CHAPTER 4

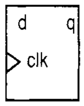
REGULAR SEQUENTIAL CIRCUIT

4.1 INTRODUCTION

A sequential circuit is a circuit with *memory*, which forms the *internal state* of the circuit. Unlike a combinational circuit, in which the output is a function of input only, the output of a sequential circuit is a function of the input and the internal state. The *synchronous design methodology* is the most commonly used practice in designing a sequential circuit. In this methodology, all storage elements are controlled (i.e., synchronized) by a global clock signal and the data is sampled and stored at the rising or falling edge of the clock signal. It allows designers to separate the storage components from the circuit and greatly simplifies the development process. This methodology is the most important principle in developing a large, complex digital system and is the foundation of most synthesis, verification, and testing algorithms. All of the designs in the book follow this methodology.

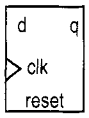
4.1.1 D FF and register

The most basic storage component in a sequential circuit is a D-type flip-flop (D FF). The symbol and function table of a positive edge-triggered D FF are shown in Figure 4.1(a). The value of the `d` signal is sampled at the rising edge of the `clk` signal and stored to FF. A D FF may contain an asynchronous reset signal to clear the FF to '0'. Its symbol and function table are shown in Figure 4.1(b). Note that the reset operation is independent of the clock signal.



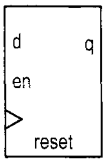
clk	q*
0	q
1	q
↑	d

(a) D FF



reset	clk	q*
1	-	0
0	0	q
0	1	q
0	↑	d

(b) D FF with asynchronous reset



reset	clk	en	q*
1	-	-	0
0	0	-	q
0	1	-	q
0	↑	0	q
0	↑	1	d

(c) D FF with synchronous enable

Figure 4.1 Block diagram and functional table of a D FF.

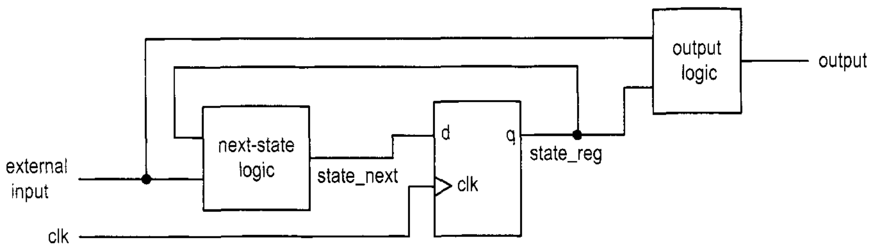


Figure 4.2 Block diagram of a synchronous system.

The three main timing parameters of a D FF are T_{cq} (clock-to-q delay), T_{setup} (setup time), and T_{hold} (hold time). T_{cq} is the time required to propagate the value of d to q at the rising edge of the clock signal. The d signal must be stable around the sampling edge to prevent the FF from entering the metastable state. T_{setup} and T_{hold} specify the time intervals before or after the sampling edge.

A D FF provides 1-bit storage. A collection of D FFs can be grouped together to store multiple bits and is known as a *register*.

4.1.2 Synchronous system

Block diagram The block diagram of a synchronous system is shown in Figure 4.2. It consists of the following parts:

- *State register*: a collection of D FFs controlled by the same clock signal

- *Next-state logic*: combinational logic that uses the external input and internal state (i.e., the output of register) to determine the new value of the register
- *Output logic*: combinational logic that generates the output signal

Maximal operating frequency One of the most difficult design aspects of a sequential circuit is to ensure that the system timing does not violate the setup and hold time constraints. In a synchronous system, the storage components are grouped together and treated as a single register, as shown in Figure 4.2. We need to perform timing analysis on only one memory component.

The timing of a sequential circuit is characterized by f_{max}, *the maximal clock frequency*, which specifies how fast the circuit can operate. The reciprocal of f_{max} specifies T_{clock}, the minimal clock period, which can be interpreted as the interval between two sampling edges of the clock. To ensure correct operation, the next value must be generated and stabilized within this interval. Assume that the maximal propagation delay of next-state logic is T_{comb}. The minimal clock period can be obtained by adding the propagation delays and setup time constraint of the closed loop in Figure 4.2:

$$T_{clock} = T_{cq} + T_{comb} + T_{setup}$$

and the maximal clock rate is the reciprocal:

$$f_{max} = \frac{1}{T_{clock}} = \frac{1}{T_{cq} + T_{comb} + T_{setup}}$$

Timing constraint in Xilinx ISE$^{Xilinx\ specific}$ During synthesis, Xilinx software will analyze the synthesized circuit and show f_{max} in a report. We can also specify the desired operating frequency as a synthesis constraint, and the synthesis software will try to obtain a circuit to satisfy this requirement (i.e., a circuit whose f_{max} is equal to or greater than the desired operating frequency). For example, if we use the 50-MHz (i.e., 20-ns period) oscillator on the prototyping board as the clock source, f_{max} of a sequential circuit must exceed this frequency (i.e., the period must be smaller than 20 ns). The following lines can be added to the constraint file:

```
NET "clk" TNM_NET = "clk";
TIMESPEC "TS_clk" = PERIOD "clk" 20 ns HIGH 50 %;
```

This indicates that the `clk` signal has a maximal period of 20 ns (i.e., 50 MHz) and a duty cycle of 50%.

After synthesis, we can check the relevant timing information by invoking the View Design Summary process from the ISE's Processes window. The Timing Constraints section shows whether the imposed constraints are met, and the Static Timing Report section provides more detailed timing information.

4.1.3 Code development

Our code development follows the basic block diagram in Figure 4.2. The key is to separate the memory component (i.e., the register) from the system. Once the register is isolated, the remaining portion is a pure combinational circuit, and the coding and analysis schemes discussed in previous chapters can be applied accordingly. While this approach may make the code a little bit more cumbersome at times, it helps us to better visualize the circuit architecture and avoid unintended memory and subtle mistakes.

Based on the characteristics of the next-state logic, we divide sequential circuits into three categories:

- *Regular sequential circuit.* The state transitions in the circuit exhibit a "regular" pattern, as in a counter or shift register. The next-state logic is constructed primarily by a predesigned, "regular" component, such as an incrementor or shifter.
- *FSM.* The state transitions in the circuit do not exhibit a simple, repetitive pattern. The next-state logic is constructed by "random logic" and synthesized from scratch. It should be called a random sequential circuit, but is commonly known as an FSM (*finite state machine*).
- *FSMD.* The circuit consists of a regular sequential circuit and an FSM. The two parts are known as a *data path* and a *control path*, and the complete circuit is known as an FSMD (*FSM with data path*). This type of circuit is used to implement an algorithm represented by *register-transfer* (RT) methodology, which describes system operation by a sequence of data transfers and manipulations among registers.

The three types of circuits are discussed in this and two subsequent chapters.

4.2 HDL CODE OF THE FF AND REGISTER

Describing storage components in HDL is a subtle procedure, and there are many ways to do it. In fact, one common problem encountered by a new HDL user is the inference of unintended latches and buffers. Instead of covering all possible forms of syntactic descriptions, we introduce the code segments for several commonly used memory components. Since our development process separates the register and the combinational circuit, these components are sufficient for all designs in this book. The components are:

- D FF
- Register
- Register file

4.2.1 D FF

We consider three types of D FFs:

- D FF without asynchronous reset
- D FF with asynchronous reset
- D FF with synchronous enable

The first two are the most basic memory components and can be found in the library of any device technology. The third can be constructed from a simple D FF. We include the code since it is a frequently used memory component and can be mapped to the FF of the Spartan-3 device's logic cell.

D FF without asynchronous reset The function table of a D FF is shown in Figure 4.1(a) and the code is shown in Listing 4.1.

Listing 4.1 D FF without asynchronous reset

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

architecture arch of d_ff is
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

The rising edge is checked by the `clk'event` **and** `clk='1'` expression, which represents that there is a change in the `clk` signal (i.e., an "event") and the new value is '1'. If this condition is `true`, the value of d is stored to q, and if this condition is `false`, q keeps its previous value (i.e., memorizes the value sampled earlier). Note that only the `clk` signal is included in the sensitive list. This is consistent with the fact that the d signal is sampled only at the rising edge of the `clk` signal, and change in its value does not trigger any immediate response.

