t_4$$

From the timing diagram, we see that

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

and

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

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

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

and the minimal clock period is

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

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

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

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

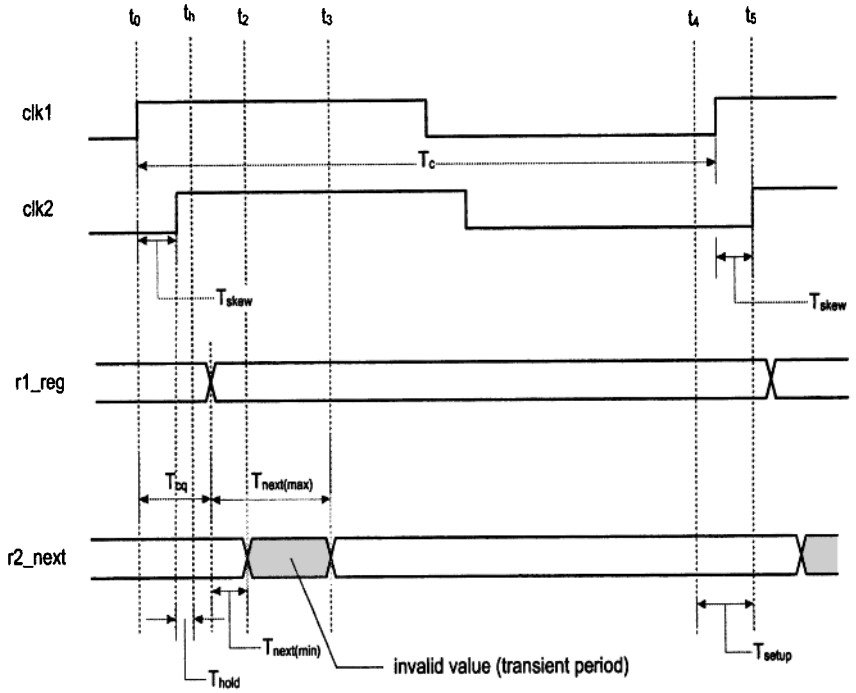


Figure 16.5 Timing analysis of positive clock skew.

and the minimal clock period becomes

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

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

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

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

and

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

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

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

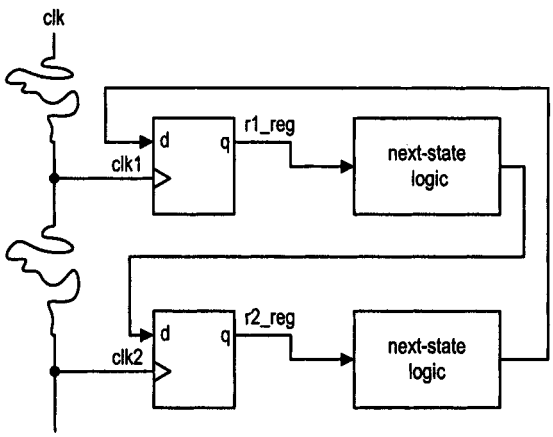


Figure 16.6 Circuit with multiple feedback paths.

16.2.2 Effect on hold time constraint

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

$$t_h < t_2$$

From the timing diagram, we see that

$$t_2 = t_0 + T_{cq} + T_{next(min)}$$

and

$$t_h = t_0 + T_{hold} + T_{skew}$$

After substitution, the inequality equation can be simplified to

$$T_{hold} < T_{cq} + T_{next(min)} - T_{skew}$$

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

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

$$T_{hold} < T_{cq} - T_{skew}$$

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

Unfortunately, there is no fix from the RT-level design. T_{hold}, T_{cq} and T_{skew} are the three parameters in the inequality equation. The first two are inherent characteristics of the device, and the third can only be obtained after synthesis of the clock distribution network. Although in theory we can add some redundant combinational logic (such as a pair of cascading invertors) to introduce artificial propagation delay, this approach is delay sensitive and is not recommended. This problem is better left for the physical design. After

construction of the clock distribution network and completion of the placement and routing, accurate clock skew information can be obtained. The physical design software will check for hold time violations. It should correct the problem by rearranging the clock routing or inserting a delay element in the violated path.

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

$$T_{hold} < T_{cq} + T_{next(min)} + T_{skew}$$

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

16.3 OVERVIEW OF A MULTIPLE-CLOCK SYSTEM

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

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

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

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

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

As discussed in Section 8.2, the synchronous methodology is the fundamental principle in today's digital system development, and most design and analysis schemes are based on

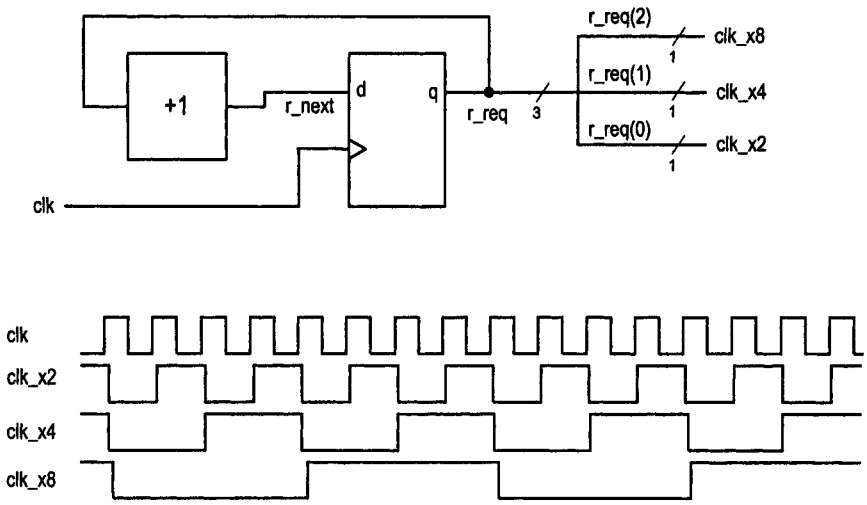


Figure 16.7 Poorly conceived clock divider.

this methodology. Thus, even with multiple clocks, we still want to apply this methodology as much as possible. The basic approach is to divide a system into multiple synchronous subsystems and design a special interface between the subsystems. This allows us to continuously apply the synchronous methodology to design a much larger system.

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

16.3.1 System with derived clock signals

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

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

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

A system with derived clock signals is subjected to the same setup and hold time constraints. Once the clock skew between the two clock signals is known, we can apply the technique in Section 16.2 to analyze timing. The derivation procedure is similar except that we must take into consideration the difference in the clock periods.

In a multiple-clock system, we use the term *clock domain* to describe use of the clock signal. A clock domain is a block of circuitry in which the FFs are controlled by the same clock signal. Although a derived clock signal has a different clock frequency or phase, its relationship to the original clock signal is known. The design and analysis techniques of synchronous sequential circuit can be modified and applied in such a system. Because of this, we consider that subsystems with derived clock signals are in the same clock domain. Note that these derive clock signals need their own individual clock distribution networks even though they are in the same clock domain.

16.3.2 GALS system

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

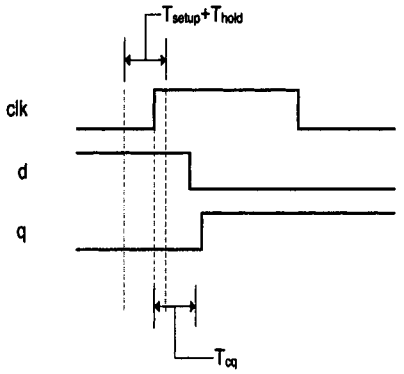
Within a clock domain, the circuit operation is completely synchronous and its design follows the synchronous design methodology. Interface between the two clock domains involves two independent clocks and thus is asynchronous. This configuration is sometimes known as a *globally asynchronous locally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

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

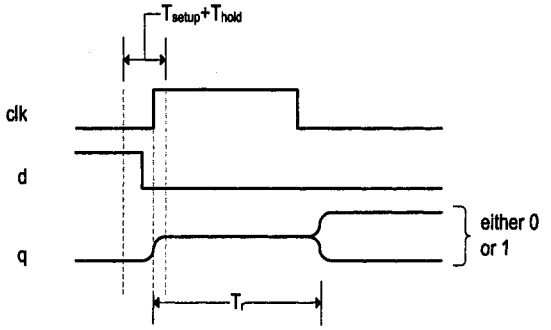
The interface between clock domains is very different from a regular synchronous system or a system with derived clock signals. Its design cannot be automated and usually needs detailed manual analysis. It is more difficult and error-prone. Furthermore, the existence of multiple clock domains affects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

16.4 METASTABILITY AND SYNCHRONIZATION FAILURE

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



(a) Input stable during setup and hold time



(b) Input changing during setup or hold time

Figure 16.8 Timing diagrams of a D FF.

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

16.4.1 Nature of metastability

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

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

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

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

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

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

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

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

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

16.4.2 Analysis of MTBF(T_τ)

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

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

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

- R_{meta}: The average rate at which an FF enters the metastable state.
- $P(T_\tau)$: The probability that an FF cannot resolve the metastable condition within T_τ.

R_{meta} is determined by the formula

$$R_{meta} = w * f_{clk} * f_d$$

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

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

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

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

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

For a given T_r, MTBF(T_r) becomes

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

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

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

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

After simple mathematical manipulation, we can easily convert MTBF(T_{r2}) to MTBF(T_r), and vice versa.

Table 16.1 Sample MTBF(T_r) computation

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

16.4.3 Unique characteristics of MTBF(T_r)

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

Another observation about the resolution time relates to its highly non-linear characteristics. Note that T_r is in the exponent position of the MTBF(T_r) formula. A small variation over T_r leads to drastic change in the value of MTBF. For example, consider an FF with a w of 0.1 ns and a τ of 0.5 ns and assume that the system clock frequency (f_{clk}) is 50 MHz and the data rate (f_d) is $0.1 f_{clk}$. The resolution time of a synchronizer is normally ranged between a fraction of a clock period to one or two clock periods (discussed in the next section). Table 16.1 lists the MTBF values of T_r from 0 to 35 ns at increments of 2.5 ns. Note that the period of a 50-MHz clock signal is 20 ns. When no resolution time is provided (i.e., $T_r = 0$), the MTBF is an unacceptable 0.04 ms. If we can use a T_r value of half a clock period (i.e., 10 ns), the MTBF becomes about 5 hours. Because of the exponential rate, each extra 2.5 ns can increase the MTBF more than 100 times. When T_r reaches 17.5 ns, the MTBF reaches about 2000 years. If we provide 1.5 times the clock period (i.e., 30 ns), the MTBF becomes about 10^{14} years (for comparison, the age the universe is on the order of 10^{11} years, and the appearance of the human being is on the order of 10^{5} years).

This phenomenon is a mixed blessing. On the positive side, while the synchronizing failure cannot be eliminated, we can make the probability of occurrence extremely small.

On the negative side, because of the sensitivity of the resolution time, a small decrease in the resolution time can significantly degrade the value of MTBF. Thus, minor revisions in the system, such as the slight increment of the system clock rate or use of an FF with a slightly larger setup time, may lead to a drastic consequence.

