CHAPTER 9

SEQUENTIAL CIRCUIT DESIGN: PRACTICE

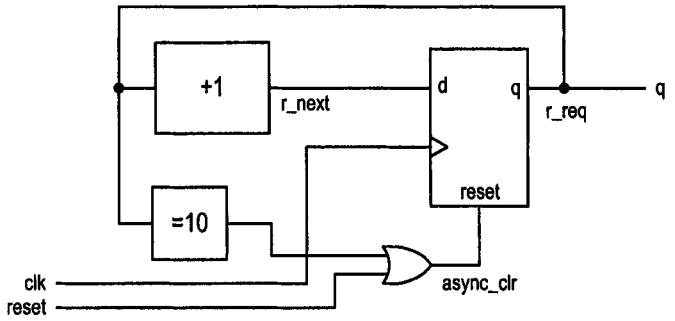
After learning the basic model and coding style, we explore more sophisticated regular sequential circuits in this chapter. The design examples show the implementation of a variety of counters, the use of registers as fast, temporary storage, and the construction of a "pipeline" to increase the throughput of certain combinational circuits.

9.1 POOR DESIGN PRACTICES AND THEIR REMEDIES

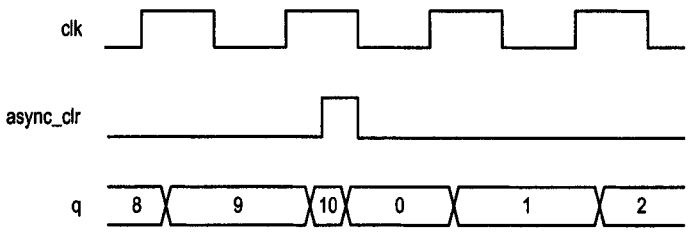
Synchronous design is the most important design methodology for developing a large, complex, reliable digital system. In the past, some poor, non-synchronous design practices were used. Those techniques failed to follow the synchronous principle and should be avoided in RT-level design. Before continuing with more examples, we examine those practices and their remedies. The most common problems are:

- Misuse of the asynchronous reset.
- Misuse of the gated clock.
- Misuse of the derived clock.

Some of those practices were used when a system was realized by SSI and MSI devices and the silicon real estate and printed circuit board were a premium. Designers tended to cut corners to save a few chips. These legacy practices are no longer applicable in today's design environment and should be avoided. The following subsections show how to remedy these poor non-synchronous design practices.



(a) Block diagram



(b) Timing diagram

Figure 9.1 Decade counter using asynchronous reset.

In few special situations, such as the interface to an external system and low power design, the use of multiple clocks and asynchrony may be unavoidable. This kind of design cannot easily be incorporated into the regular synthesis and testing flow. It should be treated differently and separated from the regular sequential system development. We discuss the asynchronous aspect in Chapter 16.

9.1.1 Misuse of asynchronous signals

In a synchronous design, we utilize only asynchronous reset or preset signals of FFs for system initialization. These signals should not be used in regular operation. A decade (mod-10) counter based on asynchronous reset is shown in Figure 9.1(a). The idea behind the design is to clear the counter to "0000" immediately after the counter reaches "1010". The timing diagram is shown in Figure 9.1(b). If we want, we can write VHDL code for this design, as in Listing 9.1.

Listing 9.1 Decade counter using an asynchronous reset signal

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod10_counter is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end mod10_counter;
```

```
10
  architecture poor_async_arch of mod10_counter is
     signal r_reg: unsigned(3 downto 0);
     signal r_next: unsigned(3 downto 0);
     signal async_clr: std_logic;
15 begin
     -- register
     process(clk,async_clr)
     begin
        if (async_clr='1') then
20         r_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
           r_reg <= r_next;
        end if;
     end process;
25   -- asynchronous clear
     async_clr <= '1' when (reset='1' or r_reg="1010") else
                  '0';
     -- next state logic
     r_next <= r_reg + 1;
30   -- output logic
     q <= std_logic_vector(r_reg);
  end poor_async_arch;
```

There are several problems with this design. First, the transition from state "1001" (9) to "0000" (0) is noisy, as shown in Figure 9.1(b). In that clock period, the counter first changes from "1001" (9) to "1010" (10) and then clears to "0000" (0) after the propagation delay of the comparator and reset. Second, this design is not very reliable. A combinational circuit is needed to generate the clear signal, and glitches may exist. Since the signal is connected to the asynchronous reset of the register, the register will be cleared to "0000" whenever a glitch occurs. Finally, because the asynchronous reset is used in normal operation, we cannot apply the timing analysis technique of Section 8.6. It is very difficult to determine the maximal operation clock rate for this design.

The remedy for this design is to load "0000" in a synchronous fashion. We can use a multiplexer to route "0000" or the incremented result to the input of the register. The code was discussed in Section 8.5.5 and is listed in Listing 9.2 for comparison. In terms of the circuit complexity, the synchronous design requires an additional 4-bit 2-to-1 multiplexer.

Listing 9.2 Decade counter using a synchronous clear signal

```
architecture two_seg_arch of mod10_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
begin
5  -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
10    elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
```

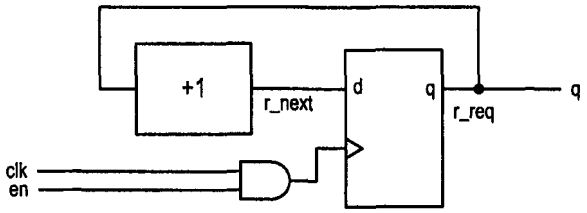


Figure 9.2 Disabling FF with a gated clock.

```
   -- next-state logic
   r_next <= (others=>'0') when r_reg=9 else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

9.1.2 Misuse of gated clocks

Correct operation of a synchronous circuit relies on an accurate clock signal. Since the clock signal needs to drive hundreds or even thousands of FFs, it uses a special distribution network and its treatment is very different from that of a regular signal. We should not manipulate the clock signal in RT-level design.

One bad RT-level design practice is to use a gated clock to suspend system operation, as shown in Figure 9.2. The intention of the design is to pause the counter operation by disabling the clock signal. The design suffers from several problems. First, since the enable signal, en, changes independent of the clock signal, the output pulse can be very narrow and cause the counter to malfunction. Second, if the en signal is not glitch-free, the glitches will be passed through the and cell and be treated as clock edges by the counter. Finally, since the and cell is included in the clock path, it may interfere with the construction and analysis of the clock distribution network.

The remedy for this design is to use a synchronous enable signal for the register, as discussed in Section 8.5.1. We essentially route the register output as a possible input. If the en signal is low, the same value is sampled and stored back to the register and the counter appears to be "paused." The VHDL codes for the original and revised designs are shown in Listings 9.3 and 9.4. In terms of the circuit complexity, the synchronous design requires an additional 2-to-1 multiplexer.

Listing 9.3 Binary counter with a gated clock

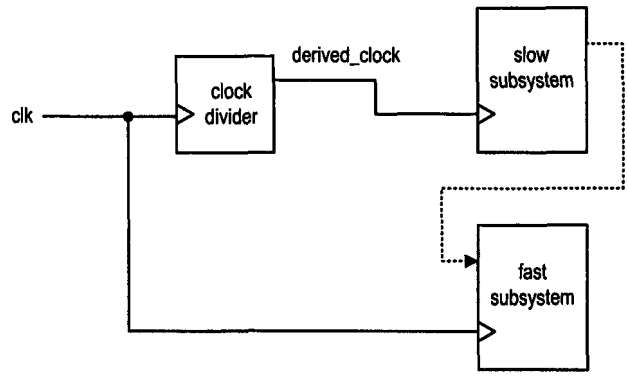
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity binary_counter is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end binary_counter;
```

```
architecture gated_clk_arch of binary_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal gated_clk: std_logic;
begin
   -- register
   process(gated_clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (gated_clk'event and gated_clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- gated clock
   gated_clk <= clk and en;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
end gated_clk_arch;
```

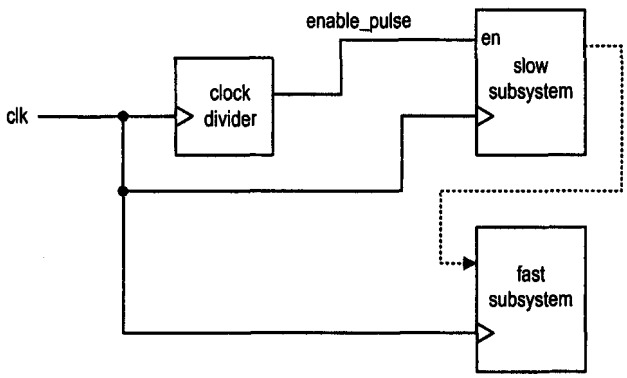
Listing 9.4 Binary counter with a synchronous enable signal

```
architecture two_seg_arch of binary_counter is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 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 en='1' else
             r_reg;
   -- output logic
   q <= std_logic_vector(r_reg);
end two_seg_arch;
```

Power consumption is one important design criterion in today's digital system. A commonly used technique is to gate the clock to reduce the unnecessary transistor switching activities. However, this practice should not be done in RT-level code. The system should be developed and coded as a normal sequential circuit. After synthesis and verification, we can apply special power optimization software to replace the enable logic with a gated clock systematically.



(a) System with a derived clock



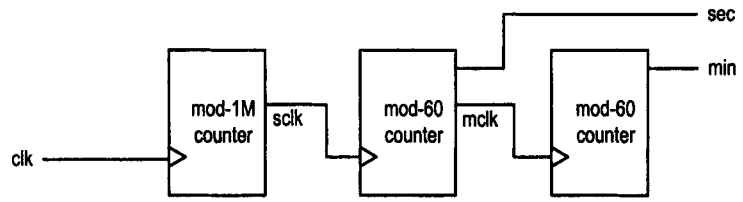
(b) System with a single synchronous clock

Figure 9.3 System composed of fast and slow subsystems.

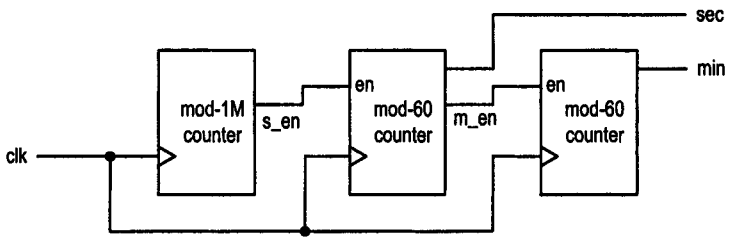
9.1.3 Misuse of derived clocks

A large digital system may consist of subsystems that operate in different paces. For example, a system may contain a fast processor and a relatively slow I/O subsystem. One way to accommodate the slow operation is to use a clock divider (i.e., a counter) to derive a slow clock for the subsystem. The block diagram of this approach is shown in Figure 9.3(a). There are several problems with this approach. The most serious one is that the system is no longer synchronous. If the two subsystems interact, as shown by the dotted line in Figure 9.3(a), the timing analysis becomes very involved. The simple timing model of Section 8.6 can no longer be applied and we must consider two clocks that have different frequencies and phases. Another problem is the placement and routing of the multiple clock signals. Since a clock signal needs a special driver and distribution network, adding derivative clock signals makes this process more difficult. A better alternative is to add a synchronous enable signal to the slow subsystem and drive the subsystem with the same clock signal. Instead of generating a derivative clock signal, the clock divider generates a low-rate single-clock enable pulse. This scheme is shown in Figure 9.3(b).

Let us consider a simple example. Assume that the system clock is 1 MHz and we want a timer that counts in minutes and seconds. The first design is shown in Figure 9.4(a). It first utilizes a mod-1000000 counter to generate a 1-Hz squared wave, which is used as a



(a) Design with a derived clock



(b) Design with a single synchronous clock

Figure 9.4 Second and minute counter.

1-Hz clock to drive the second counter. The second counter is a mod-60 counter, which in turn generates a $\frac{1}{60}$-Hz signal to drive the clock of the minute counter. The VHDL code is shown in Listing 9.5. It consists of a mod-1000000 counter and two mod-60 counters. The output logic of the mod-1000000 counter and one mod-60 counter utilizes comparators to generate 50% duty-cycle pulses, which are used as the clocks in successive stages.

Listing 9.5 Second and minute counter with derived clocks

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity timer is
   port(
      clk, reset: in std_logic;
      sec,min: out std_logic_vector(5 downto 0)
   );
end timer;

architecture multi_clock_arch of timer is
   signal r_reg: unsigned(19 downto 0);
   signal r_next: unsigned(19 downto 0);
   signal s_reg, m_reg: unsigned(5 downto 0);
   signal s_next, m_next: unsigned(5 downto 0);
   signal sclk, mclk: std_logic;
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 <= (others=>'0') when r_reg=999999 else
             r_reg + 1;
   -- output logic
   sclk <= '0' when r_reg < 500000 else
           '1';
   -- second divider
   process(sclk,reset)
   begin
      if (reset='1') then
         s_reg <= (others=>'0');
      elsif (sclk'event and sclk='1') then
         s_reg <= s_next;
      end if;
   end process;
   -- next-state logic
   s_next <= (others=>'0') when s_reg=59 else
             s_reg + 1;
   -- output logic
   mclk <= '0' when s_reg < 30 else
           '1';
   sec <= std_logic_vector(s_reg);
   -- minute divider
   process(mclk,reset)
   begin
      if (reset='1') then
         m_reg <= (others=>'0');
      elsif (mclk'event and mclk='1') then
         m_reg <= m_next;
      end if;
   end process;
   -- next-state logic
   m_next <= (others=>'0') when m_reg=59 else
             m_reg + 1;
   -- output logic
   min <= std_logic_vector(m_reg);
end multi_clock_arch;
```

To convert the design to a synchronous circuit, we need to make two revisions. First, we add a synchronous enable signal for the mod-60 counter. The enable signal functions as the en signal discussed in examples in Section 8.5.1. When it is deasserted, the counter will pause and remain in the same state. Second, we have to replace the 50% duty cycle clock pulse with a one-clock-period enable pulse, which can be obtained by decoding a specific value of the counter. The revised diagram is shown in Figure 9.4(b), and the VHDL code is shown in Listing 9.6.

Listing 9.6 Second and minute counter with enable pulses

```
architecture single_clock_arch of timer is
   signal r_reg: unsigned(19 downto 0);
   signal r_next: unsigned(19 downto 0);
```

```
    signal s_reg, m_reg: unsigned(5 downto 0);
    signal s_next, m_next: unsigned(5 downto 0);
    signal s_en, m_en: std_logic;
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
         s_reg <= (others=>'0');
         m_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
         s_reg <= s_next;
         m_reg <= m_next;
      end if;
   end process;
   -- next-state logic/output logic for mod-1000000 counter
   r_next <= (others=>'0') when r_reg=999999 else
             r_reg + 1;
   s_en <= '1' when r_reg = 500000 else
           '0';
   -- next state logic/output logic for second divider
   s_next <= (others=>'0') when (s_reg=59 and s_en='1') else
             s_reg + 1     when s_en='1' else
             s_reg;
   m_en <= '1' when s_reg=30 and s_en='1' else
           '0';
   -- next-state logic for minute divider
   m_next <= (others=>'0') when (m_reg=59 and m_en='1') else
             m_reg + 1     when m_en='1' else
             m_reg;
   -- output logic
   sec <= std_logic_vector(s_reg);
   min <= std_logic_vector(m_reg);
end single_clock_arch;
```

9.2 COUNTERS

A counter can be considered as a circuit that circulates its internal state through a set of patterns. The patterns dictate the complexity of the next-state logic and the performance of the counter. Some applications require patterns with specific characteristics. We studied several counters in Sections 8.5. These counters are variations of the binary counter, which follows the basic binary counting sequence. This section introduces several other types of commonly used counters.