D FF with asynchronous reset A D FF may contain an asynchronous reset signal, as shown in the function table of Figure 4.1(b). The signal clears the D FF to '0' any time and is not controlled by the clock signal. It actually has a higher priority than the regularly sampled input. Using an asynchronous reset signal violates the synchronous design methodology and thus should be avoided in normal operation. Its major application is to perform system initialization. For example, we can generate a short reset pulse to force a system to an initial state after turning on the power. The code for a D FF with asynchronous reset is shown in Listing 4.2.

Listing 4.2 D FF with asynchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_reset is
   port(
      clk, reset: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_ff_reset;

architecture arch of d_ff_reset is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

Note that the `reset` signal is included in the sensitivity list, and its condition is checked before the rising-edge condition.

D FF with synchronous enable A D FF may include an additional control signal, en, to enable the FF to sample the input value. Its symbol and functional table are shown in Figure 4.1(c). Note that the en signal is examined only at the rising edge of the clock and thus is synchronous. If it is not asserted, the FF keeps its previous value. The code is shown in Listing 4.3.

Listing 4.3 One-process coding style for a D FF with synchronous enable

```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_en is
   port(
      clk, reset: in std_logic;
      en: in std_logic;
      d: in std_logic;
      q: out std_logic
   );
end d_ff_en;

architecture arch of d_ff_en is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <='0';
      elsif (clk'event and clk='1') then
         if (en='1') then
            q <= d;
         end if;
      end if;
   end process;
end arch;
```

The enabling feature of this D FF is useful in maintaining synchronism between a fast subsystem and a slow subsystem. For example, assume that the operation rates of a fast and a slow subsystem are 50 MHz and 1 MHz. Instead of using a derived 1-MHz clock to drive the slow subsystem, we can generate a periodic enable tick that is asserted one clock cycle every 50 clock cycles. The slow subsystem is disabled (i.e., keep the previous state) for the remaining 49 clock cycles. The same scheme can also be applied to eliminate a gated clock signal.

Since the enable signal is synchronous, this circuit can be constructed by a regular D FF and simple next-state logic. The code is shown in Listing 4.4, and its block diagram is shown in Figure 4.3.

Listing 4.4 Two-segment coding style for a D FF with synchronous enable

```
architecture two_seg_arch of d_ff_en is
   signal r_reg, r_next: std_logic;
```

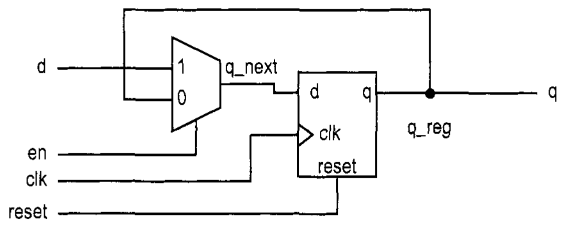


Figure 4.3 D FF with synchronous enable.

```
begin
   -- D FF
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <='0';
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= d when en ='1' else
             r_reg;
   -- output logic
   q <= r_reg;
end two_seg_arch;
```

For clarity, we use suffixes _next and _reg to emphasize the next input value and the registered output of an FF. They are connected to the d and q signals of a D FF. The earlier one-process code can be considered as shorthand for this more explicit description.

4.2.2 Register

A register is a collection of D FFs that are controlled by the same clock and reset signals. Like a D FF, a register can have an optional asynchronous reset signal and a synchronous enable signal. The code is identical to that of a D FF except that the array data type, std_logic_vector, is needed for the relevant input and output signals. For example, an 8-bit register with asynchronous reset is shown in Listing 4.5.

Listing 4.5 Register

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_reset is
   port(
      clk, reset: in std_logic;
      d: in std_logic_vector(7 downto 0);
      q: out std_logic_vector(7 downto 0)
   );
end reg_reset;

```

```
architecture arch of reg_reset is
begin
   process(clk,reset)
   begin
      if (reset='1') then
         q <=(others=>'0');
      elsif (clk'event and clk='1') then
         q <= d;
      end if;
   end process;
end arch;
```

Note that the expression (**others**=>'0') means that all elements are assigned to '0' and is equivalent to "00000000" in this case.

4.2.3 Register file

A register file is a collection of registers with one input port and one or more output ports. The write address signal, w_addr, specifies where to store data, and the read address signal, r_addr, specifies where to retrieve data. The register file is generally used as fast, temporary storage. The code for a parameterized 2^W-by-B register file is shown in Listing 4.6. Two generics are defined in this design. The W generic specifies the number of address bits, which implies that there are 2^W words in the file, and the B generic specifies the number of bits in a word.

Listing 4.6 Parameterized register file

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity reg_file is
   generic(
      B: integer:=8; -- number of bits
      W: integer:=2  -- number of address bits
   );
   port(
      clk, reset: in std_logic;
      wr_en: in std_logic;
      w_addr, r_addr: in std_logic_vector (W-1 downto 0);
      w_data: in std_logic_vector (B-1 downto 0);
      r_data: out std_logic_vector (B-1 downto 0)
   );
end reg_file;

architecture arch of reg_file is
   type reg_file_type is array (2**W-1 downto 0) of
        std_logic_vector(B-1 downto 0);
   signal array_reg: reg_file_type;
begin
   process(clk,reset)
   begin
      if (reset='1') then
         array_reg <= (others=>(others=>'0'));
      elsif (clk'event and clk='1') then
         if wr_en='1' then
            array_reg(to_integer(unsigned(w_addr))) <= w_data;
         end if;
      end if;
   end process;
   -- read port
   r_data <= array_reg(to_integer(unsigned(r_addr)));
end arch;
```

The code includes several new features. First, since no built-in two-dimensional array is defined in the `std_logic_1164` package a user-defined array-of-array data type, `reg_file_type`, is introduced. It is first defined by a type statement and is then used by the `array_reg` signal. Second, a signal is used as an index to access an element in the array, as in `array_reg(..w_addr..)`. Although the description is very abstract, Xilinx software recognizes this language construct and can derive the correct implementation accordingly. The `array_reg(...) <= ...` and `... <= array_reg(...)` statements infer decoding and multiplexing logic, respectively.

Some applications may need to retrieve multiple data words at the same time. This can be done by adding an additional read port:

```
r_data2 <= array_reg(to_integer(unsigned(r_addr_2)));
```

4.2.4 Storage components in a Spartan-3 device$^{Xilinx\ specific}$

In a Spartan-3 device, each logic cell contains a D FF with asynchronous reset and synchronous enable. These D FFs basically constitute the register of Figure 4.2. Since a logic cell also contains a four-input LUT, it will be wasteful if the cell is just used simply as 1 bit of a massive storage. The Spartan-3 device also has distributed RAM (random access memory) and block RAM modules, and they can be used for larger storage requirements. These modules can be configured for synchronous operation, and their characteristics are somewhat like a restricted version of the register file. The configuration and inference of these modules are discussed in Chapter 11.

4.3 SIMPLE DESIGN EXAMPLES

We illustrate the construction of several simple, representative sequential circuits in this section.

4.3.1 Shift register

Free-running shift register A free-running shift register shifts its content to the left or right by one position in each clock cycle. There is no other control signal. The code for an N-bit free-running shift-right register is shown in Listing 4.7.

Listing 4.7 Free-running shift register

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

```
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      s_in: in std_logic;
      s_out: out std_logic
   );
end free_run_shift_reg;

architecture arch of free_run_shift_reg is
   signal r_reg: std_logic_vector(N-1 downto 0);
   signal r_next: std_logic_vector(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic (shift right 1 bit)
   r_next <= s_in & r_reg(N-1 downto 1);
   -- output
   s_out <= r_reg(0);
end arch;
```

The next-state logic is a 1-bit shifter, which shifts r_reg right one position and inserts the serial input, s_in, to the MSB. Since the 1-bit shifter involves only reconnection of the input and output signals, no real logic is needed. Its propagation delay represents the smallest possible T_{comb}, and the corresponding f_{max} represents the highest clock rate that can be achieved for a given device technology.

Universal shift register A universal shift register can load parallel data, shift its content left or right, or remain in the same state. It can perform parallel-to-serial operation (first loading parallel input and then shifting) or serial-to-parallel operation (first shifting and then retrieving parallel output). The desired operation is specified by a 2-bit control signal, ctrl. The code is shown in Listing 4.8.

