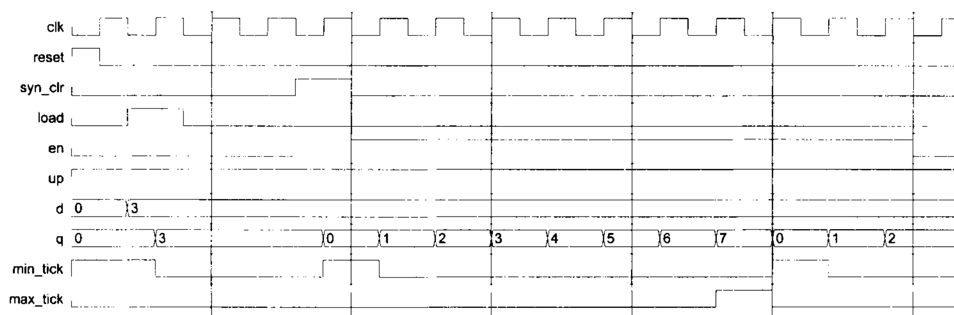


Figure 4.4 Testbench waveform.

```
wait(q==2);
```

or wait until a signal changes, such as

```
wait(min_tick);
```

or wait for an absolute time, such as

```
#(4*T);  // wait for 4T
```

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

```
@(negedge 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 a "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.

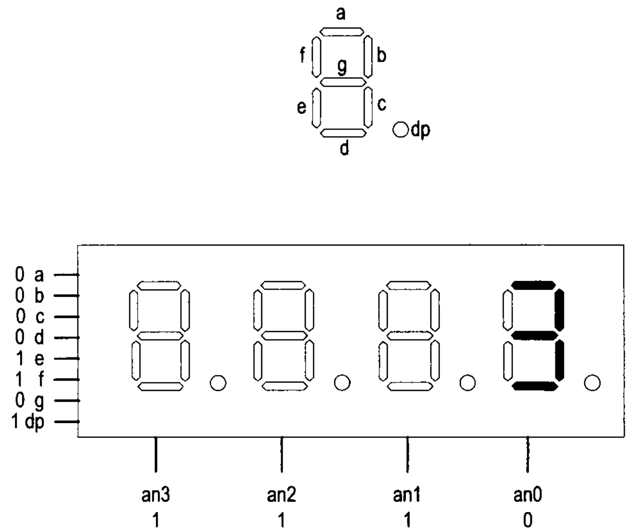


Figure 4.5 Time-multiplexed seven-segment LED display.

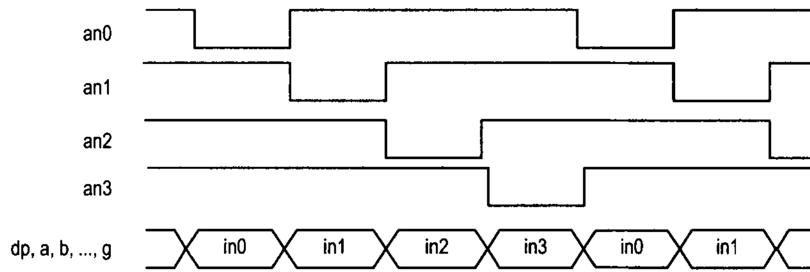
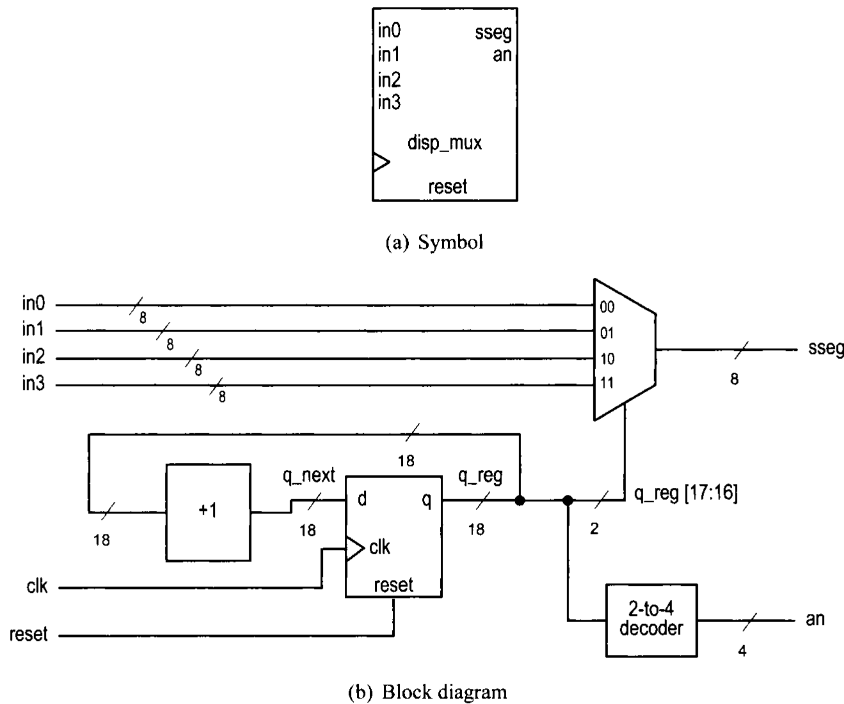


Figure 4.6 Timing diagram of a time-multiplexed seven-segment LED display.



(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