16.5 BASIC SYNCHRONIZER

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

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

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

16.5.1 The danger of no synchronizer

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

16.5.2 One-FF synchronizer and its deficiency

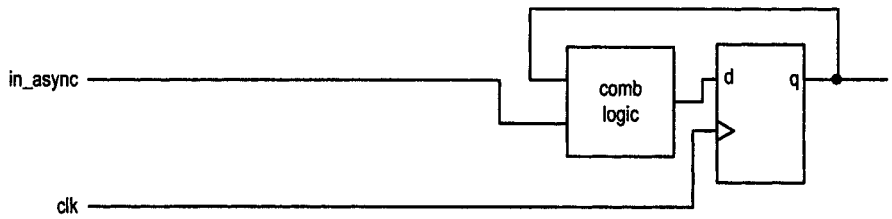
The first design of a synchronizer is to use a single D FF, as shown in Figure 16.9(b). Let T_c, T_{setup} and T_{comb} be the clock period of the system, the setup time of the FF and the propagation delay of the combinational circuit respectively. Consider the path from the synchronizer D FF to the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

$$T_r = T_c - (T_{comb} + T_{setup})$$

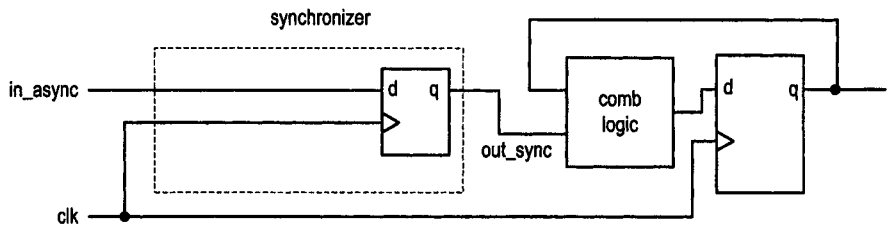
Assume that T_{setup} of the system register is 2.5 ns. The resolution time of this circuit becomes

$$T_r = 20 - (T_{comb} + 2.5) = 17.5 - T_{comb}$$

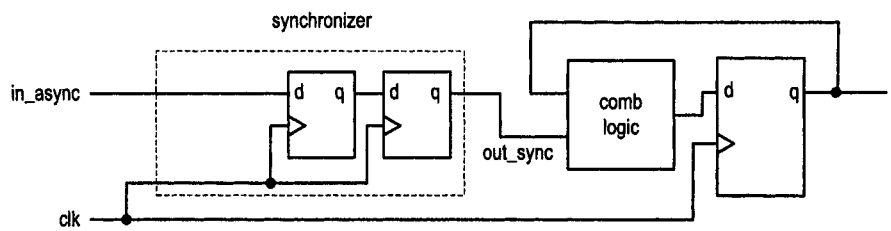
T_r and MTBF depend on T_{comb}, the propagation delay of the combinational circuit. For a simple combination circuit, the T_r will be relatively large. For example, if T_{comb} is 1 ns,



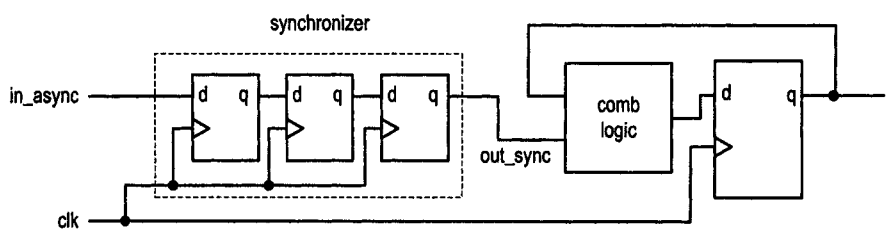
(a) No synchronizer



(b) One-FF synchronizer



(c) Two-FF synchronizer



(d) Three-FF synchronizer

Figure 16.9 Synchronizers.

T_r becomes 16.5 ns and MTBF(16.5 ns) is about 272 years. On the other hand, a complex combination circuit can drastically reduce the MTBF value. If T_{comb} is 12.5 ns, T_r becomes 5 ns and MTBF(5 ns) is dropped to about 0.88 second.

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

16.5.3 Two-FF synchronizer

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

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

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

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

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

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

Listing 16.1 Two-FF synchronizer

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

architecture two_ff_arch of synchronizer is
   signal meta_reg, sync_reg: std_logic;
   signal meta_next, sync_next: std_logic;
begin
   -- two D FFs
   process(clk,reset)
```

```
    begin
       if (reset='1') then
          meta_reg <= '0';
          sync_reg <= '0';
       elsif (clk'event and clk='1') then
          meta_reg <= meta_next;
          sync_reg <= sync_next;
       end if;
    end process;
    -- next-state logic
    meta_next <= in_async;
    sync_next <= meta_reg;
    -- output
    out_sync <= sync_reg;
end two_ff_arch;
```

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

16.5.4 Three-FF synchronizer

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

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

$$R_{meta} = w * f_{clk} * f_d$$

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

$$P1 = e^{-\frac{T_c - T_{setup}}{\tau}}$$

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

$$P2 = e^{-\frac{T_c - T_{setup}}{\tau}}$$

The MTBF of this circuit becomes

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

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

$$\text{MTBF} = \frac{1}{R_{meta} * P1} = \frac{e^{\frac{(T_c - T_{setup})}{\tau}}}{w * f_{clk} * f_d}$$

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

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

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

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

16.5.5 Proper use of a synchronizer

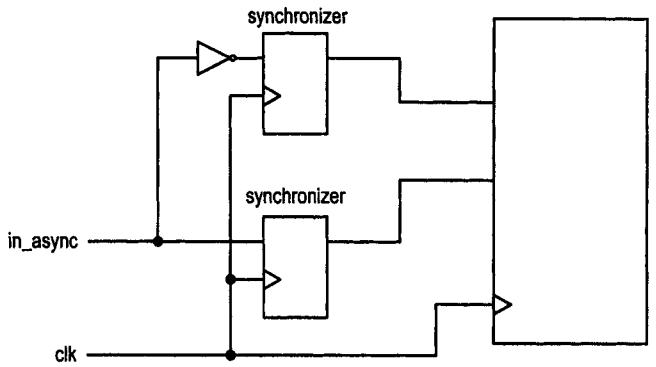
The function of a synchronizer is to provide a non-metastable output value. We must use it properly to obtain a reliable synchronized result. Good design practices can help us to achieve this goal and avoid subtle errors:

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

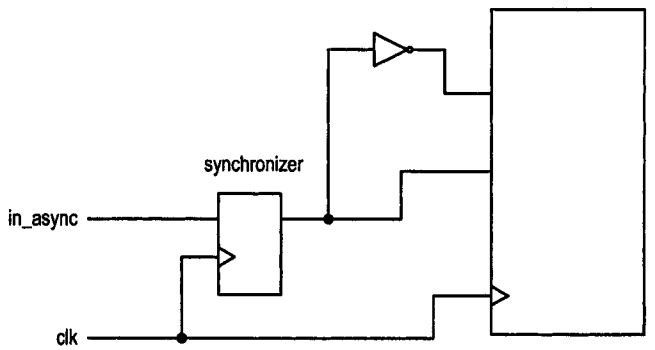
These practices are discussed in the following paragraphs.

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

Synchronize a signal in a single place The function of a synchronizer is to generate a stable output value. The synchronizer, however, cannot guarantee which value will be reached. For example, if a timing violation occurs when the input changes from '0' to '1', the synchronized input value can be '0' or '1' at the current sampling clock edge. Assume that the input signal does not change. It will be sampled again at the next rising edge of the clock and a stable '1' will be obtained. This implies that the arrival time of a synchronized asynchronous input signal may exhibit a random one-clock delay. We must take the random delay into consideration when using a synchronizer.



(a) Synchronizing a signal in two places



(b) Synchronizing a signal in one place

Figure 16.10 Synchronization at multiple places.

An asynchronous input signal may be used in multiple places in a clock domain. It should be synchronized in a single entry point. An example of a poor design is shown in Figure 16.10(a), in which the `in_async` signal and its derivative are synchronized by two individual synchronizers. The potentially random one-clock delay may introduce inconsistent values to the system and lead to incorrect operation. A better alternative is shown in Figure 16.10(b). The signal is synchronized by a single synchronizer, and the system is always fed with the same value.

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

There are two ways to correct the problem. The first is to apply special coding patterns, such as Gray code, to ensure that only one bit changes during the transition. One example

is given in Section 16.9.1. A better, more systematic alternative is to bundle all signals and pass them as a single data item. The data transfer between two clock domains is discussed in Section 16.8.

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

16.6 SINGLE ENABLE SIGNAL CROSSING CLOCK DOMAINS

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

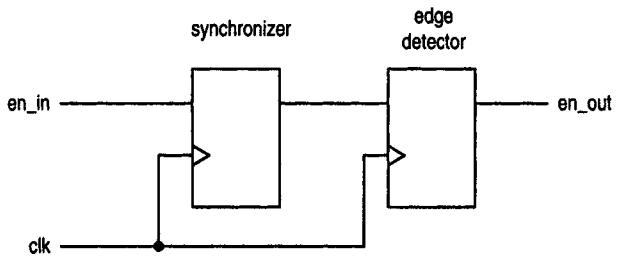
16.6.1 Edge detection scheme

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

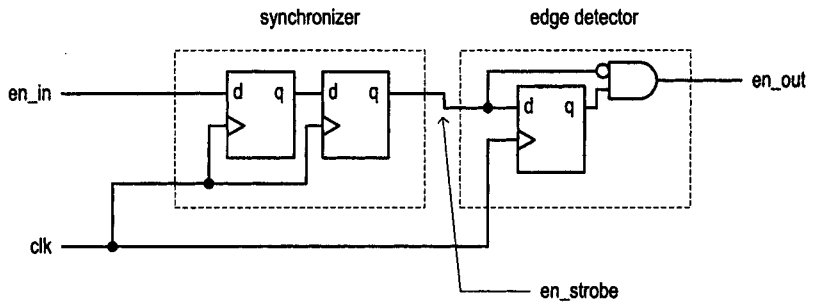
While using an enable pulse between two synchronous subsystems is straightforward, it is much harder to pass the pulse crossing the clock domains. We must consider the synchronization and the difference in clock rates. The following subsections discuss several ad hoc edge detection schemes to regenerate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

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

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



(a) Block diagram



(b) Correct circuit diagram

(c) Incorrect implementation

Figure 16.11 Regeneration of a wide enable signal.

Listing 16.2 Rising-edge detection circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity rising_edge_detector is
   port(
      clk, reset: in std_logic;
      strobe: in std_logic;
      pulse: out std_logic
   );
end rising_edge_detector;

architecture direct_arch of rising_edge_detector is
   signal delay_reg: std_logic;
begin
   -- delay register
   process(clk,reset)
   begin
      if (reset='1') then
         delay_reg <= '0';
```

```
      elsif (clk'event and clk='1') then
         delay_reg <= strobe;
      end if;
   end process;
   -- decoding logic
   pulse <= (not delay_reg) and strobe;
end direct_arch;
```

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

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

Listing 16.3 Enable pulse regenerator for a wide enable signal

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

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

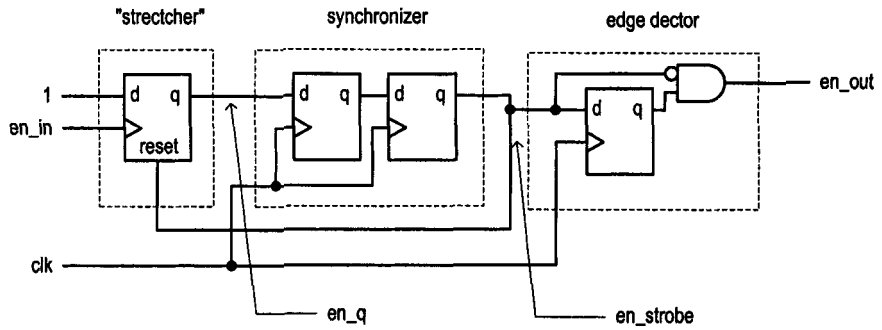


Figure 16.12 Regeneration of a narrow enable signal.

We may be tempted to use the second D FF of the synchronizer to function as the edge detection circuit to save a D FF and to reduce the propagation delay, as shown in Figure 16.11(c). This is a poor design since the unresolved signal may leak through the and cell and propagate to the downstream logic.