Listing 4.8 Universal shift register

```
library ieee;
use ieee.std_logic_1164.all;
entity univ_shift_reg is
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      ctrl: in std_logic_vector(1 downto 0);
      d: in std_logic_vector(N-1 downto 0);
      q: out std_logic_vector(N-1 downto 0)
   );
end univ_shift_reg;

architecture arch of univ_shift_reg is
```

```
   signal r_reg: std_logic_vector(N-1 downto 0);
   signal r_next: std_logic_vector(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   with ctrl select
    r_next <=
      r_reg                       when "00", --no op
      r_reg(N-2 downto 0) & d(0)  when "01", --shift left;
      d(N-1) & r_reg(N-1 downto 1) when "10", --shift right;
      d                           when others; -- load
   -- output
   q <= r_reg;
end arch;
```

The next-state logic uses a 4-to-1 multiplexer to select the desired next value of the register. Note that the LSB and MSB of d (i.e., d(0) and d(N-1)) are used as serial input for the shift-left and shift-right operations.

In a Xilinx Spartan-3 device, a logic cell's 4-input LUT is implemented by a 16-by-1 SRAM. The same SRAM can also be configured as a cascading chain of sixteen 1-bit SRAM cells, which resembles a 16-bit shift register. This can be used to construct certain forms of shift register and leads to very efficient implementation. **Xilinx specific**

4.3.2 Binary counter and variant

Free-running binary counter A free-running binary counter circulates through a binary sequence repeatedly. For example, a 4-bit binary counter counts from "0000", "0001", ..., to "1111" and wraps around. The code for a parameterized N-bit free-running binary counter is shown in Listing 4.9.

Listing 4.9 Free-running binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity free_run_bin_counter is
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      max_tick: out std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end free_run_bin_counter;

architecture arch of free_run_bin_counter is
```

Table 4.1 Function table of a universal binary counter

syn_clr	load	en	up	q*	**Operation**
1	–	–	–	00···00	synchronous clear
0	1	–	–	d	parallel load
0	0	1	1	q+1	count up
0	0	1	0	q-1	count down
0	0	0	–	q	pause

```
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_tick <= '1' when r_reg=(2**N-1) else '0';
end arch;
```

The next-state logic is an incrementor, which adds 1 to the register's current value. By definition of the + operator in the IEEE numeric_std package, the operation implicitly wraps around after the r_reg reaches "1...1". The circuit also consists of an output status signal, max_tick, which is asserted when the counter reaches the maximal value, "1...1" (which is equal to $2^N - 1$).

The max_tick signal represents a special type of signal that is asserted for a single clock cycle. In this book, we call this type of signal a *tick* and use the suffix _tick to indicate a signal with this property. It is commonly used to interface with the enable signal of other sequential circuits.

Universal binary counter A universal binary counter is more versatile. It can count up or down, pause, be loaded with a specific value, or be synchronously cleared. Its functions are summarized in Table 4.1. Note the difference between the reset and syn_clr signals. The former is asynchronous and should only be used for system initialization. The latter is sampled at the rising edge of the clock and can be used in normal synchronous design. The code for this counter is shown in Listing 4.10.

Listing 4.10 Universal binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity univ_bin_counter is
```

```
   generic(N: integer := 8);
   port(
      clk, reset: in std_logic;
      syn_clr, load, en, up: in std_logic;
      d: in std_logic_vector(N-1 downto 0);
      max_tick, min_tick: out std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end univ_bin_counter;

architecture arch of univ_bin_counter is
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= (others=>'0') when syn_clr='1' else
             unsigned(d)   when load='1' else
             r_reg + 1     when en ='1' and up='1' else
             r_reg - 1     when en ='1' and up='0' else
             r_reg;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_tick <= '1' when r_reg=(2**N-1) else '0';
   min_tick <= '1' when r_reg=0 else '0';
end arch;
```

The next-state logic follows the function table and uses a conditional signal assignment to prioritize the desired operations.

Mod-m counter A mod-m counter counts from 0 to $m-1$ and wraps around. A parameterized mod-m counter is shown in Listing 4.11. It has two generics. One is M, which specifies the limit, m, and the other is N, which specifies the number of bits needed and should be equal to $\lceil \log_2 M \rceil$. The code is shown in Listing 4.11, and the default value is for a mod-10 counter.

Listing 4.11 Mod-m counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity mod_m_counter is
   generic(
      N: integer := 4;     -- number of bits
      M: integer := 10     -- mod-M
   );
   port(
      clk, reset: in std_logic;
      max_tick: out std_logic;
      q: out std_logic_vector(N-1 downto 0)
   );
end mod_m_counter;

architecture arch of mod_m_counter is
   signal r_reg: unsigned(N-1 downto 0);
   signal r_next: unsigned(N-1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if (reset='1') then
         r_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         r_reg <= r_next;
      end if;
   end process;
   -- next-state logic
   r_next <= (others=>'0') when r_reg=(M-1) else
             r_reg + 1;
   -- output logic
   q <= std_logic_vector(r_reg);
   max_tick <= '1' when r_reg=(M-1) else '0';
end arch;
```

The next-state logic is constructed by a conditional signal assignment statement. If the counter reaches M-1, the new value is cleared to 0. Otherwise, it is incremented by 1.

Inclusion of the N parameter in the code is somewhat redundant since its value depends on M. A more elegant way is to define a function that calculates N from M automatically. In VHDL, this can be done by creating a user-defined *function* in a *package* and invoking the package before the entity declaration. This is beyond the scope of this book and the details may be found in the references cited in the Bibliographic section.

4.4 TESTBENCH FOR SEQUENTIAL CIRCUITS

A testbench is a program that mimics a physical lab bench, as discussed in Section 1.4. Developing a comprehensive testbench is beyond the scope of this book. We discuss a simple testbench for the previous universal binary counter in this section. It can serve as a template for other sequential circuits. The code for the testbench is shown in Listing 4.12.

Listing 4.12 Testbench for a universal binary counter

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

entity bin_counter_tb is
end bin_counter_tb;

architecture arch of bin_counter_tb is
```

```
   constant THREE: integer := 3;
   constant T: time := 20 ns; -- clk period
   signal clk, reset: std_logic;
   signal syn_clr, load, en, up: std_logic;
   signal d: std_logic_vector(THREE-1 downto 0);
   signal max_tick, min_tick: std_logic;
   signal q: std_logic_vector(THREE-1 downto 0);
begin
   --************************
   -- instantiation
   --************************
   counter_unit: entity work.univ_bin_counter(arch)
      generic map(N=>THREE)
      port map(clk=>clk, reset=>reset, syn_clr=>syn_clr,
               load=>load, en=>en, up=>up, d=>d,
               max_tick=>max_tick, min_tick=>min_tick, q=>q);

   --************************
   -- clock
   --************************
   -- 20 ns clock running forever
   process
   begin
      clk <= '0';
      wait for T/2;
      clk <= '1';
      wait for T/2;
   end process;
   --************************
   -- reset
   --************************
   -- reset asserted for T/2
   reset <= '1', '0' after T/2;

   --************************
   -- other stimulus
   --************************
   process
   begin
      --************************
      -- initial input
      --************************
      syn_clr <= '0';
      load <= '0';
      en <= '0';
      up <= '1';  -- count up
      d <= (others=>'0');
      wait until falling_edge(clk);
      wait until falling_edge(clk);
      --************************
      -- test load
      --************************
      load <= '1';
```

```
d <= "011";
wait until falling_edge(clk);
load <= '0';
-- pause 2 clocks
wait until falling_edge(clk);
wait until falling_edge(clk);
--************************
-- test syn_clear
--************************
syn_clr <= '1';  -- clear
wait until falling_edge(clk);
syn_clr <= '0';
--************************
-- test up counter and pause
--************************
en <= '1'; -- count
up <= '1';
for i in 1 to 10 loop -- count 10 clocks
   wait until falling_edge(clk);
end loop;
en <='0';
wait until falling_edge(clk);
wait until falling_edge(clk);
en <='1';
wait until falling_edge(clk);
wait until falling_edge(clk);
--************************
-- test down counter
--************************
up <= '0';
for i in 1 to 10 loop -- run 10 clocks
   wait until falling_edge(clk);
end loop;
--************************
-- other wait conditions
--************************
-- continue until q=2
wait until q="010";
wait until falling_edge(clk);
up <= '1';
-- continue until min_tick changes value
wait on min_tick;
wait until falling_edge(clk);
up <= '0';
wait for 4*T;  -- wait for 80 ns
en <= '0';
wait for 4*T;
--************************
-- terminate simulation
--************************
assert false
   report "Simulation Completed"
 severity failure;
```

```
   end process ;
end arch;
```

The code consists of a component instantiation statement, which creates an instance of a 3-bit counter, and three segments, which generate a stimulus for clock, reset, and regular inputs. Since operation of a synchronous system is synchronized by a clock signal, we define a constant with the built-in data type `time` for the clock period:

```
constant T: time := 20 ns; -- clk period
```

The clock generation is specified by a process:

```
process
begin
   clk <= '0';
   wait for T/2;
   clk <= '1';
   wait for T/2;
end process;
```

The `clk` signal is assigned between '0' and '1' alternatively, and each value lasts for half a period. Note that the process has no sensitivity list and repeats itself forever.

