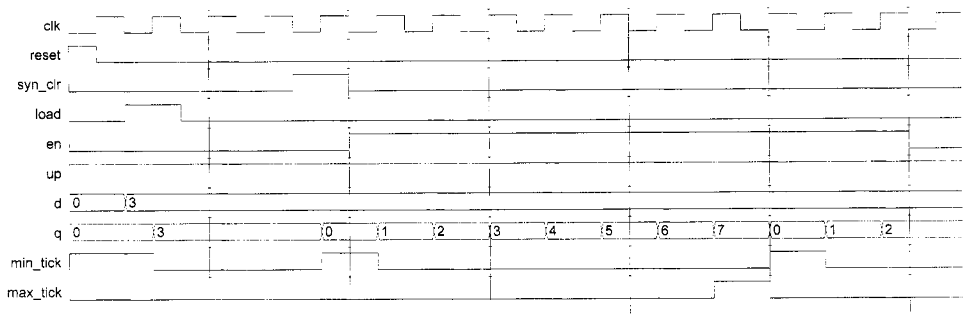


Figure 4.4 Testbench waveform.

```
      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.

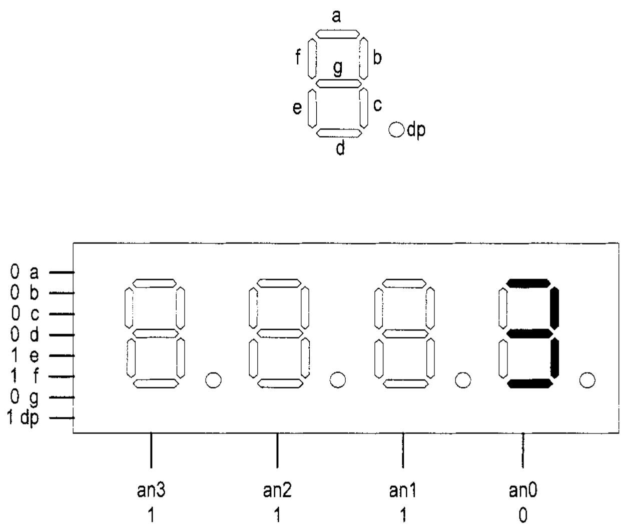


Figure 4.5 Time-multiplexed seven-segment LED display.

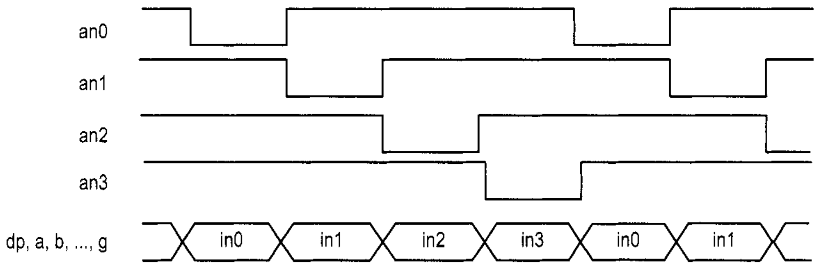
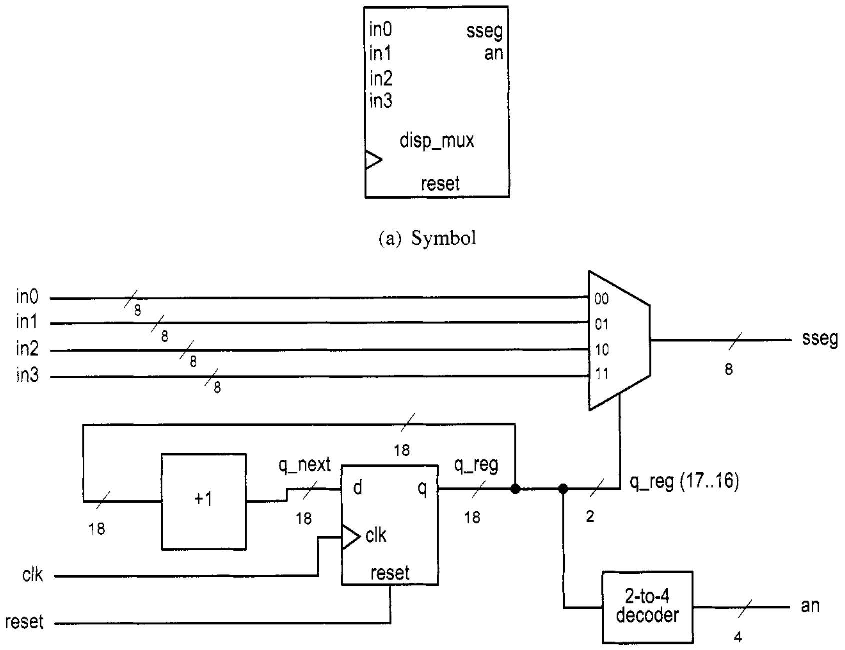


Figure 4.6 Timing diagram of a time-multiplexed seven-segment LED display.



(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(
```