```
module disp_mux
   (
    input wire clk, reset,
    input [7:0] in3, in2, in1, in0,
 5  output reg [3:0] an,   // enable, 1-out-of-4 asserted low
    output reg [7:0] sseg  // led segments
   );

   // constant declaration
10 // refreshing rate around 800 Hz (50 MHz/2^16)
   localparam N = 18;
```

```
   // signal declaration
   reg [N-1:0] q_reg;
15 wire [N-1:0] q_next;

   // N-bit counter
   // register
   always @(posedge clk,  posedge reset)
20    if (reset)
         q_reg <= 0;
      else
         q_reg <= q_next;

25 // next-state logic
   assign q_next = q_reg + 1;

   // 2 MSBs of counter to control 4-to-1 multiplexing
   // and to generate active-low enable signal
30 always @*
      case (q_reg[N-1:N-2])
         2'b00:
            begin
               an = 4'b1110;
35             sseg = in0;
            end
         2'b01:
            begin
               an =  4'b1101;
40             sseg = in1;
            end
         2'b10:
            begin
               an =  4'b1011;
45             sseg = in2;
            end
         default:
            begin
               an =  4'b0111;
50             sseg = in3;
            end
       endcase

endmodule
```

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 signals. When we press a pushbutton, 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

```
module disp_mux_test
   (
    input wire clk,
```

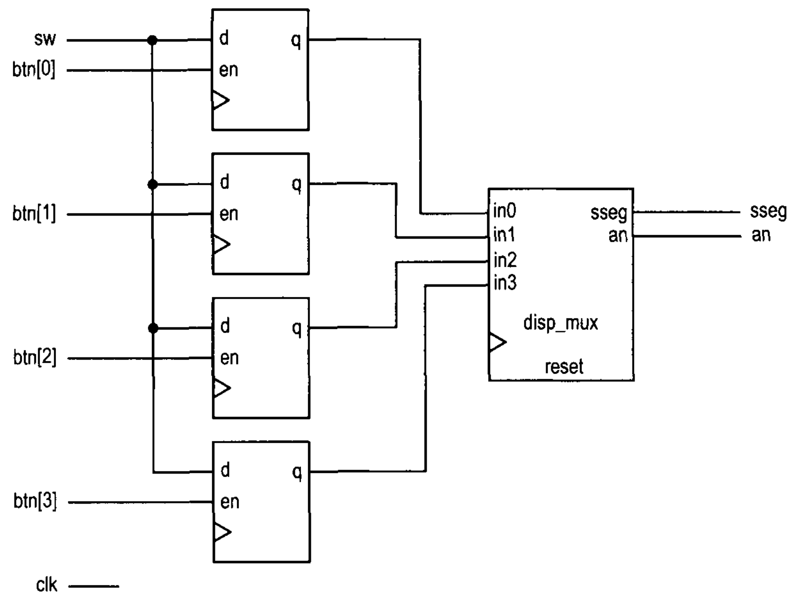


Figure 4.8 LED time-multiplexing testing circuit.