The reset stimulus involves one statement,

```
reset <= '1', '0' after T/2;
```

It indicates that the `reset` signal is set to '1' initially and changed to '0' after half a period. The statement represents the "power-on" condition, in which the `reset` signal is asserted momentarily to clear the system to the initial state. Note that, by default, the `'U'` value (for uninitialized), not `'0'`, is assigned to a signal with the `std_logic` type. Using a short reset pulse is a good mechanism to perform system initialization.

The last process statement generates a stimulus for other input signals. We first test the load and clear operations and then exercise counting in both directions. The final **assert** `false` statement forces the simulator to terminate simulation, as discussed in Section 2.7.

For a synchronous system with positive edge-triggered FFs, an input signal must be stable around the rising edge of the clock signal to satisfy the setup and hold time constraints. One easy way to achieve this is to change an input signal's value during the '1'-to-'0' transition of the `clk` signal. The `falling_edge` function of the `std_logic_1164` package checks this condition, and we can use it in a wait statement:

```
wait until falling_edge(clk);
```

Note that each statement represents a new falling edge, which corresponds to the advancement of one clock cycle. In our template, we generally use this statement to specify the progress of time. For multiple clock cycles, we can use a loop statement:

```
for i in 1 to 10 loop -- count 10 clocks
   wait until falling_edge(clk);
end loop;
```

There are other useful forms of wait statements, as shown at the end of the process. We can wait until a special condition, such as "when `q` is equal to 2",

```
wait until q="010";
```

or wait until a signal changes, such as

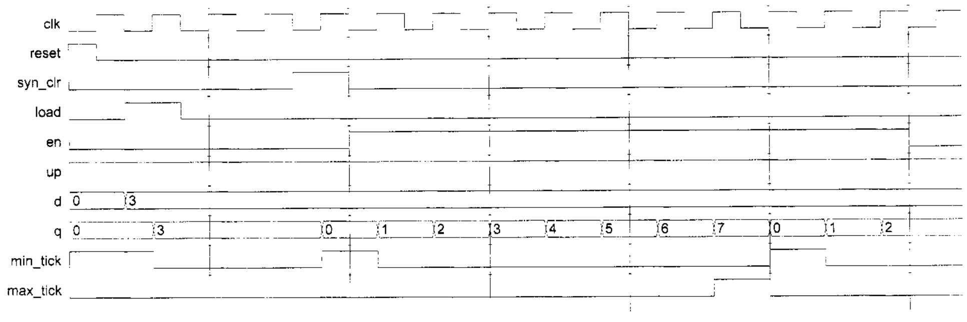


Figure 4.4 Testbench waveform.

```
wait on min_tick;
```

or wait for an absolute time, such as

```
wait for 4*T;  -- wait for 4 clock periods
```

If an input signal is modified after these statements, we need to make sure that the input change does not occur at the rising edge of the clock. An additional

```
wait until falling_edge(clk);
```

statement should be added when needed.

We can compile the code and perform simulation. Part of the simulated waveform is shown in Figure 4.4.

4.5 CASE STUDY

After examining several simple circuits, we discuss the design of more sophisticated examples in this section.

4.5.1 LED time-multiplexing circuit

The S3 board has four seven-segment LED displays, each containing seven bars and one small round dot. To reduce the use of FPGA's I/O pins, the S3 board uses a time-multiplexing sharing scheme. In this scheme, the four displays have their individual enable signals but share eight common signals to light the segments. All signals are active-low (i.e., enabled when a signal is '0'). The schematic of displaying '3' on the rightmost LED is shown in Figure 4.5. Note that the enable signal (i.e., an) is "1110". This configuration clearly can enable only one display at a time. We can *time-multiplex* the four LED patterns by enabling the four displays in turn, as shown in the simplified timing diagram in Figure 4.6. If the refreshing rate of the enable signal is fast enough, the human eye cannot distinguish the on and off intervals of the LEDs and perceives that all four displays are lit simultaneously. This scheme reduces the number of I/O pins from 32 to 12 (i.e., eight LED segments plus four enable signals) but requires a time-multiplexing circuit. Two variations of the circuit are discussed in the following subsections.

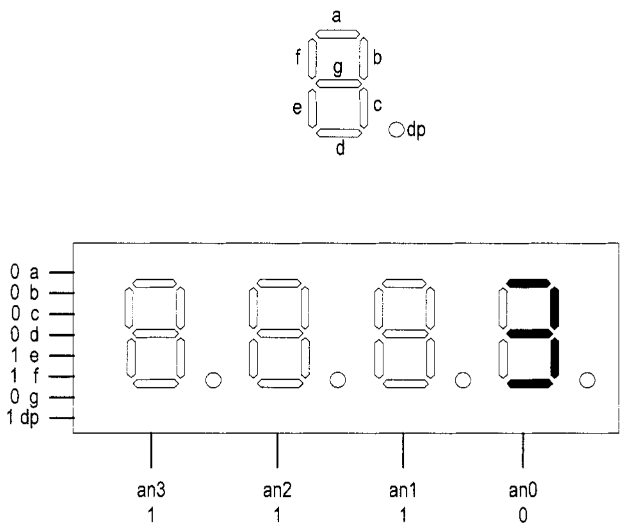


Figure 4.5 Time-multiplexed seven-segment LED display.

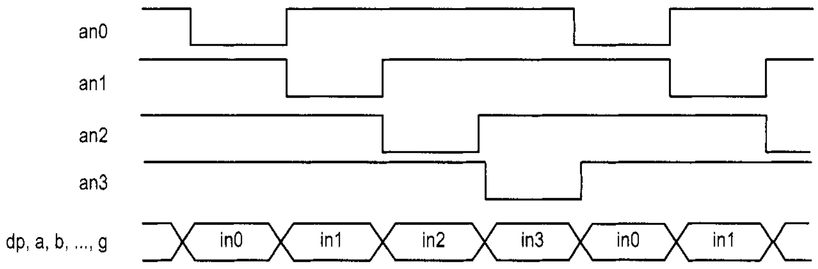
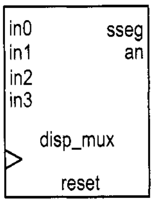
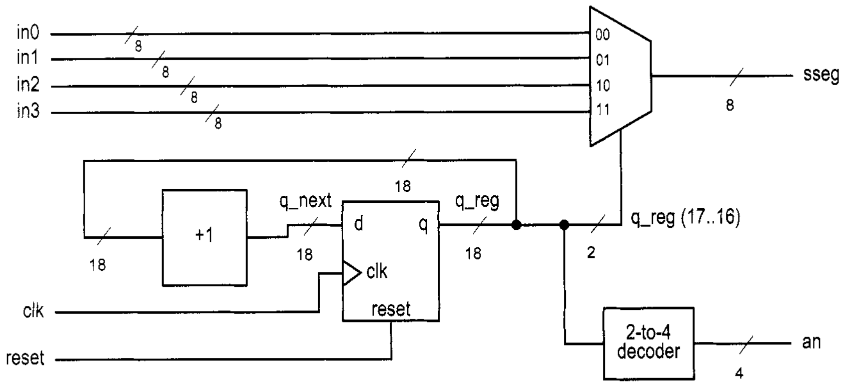


Figure 4.6 Timing diagram of a time-multiplexed seven-segment LED display.



(a) Symbol



(b) Block diagram

Figure 4.7 Symbol and block diagram of a time-multiplexing circuit.

Time multiplexing with LED patterns The symbol and block diagram of the time-multiplexing circuit are shown in Figure 4.7. It takes four seven-segment LED patterns, in3, in2, in1, and in0, and passes them to the output, sseg, in accordance with the enable signal.

The refresh rate of the enable signal has to be fast enough to fool our eyes but should be slow enough so that the LEDs can be turned on and off completely. The rate around the range 1000 Hz should work properly. In our design, we use an 18-bit binary counter for this purpose. The two MSBs are decoded to generate the enable signal and are used as the selection signal for multiplexing. The refreshing rate of an individual bit, such as an(0), becomes $\frac{50M}{2^{16}}$Hz, which is about 800 Hz. The code is shown in Listing 4.13.