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

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

Listing 16.4 Enable pulse regenerator for a narrow enable signal

```
architecture fast_en_arch of sync_en_pulse is
   component synchronizer
      port(
         clk, reset: in std_logic;
         in_async: in std_logic;
         out_sync: out std_logic
      );
   end component;
   component rising_edge_detector
      port(
         clk, reset: in std_logic;
         strobe: in std_logic;
         pulse: out std_logic
      );
```

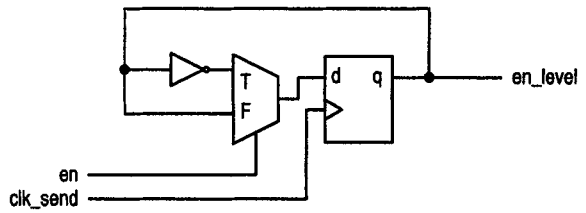
```
    end component;
    signal en_strobe: std_logic;
    signal en_q: std_logic;
begin
    -- ad hoc stretcher
    process(en_in,en_strobe)
    begin
       if (en_strobe='1') then
          en_q <= '0';
       elsif (en_in'event and en_in='1') then
          en_q <= '1';
       end if;
    end process;
    -- slow enable pulse regenerator
    sync: synchronizer
       port map (clk=>clk, reset=>reset, in_async=>en_q,
                 out_sync=>en_strobe);
    edge_detect: rising_edge_detector
       port map (clk=>clk, reset=>reset, strobe=>en_strobe,
                 pulse=>en_out);
end fast_en_arch;
```

Since the function of the first D FF depends only on the rising edge, not on the duration, of the incoming pulse, this scheme can be applied to a wide enable pulse as well. Note that the incoming enable pulse must be glitch-free to prevent false triggering.

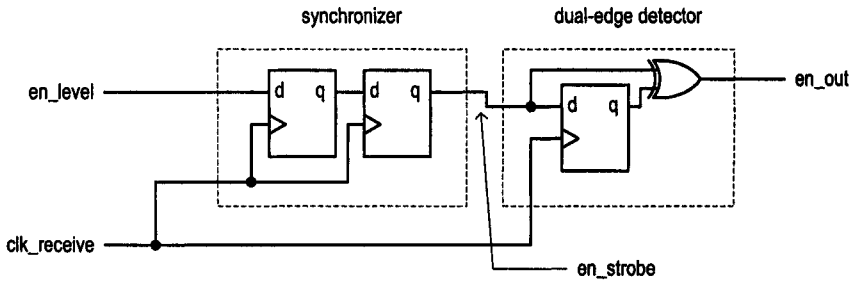
16.6.2 Level-alternation scheme

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

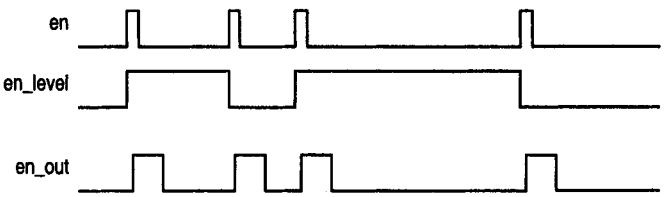
```
. . .
-- D FF
process(clk, reset)
begin
   if (reset='1') then
      t_next <= '0';
   elsif (clk'event and clk='1') then
      t_reg <=  t_next;
   end if;
end process;
-- next-state logic
t_next <= not (t_reg) when en='1' else
          t_reg;
-- output logic
en_level <= t_reg
. . .
```



(a) Block diagram of the sending subsystem



(b) Block diagram of the receiving subsystem



(c) Simplified timing diagram

Figure 16.13 Pulse regeneration with level alternation.

In the domain that receives the enable pulse, it needs a synchronizer and a dual-edge detection circuit that can detect both the rising and falling edges of an input signal. The edge detection circuit senses the change in signal level and converts it back to a single one-clock-period pulse. We can derive the dual-edge detection circuit by using an FSM or direct implementation. One possible direct implementation is to perform an xor operation over the current input value and the previous input value stored in a D FF, as shown in Figure16.13(b). The output waveform of the regenerator is demonstrated in the bottom part of the timing diagram in Figure 16.13(c). For clarity, the synchronizer delay is not included in the diagram.

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

Listing 16.5 Dual-edge detection circuit

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

```
5       clk, reset: in std_logic;
        strobe: in std_logic;
        pulse: out std_logic
    );
end dual_edge_detector;
10
architecture direct_arch of dual_edge_detector is
  signal delay_reg: std_logic;
begin
    -- delay register
15  process(clk,reset)
    begin
       if (reset='1') then
          delay_reg <= '0';
       elsif (clk'event and clk='1') then
20        delay_reg <= strobe;
       end if;
    end process;
    -- decoding logic
    pulse <= delay_reg xor strobe;
25 end direct_arch;
```

Listing 16.6 Enable pulse regenerator using the level-alternation scheme

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

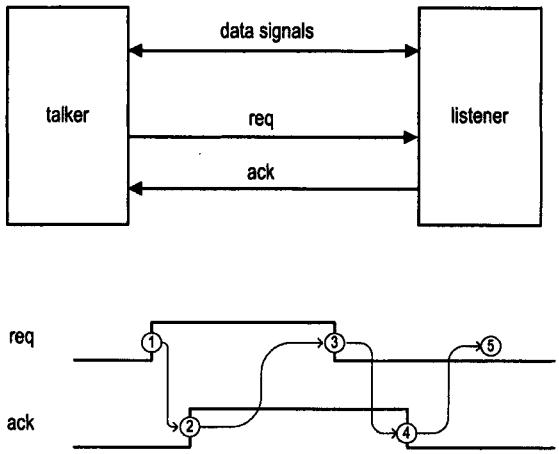


Figure 16.14 Basic conceptual and timing diagrams of the four-phase handshaking protocol.

16.7 HANDSHAKING PROTOCOL

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

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

16.7.1 Four-phase handshaking protocol

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

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

1. The talker activates the `req` signal.

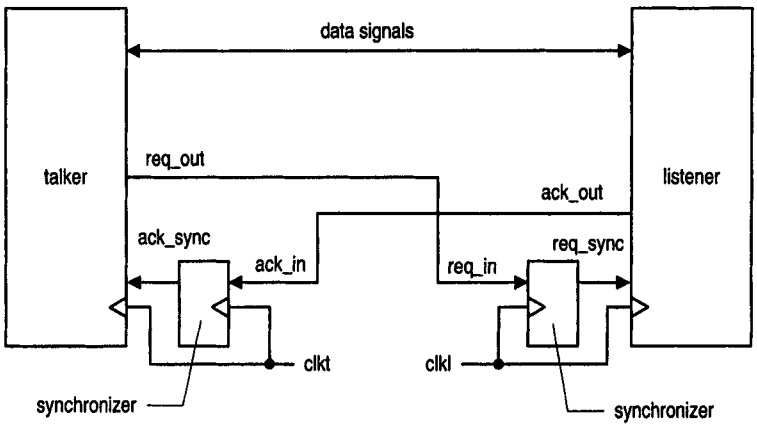


Figure 16.15 Handshaking system with synchronizers.

2. When the listener detects activation of the `req` signal, it activates the `ack` signal to inform the talker.
3. When the talker senses activation of the `ack` signal, it deactivates the `req` signal.
4. After the listener detects deactivation of the `req` signal, it deactivates the `ack` signal accordingly.
5. Once the talker senses deactivation of the `ack` signal, it returns to the initial state. The talker can issue a new request if needed.

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

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

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

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

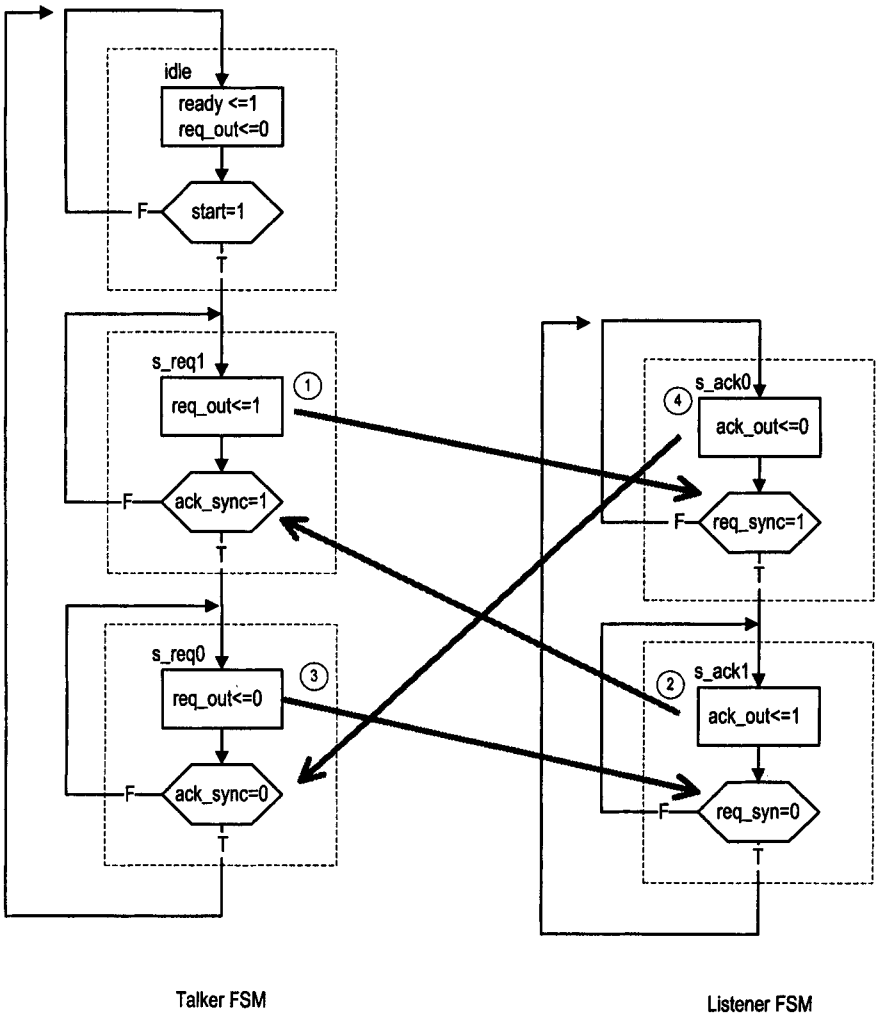


Figure 16.16 ASM charts of the talker and listener of the four-phase handshaking protocol.