```
    input wire [3:0] btn,
5   input wire [7:0] sw,
    output wire [3:0] an,
    output wire [7:0] sseg
   );

10 // signal declaration
   reg [7:0] d3_reg, d2_reg, d1_reg, d0_reg;

   // instantiate 7-seg LED display time-multiplexing module
   disp_mux disp_unit
15    (.clk(clk), .reset(1'b0), .in0(d0_reg), .in1(d1_reg),
       .in2(d2_reg), .in3(d3_reg), .an(an), .sseg(sseg));

   // registers for 4 led patterns
   always @(posedge clk)
20 begin
      if (btn[3])
         d3_reg <= sw;
      if (btn[2])
         d2_reg <= sw;
25    if (btn[1])
         d1_reg <= sw;
      if (btn[0])
         d0_reg <= sw;
    end
30
endmodule
```

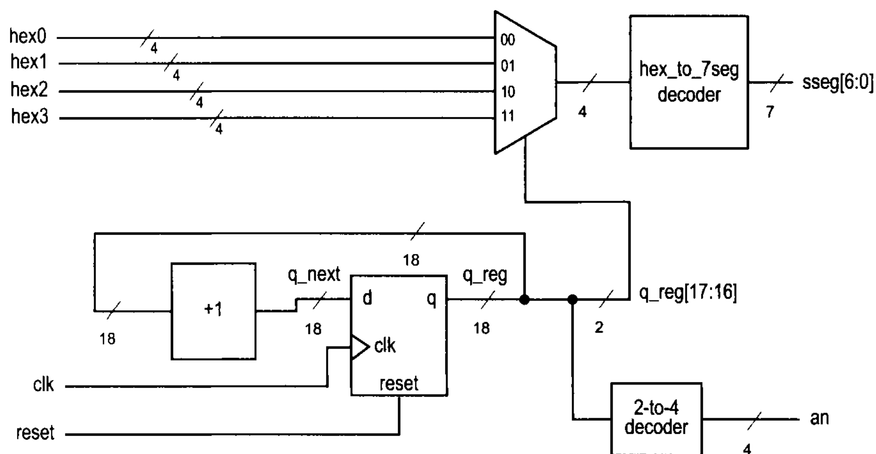


Figure 4.9 Block diagram of a hexadecimal time-multiplexing circuit.

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 3.9.1. 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

```
module disp_hex_mux
   (
    input wire clk, reset,
    input wire [3:0] hex3, hex2, hex1, hex0,  // hex digits
5   input wire [3:0] dp_in,                   // 4 decimal points
    output reg [3:0] an,  // enable 1-out-of-4 asserted low
    output reg [7:0] sseg // led segments
   );

10  // constant declaration
   // refreshing rate around 800 Hz (50 MHz/2^16)
   localparam N = 18;
   // internal signal declaration
   reg [N-1:0] q_reg;
15 wire [N-1:0] q_next;
   reg [3:0] hex_in;
   reg dp;

   // N-bit counter
20 // register
   always @(posedge clk, posedge reset)
```

```
         if (reset)
            q_reg <= 0;
         else
            q_reg <= q_next;

   // next-state logic
   assign q_next = q_reg + 1;

   // 2 MSBs of counter to control 4-to-1 multiplexing
   // and to generate active-low enable signal
   always @*
      case (q_reg[N-1:N-2])
         2'b00:
            begin
               an =  4'b1110;
               hex_in = hex0;
               dp = dp_in[0];
            end
         2'b01:
            begin
               an =  4'b1101;
               hex_in = hex1;
               dp = dp_in[1];
            end
         2'b10:
            begin
               an =  4'b1011;
               hex_in = hex2;
               dp = dp_in[2];
            end
         default:
            begin
               an =  4'b0111;
               hex_in = hex3;
               dp = dp_in[3];
            end
       endcase