Listing 4.13 LED time-multiplexing circuit with LED patterns

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity disp_mux is
   port(
      clk, reset: in std_logic;
      in3, in2, in1, in0: in std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end disp_mux ;
```

```
architecture arch of disp_mux is
   -- refreshing rate around 800 Hz (50MHz/2^16)
   constant N: integer:=18;
   signal q_reg, q_next: unsigned(N-1 downto 0);
   signal sel: std_logic_vector(1 downto 0);
begin
   -- register
   process(clk,reset)
   begin
      if reset='1' then
         q_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         q_reg <= q_next;
      end if;
   end process;

   -- next-state logic for the counter
   q_next <= q_reg + 1;

   -- 2 MSBs of counter to control 4-to-1 multiplexing
   -- and to generate active-low enable signal
   sel <= std_logic_vector(q_reg(N-1 downto N-2));
   process(sel,in0,in1,in2,in3)
   begin
      case sel is
         when "00" =>
            an <= "1110";
            sseg <= in0;
         when "01" =>
            an <= "1101";
            sseg <= in1;
         when "10" =>
            an <= "1011";
            sseg <= in2;
         when others =>
            an <= "0111";
            sseg <= in3;
      end case;
   end process;
end arch;
```

We use the testing circuit in Figure 4.8 to verify operation of the LED time-multiplexing circuit. It uses four 8-bit registers to store the LED patterns. The registers use the same 8-bit switch as input but are controlled by individual enable signal. When we press a button, the corresponding register is enabled and the switch pattern is loaded to that register. The code is shown in Listing 4.14.

Listing 4.14 Testing circuit for time multiplexing with LED patterns

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

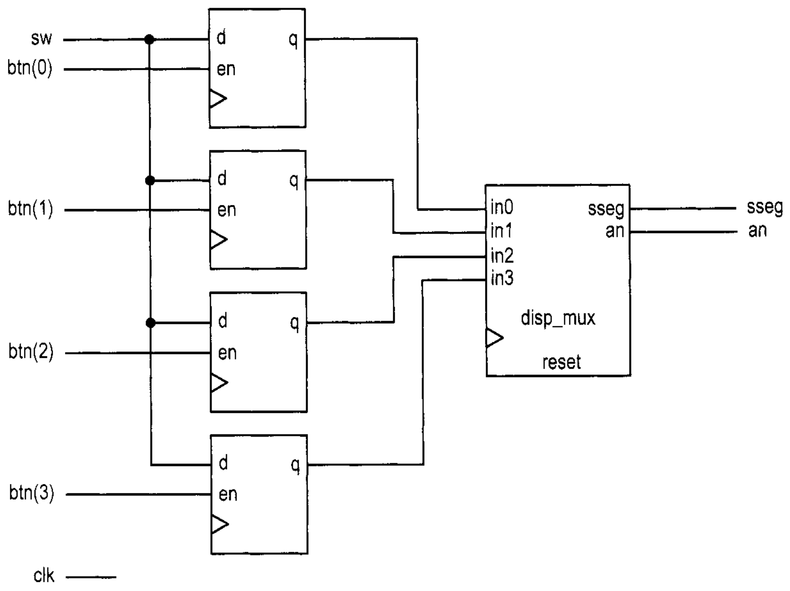


Figure 4.8 LED time-multiplexing testing circuit.

```
      clk: in std_logic;
      btn: in std_logic_vector(3 downto 0);
      sw: in std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end disp_mux_test;

architecture arch of disp_mux_test is
   signal d3_reg, d2_reg: std_logic_vector(7 downto 0);
   signal d1_reg, d0_reg: std_logic_vector(7 downto 0);
begin
   disp_unit: entity work.disp_mux
      port map(
         clk=>clk, reset=>'0',
         in3=>d3_reg, in2=>d2_reg, in1=>d1_reg,
         in0=>d0_reg, an=>an, sseg=>sseg);
   -- registers for 4 led patterns
   process (clk)
   begin
      if (clk'event and clk='1') then
         if (btn(3)='1') then
            d3_reg <= sw;
         end if;
         if (btn(2)='1') then
            d2_reg <= sw;
         end if;
         if (btn(1)='1') then
            d1_reg <= sw;
```

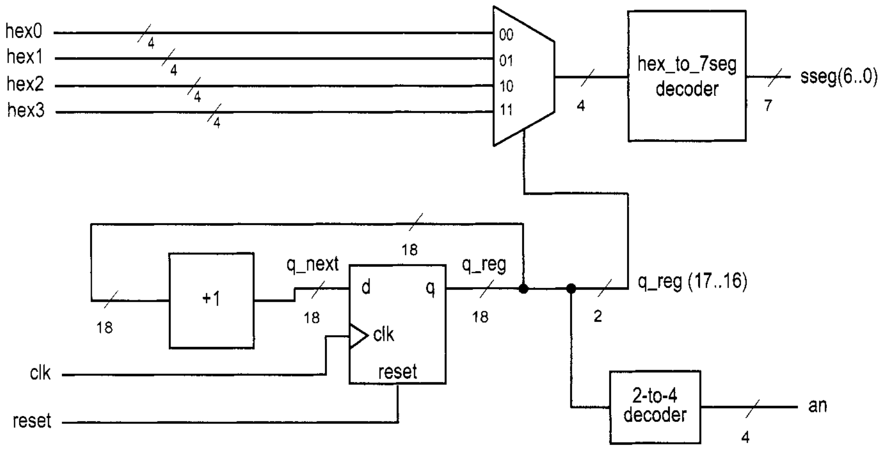


Figure 4.9 Block diagram of a hexadecimal time-multiplexing circuit.

```
         end if;
         if (btn(0)='1') then
            d0_reg <= sw;
         end if;
      end if;
   end process;
end arch;
```

Time multiplexing with hexadecimal digits The most common application of a seven-segment LED is to display a hexadecimal digit. The decoding circuit is discussed in Section 3.7.1. To display four hexadecimal digits with the previous time-multiplexing circuit, four decoding circuits are needed. A better alternative is first to multiplex the hexadecimal digits and then decode the result, as shown in Figure 4.9.

This scheme requires only one decoding circuit and reduces the width of the 4-to-1 multiplexer from 8 bits to 5 bits (i.e., 4 bits for the hexadecimal digit and 1 bit for the decimal point). The code is shown in Listing 4.15. In addition to clock and reset, the input consists of four 4-bit hexadecimal digits, hex3, hex2, hex1, and hex0, and four decimal points, which are grouped as one signal, dp_in.

Listing 4.15 LED time-multiplexing circuit with hexadecimal digits

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity disp_hex_mux is
   port(
      clk, reset: in std_logic;
      hex3, hex2, hex1, hex0: in std_logic_vector(3 downto 0);
      dp_in: in std_logic_vector(3 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end disp_hex_mux ;
```

```
architecture arch of disp_hex_mux is
   -- each 7-seg led enabled (2^18/4)*25 ns (40 ms)
   constant N: integer:=18;
   signal q_reg, q_next: unsigned(N-1 downto 0);
   signal sel: std_logic_vector(1 downto 0);
   signal hex: std_logic_vector(3 downto 0);
   signal dp: std_logic;
begin
   -- register
   process(clk,reset)
   begin
      if reset='1' then
         q_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         q_reg <= q_next;
      end if;
   end process;

   -- next-state logic for the counter
   q_next <= q_reg + 1;

   -- 2 MSBs of counter to control 4-to-1 multiplexing
   sel <= std_logic_vector(q_reg(N-1 downto N-2));
   process(sel,hex0,hex1,hex2,hex3,dp_in)
   begin
      case sel is
         when "00" =>
            an <= "1110";
            hex <= hex0;
            dp <= dp_in(0);
         when "01" =>
            an <= "1101";
            hex <= hex1;
            dp <= dp_in(1);
         when "10" =>
            an <= "1011";
            hex <= hex2;
            dp <= dp_in(2);
         when others =>
            an <= "0111";
            hex <= hex3;
            dp <= dp_in(3);
      end case;
   end process;
   -- hex-to-7-segment led decoding
   with hex select
      sseg(6 downto 0) <=
         "0000001" when "0000",
         "1001111" when "0001",
         "0010010" when "0010",
         "0000110" when "0011",
         "1001100" when "0100",
            "0100100" when "0101",
            "0100000" when "0110",
            "0001111" when "0111",
            "0000000" when "1000",
            "0000100" when "1001",
            "0001000" when "1010", --a
            "1100000" when "1011", --b
            "0110001" when "1100", --c
            "1000010" when "1101", --d
            "0110000" when "1110", --e
            "0111000" when others; --f
   -- decimal point
   sseg(7) <= dp;
end arch;
```

To verify operation of this circuit, we define the 8-bit switch as two 4-bit unsigned numbers, add the two numbers, and show the two numbers and their sum on the four-digit seven-segment LED display. The code is shown in Listing 4.16.