9.2.1 Gray counter

An n-bit Gray counter also circulates through all 2^n states. Its counting sequence follows the Gray code sequence, in which only one bit is changed between successive code words.

The design and VHDL description are similar to those of a binary counter except that we need to replace the binary incrementor with the Gray code incrementor of Section 7.5.1. The VHDL code of a 4-bit Gray counter is shown in Listing 9.7.

Listing 9.7 Gray counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity gray_counter4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end gray_counter4;

architecture arch of gray_counter4 is
   constant WIDTH: natural := 4;
   signal g_reg: unsigned(WIDTH-1 downto 0);
   signal g_next, b, b1: unsigned(WIDTH-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         g_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         g_reg <= g_next;
      end if;
   end process;
   -- next-state logic
   -- Gray to binary
   b <= g_reg xor ('0' & b(WIDTH-1 downto 1));
   -- binary increment
   b1 <= b+1;
   -- binary to Gray
   g_next <= b1 xor ('0' & b1(WIDTH-1 downto 1));
   -- output logic
   q <= std_logic_vector(g_reg);
end arch;
```

9.2.2 Ring counter

A ring counter is constructed by connecting the serial-out port to the serial-in port of a shift register. The basic sketch of a 4-bit shift-to-right ring counter and its timing diagram are shown in Figure 9.5. After the "0001" pattern is loaded, the counter circulates through "1000", "0100", "0010" and "0001" states, and then repeats.

There are two methods of implementing a ring counter. The first method is to load the initial pattern during system initialization. Consider a 4-bit ring counter. We can set the counter to "0001" when the `reset` signal is asserted. After initialization, the `reset` signal is deasserted and the counter enters normal synchronous operation and circulates through the patterns. The VHDL code of a 4-bit ring counter is shown in Listing 9.8.

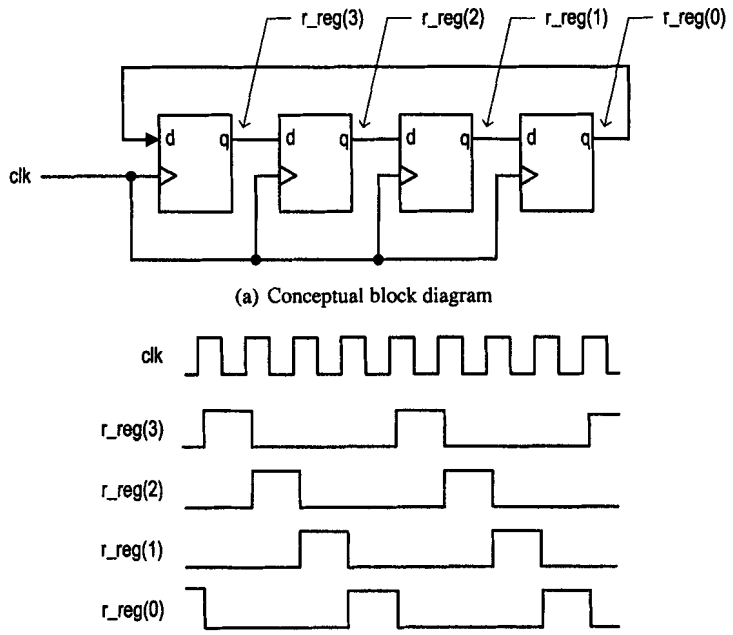


Figure 9.5 Sketch of a 4-bit ring counter.

Listing 9.8 Ring counter using asynchronous initialization

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

architecture reset_arch of ring_counter is
   constant WIDTH: natural := 4;
   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);
```

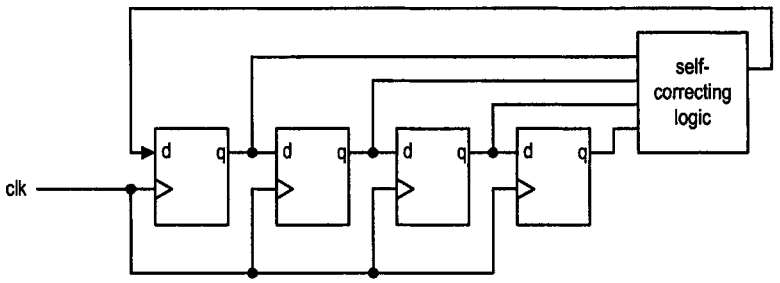


Figure 9.6 Block diagram of a self-correcting ring counter.

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

Note that the q_reg is initialized with "0001" by the statement

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

The alternative method is to utilize a self-correcting logic to feed the serial-in port with the correct pattern. The block diagram of a 4-bit self-correcting ring counter is shown in Figure 9.6. The design is based on the observation that a '1' can only be shifted into the shift register if the current three MSBs of the register are "000". If any of the three MSBs is not '0', the self-correcting logic generates a '0' and shifts it into the register. This process continues until the three MSBs become "000" and a '1' is shifted in afterward. Note that this scheme works even when the register contains an invalid pattern initially. For example, if the initial value of the register is "1101", the logic will gradually shift in 0's and return to the normal circulating sequence. Because of this property, the circuit is known as *self-correcting*.

The VHDL code for this design is shown in Listing 9.9. Note that no special input pattern is needed during system initialization, and the all-zero pattern is used in the code.

Listing 9.9 Ring counter using self-correcting logic

```
architecture self_correct_arch of ring_counter is
   constant WIDTH: natural := 4;
   signal r_reg, r_next: std_logic_vector(WIDTH-1 downto 0);
   signal s_in: std_logic;
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_reg(WIDTH-1 downto 1)="000" else
           '0';
```

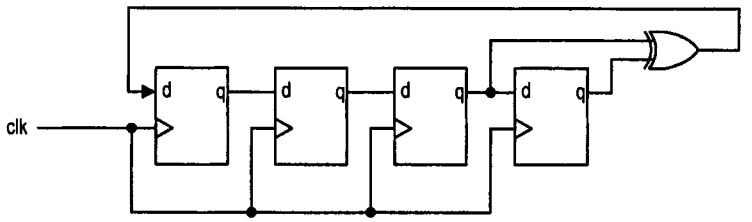


Figure 9.7 Block diagram of a 4-bit LFSR.

```
    r_next <= s_in & r_reg(WIDTH-1 downto 1);
    -- output logic
    q <= r_reg;
end self_correct_arch;
```

In a ring counter, an n-bit register can generate only n states, which is much smaller than the possible 2^n states of a binary counter. Despite its inefficiency, a ring counter offers several benefits. First, each bit of a ring counter is in the 1-out-of-n format. It requires no decoding logic and is glitch-free. Second, the output of a ring counter is out of phase, and the n output bits of an n-bit ring counter form a set of n-phase signals. For example, in the timing diagram of the 4-bit ring counter, each bit is activated for one-fourth of the period and only one bit is activated at a particular phase. Finally, the ring counter is extremely fast. For the `reset_arch` architecture, the next-state logic consists only of connection wires. If we assume that the wiring delay is negligible, the maximal clock rate becomes $\frac{1}{T_{cq}+T_{setup}}$, which is the fastest clock rate that can be achieved by a sequential circuit for a given technology.

9.2.3 LFSR (linear feedback shift register)

The *linear feedback shift register* (*LFSR*) is a shift register that utilizes a special feedback circuit to generate the serial input value. The feedback circuit is essentially the next-state logic. It performs xor operation on certain bits of the register and forces the register to cycle through a set of unique states. In a properly designed n-bit LFSR, we can use a few xor gates to force the register to circulate through $2^n - 1$ states. The diagram of a 4-bit LFSR is shown in Figure 9.7. The two LSB signals of the register are xored to generate a new value, which is fed back to the serial-in port of the shift register. Assume that the initial state of register is "1000". The circuit will circulate through the 15 (i.e., $2^4 - 1$) states as follows: "1000", "0100", "0010", "1001", "1100", "0110", "1011", "0101", "1010", "1101", "1110", "1111", "0111", "0011", "0001".

Note that the "0000" state is not included and constitutes the only missing state. If the LFSR enters this state accidentally, it will be stuck in this state.

The construction of LFSRs is based on the theoretical study of finite fields. The term *linear* comes from the fact that the general feedback equation of an LFSR is described by an expression of the *and* and *xor* operators, which form a linear system in algebra. The theoretical study shows some interesting properties of LFSRs:

- An n-bit LFSR can cycle through up to $2^n - 1$ states. The all-zero state is excluded from the sequence.
- A feedback circuit to generate maximal number of states exists for any n.

Table 9.1 Feedback expression of LFSR

Register size	Feedback expression
2	$q_1 \oplus q_0$
3	$q_1 \oplus q_0$
4	$q_1 \oplus q_0$
5	$q_2 \oplus q_0$
6	$q_1 \oplus q_0$
7	$q_3 \oplus q_0$
8	$q_4 \oplus q_3 \oplus q_2 \oplus q_0$
16	$q_5 \oplus q_4 \oplus q_3 \oplus q_0$
32	$q_{22} \oplus q_2 \oplus q_1 \oplus q_0$
64	$q_4 \oplus q_3 \oplus q_1 \oplus q_0$
128	$q_{29} \oplus q_{17} \oplus q_2 \oplus q_0$

- The sequence generated by the feedback circuit is *pseudorandom*, which means that the sequence exhibits a certain statistical property and appears to be random.

The feedback circuit depends on the number of bits of the LFSR and is determined on an ad hoc basis. Despite its irregular pattern, the feedback expressions are very simple, involving either one or three xor operators most of the time. Table 9.1 lists the feedback expressions for register sizes between 2 and 8 as well as several larger values. We assume that the output of the n-bit shift register is $q_{n-1}, q_{n-2}, \ldots, q_1, q_0$. The result of the feedback expression is to be connected to the serial-in port of the shift register (i.e., the input of the $(n-1)$th FF).

Once we know the feedback expression, the coding of LFSR is straightforward. The VHDL code for a 4-bit LFSR is shown in Listing 9.10. Note that the LFSR cannot be initialized with the all-zero pattern. In pseudo number generation, the initial value of the sequence is known as a *seed*. We use a constant to define the initial value and load it into the LFSR during system initialization.

Listing 9.10 LFSR

```
library ieee;
use ieee.std_logic_1164.all;
entity lfsr4 is
   port(
      clk, reset: in std_logic;
      q: out std_logic_vector(3 downto 0)
   );
end lfsr4;

architecture no_zero_arch of lfsr4 is
   signal r_reg, r_next: std_logic_vector(3 downto 0);
   signal fb: std_logic;
   constant SEED: std_logic_vector(3 downto 0):="0001";
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
```

```
            r_reg <= SEED;
20        elsif (clk'event and clk='1') then
            r_reg <= r_next;
          end if;
      end process;
      -- next-state logic
25    fb <= r_reg(1) xor r_reg(0);
      r_next <= fb & r_reg(3 downto 1);
      -- output logic
      q <= r_reg;
  end no_zero_arch;
```

The unique properties of LFSR make them useful in a variety of applications. The first type of application utilizes its pseudorandomness property to scramble and descramble data, as in testing, encryption and modulation. The second type takes advantages of its simple combinational feedback circuit. For example, we can use just three xor gates in a 32-bit LFSR to cycle through $2^{32} - 1$ states. By comparison, we need a fairly large 32-bit incrementor to cycle through 2^{32} states in a binary counter. By the component information of Table 6.2, the gate counts for three xor gates and a 32-bit incrementor are 9 and 113 respectively, and their propagation delays are 0.8 and 11.6 ns respectively. Thus, an LFSR can replace other counters for applications in which the order of the counting states is not important. This is a clever design technique. For example, we can use three xor gates to implement a 128-bit LFSR, and it takes about 10^{12} years for a 100-GHz system to circulate all the possible $2^{128} - 1$ states.

The all-zero state is excluded in a pure LFSR. It is possible to use an additional circuit to insert the all-zero state into the counting sequence so that an n-bit LFSR can circulate through all 2^n states. This scheme is based on the following observation. In any LFSR, a '1' will be shifted in after the "00$\cdots$01" state since the all-zero state is not possible. In other words, the feedback value will be '1' when the $n-1$ MSBs are 0's and the state following "00$\cdots$01" will always be "10$\cdots$00". The revised design will insert the all-zero state, "00$\cdots$00", between the "00$\cdots$01" and "10$\cdots$00" states. Let the output of an n-bit shift register be $q_{n-1}, q_{n-2}, \ldots, q_1, q_0$ and the original feedback signal of the LFSR be f_b. The modified feedback value f_{zero} has the expression

$$f_{zero} = f_b \oplus (q'_{n-1} \cdot q'_{n-2} \cdots q'_2 \cdot q'_1)$$

This expression can be analyzed as follows:

- The expression $q'_{n-1} \cdot q'_{n-2} \cdots q'_2 \cdot q'_1$ indicates the condition that the $n-1$ MSBs are 0's. This condition can only be true when the LFSR is in "00$\cdots$01" or "00$\cdots$00" state.
- If the condition above is false, the value of f_{zero} is f_b since $f_b \oplus 0 = f_b$. This implies that the circuit will shift in a regular feedback value and follow the original sequence except for the "00$\cdots$01" or "00$\cdots$00" states.
- If the current state of the register is "00$\cdots$01", the value of f_b should be '1' and the expression $f_b \oplus (q'_{n-1} \cdot q'_{n-2} \cdots q'_2 \cdot q'_1)$ becomes $1 \oplus 1$. Thus, a '0' will be shifted into the register at the next rising edge of the clock and the next state will be "00$\cdots$00".
- If the current state of the register is "00$\cdots$00", the value of f_b should be '0' and the expression $f_b \oplus (q'_{n-1} \cdot q'_{n-2} \cdots q'_2 \cdot q'_1)$ becomes $0 \oplus 1$. Thus, a '1' will be shifted into the register at the next rising edge of the clock and the next state will be "10$\cdots$00".
- Once the shift register reaches "10$\cdots$00", it returns to the regular LFSR sequence.

The analysis clearly shows that the modified feedback circuit can insert the all-zero state between the "00···01" and "10···00" states. Technically, the and operation of the revised feedback expression destroys the "linearity," and thus the circuit is no longer a linear feedback shift register. The modified design is sometimes known as a *Bruijn counter.*

Once understanding the form and operation of the modified feedback expression, we can easily incorporate it into the VHDL code. The revised code is shown in Listing 9.11. We use the statement

```
zero <= '1' when r_reg(3 downto 1)="000" else
        '0';
```

to obtain the result of the $q'_{n-1} \cdot q'_{n-2} \cdots q'_2 \cdot q'_1$ expression. Note that the all-zero state can be loaded into the register during system initialization.

Listing 9.11 LFSR with the all-zero state

```
architecture with_zero_arch of lfsr4 is
   signal r_reg, r_next: std_logic_vector(3 downto 0);
   signal fb, zero, fzero: std_logic;
   constant seed: std_logic_vector(3 downto 0):="0000";
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
   fb <= r_reg(1) xor r_reg(0);
   zero <= '1' when r_reg(3 downto 1)="000" else
           '0';
   fzero <= zero xor fb;
   r_next <= fzero & r_reg(3 downto 1);
   -- output logic
   q <= r_reg;
end with_zero_arch;
```

9.2.4 Decimal counter

A decimal counter circulates the patterns in binary-coded decimal (BCD) format. The BCD code uses 4 bits to represent a decimal number. For example, the BCD code for the three-digit decimal number 139 is "0001 0011 1001". The decimal counter follows the decimal counting sequence and the number following 139 is 140, which is represented as "0001 0100 0000".