   // hex to seven-segment led display
   always @*
   begin
      case(hex_in)
         4'h0: sseg[6:0] = 7'b0000001;
         4'h1: sseg[6:0] = 7'b1001111;
         4'h2: sseg[6:0] = 7'b0010010;
         4'h3: sseg[6:0] = 7'b0000110;
         4'h4: sseg[6:0] = 7'b1001100;
         4'h5: sseg[6:0] = 7'b0100100;
         4'h6: sseg[6:0] = 7'b0100000;
         4'h7: sseg[6:0] = 7'b0001111;
         4'h8: sseg[6:0] = 7'b0000000;
         4'h9: sseg[6:0] = 7'b0000100;
         4'ha: sseg[6:0] = 7'b0001000;
```

```
75          4'hb: sseg[6:0] = 7'b1100000;
            4'hc: sseg[6:0] = 7'b0110001;
            4'hd: sseg[6:0] = 7'b1000010;
            4'he: sseg[6:0] = 7'b0110000;
            default: sseg[6:0] = 7'b0111000;  //4'hf
80      endcase
        sseg[7] = dp;
     end

endmodule
```

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

```
module hex_mux_test
   (
    input wire clk,
    input wire [7:0] sw,
5   output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
10 wire [3:0] a, b;
   wire [7:0] sum;

   // instantiate 7-seg LED display module
   disp_hex_mux disp_unit
15    (.clk(clk), .reset(1'b0),
       .hex3(sum[7:4]), .hex2(sum[3:0]), .hex1(b), .hex0(a),
       .dp_in(4'b1011), .an(an), .sseg(sseg));

   // adder
20 assign a = sw[3:0];
   assign b = sw[7:4];
   assign sum = {4'b0,a} + {4'b0,b};

endmodule
```

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:

```
localparam N = 18;
reg [N-1:0] q_reg;
wire [N-1:0] q_next;
. . .
assign 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

```
localparam N = 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 and modifying code between simulation and synthesis, an alternative is to define N as a 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

```
module stop_watch_cascade
   (
    input wire clk,
    input wire go, clr,
5   output wire [3:0] d2, d1, d0
   );

   // declaration
   localparam  DVSR = 5000000;
10 reg [22:0] ms_reg;
   wire [22:0] ms_next;
   reg [3:0] d2_reg, d1_reg, d0_reg;
   wire [3:0] d2_next, d1_next, d0_next;
   wire d1_en, d2_en, d0_en;
15 wire ms_tick, d0_tick, d1_tick;

   // body
   // register
   always @(posedge clk)
```

```
20     begin
          ms_reg <= ms_next;
          d2_reg <= d2_next;
          d1_reg <= d1_next;
          d0_reg <= d0_next;
25     end

       // next-state logic
       // 0.1 sec tick generator: mod-5000000
       assign ms_next = (clr || (ms_reg==DVSR && go)) ? 4'b0 :
30                      (go) ? ms_reg + 1 :
                               ms_reg;
       assign ms_tick = (ms_reg==DVSR) ? 1'b1 : 1'b0;
       // 0.1 sec counter
       assign d0_en = ms_tick;
35     assign d0_next = (clr || (d0_en && d0_reg==9)) ? 4'b0 :
                        (d0_en) ? d0_reg + 1 :
                                  d0_reg;
       assign d0_tick = (d0_reg==9) ? 1'b1 : 1'b0;
       // 1 sec counter
40     assign d1_en = ms_tick & d0_tick;
       assign d1_next = (clr || (d1_en && d0_reg==9)) ? 4'b0 :
                        (d1_en) ? d1_reg + 1 :
                                  d1_reg;
       assign d1_tick = (d1_reg==9) ? 1'b1 : 1'b0;
45
       // 10 sec counter
       assign d2_en = ms_tick & d0_tick & d1_tick;
       assign d2_next = (clr || (d2_en && d2_reg==9)) ? 4'b0 :
                        (d2_en) ? d2_reg + 1 :
50                                d2_reg;