Listing 4.16 Testing circuit for time multiplexing with hexadecimal digits

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity hex_mux_test is
   port(
      clk: in std_logic;
      sw: in std_logic_vector(7 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end hex_mux_test;

architecture arch of hex_mux_test is
   signal a, b: unsigned(7 downto 0);
   signal sum: std_logic_vector(7 downto 0);
begin
   disp_unit: entity work.disp_hex_mux
      port map(
         clk=>clk, reset=>'0',
         hex3=>sum(7 downto 4), hex2=>sum(3 downto 0),
         hex1=>sw(7 downto 4), hex0=>sw(3 downto 0),
         dp_in=>"1011", an=>an, sseg=>sseg);
   a <= "0000" & unsigned(sw(3 downto 0));
   b <= "0000" & unsigned(sw(7 downto 4));
   sum <= std_logic_vector(a + b);
end arch;
```

Simulation consideration Many sequential circuit examples in the book operate at a relatively slow rate, as does the enable pulse of the LED time-multiplexing circuit. This can be done by generating a single-clock enable tick from a counter. An 18-bit counter is used in this circuit:

```
   constant N: integer:=18;
```

```
signal q_reg, q_next: unsigned(N-1 downto 0);
. . .
q_next <= q_reg + 1;
```

Because of the counter's size, simulating this type of circuit consumes a significant amount of computation time (i.e., 2^{18} clock cycles for one iteration). Since our main interest is in the multiplexing part of the code, most simulation time is wasted. It is more efficient to use a smaller counter in simulation. We can do this by modifying the constant statement

```
constant N: integer:=4;
```

when constructing the testbench. This requires only 2^4 clock cycles for one iteration and allows us to better exercise and observe the key operations.

Instead of using a constant statement and modifying code between simulation and synthesis, an alternative is to define a generic for the relevant parameter. During instantiation, we can assign different values for simulation and synthesis.

4.5.2 Stopwatch

We consider the design of a stopwatch in this subsection. The watch displays the time in three decimal digits, and counts from 00.0 to 99.9 seconds and wraps around. It contains a synchronous clear signal, `clr`, which returns the count to 00.0, and an enable signal, `go`, which enables and suspends the counting. This design is basically a BCD (binary-coded decimal) counter, which counts in BCD format. In this format, a decimal number is represented by a sequence of 4-bit BCD digits. For example, 139_{10} is represented as "0001 0011 1001" and the next number in sequence is 140_{10}, which is represented as "0001 0100 0000".

Since the S3 board has a 50-MHz clock, we first need a mod-5,000,000 counter that generates a one-clock-cycle tick every 0.1 second. The tick is then used to enable counting of the three-digit BCD counter.

Design I Our first design of the BCD counter uses a cascading structure of three decade (i.e., mod-10) counters, representing counts of 0.1, 1, and 10 seconds, respectively. The decade counter has an enable signal and generates a one-clock-cycle tick when it reaches 9. We can use these signals to "hook" the three counters. For example, the 10-second counter is enabled only when the enable tick of the mod-5,000,000 counter is asserted and both the 0.1- and 1-second counters are 9. The code is shown in Listing 4.17.

Listing 4.17 Cascading description for a stopwatch

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity stop_watch is
   port(
      clk: in std_logic;
      go, clr: in std_logic;
      d2, d1, d0: out std_logic_vector(3 downto 0)
   );
end stop_watch;

architecture cascade_arch of stop_watch is
   constant DVSR: integer:=5000000;
```

```vhdl
   signal ms_reg, ms_next: unsigned(22 downto 0);
   signal d2_reg, d1_reg, d0_reg: unsigned(3 downto 0);
   signal d2_next, d1_next, d0_next: unsigned(3 downto 0);
   signal d1_en, d2_en, d0_en: std_logic;
   signal ms_tick, d0_tick, d1_tick: std_logic;
begin
   -- register
   process(clk)
   begin
      if (clk'event and clk='1') then
         ms_reg <= ms_next;
         d2_reg <= d2_next;
         d1_reg <= d1_next;
         d0_reg <= d0_next;
      end if;
   end process;

   -- next-state logic
   -- 0.1 sec tick generator: mod-5000000
   ms_next <=
      (others=>'0') when clr='1' or
                         (ms_reg=DVSR and go='1') else
      ms_reg + 1 when go='1' else
      ms_reg;
   ms_tick <= '1' when ms_reg=DVSR else '0';
   -- 0.1 sec counter
   d0_en <= '1' when ms_tick='1' else '0';
   d0_next <=
      "0000" when (clr='1') or (d0_en='1' and d0_reg=9) else
      d0_reg + 1 when d0_en='1' else
      d0_reg;
   d0_tick <= '1' when d0_reg=9 else '0';
   -- 1 sec counter
   d1_en <= '1' when ms_tick='1' and d0_tick='1' else '0';
   d1_next <=
      "0000" when (clr='1') or (d1_en='1' and d1_reg=9) else
      d1_reg + 1 when d1_en='1' else
      d1_reg;
   d1_tick <= '1' when d1_reg=9 else '0';
   -- 10 sec counter
   d2_en <=
      '1' when ms_tick='1' and d0_tick='1' and d1_tick='1' else
      '0';
   d2_next <=
      "0000" when (clr='1') or (d2_en='1' and d2_reg=9) else
      d2_reg + 1 when d2_en='1' else
      d2_reg;

   -- output logic
   d0 <= std_logic_vector(d0_reg);
   d1 <= std_logic_vector(d1_reg);
   d2 <= std_logic_vector(d2_reg);
end cascade_arch;
```

Note that all registers are controlled by the same clock signal. This example illustrates how to use a one-clock-cycle enable tick to maintain synchronicity. An inferior approach is to use the output of the lower counter as the clock signal for the next stage. Although it may appear to be simpler, it violates the synchronous design principle and is a very poor practice.

Design II An alternative for the three-digit BCD counter is to describe the entire structure in a nested if statement. The nested conditions indicate that the counter reaches .9, 9.9, and 99.9 seconds. The code is shown in Listing 4.18.

Listing 4.18 Nested if-statement description for a stopwatch

```
architecture if_arch of stop_watch is
   constant DVSR: integer:=5000000;
   signal ms_reg, ms_next: unsigned(22 downto 0);
   signal d2_reg, d1_reg, d0_reg: unsigned(3 downto 0);
   signal d2_next, d1_next, d0_next: unsigned(3 downto 0);
   signal ms_tick: std_logic;
begin
   -- register
   process(clk)
   begin
      if (clk'event and clk='1') then
         ms_reg <= ms_next;
         d2_reg <= d2_next;
         d1_reg <= d1_next;
         d0_reg <= d0_next;
      end if;
   end process;

   -- next-state logic
   -- 0.1 sec tick generator: mod-5000000
   ms_next <=
      (others=>'0') when clr='1' or
                         (ms_reg=DVSR and go='1') else
      ms_reg + 1 when go='1' else
      ms_reg;
   ms_tick <= '1' when ms_reg=DVSR else '0';
   --3-digit incrementor
   process(d0_reg,d1_reg,d2_reg,ms_tick,clr)
   begin
      -- default
      d0_next <= d0_reg;
      d1_next <= d1_reg;
      d2_next <= d2_reg;
      if clr='1' then
         d0_next <= "0000";
         d1_next <= "0000";
         d2_next <= "0000";
      elsif ms_tick='1' then
         if (d0_reg/=9) then
            d0_next <= d0_reg + 1;
         else       -- reach XX9
            d0_next <= "0000";
```

```
               if (d1_reg/=9) then
                  d1_next <= d1_reg + 1;
               else    -- reach X99
                  d1_next <= "0000";
                  if (d2_reg/=9) then
                     d2_next <= d2_reg + 1;
                  else -- reach 999
                     d2_next <= "0000";
                  end if;
               end if;
            end if;
         end if;
      end process;
      -- output logic
      d0 <= std_logic_vector(d0_reg);
      d1 <= std_logic_vector(d1_reg);
      d2 <= std_logic_vector(d2_reg);
   end if_arch;
```

Verification circuit To verify operation of the stopwatch, we can combine it with the previous hexadecimal LED time-multiplexing circuit to display the output of the watch. The code is shown in Listing 4.19. Note that the first digit of the LED is assigned to 0 and the go and clr signals are mapped to two buttons of the S3 board.