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

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

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

Listing 16.7 Talker FSM of the four-phase handshaking protocol

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

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

```
            if ack_sync='1' then
               state_next <= s_req0;
            end if;
         when s_req0 =>
            if ack_sync='0' then
               state_next <= idle;
            end if;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      case state_next is
         when idle =>
            req_buf_next <= '0';
         when s_req1 =>
            req_buf_next <= '1';
         when s_req0 =>
            req_buf_next <= '0';
      end case;
   end process;
   req_out<= req_buf_reg;
end arch;
```

Listing 16.8 Listener FSM of the four-phase handshaking protocol

```
library ieee;
use ieee.std_logic_1164.all;
entity listener_fsm is
   port(
      clk, reset: in std_logic;
      req_sync: in std_logic;
      ack_out: out std_logic
   );
end listener_fsm;

architecture arch of listener_fsm is
   type l_state_type is (s_ack0, s_ack1);
   signal state_reg, state_next: l_state_type;
   signal ack_buf_reg, ack_buf_next: std_logic;
begin
   -- state register and output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         state_reg <= s_ack0;
         ack_buf_reg <='0';
      elsif (clk'event and clk='1') then
         state_reg <= state_next;
         ack_buf_reg <= ack_buf_next;
      end if;
   end process;
   -- next-state logic
```

```
    process(state_reg,req_sync)
    begin
30      state_next <= state_reg;
        case state_reg is
            when s_ack0 =>
                if req_sync='1' then
                    state_next <= s_ack1;
35              end if;
            when s_ack1 =>
                if req_sync='0' then
                    state_next <= s_ack0;
                end if;
40          end case;
    end process;
    -- look-ahead output logic
    process(state_next)
    begin
45      case state_next is
            when s_ack0 =>
                ack_buf_next <= '0';
            when s_ack1 =>
                ack_buf_next <= '1';
50          end case;
    end process;
    ack_out<= ack_buf_reg;
end arch;
```

To complete the design, synchronizers are needed for the input signals. Again, to emphasize the unique characteristics of the synchronization circuits, we separate them from the regular sequential circuits and instantiate the synchronizers in the top level. The VHDL codes of the complete design follow the block diagram in Figure 16.15 and are shown in Listings 16.9 and 16.10 respectively.

Listing 16.9 Talker of the four-phase handshaking protocol

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

architecture str_arch of talker_str is
15  signal ack_sync: std_logic;
    component synchronizer
        port(
            clk: in std_logic;
            in_async: in std_logic;
```

```
        reset: in std_logic;
        out_sync: out std_logic
    );
  end component;
  component talker_fsm
    port(
        ack_sync: in std_logic;
        clk: in std_logic;
        reset: in std_logic;
        start: in std_logic;
        ready: out std_logic;
        req_out: out std_logic
    );
  end component;
begin
  sync_unit: synchronizer
    port map (clk=>clkt, reset=>resett, in_async=>ack_in,
              out_sync=>ack_sync);
  fsm_unit: talker_fsm
    port map (clk=>clkt, reset=>resett, start=>start,
              ack_sync=>ack_sync, ready=>ready,
              req_out=>req_out);
end str_arch;
```

Listing 16.10 Listener of the four-phase handshaking protocol

```
library ieee;
use ieee.std_logic_1164.all;
entity listener_str is
  port(
    clkl: in std_logic;
    resetl: in std_logic;
    req_in: in std_logic;
    ack_out: out std_logic
  );
end listener_str;

architecture str_arch of listener_str is
  signal req_sync: std_logic;
  component listener_fsm
    port (
        clk: in std_logic;
        req_sync: in std_logic;
        reset: in std_logic;
        ack_out: out std_logic
    );
  end component;
  component synchronizer
    port (
        clk: in std_logic;
        in_async: in std_logic;
        reset: in std_logic;
        out_sync: out std_logic
```

```
      );
   end component;
begin
   sync_unit: synchronizer
      port map (clk=>clk1, reset=>reset1, in_async=>req_in,
                out_sync=> req_sync);
   fsm_unit: listener_fsm
      port map (clk=>clk1, reset=>reset1, req_sync=>req_sync,
                ack_out => ack_out);
end str_arch;
```

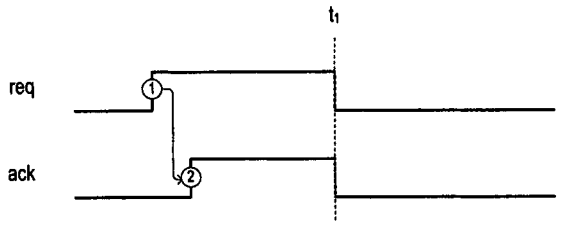
We can use this protocol to pass an enable pulse across the clock domain by connecting the en signal to the start signal of the talker FSM. When an enable pulse arrives, the talker initiates the handshaking operation. When the listener detects the activation edge of the req_in signal, it can also generate an output pulse, which corresponds to the regenerated enable pulse in the new clock domain. Since the sending subsystem cannot generate another enable pulse until the handshaking operation is completed, the sending subsystem will not overrun the the receiving subsystem.

At first glance, the four phases may appear to be somewhat redundant. We may be tempted to discard the second half of the handshaking to simplify the FSMs and let the talker and listener return to the initial state automatically. Let us consider what happens if this is done. Assume that the talker and the listener deactivate the req and ack signals automatically after the system reaches the "11" phase. There will be no problem if the deactivations are done simultaneously, as shown in the timing diagram of Figure 16.17(a). However, since the two subsystems are driven by different clocks, this is hardly possible. If the talker is much slower, the listener may be fooled into thinking that the asserted req signal is the initiation of a new request, as shown in the timing diagram of Figure 16.17(b). At time t_2, the listener deactivates the ack signal. It then senses the activation of the req signal and mistakenly treats the condition as a new round of handshaking and responds accordingly. Thus, the same incoming request pulse will be incorrectly regenerated again. On the other hand, if the listener is too slow, the talker may start to send a new request when the ack signal is still asserted, as shown in the timing diagram of Figure 16.17(c). At time t_3, the talker mistakenly thinks that the handshaking is completed and starts a new round shortly after. Since the listener still processes the first request, the new request will be lost. These examples show that all steps are needed in the original four-phase handshaking protocol.

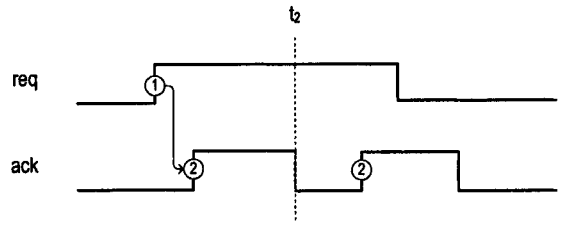
16.7.2 Two-phase handshaking protocol

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

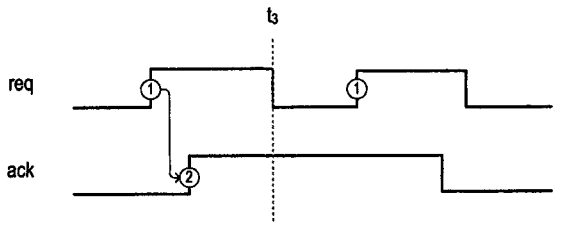
We can make the handshaking scheme more efficient by including only a single information exchange in the protocol. In this scheme, we do not require the system to return to the original state and define that the system is idle when the req and ack signals are



(a) Ideal scenario



(b) Error due to slow talker response



(c) Error due to slow listener response

Figure 16.17 Timing diagrams of an erroneous protocol.

both '0' or both '1'. The system will alternate between the two representations of the idle state.

The operation sequence of the new protocol includes the following steps:

1. The talker activates the `req` signal.
2. When the listener detects activation of the `req` signal, it activates the `ack` signal to inform the talker.
3. After the talker senses activation of the `ack` signal, it knows that the handshaking is completed and the system reaches the idle state.

Note that the values of the `req` and `ack` signals are "11". When a new round of handshaking is initiated, the system starts from the "11" state and the steps are:

1. The talker deactivates the `req` signal.
2. When the listener detects deactivation of the `req` signal, it deactivates the `ack` signal to inform the talker.
3. After the the talker senses deactivation of the `ack` signal, it knows that the handshaking is completed and the system reaches the idle state.

Note that the values of the `req` and `ack` signals are "00" now, and thus the system returns to its initial state. The timing diagram is shown in Figure 16.18. Although the appearance of the four-phase and two-phase timing diagrams are similar, interpretation of the `req` and `ack` signals (i.e., system state) is very different.

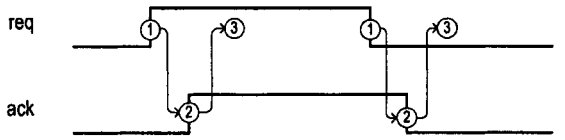


Figure 16.18 Timing diagram of the two-phase handshaking protocol.

We can follow the previous procedure to derive the talker and listener FSMs for the two-phase handshaking protocol. The revised the talker and listener ASM charts are shown in Figure 16.19. Note that the talker FSM stays in the `s_req1` and `s_req0` states until a new round of handshaking is initiated (i.e., when the `start` signal is '1'). Closer observation shows that the `idle` and `s_req0` states of the talker FSM are equivalent, and we can merge the two states and remove the `idle` state.

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

16.8 DATA TRANSFER CROSSING CLOCK DOMAINS

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

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

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

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

The four-phase handshaking transfer has the highest overhead but is most robust. It assumes that the two subsystems have minimal information about each other. One-phase transfer uses a single enable signal with no feedback. It involves minimal overhead, but its operation is based on the assumption that the two subsystems have prior knowledge of the other's timing characteristics.

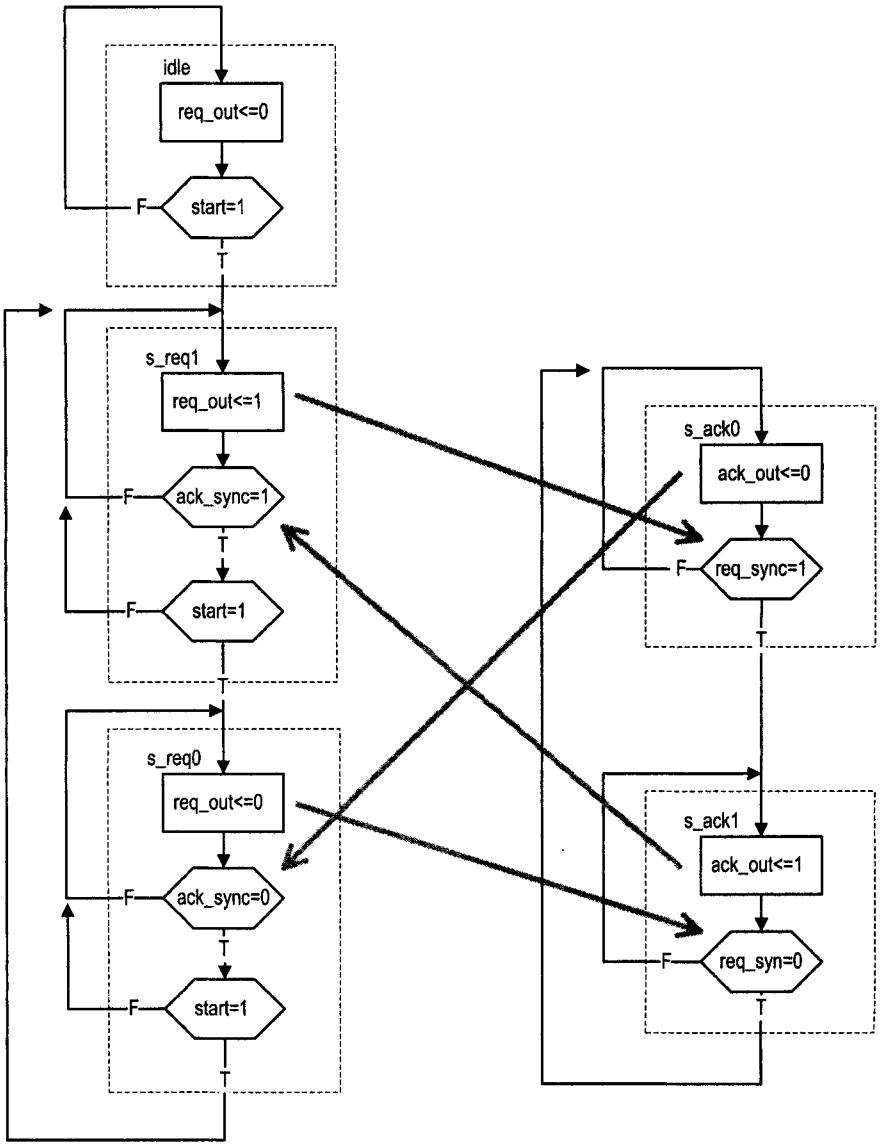


Figure 16.19 ASM charts of the talker and listener of the two-phase handshaking protocol.

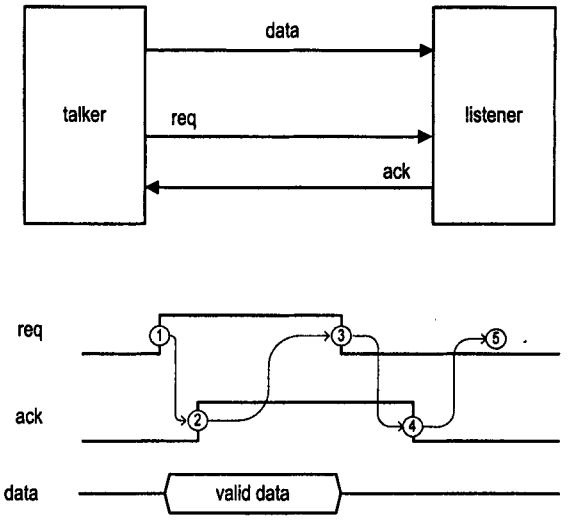


Figure 16.20 Push operation using the four-phase handshaking protocol.

For an asynchronous subsystem, storing data into another subsystem is known as a *push* operation and retrieving data from another subsystem is known as a *pull* operation. Many applications process data stage after stage, and thus the push operation is more common.

16.8.1 Four-phase handshaking protocol data transfer

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

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

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

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

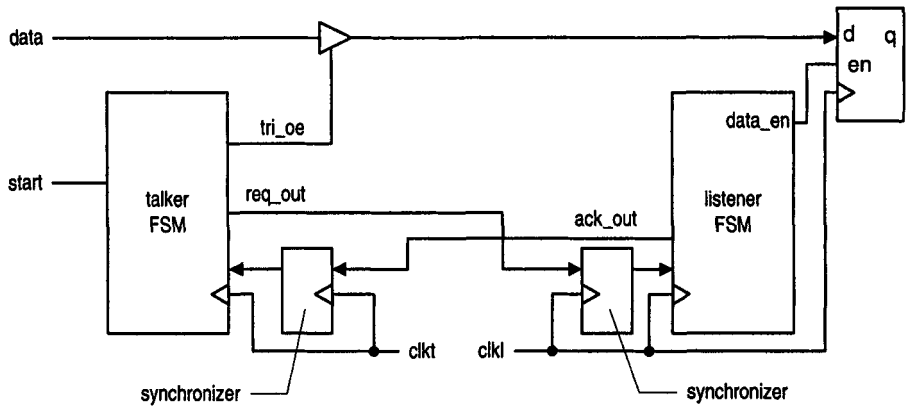


Figure 16.21 Block diagram of the push operation.

the req_out signal is also asserted. We can actually use the req_out signal to control the tri-state buffer.

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

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

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

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

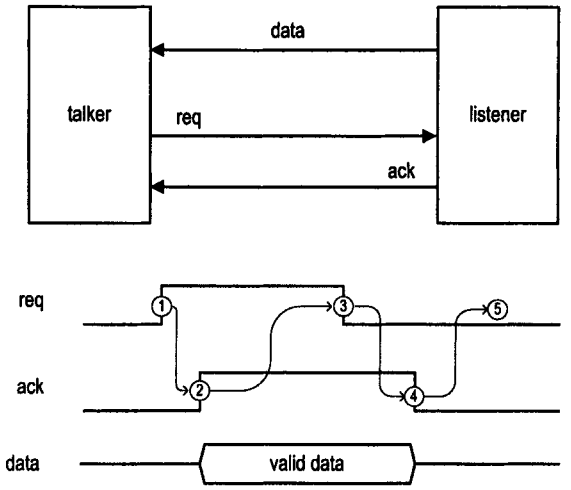


Figure 16.22 Pull operation using the four-phase handshaking protocol.

delayed by the synchronizer, the data signal should be stabilized when the `ack_sync` signal is activated.

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

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

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

The data path of the talker includes a register for the address, a register for the `rw` signal and two data registers to store the transmitted and received data. The data path of the listener is an eight-word register file. In a realistic scenario, the I/O controller should also be able

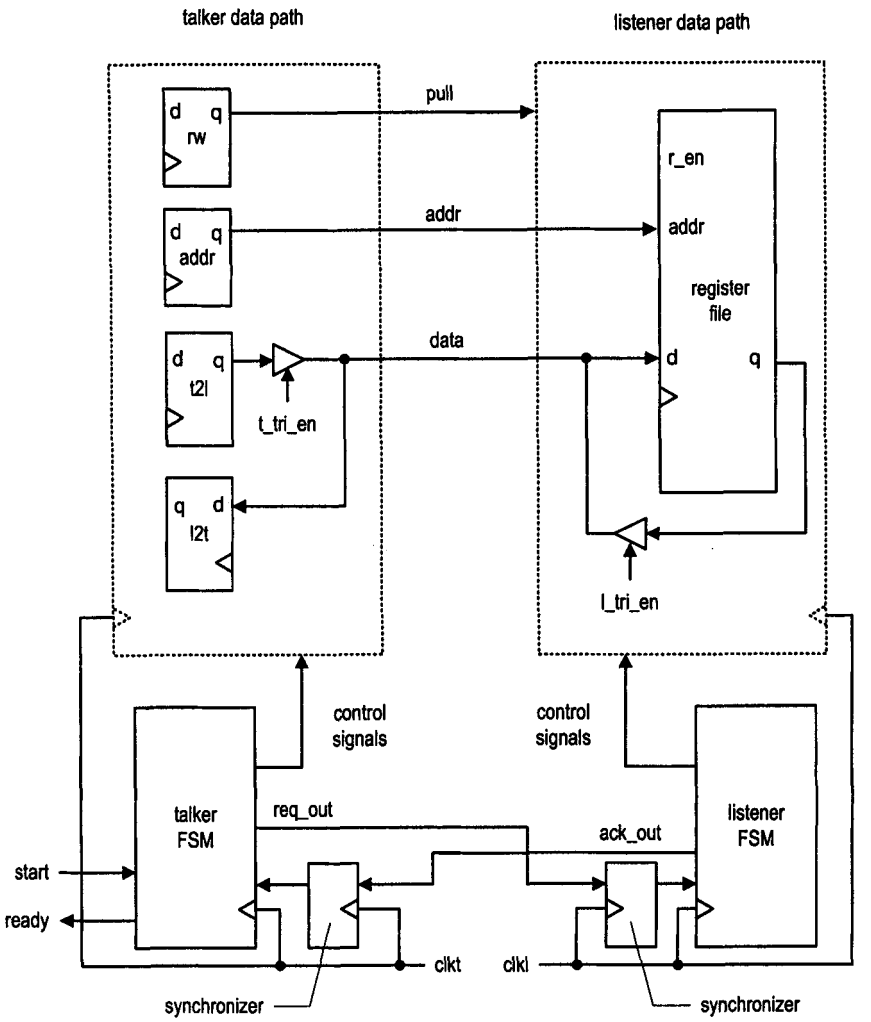


Figure 16.23 Block diagram of a push-and-pull system using the four-phase handshaking protocol.

to access the register file, and thus all signals should be multiplexed. For simplicity, the signals from the I/O controller to the register file are not shown.

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

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

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

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

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

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