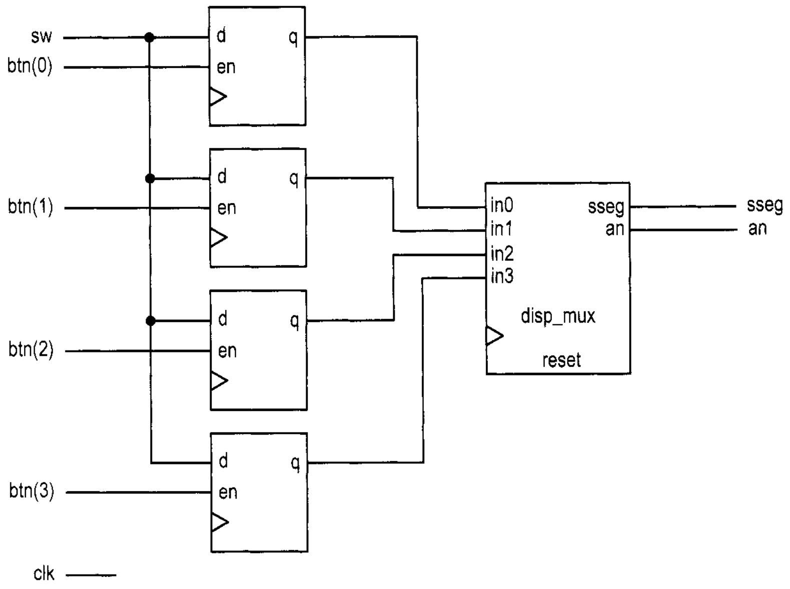


Figure 4.8 LED time-multiplexing testing circuit.

```
      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;
```

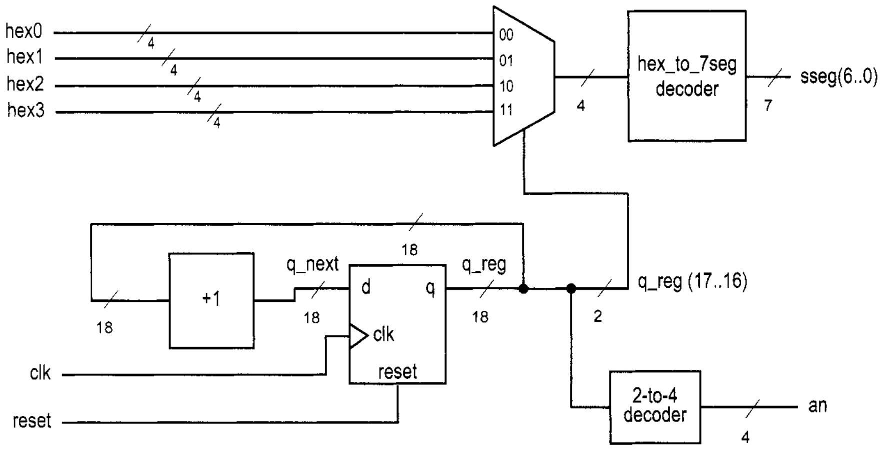


Figure 4.9 Block diagram of a hexadecimal time-multiplexing circuit.

```
            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;
   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;
```

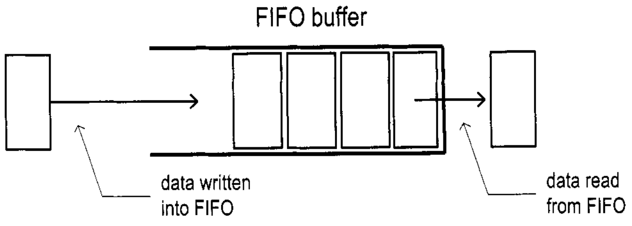


Figure 4.10 Conceptual diagram of a FIFO buffer.