Listing 4.19 Testing circuit for a stopwatch

```
library ieee;
use ieee.std_logic_1164.all;
entity stop_watch_test is
   port(
      clk: in std_logic;
      btn: in std_logic_vector(3 downto 0);
      an: out std_logic_vector(3 downto 0);
      sseg: out std_logic_vector(7 downto 0)
   );
end stop_watch_test;

architecture arch of stop_watch_test is
   signal d2, d1, d0: std_logic_vector(3 downto 0);
begin
   disp_unit: entity work.disp_hex_mux
      port map(
         clk=>clk, reset=>'0',
         hex3=>"0000", hex2=>d2,
         hex1=>d1, hex0=>d0,
         dp_in=>"1101", an=>an, sseg=>sseg);

  watch_unit: entity work.stop_watch(cascade_arch)
     port map(
        clk=>clk, go=>btn(1), clr=>btn(0),
        d2 =>d2, d1=>d1, d0=>d0 );
end arch;
```

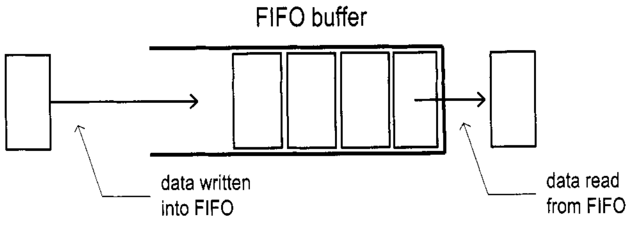


Figure 4.10 Conceptual diagram of a FIFO buffer.

4.5.3 FIFO buffer

A FIFO (first-in-first-out) buffer is an "elastic" storage between two subsystems, as shown in the conceptual diagram of Figure 4.10. It has two control signals, `wr` and `rd`, for write and read operations. When `wr` is asserted, the input data is written into the buffer. The read operation is somewhat misleading. The head of the FIFO buffer is normally always available and thus can be read at any time. The `rd` signal actually acts like a "remove" signal. When it is asserted, the first item (i.e., head) of the FIFO buffer is removed and the next item becomes available.

FIFO buffer is a critical component in many applications and the optimized implementation can be quite complex. In this subsection, we introduce a simple, genuine circular-queue-based design. More efficient, device-specific implementation can be found in the Xilinx literature.

Circular-queue-based implementation One way to implement a FIFO buffer is to add a control circuit to a register file. The registers in the register file are arranged as a circular queue with two pointers. The *write pointer* points to the head of the queue, and the *read pointer* points to the tail of the queue. The pointer advances one position for each write or read operation. The operation of an eight-word circular queue is shown in Figure 4.11.

A FIFO buffer usually contains two status signals, `full` and `empty`, to indicate that the FIFO is full (i.e., cannot be written) and empty (i.e., cannot be read), respectively. One of the two conditions occurs when the read pointer is equal to the write pointer, as shown in Figure 4.11(a), (f), and (i). The most difficult design task of the controller is to derive a mechanism to distinguish the two conditions. One scheme is to use two FFs to keep track of the empty and full statuses. The FFs are set to '1' and '0' during system initialization and then modified in each clock cycle according to the values of the `wr` and `rd` signals. The code is shown in Listing 4.20.

Listing 4.20 FIFO buffer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity fifo is
   generic(
      B: natural:=8; -- number of bits
      W: natural:=4 -- number of address bits
   );
   port(
      clk, reset: in std_logic;
      rd, wr: in std_logic;
```

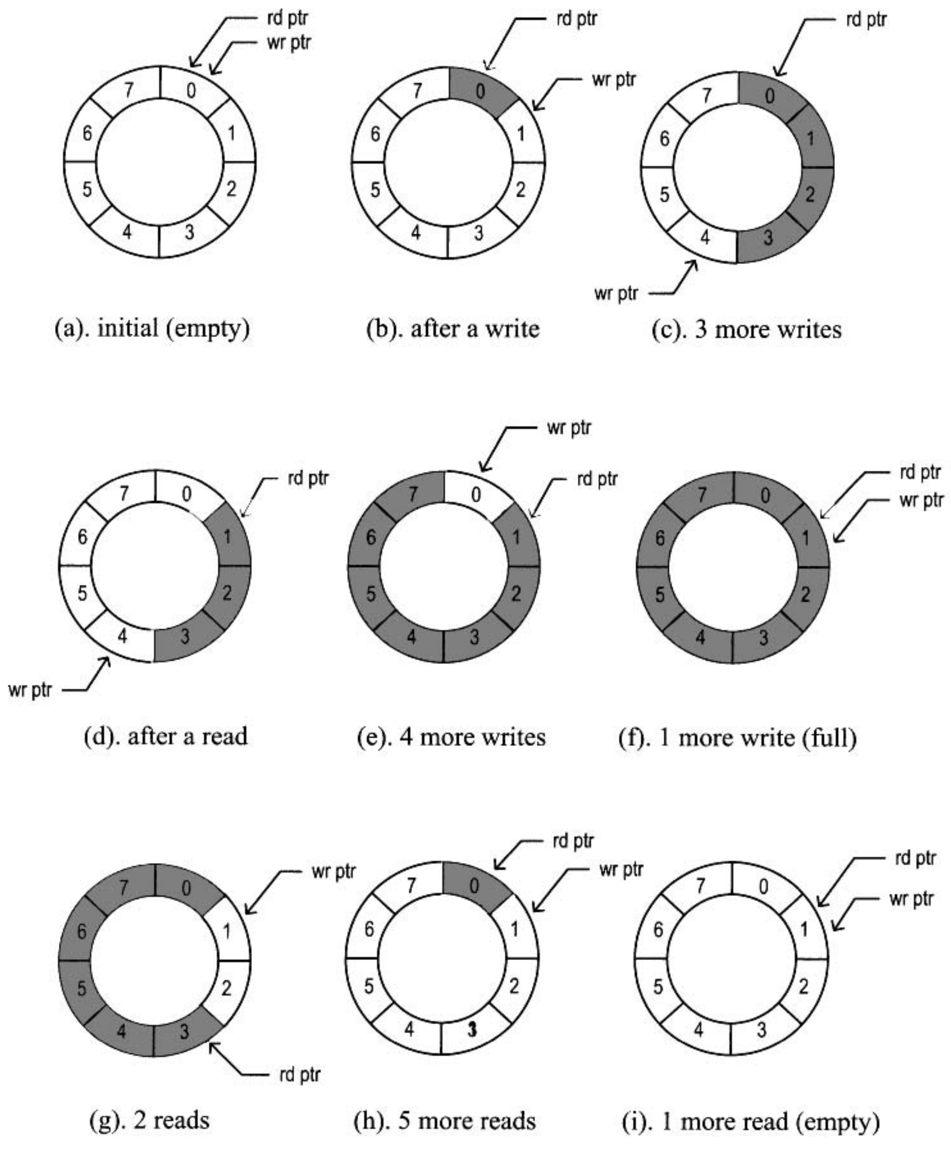


Figure 4.11 FIFO buffer based on a circular queue.