       // output logic
       assign d0 = d0_reg;
       assign d1 = d1_reg;
55     assign d2 = d2_reg;

    endmodule
```

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

```
module stop_watch_if
   (
    input wire clk,
```

```
   input wire go, clr,
   output wire [3:0] d2, d1, d0
  );

  // declaration
  localparam  DVSR = 5000000;
  reg [22:0] ms_reg;
  wire [22:0] ms_next;
  reg [3:0] d2_reg, d1_reg, d0_reg;
  reg [3:0] d2_next, d1_next, d0_next;
  wire ms_tick;

  // body
  // register
  always @(posedge clk)
  begin
     ms_reg <= ms_next;
     d2_reg <= d2_next;
     d1_reg <= d1_next;
     d0_reg <= d0_next;
  end

  // next-state logic
  // 0.1 sec tick generator: mod-5000000
  assign ms_next = (clr || (ms_reg==DVSR && go)) ? 4'b0 :
                   (go) ? ms_reg + 1 :
                          ms_reg;
  assign ms_tick = (ms_reg==DVSR) ? 1'b1 : 1'b0;
  // 3-digit bcd counter
  always @*
  begin
     // default: keep the previous value
     d0_next = d0_reg;
     d1_next = d1_reg;
     d2_next = d2_reg;
     if (clr)
        begin
           d0_next = 4'b0;
           d1_next = 4'b0;
           d2_next = 4'b0;
        end
     else if (ms_tick)
        if (d0_reg != 9)
           d0_next = d0_reg + 1;
        else              // reach XX9
           begin
              d0_next = 4'b0;
              if (d1_reg != 9)
                 d1_next = d1_reg + 1;
              else       // reach X99
                 begin
                    d1_next = 4'b0;
                    if (d2_reg != 9)
```

```
                        d2_next = d2_reg + 1;
                     else // reach 999
                        d2_next = 4'b0;
60                end
            end
   end

   // output logic
65 assign d0 = d0_reg;
   assign d1 = d1_reg;
   assign d2 = d2_reg;

endmodule
```

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 pushbuttons of the S3 board.

Listing 4.19 Testing circuit for a stopwatch

```
module stop_watch_test
   (
    input wire clk,
    input wire [1:0] btn,
5   output wire [3:0] an,
    output wire [7:0] sseg
   );

   // signal declaration
10 wire [3:0]  d2, d1, d0;

   // instantiate 7-seg LED display module
   disp_hex_mux disp_unit
      (.clk(clk), .reset(1'b0),
15     .hex3(4'b0), .hex2(d2), .hex1(d1), .hex0(d0),
       .dp_in(4'b1101), .an(an), .sseg(sseg));

   // instantiate stopwatch
   stop_watch_if counter_unit
20    (.clk(clk), .go(btn[1]), .clr(btn[0]),
       .d2(d2), .d1(d1), .d0(d0) );

endmodule
```

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"

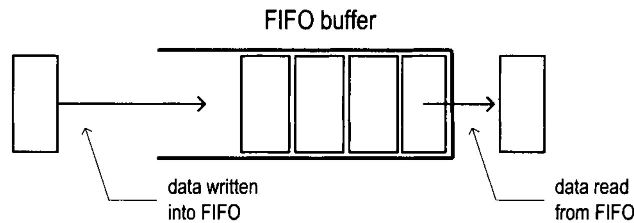


Figure 4.10 Conceptual diagram of a FIFO buffer.

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

```
module fifo
   #(
    parameter B=8, // number of bits in a word
              W=4  // number of address bits
5  )
   (
    input wire clk, reset,
    input wire rd, wr,
    input wire [B-1:0] w_data,
10  output wire empty, full,
    output wire [B-1:0] r_data
   );