One possible way to construct a decimal counter is to design a BCD incrementor for the next-state logic, just like a regular incrementor in a binary counter. Because of the cumbersome implementation of the BCD incrementor, this method is not efficient. A better alternative is to divide the counter into stages of decade counters and use special enable logic to control the increment of the individual decade counters.

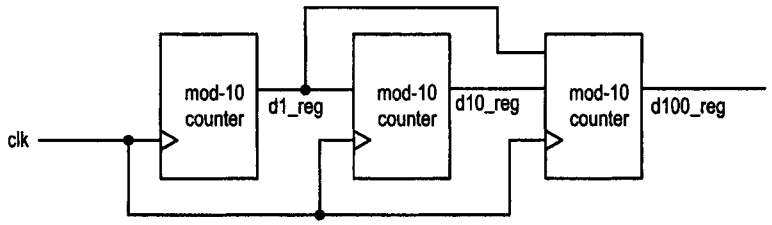


Figure 9.8 Block diagram of a BCD counter.

Consider a 3-digit (12-bit) decimal counter that counts from 000 to 999 and then repeats. It can be implemented by cascading three special decade counters, and the sketch is shown in Figure 9.8. The leftmost decade counter represents the least significant decimal digit. It is a regular mod-10 counter that counts from 0 to 9 (i.e., from "0000" to "1001") and repeats. The middle decade counter is a mod-10 counter with a special enable circuit. It increments only when the least significant decimal digit reaches 9. The rightmost decade counter represents the most significant decimal digit, and it increments only when the two least significant decimal digits are equal to 99. Note that when the counter reaches 999, it will return to 000 at the next rising edge of the clock. The VHDL codes are shown in Listings 9.12 and 9.13. The former uses three conditional signal assignment statements and the latter uses a nested if statement to check whether the counter reaches --9, -99 or 999.

Listing 9.12 Three-digit decimal counter using conditional concurrent statements

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity decimal_counter is
   port(
      clk, reset: in std_logic;
      d1, d10, d100: out std_logic_vector(3 downto 0)
   );
end decimal_counter;

architecture concurrent_arch of decimal_counter is
   signal d1_reg, d10_reg, d100_reg: unsigned(3 downto 0);
   signal d1_next, d10_next, d100_next: unsigned(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         d1_reg <= (others=>'0');
         d10_reg <= (others=>'0');
         d100_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         d1_reg <= d1_next;
         d10_reg <= d10_next;
         d100_reg <= d100_next;
      end if;
   end process;
```

```
   -- next-state logic
   d1_next <= "0000" when d1_reg=9 else
              d1_reg + 1;
   d10_next <= "0000" when (d1_reg=9 and d10_reg=9) else
               d10_reg + 1 when d1_reg=9 else
               d10_reg;
   d100_next <=
     "0000" when (d1_reg=9 and d10_reg=9 and d100_reg=9) else
     d100_reg + 1 when (d1_reg=9 and d10_reg=9) else
     d100_reg;
   -- output
   d1 <= std_logic_vector(d1_reg);
   d10 <= std_logic_vector(d10_reg);
   d100 <= std_logic_vector(d100_reg);
end concurrent_arch;
```

Listing 9.13 Three-digit decimal counter using a nested if statement

```
architecture if_arch of decimal_counter is
   signal d1_reg, d10_reg, d100_reg: unsigned(3 downto 0);
   signal d1_next, d10_next, d100_next: unsigned(3 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         d1_reg <= (others=>'0');
         d10_reg <= (others=>'0');
         d100_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         d1_reg <= d1_next;
         d10_reg <= d10_next;
         d100_reg <= d100_next;
      end if;
   end process;
   -- next-state logic
   process(d1_reg,d10_reg,d100_reg)
   begin
      d10_next <= d10_reg;
      d100_next <= d100_reg;
      if d1_reg/=9 then
         d1_next <= d1_reg + 1;
      else -- reach --9
         d1_next <= "0000";
         if d10_reg/=9 then
            d10_next <= d10_reg + 1;
         else -- reach -99
            d10_next <= "0000";
            if d100_reg/=9 then
               d100_next <= d100_reg + 1;
            else -- reach 999
               d100_next <= "0000";
            end if;
```

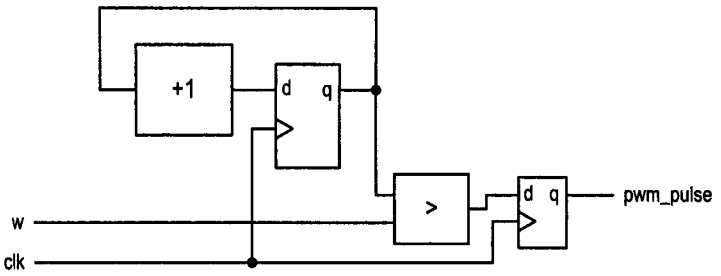


Figure 9.9 Block diagram of a PWM circuit.

```
        end if;
      end if;
    end process;
    -- output
    d1 <= std_logic_vector(d1_reg);
    d10 <= std_logic_vector(d10_reg);
    d100 <= std_logic_vector(d100_reg);
end if_arch;
```

9.2.5 Pulse width modulation circuit

Instead of using the counting patterns directly, some applications generate output signals based on the state of the counter. One example is a pulse width modulation (PWM) circuit. In a square wave, the *duty cycle* is defined as the percentage of time that the signal is asserted as '1' in a period. For example, the duty cycle of a symmetric square wave is 50% since the signal is asserted half of the period. A PWM circuit generates an output pulse with an adjustable duty cycle. It is frequently used to control the on–off time of an external system.

Consider a PWM circuit whose duty cycle can be adjusted in increments of $\frac{1}{16}$, i.e., the duty cycle can be $\frac{1}{16}, \frac{2}{16}, \frac{3}{16}, \ldots, \frac{15}{16}, \frac{16}{16}$. A 4-bit control signal, `w`, which is interpreted as an unsigned integer, specifies the desired duty cycle. The duty cycle will be $\frac{16}{16}$ when `w` is "0000", and will be $\frac{W}{16}$ otherwise. This circuit can be implemented by a mod-16 counter with a special output circuit, and the conceptual diagram is shown in Figure 9.9.

The mod-16 counter circulates through 16 patterns. An output circuit compares the current pattern with the `w` signal and asserts the output pulse when the counter's value is smaller than `w`. The output pulse's period is 16 times the clock period, and $\frac{W}{16}$ of the period is asserted. The VHDL code is shown in Listing 9.14. Note that an additional Boolean expression, `w="0000"`, is included to accommodate the special condition. We also add an output buffer to remove any potential glitch.

Listing 9.14 PWM circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pwm is
    port(
        clk, reset: in std_logic;
        w: in std_logic_vector(3 downto 0);
```

```
      pwm_pulse: out std_logic
   );
end pwm;

architecture two_seg_arch of pwm is
   signal r_reg: unsigned(3 downto 0);
   signal r_next: unsigned(3 downto 0);
   signal buf_reg: std_logic;
   signal buf_next: std_logic;
begin
   -- register & output buffer
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
         buf_reg <= '0';
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
         buf_reg <= buf_next;
      end if;
   end process;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   buf_next <=
       '1' when (r_reg<unsigned(w)) or (w="0000") else
       '0';
   pwm_pulse <= buf_reg;
end two_seg_arch;
```

9.3 REGISTERS AS TEMPORARY STORAGE

Instead of being dedicated to a specific circuit, such as a counter, registers can also be used as general-purpose storage or buffer to store data. Since the circuit size of a D FF is several times larger than that of a RAM cell, using registers as massive storage is not cost-effective. They are normally used to construct small, fast temporal storage in a large digital system. This section examines various storage structures, including a register file, register-based first-in-first-out buffer and register-based look-up table.

9.3.1 Register file

A register file consists of a set of registers. To reduce the amount of wiring and I/O signals, the register file provides only one write port and few read ports, which are shared by all registers for data access. Each register is assigned a binary address as an identifier, and an external system uses the address to specify which register is to be involved in the operation. The storage and retrieval operations are known as the *write* and *read* operations respectively. A processor normally includes a register file as fast temporary storage.

In this subsection, we illustrate the design and coding of a register file with four 16-bit registers and three I/O ports, which include one write port and two read ports. The data signals are labeled `w_data`, `r_data0` and `r_data1`, and the port addresses are labeled

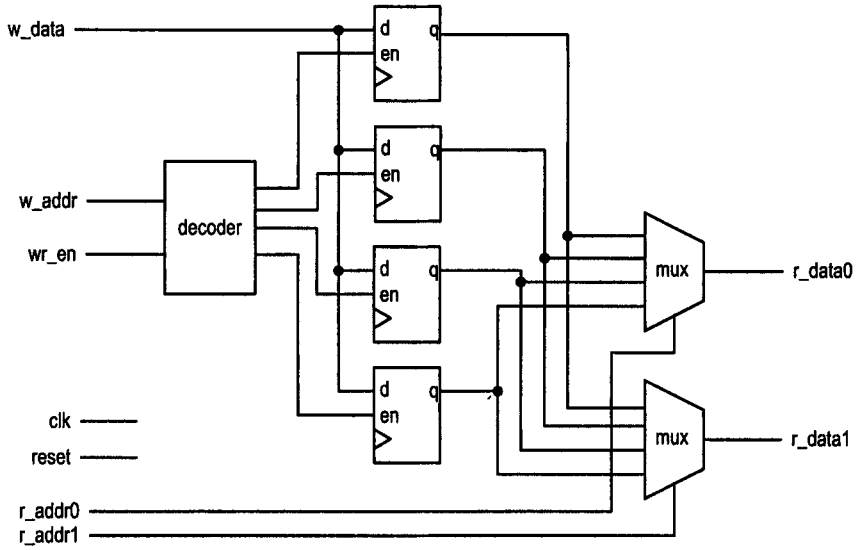


Figure 9.10 Block diagram of a register file.

`w_addr`, `r_addr0` and `r_addr1`. There is also a control signal, `wr_en`, which is the write enable signal to indicate whether a write operation can be performed.

The conceptual diagram is shown in Figure 9.10. The design consists of three major parts: registers with enable signals, a write decoding circuit, and read multiplexing circuits. There are four 16-bit registers, each register with an individual enable signal, `en`. The `en` signal is synchronous and indicates whether the input data can be stored into the register. Its function is identical to the FF example in Section 8.5.1.

The write decoding circuit examines the `wr_en` signal and decodes the write port address. If the `wr_en` signal is asserted, the decoding circuit functions as a regular 2-to-2^2 binary decoder that asserts one of the four `en` signals of the corresponding register. The `w_data` signal will be sampled and stored into the corresponding register at the rising edge of the clock.

The read multiplexing circuit consists of two 4-to-1 multiplexers. It utilizes `r_addr0` and `r_addr1` as the selection signals to route the desired register outputs to the read ports.

Note that the registers are structured as a two-dimensional 4-by-16 array of D FFs and would best be represented by a two-dimensional data type. There is no predefined two-dimensional data type in the IEEE `std_logic_1164` package, and thus we must create a user-defined data type. One way to do it is to create a user-defined "array of arrays" data type. Assume that the number of rows and columns of an array are `ROW` and `COL` respectively. The data type and signal declaration can be written as

```
type aoa_type is array (ROW-1 downto 0) of
     std_logic_vector(COL-1 downto 0);
signal s: aoa_type;
```

We can use `s[i]` to access the ith row of the array and use `s[i][j]` to access the jth bit of the ith row of the array.

Once understanding the basic block diagram and data type, we can derive the VHDL code accordingly. The VHDL code is shown in Listing 9.15. The register and corresponding

enabling circuit are described by two processes. The decoding circuit is described in another process. If wr_en is not asserted, en will be "0000" and no register will be updated. Otherwise, one bit of the en signal will asserted according to the value of the w_addr signal. The read ports are described as two multiplexers.

Listing 9.15 2^2-by-16 register file

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_file is
   port(
      clk, reset: in std_logic;
      wr_en: in std_logic;
      w_addr: in std_logic_vector(1 downto 0);
      w_data: in std_logic_vector(15 downto 0);
      r_addr0, r_addr1: in std_logic_vector(1 downto 0);
      r_data0, r_data1: out std_logic_vector(15 downto 0)
   );
end reg_file;

architecture no_loop_arch of reg_file is
   constant W: natural:=2; -- number of bits in address
   constant B: natural:=16; -- number of bits in data
   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 en: std_logic_vector(2**W-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         array_reg(3) <= (others=>'0');
         array_reg(2) <= (others=>'0');
         array_reg(1) <= (others=>'0');
         array_reg(0) <= (others=>'0');
      elsif (clk'event and clk='1') then
         array_reg(3) <=  array_next(3);
         array_reg(2) <=  array_next(2);
         array_reg(1) <=  array_next(1);
         array_reg(0) <=  array_next(0);
      end if;
   end process;
   -- enable logic for register
   process(array_reg,en,w_data)
   begin
      array_next(3) <= array_reg(3);
      array_next(2) <= array_reg(2);
      array_next(1) <= array_reg(1);
      array_next(0) <= array_reg(0);
      if en(3)='1' then
         array_next(3) <= w_data;
      end if;
```