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;
```

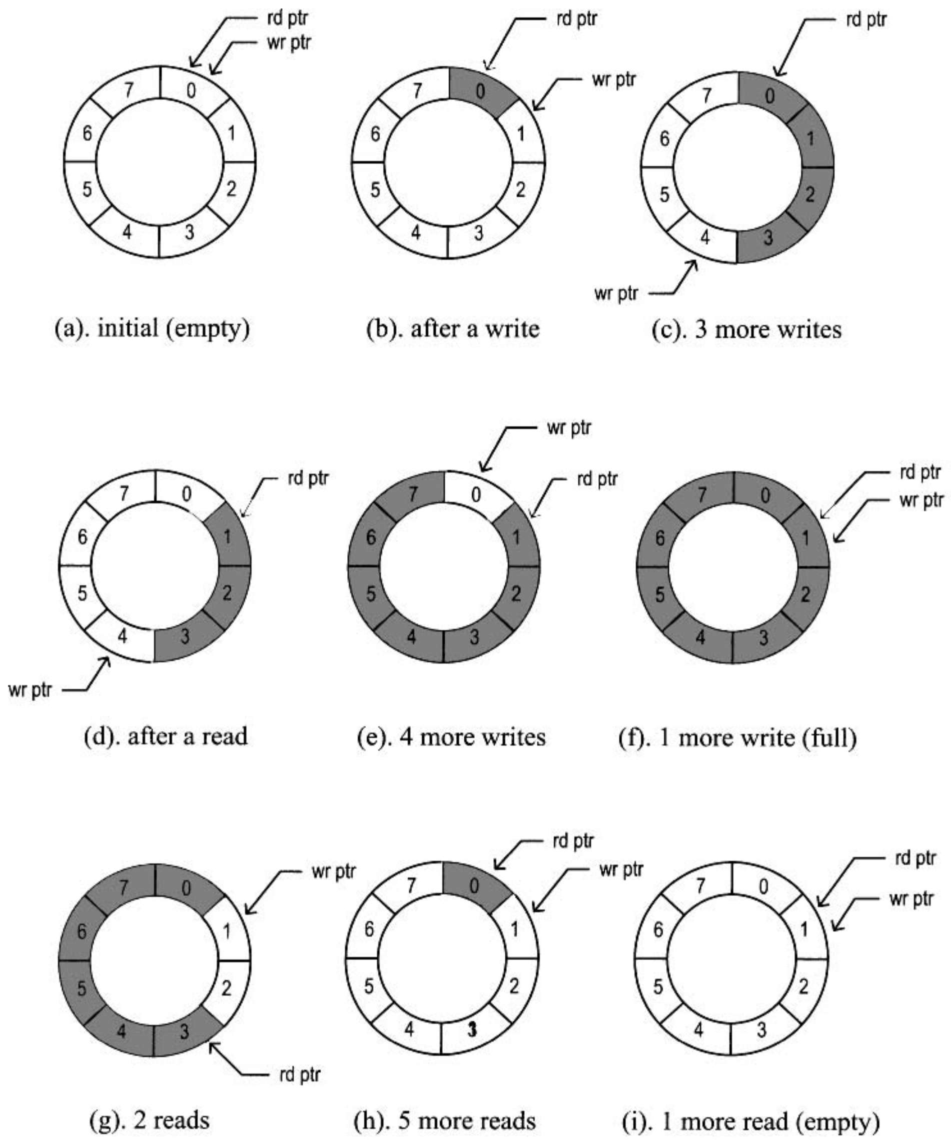


Figure 4.11 FIFO buffer based on a circular queue.

```
      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.