```
library ieee;
use ieee.std_logic_1164.all;
entity talker_interface is
   port(
      clkt, resett: in std_logic;
      start, rw: in std_logic;  --read or write to i/o
      ack_sync: in std_logic;
      ready: out std_logic;
      req_out: out std_logic;
      d_t2l: in std_logic_vector(7 downto 0);
      addr_in: in std_logic_vector(1 downto 0);
      d_l2t: out std_logic_vector(7 downto 0);
      pull: out std_logic;
      addr: out std_logic_vector(1 downto 0);
      d: inout std_logic_vector(7 downto 0)
   );
end talker_interface;
```

```
architecture arch of talker_interface is
   type t_state_type is (idle, s_req1, s_req0);
   signal state_reg, state_next: t_state_type;
   signal req_buf_reg, req_buf_next: std_logic;
   signal t_tri_en: std_logic;
   signal l2t_next, l2t_reg: std_logic_vector(7 downto 0);
   signal t2l_next, t2l_reg: std_logic_vector(7 downto 0);
   signal rw_next, rw_reg: std_logic;
   signal addr_next, addr_reg: std_logic_vector(1 downto 0);
begin
   --================
   -- talker FSM
   --================
   -- state register and output buffer
   process(clkt,resett)
   begin
      if (resett='1') then
         state_reg <= idle;
         req_buf_reg <='0';
      elsif (clkt'event and clkt='1') then
         state_reg <= state_next;
         req_buf_reg <=req_buf_next;
      end if;
   end process;
   -- next-state logic
   process(state_reg,start,ack_sync)
   begin
      ready <='0';
      state_next <= state_reg;
      case state_reg is
         when idle =>
            if start='1' then
               state_next <= s_req1;
            end if;
            ready <= '1';
         when s_req1 =>
            if ack_sync='1' then
               state_next <= s_req0;
            end if;
         when s_req0 =>
            if ack_sync='0' then
               state_next <= idle;
            end if;
      end case;
   end process;
   -- look-ahead output logic
   process(state_next)
   begin
      case state_next is
         when idle =>
            req_buf_next <= '0';
         when s_req1 =>
```

```
            req_buf_next <= '1';
         when s_req0 =>
            req_buf_next <= '0';
      end case;
   end process;
   req_out <= req_buf_reg;
   --==================
   -- talker data path
   --==================
   -- data register
   process(clkt,resett)
   begin
      if (resett='1') then
         t2l_reg <= (others=>'0');
         l2t_reg <= (others=>'0');
         addr_reg <= (others=>'0');
         rw_reg <= '0';
      elsif (clkt'event and clkt='1') then
         t2l_reg <= t2l_next;
         l2t_reg <= l2t_next;
         addr_reg <= addr_next;
         rw_reg <= rw_next;
      end if;
   end process;
   -- data path next-state logic
   process(state_reg,t2l_reg,l2t_reg,addr_reg,rw_reg,d_t2l,
           addr_in,d,rw,start,ack_sync)
   begin
      t2l_next <= t2l_reg;
      l2t_next <= l2t_reg;
      addr_next <= addr_reg;
      rw_next <= rw_reg;
      t_tri_en <= '0';
      case state_reg is
         when idle =>
            rw_next <= rw;
            addr_next <= addr_in;
            if (start='1' and rw='0') then -- write to i/o
               t2l_next <= d_t2l;
            end if;
         when s_req1 =>
            if (rw_reg='0') then -- write to i/o
               t_tri_en <= '1';
            end if;
            if (ack_sync='1') and (rw_reg='1') then
             l2t_next <= d;
            end if;
         when s_req0 =>
            t_tri_en <= '0';
      end case;
   end process;
   -- output
   d <= t2l_reg when t_tri_en='1' else (others=>'Z');
```

```
    pull <= rw_reg;
    d_l2t <= l2t_reg;
    addr <= addr_reg;
end arch;
```

Listing 16.12 Listener interface of a push-and-pull system

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity listener_interface is
    port(
        clkl, resetl: in std_logic;
        req_sync: in std_logic;
        ack_out: out std_logic;
        pull: in std_logic;
        addr: in std_logic_vector(1 downto 0);
        d: inout std_logic_vector(7 downto 0)
    );
end listener_interface;

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