```
      if en(2)='1' then
         array_next(2) <= w_data;
50    end if;
      if en(1)='1' then
         array_next(1) <= w_data;
      end if;
      if en(0)='1' then
55       array_next(0) <= w_data;
      end if;
   end process;
   -- decoding for write address
   process(wr_en,w_addr)
60 begin
      if (wr_en='0') then
         en <= (others=>'0');
      else
         case w_addr  is
65          when "00" =>   en <= "0001";
            when "01" =>   en <= "0010";
            when "10" =>   en <= "0100";
            when others => en <= "1000";
         end case;
70    end if;
   end process;
   -- read multiplexing
   with r_addr0 select
      r_data0 <=  array_reg(0) when "00",
75                 array_reg(1) when "01",
                   array_reg(2) when "10",
                   array_reg(3) when others;
   with r_addr1 select
      r_data1 <=  array_reg(0) when "00",
80                 array_reg(1) when "01",
                   array_reg(2) when "10",
                   array_reg(3) when others;
end no_loop_arch;
```

Although the description is straightforward, the code is not very compact. The code will be cumbersome and lengthy for a larger register file. A more effective description and the proper use of two-dimensional data types are discussed in Chapter 15.

9.3.2 Register-based synchronous FIFO buffer

A first-in-first-out (FIFO) buffer acts as "elastic" storage between two subsystems. The conceptual diagram is shown in Figure 9.11. One subsystem stores (i.e., writes) data into the buffer, and the other subsystem retrieves (i.e., reads) data from the buffer and removes it from the buffer. The order of data retrieval is same as the order of data being stored, and thus the buffer is known as a *first-in-first-out buffer*. If two subsystems are synchronous (i.e., driven by the same clock), we need only one clock for the FIFO buffer and it is known as a *synchronous* FIFO buffer.

The most common way to construct a FIFO buffer is to add a simple control circuit to a generic memory array, such as a register file or RAM. We can arrange the generic memory

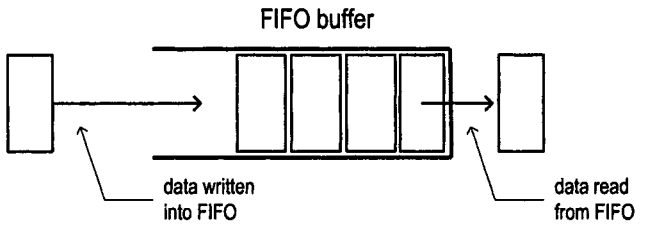


Figure 9.11 Conceptual diagram of a FIFO buffer.

array as a circular queue and use two pointers to mark the beginning and end of the FIFO buffer. The conceptual sketch is shown in Figure 9.12(c). The first pointer, known as the *write pointer* (labeled `wr ptr`), points to the first empty slot in front of the buffer. During a write operation, the data is stored in this designed slot, and the write pointer advances to the next slot (i.e., incremented by 1). The second pointer, known as the *read pointer* (labeled `rd ptr`), points to the end of the buffer. During a read operation, the data is retrieved and the read pointer advances one slot, effectively releasing the slot for future write operations.

Figure 9.12 shows a sequence of write and read operations and the corresponding growth and shrinking of the buffer. Initially, both the read and write pointers point to the 0 address, as in Figure 9.12(a). Since the buffer is empty, no read operation is allowed at this time. After a write operation, the write pointer increments and the buffer contains one item in the 0 address, as in Figure 9.12(b). After a few more write operations, the write pointer continues to increase and the buffer expands accordingly, as in Figure 9.12(c). A read operation is performed afterward. The read pointer advances in the same direction, and the previous slot is released, as in Figure 9.12(d). After several more write operations, the buffer is full, as in Figure 9.12(f), and no write operation is allowed. Several read operations are then performed, and the buffer eventually shrinks to 0, as in Figure 9.12(g), (h) and (i).

The block diagram of a register-based FIFO is shown in Figure 9.13. It consists of a register file and a control circuit, which generates proper read and write pointer values and status signals. Note that the FIFO buffer doesn't have any explicit external address signal. Instead, it utilizes two control signals, `wr` and `re`, for write and read operations. At the rising edge of the clock, if the `wr` signal is asserted and the buffer is not full, the corresponding input data will be sampled and stored into the buffer. The output data from the FIFO is always available. The `re` signal might better be interpreted as a "remove" signal. If it is asserted at the rising edge and the buffer is not empty, the FIFO's read pointer advances one position and makes the current slot available. After the internal delays of the incrementing and routing, new output data is available in FIFO's output port.

During FIFO operation, an overflow occurs when the external system attempts to write new data when the FIFO is full, and an underflow occurs when the external system attempts to read (i.e., remove) a slot when the FIFO is empty. To ensure correct operation, a FIFO buffer must include the `full` and `empty` status signals for the two special conditions. In a properly designed system, the external systems should check the status signals before attempting to access the FIFO.

The major components of a FIFO control circuit are two counters, whose outputs function as write and read pointers respectively. During regular operation, the write counter advances one position when the `wr` signal is asserted at the rising edge of the clock, and the read counter advances one position when the `re` signal is asserted. We normally prefer to add

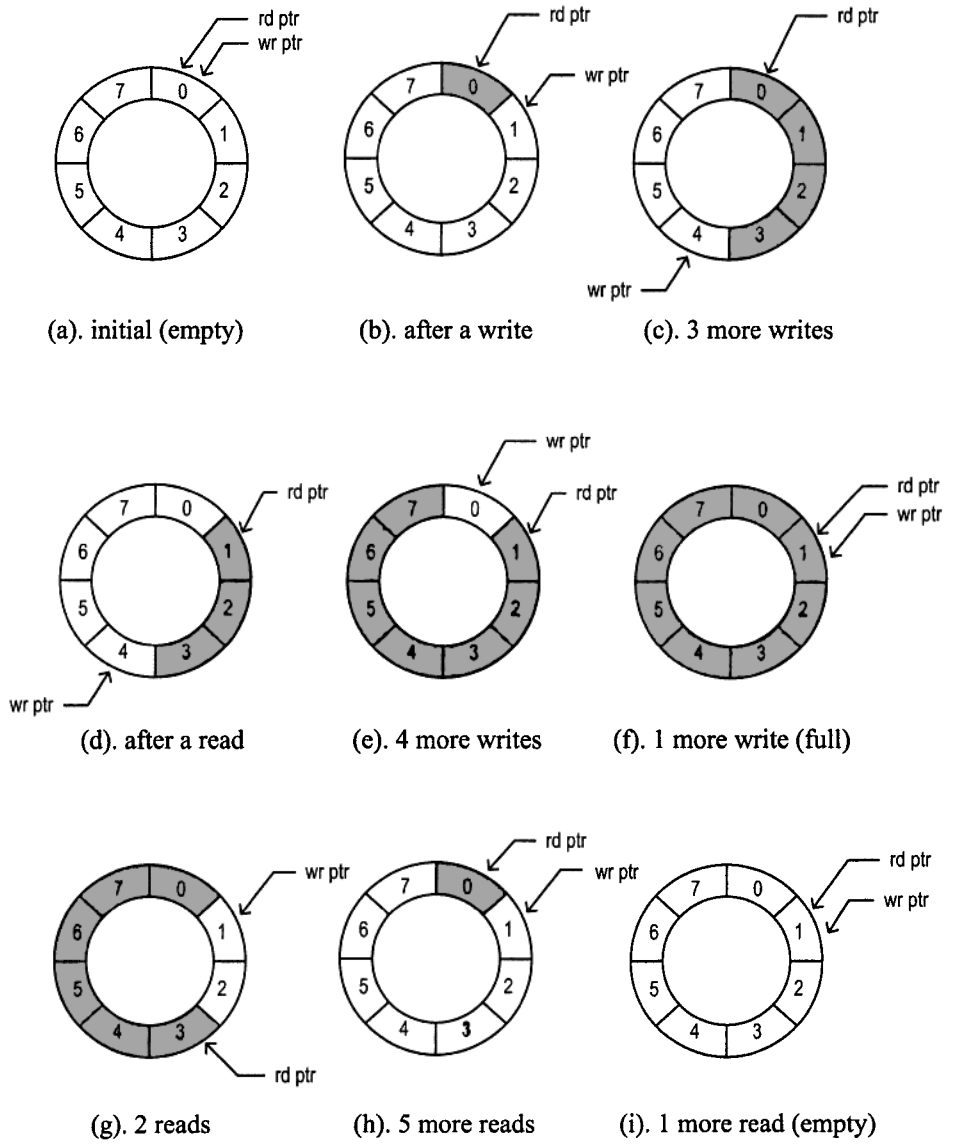


Figure 9.12 Circular-queue implementation of a FIFO buffer.

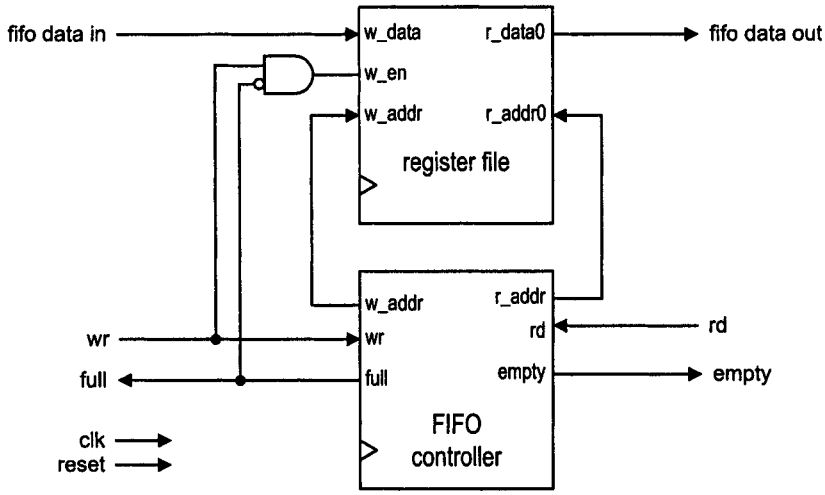


Figure 9.13 Block diagram of a register-based FIFO buffer.

some safety precautions to ensure that data will not be written into a full buffer or removed from an empty buffer. Under these conditions, the counters will retain the previous values.

The difficult part of the control circuit is the handling of two special conditions in which the FIFO buffer is empty or full. When the FIFO buffer is empty, the read pointer is the same as the write pointer, as shown in Figure 9.12(a) and (i). Unfortunately, this is also the case when the FIFO buffer is full, as shown in Figure 9.12(f). Thus, we cannot just use read and write pointers to determine full and empty conditions. There are several schemes to generate the status signals, and all of them involve additional circuitry and FFs. We examine two schemes in this subsection.

FIFO control circuit with augmented binary counters The first method is to use the binary counters for the read and write pointers and increase their sizes by 1 bit. We can determine the full or empty condition by comparing the MSBs of the two pointers. This scheme can be best explained and observed by an example. Consider a FIFO with 3-bit address (i.e., 2^3 words). Two 4-bit counters will be used for the read and write pointers. The counters and the status of a sequence of operations are shown in Table 9.2. The three LSBs of the read and write pointers are used as addresses to access the register file and wrap around after eight increments. They are equal when the FIFO is empty or full. The MSBs of the read and write pointers can be used to distinguish the two conditions. The two bits are the same when the FIFO is empty. After eight write operations, the MSB of the write pointers flips and becomes the opposite of the MSB of the read pointer. The opposite values in MSBs indicate that the FIFO is full. After eight read operations, the MSB of the read pointer flips and becomes identical to the MSB of the write pointer, which indicates that the FIFO is empty again. A more detailed block diagram of this scheme is shown in Figure 9.14.

The VHDL code of a 4-word FIFO controller is shown in Listing 9.16. A constant, `N`, is used inside the architecture body to indicate the number of address bits. Note that the `w_ptr_reg` and `r_ptr_reg` signals, which are the write and read pointers, are increased to `N` + 1 bits.

Table 9.2 Representative sequence of FIFO operations

Write pointer	Read pointer	Operation	Status
0 000	0 000	initialization	empty
0 111	0 000	after 7 writes	
1 000	0 000	after 1 write	full
1 000	0 100	after 4 reads	
1 100	0 100	after 4 writes	full
1 100	1 011	after 7 reads	
1 100	1 100	after 1 read	empty
0 011	1 100	after 7 writes	
0 100	1 100	after 1 write	full
0 100	0 100	after 8 reads	empty

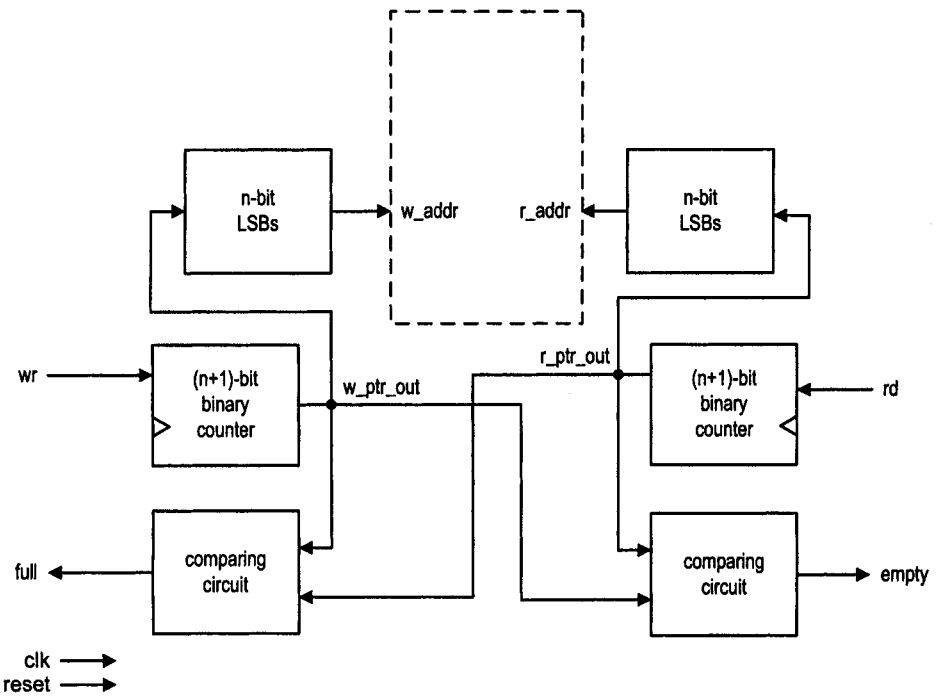


Figure 9.14 Detailed block diagram of an augmented-binary-counter FIFO control circuit.

Listing 9.16 FIFO control circuit with augmented binary counters

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo_sync_ctrl4 is
   port(
      clk, reset: in std_logic;
      wr, rd: in std_logic;
      full, empty: out std_logic;
      w_addr, r_addr: out std_logic_vector(1 downto 0)
   );
end fifo_sync_ctrl4;

architecture enlarged_bin_arch of fifo_sync_ctrl4 is
   constant N: natural:=2;
   signal w_ptr_reg, w_ptr_next: unsigned(N downto 0);
   signal r_ptr_reg, r_ptr_next: unsigned(N downto 0);
   signal full_flag, empty_flag: std_logic;
begin
   -- register
   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;
   -- write pointer next-state logic
   w_ptr_next <=
      w_ptr_reg + 1 when wr='1' and full_flag='0' else
      w_ptr_reg;
   full_flag <=
      '1' when r_ptr_reg(N) /=w_ptr_reg(N) and
            r_ptr_reg(N-1 downto 0)=w_ptr_reg(N-1 downto 0)
         else
      '0';
   -- write port output
   w_addr <= std_logic_vector(w_ptr_reg(N-1 downto 0));
   full <= full_flag;
   -- read pointer next-state logic
   r_ptr_next <=
      r_ptr_reg + 1 when rd='1' and empty_flag='0' else
      r_ptr_reg;
   empty_flag <= '1' when r_ptr_reg=w_ptr_reg else
                 '0';
   -- read port output
   r_addr <= std_logic_vector(r_ptr_reg(N-1 downto 0));
   empty <= empty_flag;
end enlarged_bin_arch;
```

To complete the FIFO buffer, we combine the control circuit and the register file, as shown in Figure 9.13. This can be done by merging the previous register file VHDL code with the FIFO controller code. A more systematic approach is to use component instantiation, which is discussed in Chapter 13.

FIFO control circuit with status FFs An alternative design of a FIFO control circuit is to keep track the state of the empty and full conditions and to use this information, combined with the `wr` and `rd` signals, to determine the new conditions. This scheme does not require augmented counters but needs two extra FFs to record the empty and full statuses. During system initialization, the full status FF is set to '0' and the empty status FF is set to '1'. After initialization, the `wr` and `rd` signals are examined at the rising edge of the clock, and the pointers and the FFs are modified according to the following rules:

- `wr` and `rd` are "00": Since no operation is specified, pointers and FFs remain in the previous state.
- `wr` and `rd` are "11": Write and read operations are performed simultaneously. Since the net size of the buffer remains the same, the empty and full conditions will not change. Both pointers advance one position.
- `wr` and `rd` are "10": This indicates that only a write operation is performed. We must first make sure that the buffer is not full. If that is the case, the write pointer advances one position and the empty status FF should be deasserted. The advancement may make the buffer full. This condition happens if the *next value* of the write pointer is equal to the current value of the read pointer (i.e., the write pointer catches up to the read pointer). If this condition is true, the full status FF will be set to '1' accordingly.
- `wr` and `rd` are "01": This indicates that only a read operation is performed. We must first make sure that the buffer is not empty. If that is the case, the read pointer advances one position and the full status FF should be deasserted. The advancement may make the buffer empty. This condition happens if the *next value* of the read pointer is equal to the current value of the write pointer (i.e., the read pointer catches up to the write pointer). If this condition is true, the empty status FF will be set to '1' accordingly.