```
      w_data: in std_logic_vector (B-1 downto 0);
      empty, full: out std_logic;
      r_data: out std_logic_vector (B-1 downto 0)
   );
end fifo;

architecture arch of fifo is
   type reg_file_type is array (2**W-1 downto 0) of
        std_logic_vector(B-1 downto 0);
   signal array_reg: reg_file_type;
   signal w_ptr_reg, w_ptr_next, w_ptr_succ:
      std_logic_vector(W-1 downto 0);
   signal r_ptr_reg, r_ptr_next, r_ptr_succ:
      std_logic_vector(W-1 downto 0);
   signal full_reg, empty_reg, full_next, empty_next:
          std_logic;
   signal wr_op: std_logic_vector(1 downto 0);
   signal wr_en: std_logic;
begin
   --=================================================
   -- register file
   --=================================================
   process(clk,reset)
   begin
     if (reset='1') then
        array_reg <= (others=>(others=>'0'));
     elsif (clk'event and clk='1') then
        if wr_en='1' then
           array_reg(to_integer(unsigned(w_ptr_reg)))
                 <= w_data;
        end if;
     end if;
   end process;
   -- read port
   r_data <= array_reg(to_integer(unsigned(r_ptr_reg)));
   -- write enabled only when FIFO is not full
   wr_en <= wr and (not full_reg);

   --=================================================
   -- fifo control logic
   --=================================================
   -- register for read and write pointers
   process(clk,reset)
   begin
      if (reset='1') then
         w_ptr_reg <= (others=>'0');
         r_ptr_reg <= (others=>'0');
         full_reg <= '0';
         empty_reg <= '1';
      elsif (clk'event and clk='1') then
         w_ptr_reg <= w_ptr_next;
         r_ptr_reg <= r_ptr_next;
         full_reg <= full_next;
         empty_reg <= empty_next;
      end if;
   end process;

   -- successive pointer values
   w_ptr_succ <= std_logic_vector(unsigned(w_ptr_reg)+1);
   r_ptr_succ <= std_logic_vector(unsigned(r_ptr_reg)+1);

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

The code is divided into a register file and a FIFO controller. The controller consists of two pointers and two status FFs. Its next-state logic examines the wr and rd signals and takes actions accordingly. For example, let us consider the "10" case, which implies that only a write operation occurs. The status FF is checked first to ensure that the buffer is not full. If this condition is met, we advance the write pointer by one position and clear the empty status FF. Storing one extra word to the buffer may make it full. This happens if the new write pointer "catches" the read pointer, which is expressed by the w_ptr_succ=r_ptr_reg expression.

Verification circuit The verification circuit examines the operation of a 2^4-by-3 FIFO buffer. We use three switches to generate the input data and use two buttons for the `wr` and `rd` signals. The 3-bit readout and the `full` and `empty` status signals are displayed in five discrete LEDs. Because of bounces of the mechanical contact, a debouncing circuit is needed to generate a clean, one-clock-cycle tick. The debouncing module, named `debounce`, is discussed in Section 5.9 but for now can be treated as a predesigned module. The original button inputs are `btn(0)` and `btn(1)`, and the debounced signals are `db_btn(0)` and `db_btn(1)`. The code is shown in Listing 4.21.

Listing 4.21 Testing circuit for a FIFO buffer

```
library ieee;
use ieee.std_logic_1164.all;
entity fifo_test is
   port(
      clk, reset: in std_logic;
      btn: std_logic_vector(1 downto 0);
      sw: std_logic_vector(2 downto 0);
      led: out std_logic_vector(7 downto 0)
   );
end fifo_test;

architecture arch of fifo_test is
   signal db_btn: std_logic_vector(1 downto 0);
begin
   -- debouncing circuit for btn(0)
   btn_db_unit0: entity work.debounce(fsmd_arch)
      port map(clk=>clk, reset=>reset, sw=>btn(0),
               db_level=>open, db_tick=>db_btn(0));
   -- debouncing circuit for btn(1)
   btn_db_unit1: entity work.debounce(fsmd_arch)
      port map(clk=>clk, reset=>reset, sw=>btn(1),
               db_level=>open, db_tick=>db_btn(1));
   -- instantiate a 2^2-by-3 fifo
   fifo_unit: entity work.fifo(arch)
      generic map(B=>3, W=>2)
      port map(clk=>clk, reset=>reset,
               rd=>db_btn(0), wr=>db_btn(1),
               w_data=>sw, r_data=>led(2 downto 0),
               full=>led(7), empty=>led(6));
   -- disable unused leds
   led(5 downto 3)<=(others=>'0');
end arch;
```

4.6 BIBLIOGRAPHIC NOTES

The bibliographic information for this chapter is similar to that for Chapter 3.

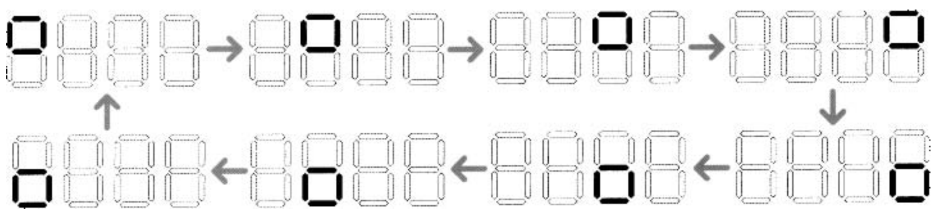


Figure 4.12 Pattern for Experiment 4.7.3.

4.7 SUGGESTED EXPERIMENTS

4.7.1 Programmable square wave generator

A programmable square wave generator is a circuit that can generate a square wave with variable on (i.e., logic '1') and off (i.e., logic '0') intervals. The durations of the intervals are specified by two 4-bit control signals, `m` and `n`, which are interpreted as unsigned integers. The on and off intervals are `m`*100 ns and `n`*100 ns, respectively (recall that the period of the S3 onboard oscillator is 20 ns). Design a programmable square wave generator circuit. The circuit should be completely synchronous. We need a logic analyzer or oscilloscope to verify its operation.

4.7.2 PWM and LED dimmer

The duty cycle of a square wave is defined as the percentage of the on interval (i.e., logic '1') in a period. A PWM (pulse width modulation) circuit can generate an output with variable duty cycles. For a PWM with 4-bit resolution, a 4-bit control signal, `w`, specifies the duty cycle. The `w` signal is interpreted as an unsigned integer and the duty cycle is $\frac{w}{16}$.

1. Design a PWM circuit with 4-bit resolution and verify its operation using a logic analyzer or oscilloscope.
2. Modify the LED time-multiplexing circuit to include the PWM circuit for the `an` signal. The PWM circuit specifies the percentage of time that the LED display is on. We can control the perceived brightness by changing the duty cycle. Verify the circuit's operation by observing 1 bit of `an` on a logic analyzer or oscilloscope.
3. Replace the LED time-multiplexing circuit of Listing 4.19 with the new design and use the lower 4 bits of the 8-bit switch to control the duty cycle. Verify operation of the circuit. It may be necessary to go to a dark area to see the effect of dimming.

4.7.3 Rotating square circuit

In a seven-segment LED display, a square pattern can be created by enabling the a, b, f, and g segments or the c, d, e, and g segments. We want to design a circuit that circulates the square patterns in the four-digit seven-segment LED display. The clockwise circulating pattern is shown in Figure 4.12. The circuit should have an input, `en`, which enables or pauses the circulation, and an input, `cw`, which specifies the direction (i.e., clockwise or counterclockwise) of the circulation.

Design the circuit and verify its operation on the prototyping board. Make sure that the circulation rate is slow enough for visual inspection.

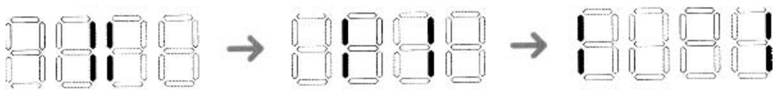


Figure 4.13 Pattern for Experiment 4.7.4.

4.7.4 Heartbeat circuit

We want to create a "heartbeat" for the prototyping board. It repeats the simple pattern in the four-digit seven-segment display, as shown in Figure 4.13, at a rate of 72 Hz. Design the circuit and verify its operation on the prototyping board.

4.7.5 Rotating LED banner circuit

The prototyping board has a four-digit seven-segment LED display, and thus only four symbols can be displayed at a time. We can show more information if the data is rotated and moved continuously. For example, assume that the message is 10 digits (i.e., "0123456789"). The display can show the message as "0123", "1234", "2345", ..., "6789", "7890", ..., "0123". The circuit should have an input, en, which enables or pauses the rotation, and an input, `dir`, which specifies the direction (i.e., rotate left or right).

Design the circuit and verify its operation on the prototyping board. Make sure that the rotation rate is slow enough for visual inspection.

4.7.6 Enhanced stopwatch

Modify the stopwatch with the following extensions:

- Add an additional signal, up, to control the direction of counting. The stopwatch counts up when the up signal is asserted and counts down otherwise.
- Add a minute digit to the display. The LED display format should be like `M.SS.D`, where `D` represents 0.1 second and its range is between 0 and 9, `SS` represents seconds and its range is between 00 and 59, and `M` represents minutes and its range is between 0 and 9.

Design the new stopwatch and verify its operation with a testing circuit.

4.7.7 Stack

A stack is a last-in-first-out buffer in which the last stored data is retrieved first. Storing a data word to a stack is known as a *push* operation, and retrieving a data word from a stack is known as a *pop* operation. The I/O signals of a stack are similar to those of a FIFO buffer except that we generally use the `push` and `pop` signals in place of the `wr` and `rd` signals. Design a stack using a register file and verify its operation with a testing circuit similar to the one in Listing 4.21.