```
        when s_ack1 =>
           if req_sync='0' then
              state_next <= s_ack0;
           end if;
        end case;
  end process;
  -- look-ahead output logic
  process(state_next)
  begin
     case state_next is
        when s_ack0 =>
           ack_buf_next <= '0';
        when s_ack1 =>
           ack_buf_next <= '1';
        end case;
  end process;
  ack_out<= ack_buf_reg;
  --====================
  -- listener data path
  --====================
  -- register file
  process(clk1,reset1)
  begin
     if (reset1='1') then
        for i in 0 to 3 loop
           r_file_reg(i) <= (others=>'0');
        end loop;
     elsif (clk1'event and clk1='1') then
        if r_en='1' then
           r_file_reg(to_integer(unsigned(addr))) <= d;
        end if;
    end if;
  end process;
  -- enable logic
  process(state_reg,req_sync,pull)
  begin
     l_tri_en <= '0';
     r_en <= '0';
     case state_reg is
        when s_ack0 =>
           if (req_sync='1')  then
              if (pull='0') then -- push
                 r_en <= '1';
              end if;
           end if;
        when s_ack1 =>
           if (pull='1')then
              l_tri_en <='1';
           end if;
     end case;
  end process;
   -- output
  d <= r_file_reg(to_integer(unsigned(addr)))
```

```
100                          when l_tri_en='1' else
        (others=>'Z');
end arch;
```

As in the handshaking code of Section 16.7.1, we must add two synchronizers for the request and acknowledge signals to complete the implementation.

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

16.8.2 Two-phase handshaking data transfer

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

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

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

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

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

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

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

Using the two-phase handshaking protocol to perform a pull operation is more difficult. The two-phase operation only allows the listener to signal the talker that it has placed the

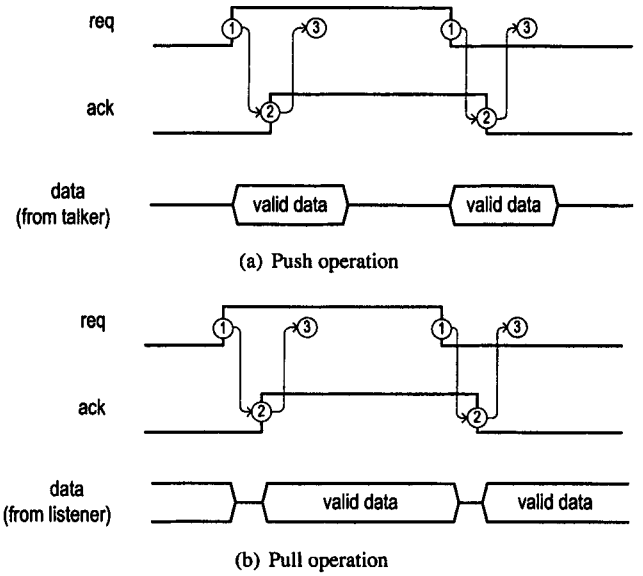


Figure 16.24 Timing diagrams of push and pull operations using two-phase handshaking protocol.

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

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

16.8.3 One-phase data transfer

If the characteristics of the listener are known in advance, we can customize the data transfer timing and eliminate the acknowledge signal. Since there is no feedback, the request signal behaves like the enable pulse discussed in Section 16.6.

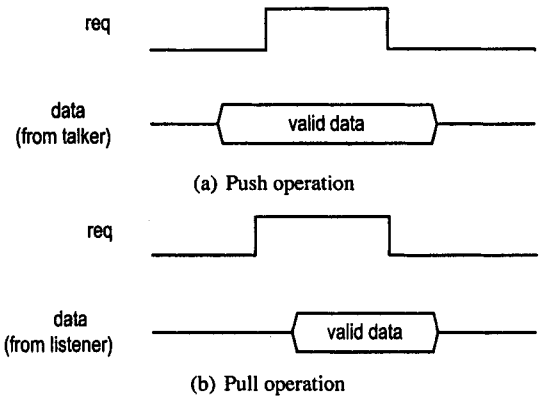


Figure 16.25 Timing diagrams of push and pull operations using one-phase protocol.

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

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

16.9 DATA TRANSFER VIA A MEMORY BUFFER

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

16.9.1 FIFO buffer

A FIFO buffer is like a one-directional pipe. The sending subsystem puts the data in one end of the pipe, and the receiving subsystem retrieves the data from the other end of the

pipe. In Section 9.3.2, we discussed the operation and design of a synchronous FIFO buffer, in which the two subsystems are controlled by the same clock signal. The operation of an asynchronous FIFO is similar, but the sending and receiving subsystems are controlled by clocks from different clock domains.

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

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

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

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

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

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

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

It is straightforward to obtain the n-bit binary counting patterns since they are the same as the n LSBs of the $(n+1)$-bit binary counter. It is more difficult for the Gray counter. For example, the counting patterns of three LSBs of the 4-bit Gray counter and the counting patterns of a 3-bit Gray counter are shown in the second and third columns of Table 16.2. Their patterns are different in the bottom half. Although the patterns are not identical, there

Table 16.2 Circulation pattern of 4-bit and 3-bit Gray counters

4-bit Gray counter	3 LSBs of 4-bit Gray counter	3-bit Gray counter
0000	000	000
0001	001	001
0011	011	011
0010	010	010
0110	110	110
0111	111	111
0101	101	101
0100	100	100
1100	100	000
1101	101	001
1111	111	011
1110	110	010
1010	010	110
1011	011	111
1001	001	101
1000	000	100

is no need to construct a separate n-bit Gray counter from scratch. Closer observation shows that the $(n-1)$ LSBs of the $(n+1)$-bit Gray counter and n-bit Gray counter are identical, and the MSB of the n-bit Gray counter can be obtained by performing an xor operation on the two MSBs of the $(n+1)$-bit Gray counter. In other words, let a_n, a_{n-1}, $\ldots$, a_0 be the bits of an $(n+1)$-bit Gray counter, and b_{n-1}, b_{n-2}, $\ldots$, b_0 be the bits of an n-bit Gray counter. We can derive the n-bit counting pattern by using

$$b_i = \begin{cases} a_{i+1} \oplus a_i & \text{if } i = n-1 \\ a_i & \text{otherwise} \end{cases}$$

The block diagram of an n-bit asynchronous FIFO control circuit is shown in Figure 16.26. In the write control part, an $(n+1)$-bit Gray counter is used as the write pointer and the derived n-bit Gray counter is used for the write address. The read pointer is obtained from the read control part. It is first synchronized by an $(n+1)$-bit synchronizer. The comparing circuit derives the n-bit read address and compares it to the write address. If the read and write addresses are the same and the MSBs of the read and write pointers are different, the FIFO is full and the `full` signal is asserted accordingly. Since all inputs of the comparing circuits are synchronized with the write controller's clock, the `full` signal will not cause a timing violation when used. The read control part essentially mirrors the write control except for the minor difference in the comparing circuit. The `empty` signal will be asserted when the read and write addresses are the same and the MSBs of the read and write pointers are the same.

The VHDL codes of the write port control and the read port control are shown in Listings 16.13 and 16.14 respectively. The code of the Gray counter is similar to the code discussed in Section 7.5.1. We use a generic, `N`, to express the number of bits of the FIFO control circuit. The code of a generic n-bit two-FF synchronizer is shown in Listing 16.15.

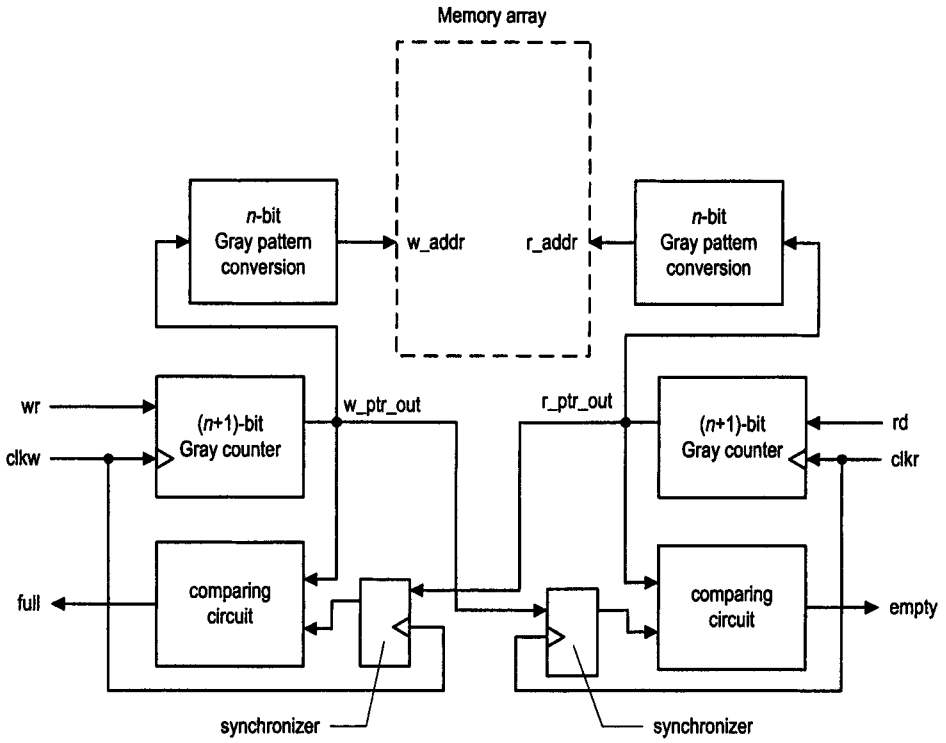


Figure 16.26 Block diagram of an asynchronous FIFO controller.

The complete asynchronous FIFO control circuit follows the basic block diagram, and its VHDL code is shown in Listing 16.16.

Listing 16.13 Write port control of an asynchronous FIFO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_write_ctrl is
   generic(N: natural);
   port(
      clkw, resetw: in std_logic;
      wr: in std_logic;
      r_ptr_in: in std_logic_vector(N downto 0);
      full: out std_logic;
      w_ptr_out: out std_logic_vector(N downto 0);
      w_addr: out std_logic_vector(N-1 downto 0)
   );
end fifo_write_ctrl;

architecture gray_arch of fifo_write_ctrl is
   signal w_ptr_reg, w_ptr_next:
      std_logic_vector(N downto 0);
   signal gray1, bin, bin1: std_logic_vector(N downto 0);
```

```
    signal waddr_all: std_logic_vector(N-1 downto 0);
    signal waddr_msb, raddr_msb: std_logic;
    signal full_flag: std_logic;
begin
   -- register
   process(clkw,resetw)
   begin
      if (resetw='1') then
          w_ptr_reg <= (others=>'0');
      elsif (clkw'event and clkw='1') then
         w_ptr_reg <= w_ptr_next;
      end if;
   end process;
   -- (N+1)-bit Gray counter
   bin <= w_ptr_reg xor ('0' & bin(N downto 1));
   bin1 <= std_logic_vector(unsigned(bin) + 1);
   gray1 <= bin1 xor ('0' & bin1(N downto 1));
   -- update write pointer
   w_ptr_next <= gray1 when wr='1' and full_flag='0' else
                 w_ptr_reg;
   -- N-bit Gray counter
   waddr_msb <=  w_ptr_reg(N) xor w_ptr_reg(N-1);
   waddr_all <= waddr_msb & w_ptr_reg(N-2 downto 0);
   -- check for FIFO full
   raddr_msb <= r_ptr_in(N) xor r_ptr_in(N-1);
   full_flag <=
     '1' when r_ptr_in(N) /=w_ptr_reg(N) and
         r_ptr_in(N-2 downto 0)=w_ptr_reg(N-2 downto 0) and
         raddr_msb = waddr_msb else
     '0';
   -- output
   w_addr <= waddr_all;
   w_ptr_out <= w_ptr_reg;
   full <= full_flag;
end gray_arch;
```