The VHDL code for this scheme is shown in Listing 9.17. In this code, we combine the next-state logic of the pointers and FFs into a single process and use a case statement to implement the desired operations under various `wr` and `rd` combinations.

Listing 9.17 FIFO controller with status FFs

```
architecture lookahead_bin_arch of fifo_sync_ctrl4 is
   constant N: natural:=2;
   signal w_ptr_reg, w_ptr_next: unsigned(N-1 downto 0);
   signal w_ptr_succ: unsigned(N-1 downto 0);
   signal r_ptr_reg, r_ptr_next: unsigned(N-1 downto 0);
   signal r_ptr_succ: unsigned(N-1 downto 0);
   signal full_reg, empty_reg: std_logic;
   signal full_next, empty_next: std_logic;
   signal wr_op: std_logic_vector(1 downto 0);
begin
   -- register
   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;
-- status 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 the write and read pointers
w_ptr_succ <= w_ptr_reg + 1;
r_ptr_succ <= r_ptr_reg + 1;
-- next-state logic
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 "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 "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 others => -- write/read;
         w_ptr_next <= w_ptr_succ;
         r_ptr_next <= r_ptr_succ;
   end case;
end process;
-- write port output
```

```
    w_addr <= std_logic_vector(w_ptr_reg);
    full <= full_reg;
    r_addr <= std_logic_vector(r_ptr_reg);
    empty <= empty_reg;
end lookahead_bin_arch;
```

FIFO control circuit with a non-binary counter For the previous two FIFO control circuit implementations, the two incrementors used in the binary counters consume the most hardware resources. If we examine operation of the read and write pointers closely, there is no need to access the register in binary sequence. Any order of access is fine as long as the two pointers circulate through the identical sequence. If we can derive a circuit to generate the status signals, other types of counters can be used for the pointers.

In the first scheme, we enlarge the binary counter and use the extra MSB to determine the status. This approach is based on the special property of the binary counting sequence and cannot easily be modified for other types of counters.

In the second scheme, the status signal relies on the successive value of the counter, and thus this scheme can be applied to any type of counter. Because of its simple next-state logic, LFSR is the best choice. It replaces the incrementor of a binary counter with a few xor cells and can significantly improve circuit size and performance, especially for a large FIFO address space.

Modifying the VHDL code is straightforward. Let us consider a FIFO controller with a 4-bit address. In the original code, the following two statements generate the successive values:

```
w_ptr_succ <= w_ptr_reg + 1;
r_ptr_succ <= r_ptr_reg + 1;
```

They can be replaced by the next-state logic of a 4-bit LFSR:

```
w_ptr_succ <=
   (w_ptr_reg(1) xor w_ptr_reg(0)) & w_ptr_reg(3 downto 1);
r_ptr_succ <=
   (r_ptr_reg(1) xor r_ptr_reg(0)) & r_ptr_reg(3 downto 1);
```

We must also revise the asynchronous reset portion of the code to initialize the counters for a non-zero value.

Recall that an n-bit LFSR circulates through only $2^n - 1$ states, and thus the size of the FIFO buffer is reduced by one accordingly. For a large n, the impact of the reduction is very small. We can also use a Bruijn counter if the entire 2^n address space is required.

9.3.3 Register-based content addressable memory

In a register file, each register in the file is assigned a unique address. When using a register to store a data item, we associate the item with the address, and access (i.e., read or write) this item via the address. An alternative way to identify a data item is to associate each item with a unique "key" and use this key to access the data item. This organization is known as *content addressable memory* (*CAM*). A CAM is used in applications that require high-speed search, such as cache memory management and network routing.

The operation of a CAM can best be explained by a simple example. Consider a network router that examines the 16-bit destination field of an incoming packet and routes it to one of the eight output ports. A 4-word CAM stores information regarding the most frequently

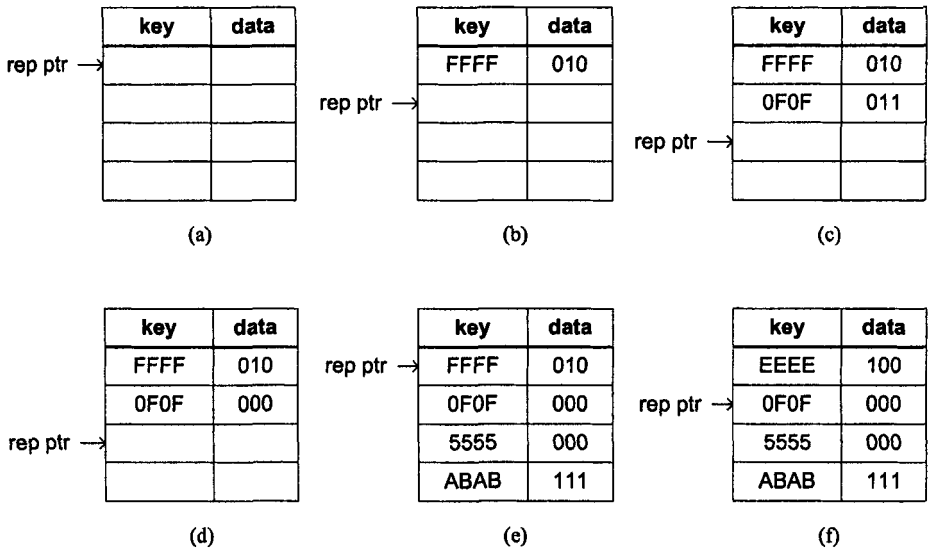


Figure 9.15 Operation of a conceptual CAM.

routed packets. The conceptual sketch is shown in Figure 9.15(a). The CAM includes four words, and each word is composed of a key–data pair. The key of the CAM is the 16-bit destination field, and the data is the 3-bit output port number. Since there are 2^{16} possible combinations for the key, which is far greater than the 4-word capacity of the CAM, we may need to remove an old key–data pair to make room for the new incoming pair. A replacement pointer, labeled `rep ptr` in the diagram, indicates the location of the word to be removed.

Let us first examine a sequence of write operations:

1. Write ($FFFF_{16}$, 010_2), which means an item with a key of $FFFF_{16}$ and data of 010_2. Since the CAM is empty and no key exists, the item is stored into the CAM, as shown in Figure 9.15(b). The replacement pointer advances accordingly.
2. Write ($0F0F_{16}$, 011_2). Since no existing key matches the new input key, the item is stored into the CAM, as shown in Figure 9.15(c).
3. Write ($0F0F_{16}$, 000_2). The input key matches an existing key in the CAM. The corresponding data of the key is replaced by the new data 000_2, as shown in Figure 9.15(d).
4. Write two new items. The CAM is now full, as shown in Figure 9.15(e). We assume that the replacement pointer moves in a round-robin fashion and thus returns to the first location of the table.
5. Write ($EEEE_{16}$, 100_2). Since the CAM is full now, a word must be removed to make place for the new item. The content of the first location is discarded for the new item, as shown in Figure 9.15(f).

To perform a read operation, we present the key as the input, and the data associated with the key will be routed to the output. For example, if the key is $EEEE_{16}$, the output of the CAM becomes 100_2. Since there is a chance that the input key does not match any stored key, a CAM usually contains an additional output signal, `hit`, to indicate whether there is a match (i.e., a *hit*).

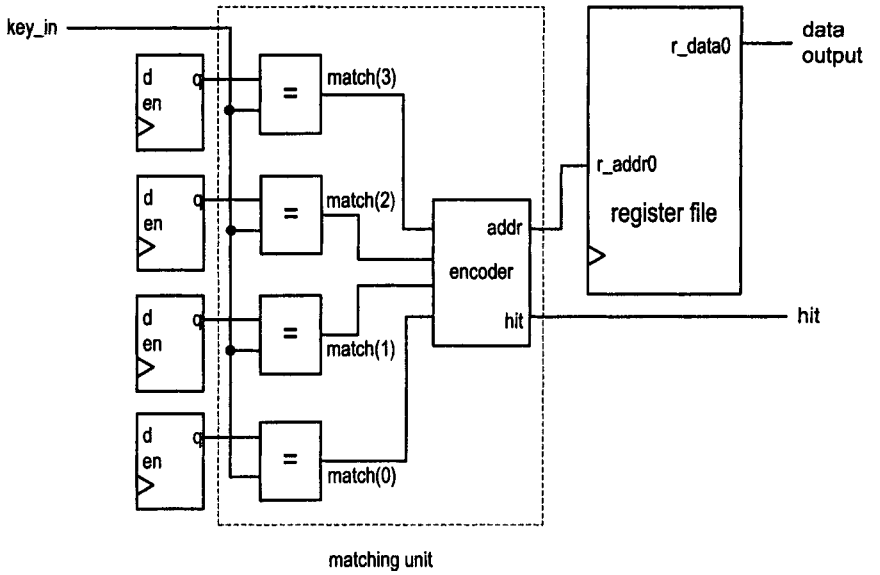


Figure 9.16 Matching circuit of a 4-word CAM.

Similar to SRAM, many technology libraries contain predesigned CAM modules that are constructed and optimized at the transistor level. Although the density of these modules is very high, accessing a CAM cell requires more time than is required by an FF. To improve performance, we sometimes want to use FFs to implement a small, fast synchronous CAM in the critical part of the system.

The major difference between a register file and a CAM is that the CAM uses a key, instead of an address, to identify a data item. One way to construct a CAM is to separate the storage into two register arrays, one for the data and one for the key. In our discussion, we call them a data file and a key file respectively. The data file is organized as a register file and uses an address to access its data. The key file contains a matching circuit, which compares the input key with the content of the key array and generates the corresponding address of the matched key. The address is then used to access the data stored in the data file.

The implementation of a CAM is fairly involved and we start with the read operation. The most unique component is the key file's matching circuit. The block diagram of the matching circuit of a 4-entry CAM and the relevant circuits is shown in Figure 9.16. The output of each register of the key array is compared with the current value of the input key (i.e., the `key_in` signal) and a 1-bit matching signal (i.e., the `match(i)` signal) is generated accordingly. Since each stored key is unique, at most one can be matched. We use the `hit` signal to indicate whether a match occurs (i.e., whether the input key is a "hit" or a "miss"). If the `key_in` signal is a hit, one bit of the 4-bit `match` signal is asserted. A 2^2-to-2 binary encoder generates the binary code of the matched location. The code is then used as the read port address of the data file, and the corresponding data item is routed to the output. Note that the output is not valid if the `hit` signal is not asserted.

The function of the key file is somewhat like a "reversed read operation" of a register file. In a register file, we present the address as an input and obtain the content of the

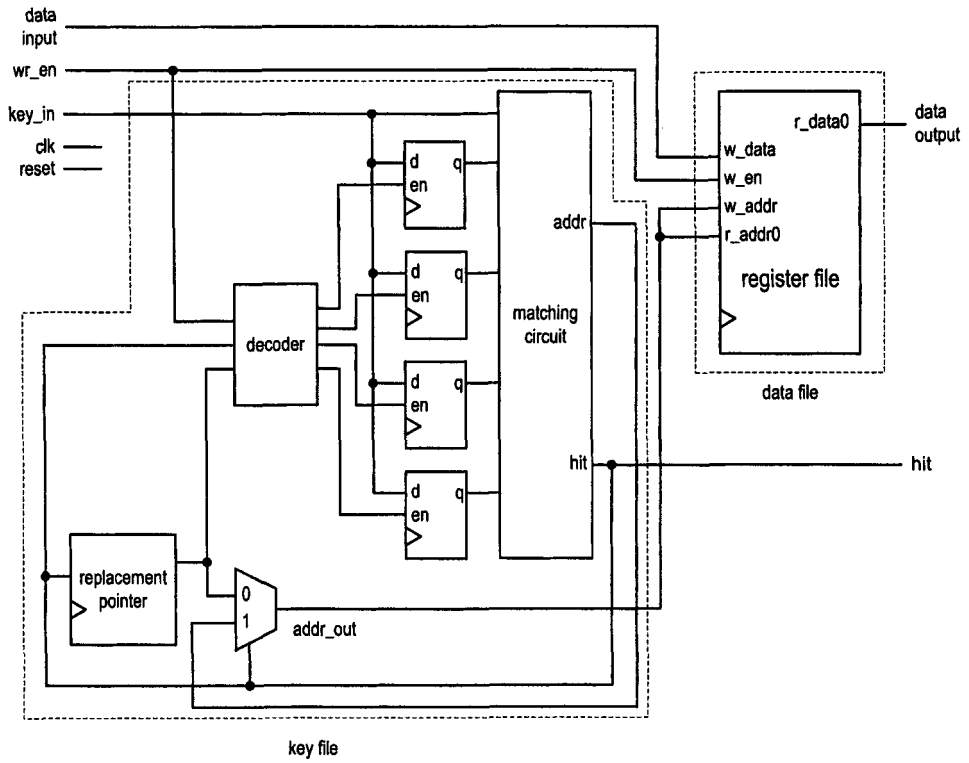


Figure 9.17 Block diagram of a 4-word register-based CAM.

corresponding register. On the other hand, we present a key (which is like the content) as the input to a key file and obtain the address where the key is stored.

The write operation is more complicated because of the possible miss condition. During a write operation, we present a key–data pair. If the key is a hit, the matching circuit will generate the write address of the data file, and the input data will be stored into the corresponding location. If the key is a miss, several tasks must be performed:

- Find an available register in the key array.
- Store the input key into this register.
- Store the data into the corresponding address in the data file.

The block diagram of the complete 4-word CAM is shown in Figure 9.17. The write operation of the data file is controlled by the external `wr_en` signal and the address is specified by the `addr_out` signal. The `addr_out` signal has two possible sources, one from the matching circuit and one from the replacement pointer. The first address is used if the input key is a hit. The *replacement pointer* is a circuit that keeps track of the available register location in the key array. The circuit updates the value when a miss occurs during the write operation. Its output value is used if the input key leads to a miss.

The writing operation of the key array is controlled by a decoding circuit similar to that of a register file. A key register can be written only if the `wr_en` signal is asserted and a miss occurs. If this is the case, the input key will be loaded into a register in the key array with the address specified by the replacement pointer.

The capacity of a register-based CAM is normally small, and the CAM is used to keep the "most frequently used" key–data pairs. When it is full and a miss occurs, a stored pair must be discarded to make place for a new pair. The *replacement policy* determines how to select the pair, and this policy is implemented by the replacement pointer circuit. One simple policy is the FIFO policy, which can be implemented by a binary counter. Initially, the CAM is empty and the counter is zero. The counter increments as the key–data pairs are stored into the CAM. The CAM is full when the counter reaches its maximal value. When a new pair comes and a miss occurs, the counter returns to 0 and wraps around. This corresponds to overwriting (i.e., discarding) the pair with the oldest key and storing the new pair in its location, achieving the desired FIFO policy.

A register file normally has one write port and several read ports. In theory, the same configuration can be achieved in a CAM by presenting several keys in parallel and using several matching circuits to generate multiple addresses. However, this is not common because of the complexity of the comparison circuit, and we normally use one input key signal, as in Figure 9.17. The read operation will be performed if the `wr_en` signal is not asserted.

The VHDL code of the data file is similar to the register file discussed in Section 9.3.1. We can even use a regular register file by connecting the `addr_out` signal to the `w_addr` and `r_addr0` signals of the register file, as in Figure 9.17. The VHDL code of the key file of a 4-word, 16-bit CAM is shown in Listing 9.18. It follows the block diagram of Figure 9.17. A 2-bit binary counter is used to implement the FIFO replacement policy.