   //signal declaration
15 reg [B-1:0] array_reg [2**W-1:0]; // register array
   reg [W-1:0] w_ptr_reg, w_ptr_next, w_ptr_succ;
   reg [W-1:0] r_ptr_reg, r_ptr_next, r_ptr_succ;
   reg full_reg, empty_reg, full_next, empty_next;
```

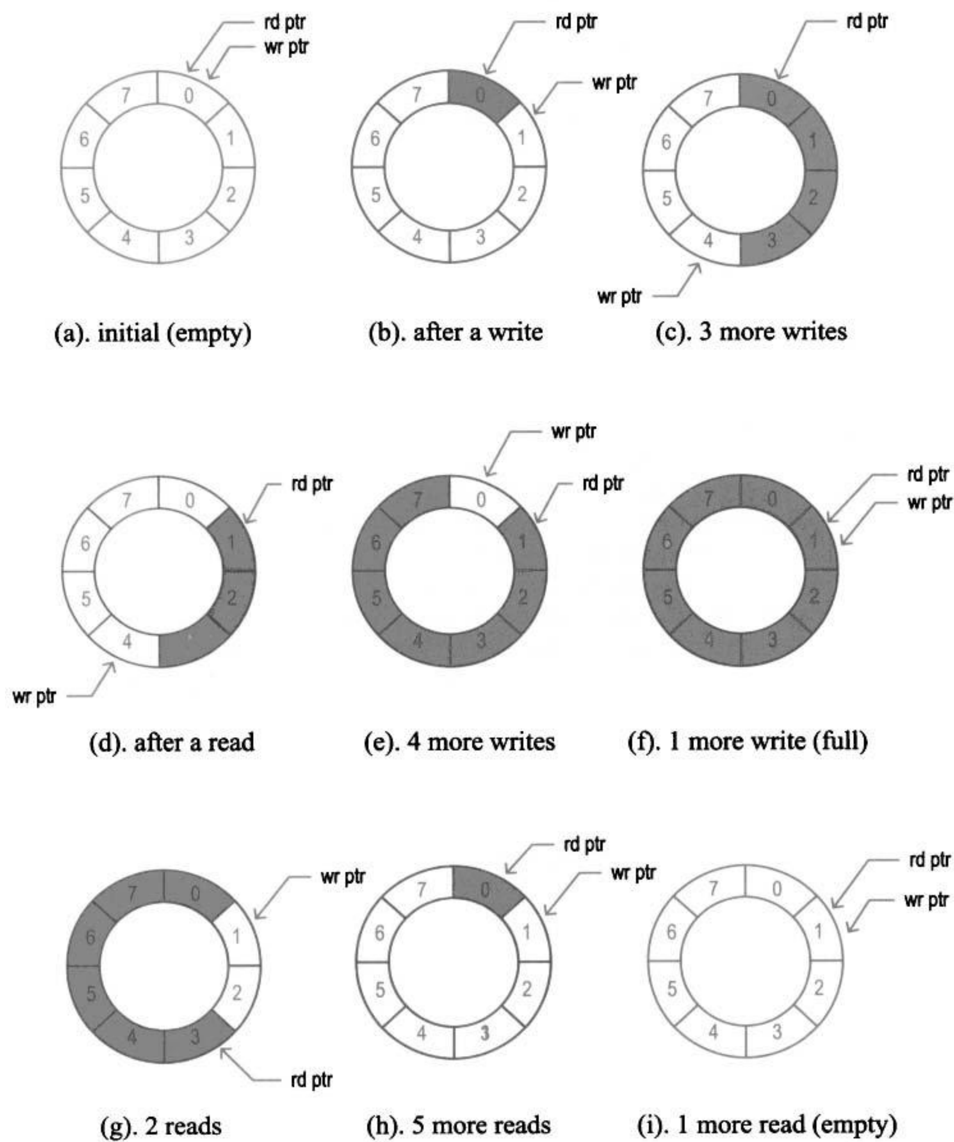


Figure 4.11 FIFO buffer based on a circular queue.