Listing 16.14 Read port control of an asynchronous FIFO

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_read_ctrl is
   generic(N: natural);
   port(
      clkr, resetr: in std_logic;
      w_ptr_in: in std_logic_vector(N downto 0);
      rd: in std_logic;
      empty: out std_logic;
      r_ptr_out: out std_logic_vector(N downto 0);
      r_addr: out std_logic_vector(N-1 downto 0)
   );
end fifo_read_ctrl;
```

```
architecture gray_arch of fifo_read_ctrl is
   signal r_ptr_reg, r_ptr_next: std_logic_vector(N downto 0);
   signal gray1, bin, bin1: std_logic_vector(N downto 0);
   signal raddr_all: std_logic_vector(N-1 downto 0);
   signal raddr_msb,waddr_msb: std_logic;
   signal empty_flag: std_logic;
begin
   -- register
   process(clkr,resetr)
   begin
      if (resetr='1') then
         r_ptr_reg <= (others=>'0');
      elsif (clkr'event and clkr='1') then
         r_ptr_reg <= r_ptr_next;
      end if;
   end process;
   -- (N+1)-bit Gray counter
   bin <= r_ptr_reg xor ('0' & bin(N downto 1));
   bin1 <= std_logic_vector(unsigned(bin) + 1);
   gray1 <= bin1 xor ('0' & bin1(N downto 1));
   -- update read pointer
   r_ptr_next <= gray1 when rd='1' and empty_flag='0' else
                 r_ptr_reg;
   -- N-bit Gray counter
   raddr_msb <= r_ptr_reg(N) xor r_ptr_reg(N-1);
   raddr_all <= raddr_msb & r_ptr_reg(N-2 downto 0);
   waddr_msb <= w_ptr_in(N) xor w_ptr_in(N-1);
   -- check for FIFO empty
   empty_flag <=
      '1' when w_ptr_in(N)=r_ptr_reg(N) and
          w_ptr_in(N-2 downto 0)=r_ptr_reg(N-2 downto 0) and
          raddr_msb = waddr_msb else
      '0';
    -- output
    r_addr <= raddr_all;
    r_ptr_out <= r_ptr_reg;
    empty <= empty_flag;
end gray_arch;
```

Listing 16.15 *n*-bit synchronizer

```
library ieee;
use ieee.std_logic_1164.all;
entity synchronizer_g is
   generic(N: natural);
   port(
      clk, reset: in std_logic;
      in_async: in std_logic_vector(N-1 downto 0);
      out_sync: out std_logic_vector(N-1 downto 0)
   );
end synchronizer_g;

architecture two_ff_arch of synchronizer_g is
```

```
    signal meta_reg, sync_reg: std_logic_vector(N-1 downto 0);
    signal meta_next, sync_next:
       std_logic_vector(N-1 downto 0);
begin
   -- two registers
   process(clk,reset)
   begin
      if (reset='1') then
         meta_reg <= (others=>'0');
         sync_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         meta_reg <= meta_next;
         sync_reg <= sync_next;
      end if;
   end process;
   -- next-state logic
   meta_next <= in_async;
   sync_next <= meta_reg;
   -- output
   out_sync <= sync_reg;
end two_ff_arch;
```

Listing 16.16 Top-level structural description of an asynchronous FIFO control circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_async_ctrl is
   generic(DEPTH: natural);
   port(
      clkw: in std_logic;
      resetw: in std_logic;
      wr: in std_logic;
      full: out std_logic;
      w_addr: out std_logic_vector (DEPTH-1 downto 0);
      clkr: in std_logic;
      resetr: in std_logic;
      rd: in std_logic;
      empty: out std_logic;
      r_addr: out std_logic_vector (DEPTH-1 downto 0)
   );
end fifo_async_ctrl ;

architecture str_arch of fifo_async_ctrl is
   signal r_ptr_in: std_logic_vector(DEPTH downto 0);
   signal r_ptr_out: std_logic_vector(DEPTH downto 0);
   signal w_ptr_in: std_logic_vector(DEPTH downto 0);
   signal w_ptr_out: std_logic_vector(DEPTH downto 0);
   -- component declarations
   component fifo_read_ctrl
      generic(N: natural);
      port(
         clkr: in std_logic;
```

```
30          rd: in std_logic;
            resetr: in std_logic;
            w_ptr_in: in std_logic_vector (N downto 0);
            empty: out std_logic;
            r_addr: out std_logic_vector (N-1 downto 0);
35        r_ptr_out: out std_logic_vector (N downto 0)
        );
    end component;
    component fifo_write_ctrl
        generic(N: natural);
40      port(
            clkw: in std_logic;
            r_ptr_in: in std_logic_vector (N downto 0);
            resetw: in std_logic;
            wr: in std_logic;
45          full: out std_logic;
            w_addr: out std_logic_vector (N-1 downto 0);
            w_ptr_out: out std_logic_vector (N downto 0)
        );
    end component;
50  component synchronizer_g
        generic(N: natural);
        port(
            clk: in std_logic;
            in_async: in std_logic_vector (N-1 downto 0);
55          reset: in std_logic;
            out_sync: out std_logic_vector (N-1 downto 0)
        );
    end component;
  begin
60  read_ctrl: fifo_read_ctrl
        generic map(N=>DEPTH)
        port map (clkr=>clkr, resetr=>resetr, rd=>rd,
                  w_ptr_in=>w_ptr_in, empty=>empty,
                  r_ptr_out=>r_ptr_out, r_addr=>r_addr);
65  write_ctrl: fifo_write_ctrl
        generic map(N =>DEPTH)
        port map(clkw=>clkw, resetw=>resetw, wr=>wr,
                 r_ptr_in=>r_ptr_in, full=>full,
                 w_ptr_out=>w_ptr_out, w_addr=>w_addr);
70  sync_w_ptr: synchronizer_g
        generic map(N=>DEPTH+1)
        port map(clk=>clkw, reset=>resetw,
                 in_async=>w_ptr_out, out_sync=>w_ptr_in);
    sync_r_ptr: synchronizer_g
75      generic map(N=>DEPTH+1)
        port map(clk=>clkr, reset=>resetr,
                 in_async=>r_ptr_out, out_sync =>r_ptr_in);
  end str_arch;
```

Because of the synchronizer, the onset of the `empty` and `full` signals may be delayed. For example, assume that the FIFO is originally empty. After a write operation is performed, the write pointer changes and the FIFO contains one data item. It takes two clock cycles

to propagate the change through the two cascading FFs of the synchronizer, and thus the onset of the `empty` signal is delayed by two clock cycles. During this interval, the reading subsystem will falsely assume that there is no data to retrieve and will stay idle. While the delay causes late data retrieval, the functionality of the FIFO remains intact and no invalid data item is retrieved though the buffer. The same situation happens to the `full` signal. The deassertion of the `full` signal is delayed by two clock cycles. During this interval, the sending subsystem falsely assumes that the FIFO is full and suspends the write operation.

The delayed `empty` and `full` signals force the subsystems to be idle unnecessarily and thus penalize the performance. The penalty is essentially due to the overhead associated with the synchronization of two clock domains and cannot be avoided. However, the idle situation occurs only when the FIFO is almost empty or almost full. There is no overhead or extra delay when the FIFO is partially full. In comparison, the handshaking scheme involves the overhead in every data transaction, and thus the FIFO buffer is more efficient.

16.9.2 Shared memory

Another frequently used buffering scheme is shared memory. The basic idea is to allow multiple subsystems to access a common memory. The sending subsystem can first write the data into the memory, and the receiving subsystem then obtains the data by reading the same memory location. This scheme is best suited when a large chunk of data, such as a high-resolution image, has to be transferred.

The shared memory scheme can be implemented by using a regular single-port memory or a special dual-port memory. For a single-port memory configuration, we can treat the memory as the shared resource and use an arbiter to resolve the conflicting requests and coordinate the memory usage. The basic design of the arbiter is similar to that in Section 10.8.2. Because the interactions are between different clock domains, synchronization circuits are needed for all request and grant signals.

The request-grant process is somewhat like the handshaking procedure and has a similar overhead. However, the overhead is associated with each resource arbitration, not each data transaction. Since one round of arbitration allows any amount of data to be transferred (up to the size of the shared memory), the average overhead of a single data transaction becomes very small.

A better alternative is to use dual-port memory. A dual-port memory has two independent access ports, each containing its own address line, data line and control signals. The multiplexing and decoding circuits are duplicated inside the memory module. Two memory accesses can be performed simultaneously as long as the memory addresses are different. A conflict occurs when two memory operations access the same memory location (i.e., two operations have the same memory addresses). An arbiter is used to resolve the condition. When there is no clock in the regular dual-port memory, the internal arbitrator is an asynchronous sequential circuit. It may enter a metastable state if the two request signals are asserted too closely. As in the synchronizer, the arbitration circuit needs to provide some time to resolve the metastable condition and thus introduces similar overhead.

As with other memory modules, dual-port memory cannot be synthesized from scratch in RT-level code. We must instantiate the predesigned module from the target device technology.

16.10 SYNTHESIS OF A MULTIPLE-CLOCK SYSTEM

The synchronous design is the most important methodology and the cornerstone of the entire design and fabrication process. The synthesis, timing analysis, verification and testing of synchronous systems are well understood, and many EDA software tools have been developed to automate the tasks.

One major motivation behind the synchronous methodology is to provide a systematic way to satisfy the timing constraints. The objective of synthesis and verification is to *identify* and *prevent* timing violations. The EDA software tools are developed to assist designers in achieving this objective. On the other hand, the metastability analysis and synchronization circuit are to deal with the scenario that a timing violation *has already occurred.* This is essentially a transistor-level phenomenon, and its behavior cannot easily be modeled at the gate or RT level. The EDA tools are not able to handle metastability, and the analysis and synthesis process cannot be automated. Most software can only detect and warn the onset of a time violation but cannot model or analyze what happens afterward. For example, in the `std_logic` data type, the `'X'` value is used to model the output after a timing violation. After a timing violation occurs, the output `'X'` value will be permanent and propagated to the downstream circuit. This is very different from the actual timing violation, in which the output signal may become '0' or '1', or be resolved to a stable value after a random period of time.

While the design considerations for a multiple-clock system are different from those of a synchronous system, it is not wise to abandon the synchronous design methodology and start from scratch. Instead, we want to incorporate the methodology into the new design flow and utilize the previous techniques and EDA tools as much as possible.

To achieve this goal, a multiple-clock system should be divided into synchronous subsystems and crossing-domain interfaces. A synchronous subsystem is within the same clock domain, and thus we can design it just as a regular synchronous system. On the other hand, the crossing domain interface involves the synchronization and data transfer protocol. Its analysis and design are very different from those of the synchronous system, and very few EDA tools are available for these tasks. We usually have to manually analyze, design and verify the interface circuit and protocols. Since the synchronization circuit depends on the device characteristics and the data transfer protocol sometimes depends on the clock rates of the domains, the interface is usually device dependent and is not portable.

The general design approach for a multiple-clock system can be summarized as follows:

1. Partition the system into locally synchronous subsystems.
2. Design and verify these subsystems following the synchronous methodology.
3. Develop protocol to pass data and exchange information between clock domains.
4. Manually analyze and design the necessary synchronization circuits between clock domains.
5. Verify operation of the entire system.

A representative top-level partition of a system with two clock domains is shown in Figure 16.27. It is a good idea to treat the synchronization circuit and data transfer interface as separate modules and instantiate them individually in VHDL code. These modules are normally device dependent and may need to be reanalyzed and redesigned when the system is ported to a different device technology or operation environment (e.g., a different clock rate).