Listing 9.18 Key file of a 4-word CAM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity key_file is
   port(
      clk, reset: in std_logic;
      wr_en: in std_logic;
      key_in: in std_logic_vector(15 downto 0);
      hit: out std_logic;
      addr_out: out std_logic_vector(1 downto 0)
   );
end key_file;

architecture no_loop_arch of key_file is
   constant WORD: natural:=2;
   constant BIT: natural:=16;
   type reg_file_type is array (2**WORD-1 downto 0) of
        std_logic_vector(BIT-1 downto 0);
   signal array_reg: reg_file_type;
   signal array_next: reg_file_type;
   signal en: std_logic_vector(2**WORD-1 downto 0);
   signal match: std_logic_vector(2**WORD-1 downto 0);
   signal rep_reg, rep_next: unsigned(WORD-1 downto 0);
   signal addr_match: std_logic_vector(WORD-1 downto 0);
   signal wr_key, hit_flag: std_logic;
begin
   -- register
   process(clk,reset)
```

```
    begin
30      if (reset='1') then
           array_reg(3) <= (others=>'0');
           array_reg(2) <= (others=>'0');
           array_reg(1) <= (others=>'0');
           array_reg(0) <= (others=>'0');
35      elsif (clk'event and clk='1') then
           array_reg(3) <=  array_next(3);
           array_reg(2) <=  array_next(2);
           array_reg(1) <=  array_next(1);
           array_reg(0) <=  array_next(0);
40      end if;
    end process;
    -- enable logic for register
    process(array_reg,en,key_in)
    begin
45      array_next(3) <= array_reg(3);
        array_next(2) <= array_reg(2);
        array_next(1) <= array_reg(1);
        array_next(0) <= array_reg(0);
        if en(3)='1' then
50         array_next(3) <= key_in;
        end if;
        if en(2)='1' then
           array_next(2) <= key_in;
        end if;
55      if en(1)='1' then
           array_next(1) <= key_in;
        end if;
        if en(0)='1' then
           array_next(0) <= key_in;
60      end if;
    end process;

    -- decoding for write address
    wr_key <= '1' when (wr_en='1' and hit_flag='0') else
65            '0';
    process(wr_key,rep_reg)
    begin
       if (wr_key='0') then
          en <= (others=>'0');
70     else
          case rep_reg  is
             when "00" =>   en <= "0001";
             when "01" =>   en <= "0010";
             when "10" =>   en <= "0100";
75           when others => en <= "1000";
          end case;
       end if;
    end process;

80  -- replacement pointer
    process(clk,reset)
```

```
   begin
      if (reset='1') then
         rep_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         rep_reg <= rep_next;
      end if;
   end process;
   rep_next <= rep_reg + 1 when wr_key='1' else
               rep_reg;

   -- key comparison
   process(array_reg,key_in)
   begin
      match <= (others=>'0');
      if array_reg(3)=key_in then
         match(3) <= '1';
      end if;
      if array_reg(2)=key_in then
         match(2) <= '1';
      end if;
      if array_reg(1)=key_in then
         match(1) <= '1';
      end if;
      if array_reg(0)=key_in then
         match(0) <= '1';
      end if;
   end process;
   -- encoding
   with match select
      addr_match <=
         "00" when "0001",
         "01" when "0010",
         "10" when "0100",
         "11" when others;
   -- hit
   hit_flag <= '1' when match /="0000" else '0';
   --output
   hit <= hit_flag;
   addr_out <= addr_match when (hit_flag='1') else
               std_logic_vector(rep_reg);
end no_loop_arch;
```

As in the register file and FIFO buffer, the code will be cumbersome for a larger CAM. A more systematic approach is discussed in Chapter 15.

9.4 PIPELINED DESIGN

Pipeline is an important technique to increase the performance of a system. The basic idea is to overlap the processing of several tasks so that more tasks can be completed in the same amount of time. If a combinational circuit can be divided into stages, we can insert buffers (i.e., registers) at proper places and convert the circuit into a pipelined design. This section

introduces the concept of pipeline and shows how to add pipeline to the combinational multiplier discussed in Chapter 8.

9.4.1 Delay versus throughput

Before we study the pipelined circuit, it will be helpful to first understand the difference between delay and throughput, two criteria used to examine the performance of a system. *Delay* is the time required to complete one task, and *throughput* is the number of tasks that can be completed per unit time. The two are related but are not identical.

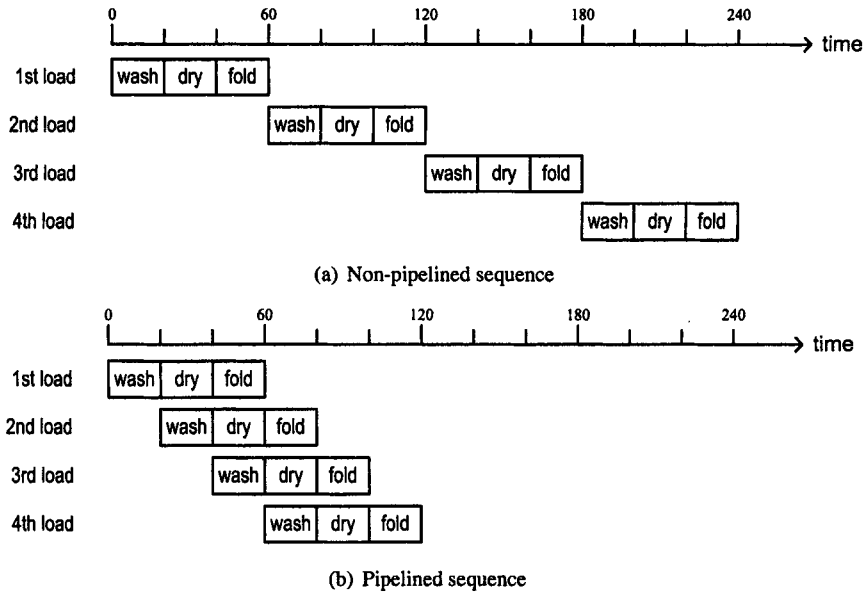
To illustrate the concept, let us use the bank ATM machine transaction as an example. Assume that a bank branch originally has only one ATM machine and it takes 3 minutes to complete a transaction. The delay to complete one transaction is 3 minutes and the maximal throughput is 20 transactions per hour. If the bank wishes to increase the performance of its ATM system, there are two options. The first option is to use a newer, faster ATM machine. For example, the bank can install a machine that takes only 1.5 minutes to complete a transaction. The second option is to add another machine so that there are two ATM machines running in parallel. For the first option, the delay becomes 1.5 minutes and the maximal throughput increases to 40 transactions per hour. For the second option, the transaction delay experienced by a user is still 3 minutes and thus remains the same. However, since there are two ATM machines, the system's maximal throughput is doubled to 40 transactions per hour. In summary, the first option reduces the delay in an individual transaction and increases the throughput at the same time, whereas the second option can only improve the throughput.

Adding pipeline to a combinational circuit is somewhat like the second option and can only increase a system's throughput. It will not reduce the delay in an individual task. Actually, because of the overhead introduced by the registers and non-ideal stage division, the delay will be worse than in the non-pipelined design.

9.4.2 Overview on pipelined design

Pipelined laundry The pipelining technique can be applied to a task that is processed in stages. To illustrate the concept, let us consider the process of doing laundry. Assume that we do a load of laundry in three stages, which are washing, drying and folding, and that each stage takes 20 minutes. For non-pipelined processing, a new load cannot start until the previous load is completed. The time line for processing four loads of laundry is shown in Figure 9.18(a). It takes 240 minutes (i.e., 4*3*20 minutes) to complete the four loads. In terms of the performance criteria, the delay of processing one load is 60 minutes and the throughput is $\frac{1}{60}$ load per minute (i.e., four loads in 240 minutes).

If we examine the process carefully, there is room for improvement. After 20 minutes, the washing stage is done and the washer is idle. We can start a new load at this point rather than waiting for completion of the entire laundry process. Since each stage takes the same amount of time, there will be no contention in subsequent stages. The time line of the pipelined version of four loads is shown in Figure 9.18(b). It takes 120 minutes to complete the four loads. In terms of performance, the delay in processing one load remains 60 minutes. However, the throughput is increased to $\frac{2}{60}$ load per minute (i.e., 4 loads in 120 minutes). If we process k loads, it will take $40+20k$ minutes. The throughput becomes $\frac{k}{40+20k}$ load per minute. If k is large, the throughput approaches $\frac{1}{20}$ load per minute, which is three times better than that of the non-pipelined process.



(a) Non-pipelined sequence

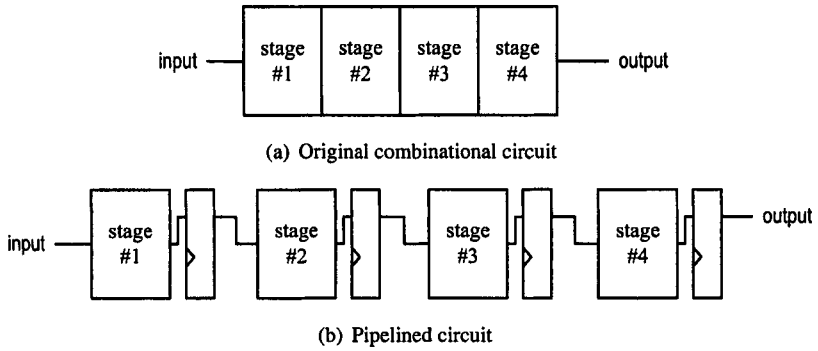
(b) Pipelined sequence

Figure 9.18 Timing diagrams of pipelined and non-pipelined laundry sequences.

This example shows an ideal pipelined situation, in which a task can be divided into stages of identical delays. If washing, drying and folding take 15, 25 and 20 minutes respectively, we must accommodate the stage with the longest delay or a conflict will occur. For example, when washing is done at the first stage, we have to wait for 10 minutes before putting a new load into the dryer. Under this scenario, we can only start a new load every 25 minutes at best. The delay to complete one load is increased from 60 minutes to 75 minutes (i.e., 3*25 minutes) now. The throughput for k loads becomes $\frac{k}{50+25k}$ load per minute and approaches $\frac{1}{25}$ load per minute when k is large. Note that while the pipelined processing helps improving the throughput, it actually increases the processing delay for a single load.

Pipelined combinational circuit The same pipeline concept can be applied to combinational circuits. We can divide a combinational circuit in stages so that the processing of different tasks can be overlapped, as in the laundry example. To ensure that the signals in each stage flow in the correct order and to prevent any potential race, we must add a register between successive stages, as shown in the four-stage pipelined combinational circuit of Figure 9.19. An output buffer is also included in the last stage. A register functions as a "flood gate" that regulates signal flow. It ensures that the signals can be passed to the next stage only at predetermined points. The clock period of the registers should be large enough to accommodate the slowest stage. At a faster stage, output will be blocked by the register even when the processing has been completed earlier. The output data at each stage will be sampled and stored into registers at the rising edge of the clock. These data will be used as input for the next stage and remain unchanged in the remaining part of the clock period. At the end of the clock period, the new output data are ready. They will be sampled and passed to the next stage (via the register) at the next rising edge of the clock.

The effectiveness of the pipelined circuit is judged by two performance criteria, the delay and the throughput. Consider the previous four-stage pipelined combinational circuit. Assume that the original propagation delays of the four stages are T_1, T_2, T_3 and T_4



(a) Original combinational circuit

(b) Pipelined circuit

Figure 9.19 Construction of a four-stage pipelined circuit.

respectively. Let T_{max} be the longest propagation delay among the four stages:

$$T_{max} = \max(T_1, T_2, T_3, T_4)$$

The clock period needs to accommodate the longest delay plus the overhead introduced by the buffer register in each stage, which includes the setup time (T_{setup}) and the clock-to-q delay (T_{cq}) of the register. Thus, the minimal clock period, T_c, is

$$T_c = T_{max} + T_{setup} + T_{cq}$$

In the original non-pipelined combinational circuit, the propagation delay in processing one data item is

$$T_{comb} = T_1 + T_2 + T_3 + T_4$$

For the pipelined design, processing a data requires four clock cycles and the propagation delay becomes

$$T_{pipe} = 4T_c = 4T_{max} + 4(T_{setup} + T_{cq})$$

This is clearly worse than the propagation delay of the original circuit.

The second performance criterion is the throughput. Since there is no overlapping when the data is processed, the maximal throughput of the non-pipelined design is $\frac{1}{T_{comb}}$. The throughput of the pipelined design can be derived by calculating the time required to complete k consecutive data items. When the process starts, the pipeline is empty. It takes $3T_c$ to fill the first three stages, and the pipeline does not generate an output in the interval. After this, the pipeline is full and the circuit can produce one output in each clock cycle. Thus, it requires $3T_c + kT_c$ to process k data items. The throughput is $\frac{k}{3T_c+kT_c}$ and approaches $\frac{1}{T_c}$ as k becomes very large.

In an ideal pipelined scenario, the propagation delay of each stage is identical (which implies that $T_{max} = \frac{T_{comb}}{4}$), and the register overhead (i.e., $T_{setup} + T_{cq}$) is comparably small and can be ignored. T_{pipe} can be simplified as

$$T_{pipe} = 4T_c \approx 4T_{max} = T_{comb}$$

The throughput becomes

$$\frac{1}{T_c} \approx \frac{1}{T_{max}} = \frac{4}{T_{comb}}$$

This implies that the pipeline imposes no penalty on the delay but increases the throughput by a factor of 4.

The discussion of four-stage pipelined design can be generalized to an N-stage pipeline. In the ideal scenario, the delay to process one data item remains unchanged and the throughput can be increased N-fold. This suggests that it is desirable to have more stages in the pipeline. However, when N becomes large, the propagation delay of each stage becomes smaller. Since $T_{setup} + T_{cq}$ of the register remains the same, its impact becomes more significant and can no longer be ignored. Thus, extremely large N has less effect and may even degrade the performance. In reality, it is also difficult, if not impossible, to keep dividing the original combinational circuit into smaller and smaller stages.

When discussing the throughput of a pipelined system, we have to be aware of the condition to obtain maximal throughput. The assumption is that the external data are fed into the pipeline at a rate of $\frac{1}{T_c}$ so that the pipeline is filled all the time. If the external input data cannot be issued fast enough, there will be slack (a "bubble") inside the pipeline, and the throughput will be decreased accordingly. If the external data is issued only sporadically, the pipelined design will not improve the performance at all.

9.4.3 Adding pipeline to a combinational circuit

Although we can add pipeline to any combinational circuit by inserting registers into the intermediate stages, the pipelined version may not provide better performance. The previous analysis shows that the good candidate circuits for effective pipeline design should include the following characteristics:

- There is enough input data to feed the pipelined circuit.
- The throughput is a main performance criterion.
- The combinational circuit can be divided into stages with similar propagation delays.
- The propagation delay of a stage is much larger than the setup time and the clock-to-q delay of the register.

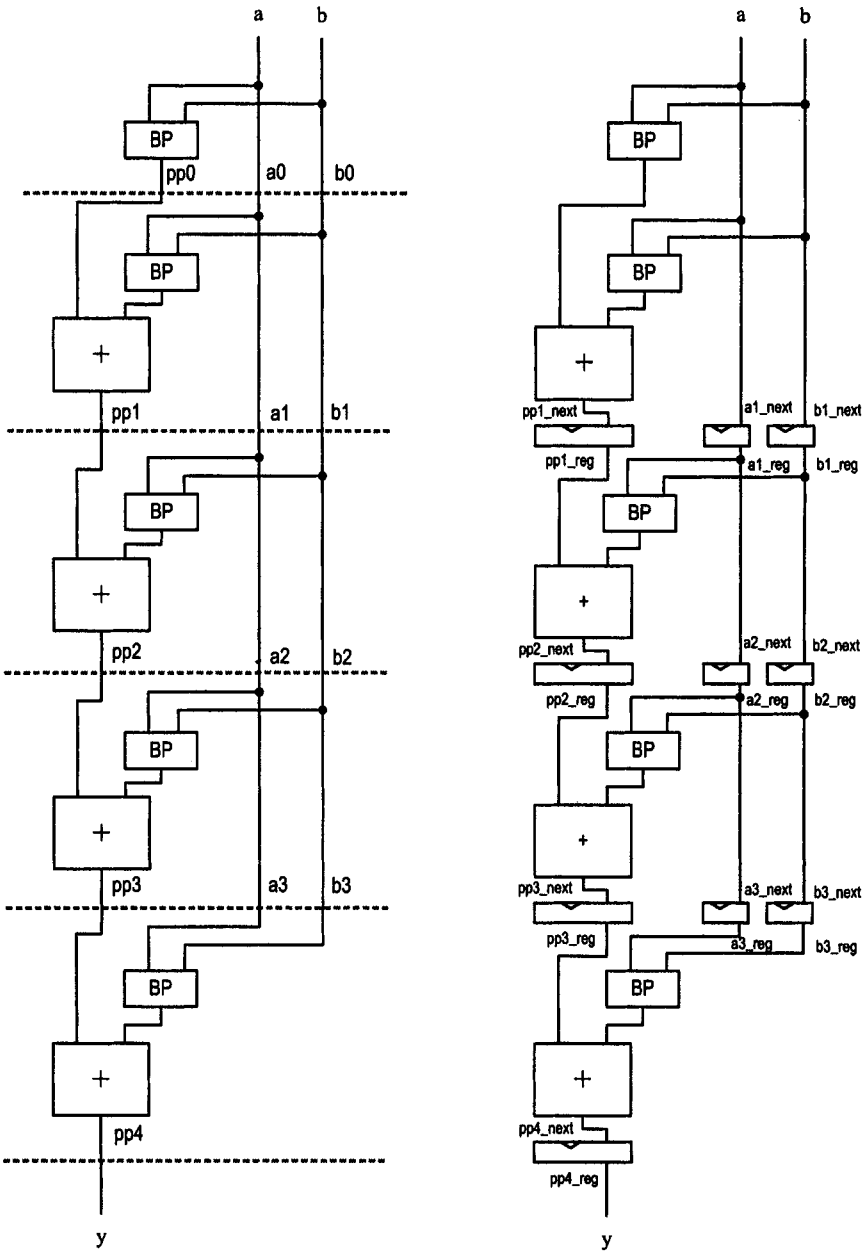
If a circuit is suitable for the pipelined design, we can convert the original circuit and derive the VHDL code by the following procedure:

1. Derive the block diagram of the original combinational circuit and arrange the circuit as a cascading chain.
2. Identify the major components and estimate the relative propagation delays of these components.
3. Divide the chain into stages of similar propagation delays.
4. Identify the signals that cross the boundary of the chain.
5. Insert registers for these signals in the boundary.

This procedure is illustrated by the examples in the following subsections.

Simple pipelined adder-based multiplier The adder-based multiplier discussed in Section 7.5.4 uses multiple adders to sum the bit products in stages and thus is a natural match for a pipelined design. Our design is based on the scheme used in the `comb1_arch` architecture of Listing 7.34. To reduce the clutter in the block diagram, we use a 5-bit multiplier to demonstrate the design process. The design approach can easily be extended to an 8-bit or larger multiplier.

The two major components are the adder and bit-product generation circuit. To facilitate the pipeline design process, we can arrange these components in cascade. The rearranged block diagram is shown in Figure 9.20(a). The circuit to generate the bit product is labeled BP in the diagram. Since the bit-product generation circuit involves only bitwise and operation



(a). Non-pipelined design

(b). Pipelined design

Figure 9.20 Block diagrams of non-pipelined and four-stage pipelined multipliers.

and the padding of 0's, its propagation delay is small. We combine it with the adder to form a stage. The division of the the circuit is shown in Figure 9.20(a), in which the boundary of each stage is shown by a dashed line. To help the coding, a signal is given a unique name in each stage. For example, the a signal is renamed a0, a1, a2 and a3 in the zeroth, first, second and third stages of the pipeline respectively. Since no addition is needed to generate the first partial product (i.e., the pp0 signal), the zeroth and first stages can be merged into a single stage later.

For a signal crossing the stage boundary line, a register is needed between the two stages. There are two types of registers. The first type of register is used to accommodate the computation flow and to store the intermediate results, which are the partial products, pp1, pp2, pp3 and pp4, in the diagram. The second type of register preserves the information needed for each stage, which are a1, a2, a3, b1, b2 and b3. The function of these registers is less obvious. In this pipeline, the processing at each stage depends on the partial product result from the preceding stage, as well as the values of the a and b signals. Note that four multiplications are performed concurrently in the pipeline, each with its own values for the a and b signals. As the partial product calculation progresses through the pipeline, these values must go with the calculation in each stage. The second type of register essentially carries the original values along the pipeline so that a correct copy of the input data is available in each stage. The completed pipelined multiplier with proper registers is shown in Figure 9.20(b).

Following the diagram, we can derive the VHDL code accordingly. The code of the rearranged combinational multiplier is shown in Listing 9.19. The creation of the new signal names is only for later use and should have no effect on synthesis.

Listing 9.19 Non-pipelined multiplier in cascading stages

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mult5 is
   port(
      clk, reset: in std_logic;
      a, b: in std_logic_vector(4 downto 0);
      y: out std_logic_vector(9 downto 0)
   );
end mult5;

architecture comb_arch of mult5 is
   constant WIDTH: integer:=5;
   signal a0, a1, a2, a3: std_logic_vector(WIDTH-1 downto 0);
   signal b0, b1, b2, b3: std_logic_vector(WIDTH-1 downto 0);
   signal bv0, bv1, bv2, bv3, bv4:
      std_logic_vector(WIDTH-1 downto 0);
   signal bp0, bp1, bp2, bp3, bp4:
      unsigned(2*WIDTH-1 downto 0);
   signal pp0, pp1, pp2, pp3, pp4:
      unsigned(2*WIDTH-1 downto 0);
begin
   -- stage 0
   bv0 <= (others=>b(0));
   bp0 <=unsigned("00000" & (bv0 and a));
   pp0 <= bp0;
```

```
    a0 <= a;
    b0 <= b;
    -- stage 1
    bv1 <= (others=>b0(1));
    bp1 <=unsigned("0000" & (bv1 and a0) & "0");
    pp1 <= pp0 + bp1;
    a1 <= a0;
    b1 <= b0;
    -- stage 2
    bv2 <= (others=>b1(2));
    bp2 <=unsigned("000" & (bv2 and a1) & "00");
    pp2 <= pp1 + bp2;
    a2 <= a1;
    b2 <= b1;
    -- stage 3
    bv3 <= (others=>b2(3));
    bp3 <=unsigned("00" & (bv3 and a2) & "000");
    pp3 <= pp2 + bp3;
    a3 <= a2;
    b3 <= b2;
    -- stage 4
    bv4 <= (others=>b3(4));
    bp4 <=unsigned("0" & (bv4 and a3) & "0000");
    pp4 <= pp3 + bp4;
    -- output
    y <= std_logic_vector(pp4);
end comb_arch;
```