```
   wire wr_en;
20
   // body
   // register file write operation
   always @(posedge clk)
      if (wr_en)
25       array_reg[w_ptr_reg] <= w_data;
   // register file read operation
   assign r_data = array_reg[r_ptr_reg];
   // write enabled only when FIFO is not full
   assign wr_en = wr & ~full_reg;
30
   // fifo control logic
   // register for read and write pointers
   always @(posedge clk, posedge reset)
      if (reset)
35       begin
            w_ptr_reg <= 0;
            r_ptr_reg <= 0;
            full_reg <= 1'b0;
            empty_reg <= 1'b1;
40       end
      else
         begin
            w_ptr_reg <= w_ptr_next;
            r_ptr_reg <= r_ptr_next;
45          full_reg <= full_next;
            empty_reg <= empty_next;
         end

   // next-state logic for read and write pointers
50 always @*
   begin
      // successive pointer values
      w_ptr_succ = w_ptr_reg + 1;
      r_ptr_succ = r_ptr_reg + 1;
55    // default: keep old values
      w_ptr_next = w_ptr_reg;
      r_ptr_next = r_ptr_reg;
      full_next = full_reg;
      empty_next = empty_reg;
60    case ({wr, rd})
         // 2'b00:  no op
         2'b01: // read
            if (~empty_reg) // not empty
               begin
65                r_ptr_next = r_ptr_succ;
                  full_next = 1'b0;
                  if (r_ptr_succ==w_ptr_reg)
                     empty_next = 1'b1;
               end
70       2'b10: // write
            if (~full_reg) // not full
```

```
                begin
                   w_ptr_next = w_ptr_succ;
                   empty_next = 1'b0;
75                 if (w_ptr_succ==r_ptr_reg)
                      full_next = 1'b1;
                end
          2'b11: // write and read
             begin
80              w_ptr_next = w_ptr_succ;
                r_ptr_next = r_ptr_succ;
             end
       endcase
    end
85
    // output
    assign full = full_reg;
    assign empty = empty_reg;

90 endmodule
```

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 6.2.1 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

```
module fifo_test
   (
    input wire clk, reset,
    input wire [1:0] btn,
5   input wire [2:0] sw,
    output wire [7:0] led
   );

   // signal declaration
10 wire [1:0] db_btn;

   // debounce circuit for btn[0]
   debounce btn_db_unit0
```

```
      (.clk(clk), .reset(reset), .sw(btn[0]),
15     .db_level(), .db_tick(db_btn[0]));
   // debounce circuit for btn[1]
   debounce btn_db_unit1
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(db_btn[1]));
20 // instantiate a 2^2-by-3 fifo
   fifo #(.B(3), .W(2)) fifo_unit
      (.clk(clk), .reset(reset),
      .rd(db_btn[0]), .wr(db_btn[1]), .w_data(sw),
      .r_data(led[2:0]), .full(led[7]), .empty(led[6]));
25 // disable unused leds
   assign led[5:3] = 3'b000;

endmodule
```

4.6 BIBLIOGRAPHIC NOTES

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

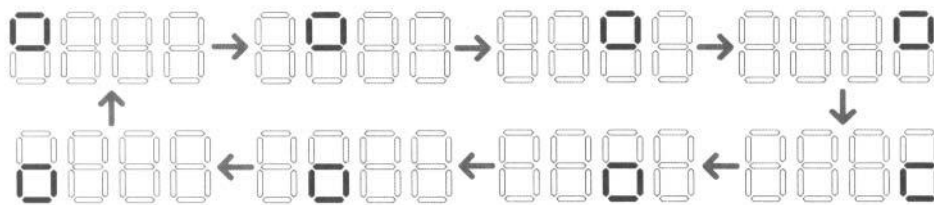


Figure 4.12 Pattern for Experiment 4.7.3.

Figure 4.13 Pattern for Experiment 4.7.4.

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.

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.