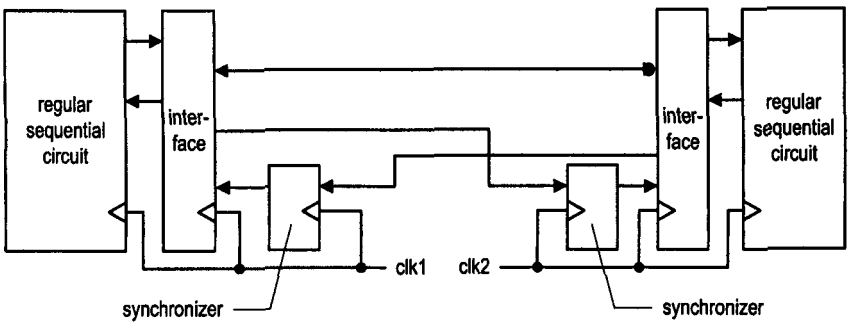


Figure 16.27 Partition of a system with two clock domains.

16.11 SYNTHESIS GUIDELINES

16.11.1 Guidelines for general use of a clock

- Do not manipulate the clock signal in regular RT-level design and synthesis.
- Minimize the number of clock signals in a system.
- Minimize the number of clock domains (i.e., the number of independent clock signals). Use a derived clock signal when possible.
- If a derived clock signal is needed, manually derive and instantiate the circuit and separate it from the regular synthesis.

16.11.2 Guidelines for a synchronizer

- Synchronize a signal in a single place.
- Avoid synchronizing related signals.
- Use a glitch-free signal for synchronization.
- Reanalyze and examine the synchronizer and MTBF when the device is changed or the clock rate is revised.

16.11.3 Guidelines for an interface between clock domains

- Clearly identify the boundary of the clock domain and the signals that cross the domain.
- Separate the synchronization circuits and asynchronous interface from the synchronous subsystems and instantiate them as individual modules.
- Use a reliable synchronizer design to provide sufficient metastability resolution time.
- Analyze the data transfer protocol over a wide range of scenarios, including faster and slower clock frequencies and different data rates.

16.12 BIBLIOGRAPHIC NOTES

The construction of the clock network involves the transmission and distribution of electronic signals. It is normally covered under the subjects of signal integration and high-speed design. Two texts by Howard Johnson, *High-Speed Digital Design: A Handbook of Black Magic* and *High-Speed Signal Propagation: Advanced Black Magic*, provide comprehensive coverage of this topic.

The study of metastability and synchronization failure relies on transistor-level model and analysis. The text, *Digital Systems Engineering* by William J. Dally and John W. Poulton, covers the theoretical foundation of this subject. The book also includes a discussion of the design of asynchronous circuit.

The more practical design materials on the synchronizer and asynchronous interface can be found in manufacturers' application notes or articles from technical conferences. Articles by Clifford E. Cummings, *Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs*, *Simulation and Synthesis Techniques for Asynchronous FIFO Design* and *Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons*, provide many practical design examples and good advice.

Problems

16.1 Assume that a sequential system with an ideal clock signal can operate at a maximal clock rate of 100 MHz. If the physical clock distribution network introduces a 1.5-ns clock skew, what is the new maximal clock rate?

16.2 Consider a D FF with w and τ.

(a) If we improve the D FF by reducing w by 10%, discuss the effect on MTBF.

(b) If we improve the D FF by reducing τ by 10%, discuss the effect on MTBF.

16.3 At the end of Section 16.4.2, we discuss the difference between T_r and T_{r2}. Assume that w and τ are identical for the calculation of MTBF(T_r) and MTBF(T_{r2}). Derive MTBF(T_r) in terms of MTBF(T_{r2}), w and τ.

16.4 For the two-FF synchronizer discussed in Section 16.5.3, determine the new MTBF for the following:

(a) The placement and routing process adds a 2.5-ns wiring delay.

(b) The system clock rate is decreased by 10%.

(c) The setup time of the D FF is reduced by 10%.

(d) The setup time of the D FF is reduced by 25%.

(e) The τ of the D FF is reduced by 10%.

(f) The τ of the D FF is reduced by 25%.

16.5 We want to regenerate the enable pulse in the listener's clock domain using the four-phase handshaking protocol. In this scheme, the listener has an output signal that is asserted once during the handshaking process.

(a) Revise the listener ASM chart of Figure 16.16 to add a Mealy output signal.

(b) Modify the VHDL code to reflect the revised ASM chart.

16.6 Repeat Problem 16.5, but add a Moore output signal.

16.7 Repeat Problem 16.5 for the two-phase handshaking protocol of Figure 16.19.

16.8 Repeat Problem 16.5, but add a Moore output signal for the two-phase handshaking protocol of Figure 16.19.

16.9 Revise the talker ASM chart of the two-phase handshaking protocol of Figure 16.19 to eliminate the `idle` state.

16.10 In a handshaking protocol, we like to include a `ready` signal in talker to indicate that the system is idle and ready to accept another operation. Revise the talker ASM chart of the two-phase handshaking protocol of Figure 16.19 to include the `ready` output signal.

16.11 We want to design a four-phase handshaking asynchronous interface for the sequential multiplier in Section 11.6. The operand width is 8 bits and the data is passed by a 16-bit bidirectional bus. After sensing the `start` signal, the talker of the sending subsystem places the data on the data bus and activates the handshaking operation. Once the receiving subsystem detects the request, it retrieves the data and performs the multiplication operation. When the operation is completed, the listener of the receiving subsystem places the result on the data bus and asserts the acknowledge signal, and the talker retrieves the result accordingly. Draw the block diagram and derive VHDL code for this system.

16.12 Repeat Problem 16.11, but use an 8-bit data bus. Since the operation involves two data transfers, we need a master control FSM to coordinate the operation. Derive VHDL code for this system.

16.13 Modify the push-and-pull system of Section 16.8.1 using the two-phase handshaking protocol (additional data lines are needed). Derive the revised block diagram and VHDL code.

16.14 Repeat Problem 16.11, but use the two-phase handshaking protocol and two 16-bit unidirectional data buses. Derive the VHDL code.

16.15 Consider the one-phase push operation in Section 16.8.3. Derive VHDL code for the sending and receiving subsystems with the following clock rates.

(a) f_{send} = 10 MHz and $f_{receive}$ = 10 MHz
(b) f_{send} = 10 MHz and $f_{receive}$ = 40 MHz
(c) f_{send} = 10 MHz and $f_{receive}$ = 2.5 MHz

16.16 Repeat Problem 16.15 for the one-phase pull operation.

16.17 Consider the FIFO buffer of Section 16.9.1, and let the clock periods of the writing and reading subsystems be T_w and T_r respectively. Assume that the sending subsystem has 10 words to pass through the FIFO buffer. Determine the total time to complete the operation with the following buffer sizes:

(a) One word.
(b) Two words.
(c) Four words.
(d) Eight words.
(e) 16 words.

REFERENCES

1. P. Alfke, "Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators," Xilinx Application Note XAPP-052, 1996.
2. M. G. Arnold, *Verilog Digital Computer Design*, Prentice Hall, 1998.
3. P. J. Ashenden, *The Designer's Guide to VHDL, 2nd ed.*, Morgan Kaufmann, 2001.
4. P. H. Bardell, *Built In Test for VLSI: Pseudorandom Techniques*, Wiley-Interscience, 1987.
5. L. Bening and H. D. Foster, *Principles of Verifiable RTL Design, 2nd ed.*, Springer-Verlag, 2001.
6. J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, Springer-Verlag, 2003.
7. M. D. Ciletti, *Advanced Digital Design with the Verilog HDL*, Prentice Hall, 2003.
8. M. D. Ciletti, *Starter's Guide to Verilog 2001*, Prentice Hall, 2003.
9. C. E. Cummings, "Coding and Scripting Techniques for FSM Designs with Synthesis-Optimized, Glitch-Free Outputs," *SNUG (Synopsys Users Group conference)*, Boston, 2000.
10. C. E. Cummings, "Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs," *SNUG (Synopsys Users Group conference)*, San Jose, 2001.
11. C. E. Cummings, "Simulation and Synthesis Techniques for Asynchronous FIFO Design," *SNUG (Synopsys Users Group conference)*, San Jose, 2002.
12. C. E. Cummings and P. Alfke, "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons," *SNUG (Synopsys Users Group conference)*, San Jose, 2002.
13. W. J. Dally and J. W. Poulton, *Digital Systems Engineering*, Cambridge University Press, 1998.
14. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
15. S. Devadas et al., *Logic Synthesis*, McGraw-Hill Professional, 1994.

16. M. D. Ercegovac and T. Lang, *Digital Arithmetic*, Morgan Kaufmann, 2003.

17. D. D. Gajski, *Principles of Digital Design*, Prentice Hall, 1997.

18. D. D. Gajski, *High-Level Synthesis: Introduction to Chip and System Design*, Springer-Verlag, 1992.

19. S. Ghosh, *Hardware Description Languages: Concepts and Principles*, Wiley-IEEE Press, 1999.

20. IEEE, *IEEE Standard for Verilog Hardware Description Language, (IEEE Std 1364-2001)*, Institute of Electrical and Electronics Engineers, 2001.

21. IEEE, *IEEE Standard VHDL Language Reference Manual (IEEE Std 1076-2001)*, Institute of Electrical and Electronics Engineers, 2001.

22. IEEE, *IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, (IEEE Std 1076.6-1999)*, Institute of Electrical and Electronics Engineers, 2000.

23. IEEE, *IEEE Standard VHDL Synthesis Packages (IEEE Std 1076.3-1997)*, Institute of Electrical and Electronics Engineers, 1997.

24. IEEE, *IEEE Standard Multivalue Logic System for VHDL Model Interoperability (IEEE Std 1164-1993)*, Institute of Electrical and Electronics Engineers, 1993.

25. H. Johnson, *High-Speed Digital Design: A Handbook of Black Magic*, Prentice Hall, 1993.

26. T. Kam, *Synthesis of Finite State Machines: Functional Optimization*, Kluwer Academic, 1997.

27. R. H. Katz and G. Borriello, *Contemporary Logic Design, 2nd ed.*, Prentice Hall, 2004.

28. M. Keating and P. Bricaud, *Methodology Manual for System-on-a-Chip Designs, 3rd ed.*, Springer-Verlag, 2002.

29. I. Koren, *Computer Arithmetic Algorithms, 2nd ed.*, A. K. Peters, 2002.

30. H. A. Landman, "Visualizing the Behavior of Logic Synthesis Algorithms," *SNUG (Synopsys Users Group conference)*, 1998.

31. C. M. Maxfield, *The Design Warrior's Guide to FPGAs*, Newnes, 2004.

32. S. Palnitkar, *Verilog HDL, 2nd ed.*, Prentice Hall, 2003.

33. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface, 3rd ed.*, Morgan Kaufmann, 2004.

34. Jan M. Rabaey, *Digital Integrated Circuits, 2nd ed.*, Prentice Hall, 2002.

35. A. Rushton, *VHDL for Logic Synthesis, 2nd ed.*, John Wiley & Sons, 1998.

36. P. Sinander, *VHDL Modelling Guidelines*, European Space Agency, 1994.

37. T. Villa et al., *Synthesis of Finite State Machines: Logic Optimization*, Kluwer Academic, 1997.

38. J. F. Wakerly, *Digital Design: Principles and Practices*, Prentice Hall, 2002.

39. W. Wolf, *FPGA-Based System Design*, Prentice Hall, 2004.

40. W. Wolf, *Modern VLSI Design: System-on-Chip Design, 3rd ed.*, Prentice Hall, 2002.

41. R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and Their Synthesis*, PhD. thesis, Swiss Federal Institute of Technology (ETH), Zurich, 1998.

42. R. Zimmermann, "VHDL Library of Arithmetic Units," *First International Forum on Design Languages (FDL'98)*, 1998.

INDEX