When converting the circuit into a pipelined version, we first add the specifications for the registers and then reconnect the input and output of each stage to the registers. Instead of using the output from the preceding stage, each stage of pipeline circuit obtains its input from the boundary register. Similarly, the output of each stage is now connected to the input of the register rather than feeding directly to the next stage. For example, the pp2 signal of the non-pipelined circuit is generated in the second stage and is then used in the third stage:

```
-- stage 2
pp2 <= pp1 + bp2;
-- stage 3
pp3 <= pp2 + bp3;
```

In the pipelined design, the signal should be stored in a register, and the code becomes

```
-- register
   if (reset ='1') then
      pp2_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      pp2_reg <= pp2_next;
   end if;
. . .
-- stage 2
pp2_next <= pp1_reg + bp2;
-- stage 3
pp3_next <= pp2_reg + bp3;
```

The complete VHDL code of the four-stage pipelined circuit is shown in Listing 9.20.

Listing 9.20 Four-stage pipelined multiplier

```
architecture four_stage_pipe_arch of mult5 is
   constant WIDTH: integer:=5;
   signal a1_reg, a2_reg, a3_reg:
      std_logic_vector(WIDTH-1 downto 0);
   signal a0, a1_next, a2_next, a3_next:
      std_logic_vector(WIDTH-1 downto 0);
   signal b1_reg, b2_reg, b3_reg:
      std_logic_vector(WIDTH-1 downto 0);
   signal b0, b1_next, b2_next, b3_next:
      std_logic_vector(WIDTH-1 downto 0);
   signal bv0, bv1, bv2, bv3, bv4:
      std_logic_vector(WIDTH-1 downto 0);
   signal bp0, bp1, bp2, bp3, bp4:
      unsigned(2*WIDTH-1 downto 0);
   signal pp1_reg, pp2_reg, pp3_reg, pp4_reg:
      unsigned(2*WIDTH-1 downto 0);
   signal pp0, pp1_next, pp2_next, pp3_next, pp4_next:
      unsigned(2*WIDTH-1 downto 0);
begin
   -- pipeline registers (buffers)
   process(clk,reset)
   begin
      if (reset ='1') then
         pp1_reg <= (others=>'0');
         pp2_reg <= (others=>'0');
         pp3_reg <= (others=>'0');
         pp4_reg <= (others=>'0');
         a1_reg <= (others=>'0');
         a2_reg <= (others=>'0');
         a3_reg <= (others=>'0');
         b1_reg <= (others=>'0');
         b2_reg <= (others=>'0');
         b3_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         pp1_reg <= pp1_next;
         pp2_reg <= pp2_next;
         pp3_reg <= pp3_next;
         pp4_reg <= pp4_next;
         a1_reg <= a1_next;
         a2_reg <= a2_next;
         a3_reg <= a3_next;
         b1_reg <= b1_next;
         b2_reg <= b2_next;
         b3_reg <= b3_next;
      end if;
   end process;

   -- merged stage 0 & 1 for pipeline
   bv0 <= (others=>b(0));
   bp0 <=unsigned("00000" & (bv0 and a));
   pp0 <= bp0;
   a0 <= a;
```

```
    b0 <= b;
    --
    bv1 <= (others=>b0(1));
    bp1 <=unsigned("0000" & (bv1 and a0) & "0");
    pp1_next <= pp0 + bp1;
    a1_next <= a0;
    b1_next <= b0;
    -- stage 2
    bv2 <= (others=>b1_reg(2));
    bp2 <=unsigned("000" & (bv2 and a1_reg) & "00");
    pp2_next <= pp1_reg + bp2;
    a2_next <= a1_reg;
    b2_next <= b1_reg;
    -- stage 3
    bv3 <= (others=>b2_reg(3));
    bp3 <=unsigned("00" & (bv3 and a2_reg) & "000");
    pp3_next <= pp2_reg + bp3;
    a3_next <= a2_reg;
    b3_next <= b2_reg;
    -- stage 4
    bv4 <= (others=>b3_reg(4));
    bp4 <=unsigned("0" & (bv4 and a3_reg) & "0000");
    pp4_next <= pp3_reg + bp4;
    -- output
    y <= std_logic_vector(pp4_reg);
end four_stage_pipe_arch;
```

We can adjust the number of stages by adding or removing buffer registers. For example, we can reduce the number of pipeline stages by removing the registers in the first and third stages and create a two-stage pipelined multiplier. The revised VHDL code is shown in Listing 9.21.

Listing 9.21 Two-stage pipelined multiplier

```
architecture two_stage_pipe_arch of mult5 is
    constant WIDTH: integer:=5;
    signal a2_reg: std_logic_vector(WIDTH-1 downto 0);
    signal a0, a1, a2_next, a3:
       std_logic_vector(WIDTH-1 downto 0);
    signal b2_reg: std_logic_vector(WIDTH-1 downto 0);
    signal b0, b1, b2_next, b3:
       std_logic_vector(WIDTH-1 downto 0);
    signal bv0, bv1, bv2, bv3, bv4:
       std_logic_vector(WIDTH-1 downto 0);
    signal bp0, bp1, bp2, bp3, bp4:
       unsigned(2*WIDTH-1 downto 0);
    signal pp2_reg, pp4_reg: unsigned(2*WIDTH-1 downto 0);
    signal pp0, pp1, pp2_next, pp3, pp4_next:
       unsigned(2*WIDTH-1 downto 0);
begin
    -- pipeline registers (buffers)
    process(clk,reset)
    begin
       if (reset ='1') then
```

```
            pp2_reg <= (others=>'0');
            pp4_reg <= (others=>'0');
            a2_reg <= (others=>'0');
            b2_reg <= (others=>'0');
        elsif (clk'event and clk='1') then
            pp2_reg <= pp2_next;
            pp4_reg <= pp4_next;
            a2_reg <= a2_next;
            b2_reg <= b2_next;
        end if;
    end process;

    -- stage 0
    bv0 <= (others=>b(0));
    bp0 <=unsigned("00000" & (bv0 and a));
    pp0 <= bp0;
    a0 <= a;
    b0 <= b;
    -- stage 1
    bv1 <= (others=>b0(1));
    bp1 <=unsigned("0000" & (bv1 and a0) & "0");
    pp1 <= pp0 + bp1;
    a1 <= a0;
    b1 <= b0;
    -- stage 2 (with buffer)
    bv2 <= (others=>b1(2));
    bp2 <=unsigned("000" & (bv2 and a1) & "00");
    pp2_next <= pp1 + bp2;
    a2_next <= a1;
    b2_next <= b1;
    -- stage 3
    bv3 <= (others=>b2_reg(3));
    bp3 <=unsigned("00" & (bv3 and a2_reg) & "000");
    pp3 <= pp2_reg + bp3;
    a3 <= a2_reg;
    b3 <= b2_reg;
    -- stage 4 (with buffer)
    bv4 <= (others=>b3(4));
    bp4 <=unsigned("0" & (bv4 and a3) & "0000");
    pp4_next <= pp3 + bp4;
    -- output
    y <= std_logic_vector(pp4_reg);
end two_stage_pipe_arch;
```

More efficient pipelined adder-based multiplier We can make some improvements of the initial pipelined design. First, as discussed in Section 7.5.4, we can use a smaller $(n+1)$-bit adder to replace the $2n$-bit adder in an n-bit multiplier. The same technique can be applied to the pipelined version. Second, we can reduce the size of the partial-product register. This is based on the observation that the valid LSBs of the partial products grow incrementally in each stage, from $n+1$ bits to $2n$ bits. There is no need to use a $2n$-bit register to carry the invalid MSBs in every stage. In the previous example, we can use a 5-bit register for the initial partial product (i.e., the pp0 signal), and increase the size by 1 in

each stage. Finally, we can reduce the size of the registers that hold the b signal. Note that only the b_i bit of b is needed to obtain the bit product at the ith stage. Once the calculation is done, the b_i bit can be discarded. Instead of using n-bit registers to carry b, we can drop one LSB bit in each stage and reduce the register size decrementally. In the previous example, we can drop the register bits for b_0 and b_1 in the first stage, the bits for b_2 in the second stage and so on. The VHDL code of the revised design is shown in Listing 9.22.

Listing 9.22 More efficient four-stage pipelined multiplier

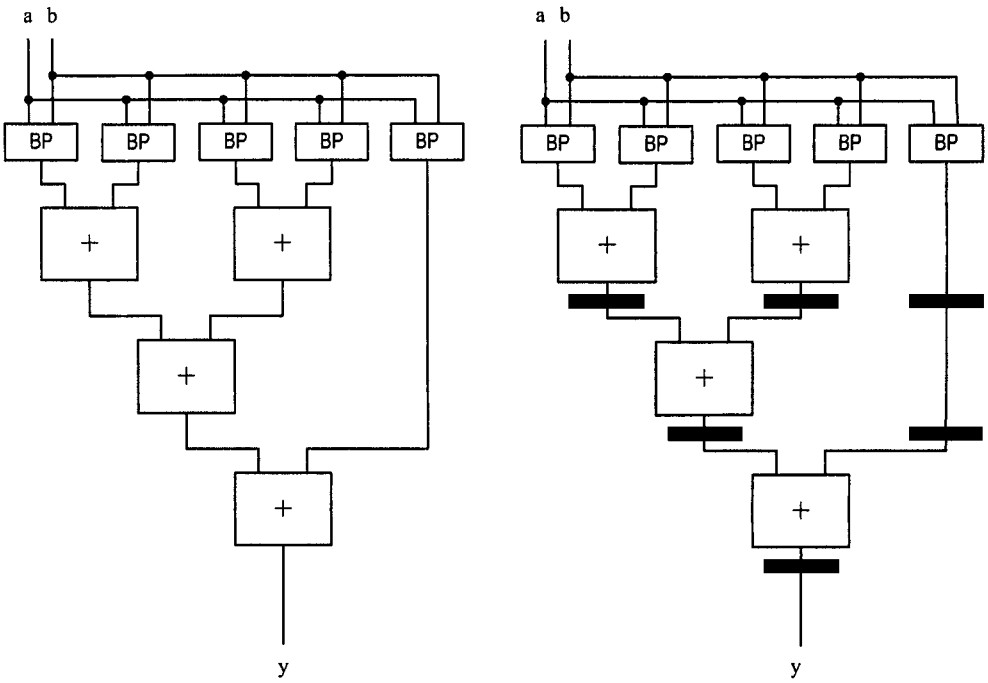
```
architecture effi_4_stage_pipe_arch of mult5 is
   signal a1_reg, a2_reg, a3_reg:
      std_logic_vector(4 downto 0);
   signal a0, a1_next, a2_next, a3_next:
      std_logic_vector(4 downto 0);
   signal b0: std_logic_vector(4 downto 1);
   signal b1_next, b1_reg: std_logic_vector(4 downto 2);
   signal b2_next, b2_reg: std_logic_vector(4 downto 3);
   signal b3_next, b3_reg: std_logic_vector(4 downto 4);
   signal bv0, bv1, bv2, bv3, bv4:
      std_logic_vector(4 downto 0);
   signal bp0, bp1, bp2, bp3, bp4: unsigned(5 downto 0);
   signal pp0: unsigned(5 downto 0);
   signal pp1_next, pp1_reg: unsigned(6 downto 0);
   signal pp2_next, pp2_reg: unsigned(7 downto 0);
   signal pp3_next, pp3_reg: unsigned(8 downto 0);
   signal pp4_next, pp4_reg: unsigned(9 downto 0);
begin
   -- pipeline registers (buffers)
   process(clk,reset)
   begin
      if (reset ='1') then
         pp1_reg <= (others=>'0');
         pp2_reg <= (others=>'0');
         pp3_reg <= (others=>'0');
         pp4_reg <= (others=>'0');
         a1_reg <= (others=>'0');
         a2_reg <= (others=>'0');
         a3_reg <= (others=>'0');
         b1_reg <= (others=>'0');
         b2_reg <= (others=>'0');
         b3_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         pp1_reg <= pp1_next;
         pp2_reg <= pp2_next;
         pp3_reg <= pp3_next;
         pp4_reg <= pp4_next;
         a1_reg <= a1_next;
         a2_reg <= a2_next;
         a3_reg <= a3_next;
         b1_reg <= b1_next;
         b2_reg <= b2_next;
         b3_reg <= b3_next;
      end if;
   end process;
```

```
    -- merged stage 0 & 1 for pipeline
    bv0 <= (others=>b(0));
    bp0 <=unsigned('0' & (bv0 and a));
    pp0 <= bp0;
    a0 <= a;
    b0 <= b(4 downto 1);
    --
    bv1 <= (others=>b0(1));
    bp1 <=unsigned('0' & (bv1 and a0));
    pp1_next(6 downto 1) <= ('0' & pp0(5 downto 1)) + bp1;
    pp1_next(0) <= pp0(0);
    a1_next <= a0;
    b1_next <= b0(4 downto 2);
    -- stage 2
    bv2 <= (others=>b1_reg(2));
    bp2 <=unsigned('0' & (bv2 and a1_reg));
    pp2_next(7 downto 2) <= ('0' & pp1_reg(6 downto 2)) + bp2;
    pp2_next(1 downto 0) <= pp1_reg(1 downto 0);
    a2_next <= a1_reg;
    b2_next <= b1_reg(4 downto 3);
    -- stage 3
    bv3 <= (others=>b2_reg(3));
    bp3 <=unsigned('0' & (bv3 and a2_reg));
    pp3_next(8 downto 3) <= ('0' & pp2_reg(7 downto 3)) + bp3;
    pp3_next(2 downto 0) <= pp2_reg(2 downto 0);
    a3_next <= a2_reg;
    b3_next(4) <= b2_reg(4);
    -- stage 4
    bv4 <= (others=>b3_reg(4));
    bp4 <=unsigned('0' & (bv4 and a3_reg));
    pp4_next(9 downto 4) <= ('0' & pp3_reg(8 downto 4)) + bp4;
    pp4_next(3 downto 0) <= pp3_reg(3 downto 0);
    -- output
    y <= std_logic_vector(pp4_reg);
end effi_4_stage_pipe_arch;
```

Tree-shaped pipelined multiplier Discussion in Section 7.5.4 shows that we can rearrange a cascading network to reduce the propagation delay. In an n-bit combinational multiplier, the critical path consists of $n-1$ adders in a cascading network. The critical path can be reduced to $\lceil \log_2 n \rceil$ adders when a tree-shaped network is used. The same scheme can be applied to the pipelined multiplier. The 5-bit tree-shaped combinational circuit is shown in Figure 9.21(a). The five bit products are first evaluated in parallel and then fed into the tree-shaped network. The pipelined version is shown in Figure 9.21(b). It is divided into three stages and the required registers are shown as dark bars. Note that one bit product has to be carried through two stages. The VHDL code is given in Listing 9.23.

Listing 9.23 Tree-shaped three-stage pipelined multiplier

```
architecture tree_pipe_arch of mult5 is
   constant WIDTH: integer:=5;
   signal bv0, bv1, bv2, bv3, bv4:
      std_logic_vector(WIDTH-1 downto 0);
```



(a). Non-pipelined design

(b). Pipelined design

Figure 9.21 Block diagrams of tree-shaped non-pipelined and pipelined multipliers.

```
    signal bp0, bp1, bp2, bp3, bp4:
       unsigned(2*WIDTH-1 downto 0);
    signal bp4_s1_reg, bp4_s2_reg:
       unsigned(2*WIDTH-1 downto 0);
    signal bp4_s1_next, bp4_s2_next:
       unsigned(2*WIDTH-1 downto 0);
    signal pp01_reg, pp23_reg, pp0123_reg, pp01234_reg:
       unsigned(2*WIDTH-1 downto 0);
    signal pp01_next, pp23_next, pp0123_next, pp01234_next:
       unsigned(2*WIDTH-1 downto 0);
begin
    -- pipeline registers (buffers)
    process(clk,reset)
    begin
       if (reset ='1') then
          pp01_reg <= (others=>'0');
          pp23_reg <= (others=>'0');
          pp0123_reg <= (others=>'0');
          pp01234_reg <= (others=>'0');
          bp4_s1_reg <= (others=>'0');
          bp4_s2_reg <= (others=>'0');
```

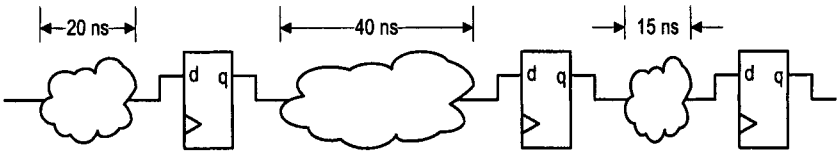
```
        elsif (clk'event and clk='1') then
           pp01_reg <= pp01_next;
           pp23_reg <= pp23_next;
           pp0123_reg <= pp0123_next;
           pp01234_reg <= pp01234_next;
           bp4_s1_reg <= bp4_s1_next;
           bp4_s2_reg <= bp4_s2_next;
        end if;
    end process;

    -- stage 1
    -- bit product
    bv0 <= (others=>b(0));
    bp0 <=unsigned("00000" & (bv0 and a));
    bv1 <= (others=>b(1));
    bp1 <=unsigned("0000" & (bv1 and a) & "0");
    bv2 <= (others=>b(2));
    bp2 <=unsigned("000" & (bv2 and a) & "00");
    bv3 <= (others=>b(3));
    bp3 <=unsigned("00" & (bv3 and a) & "000");
    bv4 <= (others=>b(4));
    bp4 <=unsigned("0" & (bv4 and a) & "0000");
    -- adder
    pp01_next <= bp0 + bp1;
    pp23_next <= bp2 + bp3;
    bp4_s1_next <= bp4;
    -- stage 2
    pp0123_next <= pp01_reg + pp23_reg;
    bp4_s2_next <= bp4_s1_reg;
    -- stage 3
    pp01234_next <= pp0123_reg + bp4_s2_reg;
    -- output
    y <= std_logic_vector(pp01234_reg);
end tree_pipe_arch;
```

In terms of performance, the delay in the tree-shaped multiplier is smaller since it has only three pipelined stages. The improvement will become more significant for a larger multiplier. On the other hand, the throughput of the two pipelined designs is similar because they have a similar clock rate. Both can generate a new multiplication result in each clock cycle.

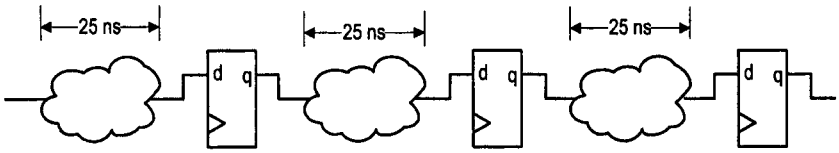
Although the division of the adder-based multiplier appears to be reasonable, it is not optimal. Examining the circuit in "finer granularity" can shed light about the data dependency on the internal structure and lead to a more efficient partition. This issue is discussed in Section 15.4.2.

9.4.4 Synthesis of pipelined circuits and retiming

The major step of adding pipeline to a combinational circuit is to divide the circuit into adequate stages. To achieve this goal, we must know the propagation delays of the relevant components. However, since the components will be transformed, merged and optimized during synthesis and wiring delays will be introduced during placement and routing, this information cannot easily be determined at the RT level.



(a) Pipelined circuit with uneven stages



(b) Rebalanced circuit

Figure 9.22 Example of circuit retiming.

Except for a highly regular structure, such as the previous adder-based multiplier example, partitioning a circuit into proper stages is difficult. We may need to synthesize major components and even some subsystems in advance to obtain rough estimations of the propagation delays, and then use this information to guide the division.

More sophisticated synthesis software can automate this task to some degree. It is known as *retiming*. For example, consider the three-stage pipelined circuit shown in Figure 9.22(a). The combinational circuits are shown as clouds with their propagation delays. The division of the original combinational circuit is not optimal and thus creates three uneven stages. In regular synthesis software, optimization can be done only for a combinational circuit, and thus the three combinational circuits of Figure 9.22(a) are processed independently. On the other hand, synthesis software with retiming capability can examine the overall circuit and move combinational circuits crossing the register boundaries. A rebalanced implementation is shown in Figure 9.22(b). This tool is especially useful if the combinational circuits are random and do not have an easily recognizable structure.

9.5 SYNTHESIS GUIDELINES

- Asynchronous reset, if used, should be only for system initialization. It should not be used to clear the registers during regular operation.
- Do not manipulate or gate the clock signal. Most desired operations can be achieved by using a register with an enable signal.
- LFSR is an effective way to construct a counter. It can be used when the counting patterns are not important.
- Throughput and delay are two performance criteria. Adding pipeline to a combinational circuit can increase the throughput but not reduce the delay.
- The main task of adding pipeline to a combinational circuit is to divide the circuit into balanced stages. Software with retiming capability can aid in this task.

9.6 BIBLIOGRAPHIC NOTES

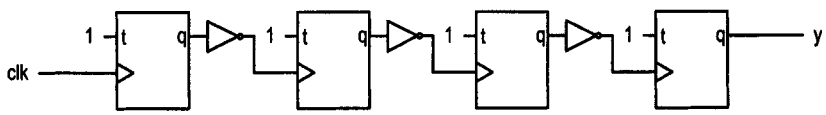
Although the implementation of an LFSR is simple, it has lots of interesting properties and a wide range of applications. The text, *Built In Test for VLSI: Pseudorandom Techniques* by Paul H. Bardell et al., has an in-depth discussion on the fundamentals and implementation of LFSRs. The application note of Xilinx, *Efficient Shift Registers, LFSR Counters, and Long Pseudorandom Sequence Generators*, includes a table that lists LFSR feedback expressions from 2 to 160 bits.

A pipeline is a commonly used technique to increase the performance of a processor. Its design is more involved because of the external data dependency. The text, *Computer Organization and Design: The Hardware/Software Interface, 3rd edition*, by David A. Patterson and John L. Hennessy, provides comprehensive coverage of this topic.

Problems

9.1 Consider the decade counter shown in Figure 9.1. Let T_{inc}, T_{comp} and T_{or} be the propagation delays of the incrementor, comparator and or cell, and T_{setup}, T_{cq} and T_{rq} be the setup time, clock-to-q delay and reset-to-q delay of the register. Determine the maximal clock rate of this counter.

9.2 Consider the following asynchronous counter constructed with T FFs:



(a) Draw the waveform for the clock and the output of four FFs.
(b) Describe the operation of this counter.
(c) Design a synchronous counter that performs the same task and derive the VHDL code accordingly.

9.3 For the 4-bit ring counter discussed in Section 9.2.2, the output of the 4 FFs appears to be out of phase. Let $T_{cq(0)}$ and $T_{cq(1)}$ be the clock-to-q delays when the q output of an FF becomes '0' and '1' respectively. Note that $T_{cq(0)}$ and $T_{cq(1)}$ may not always be equal. Perform a detailed timing analysis to determine whether a ring counter can produce true non-overlapping four-phase signals.

9.4 Design a 4-bit self-correction synchronous counter that circulates a single '0' (i.e., circulates the "1110" pattern).

9.5 Revise the design of the 4-bit LFSR in Section 9.2.3 to include the "0000" pattern but exclude the "1111" pattern.

9.6 Let the propagation delay of an xor cell be 4 ns, the propagation delay of an n-bit incrementor be $6n$ ns, and the setup time and clock-to-q delay of the register be 2 and 3 ns respectively.

(a) Determine the maximal operation rates of a 4-bit LFSR and a binary counter.
(b) Determine the maximal operation rates of an 8-bit LFSR and a binary counter.
(c) Determine the maximal operation rates of a 16-bit LFSR and a binary counter.
(d) Determine the maximal operation rates of a 64-bit LFSR and a binary counter.