CHAPTER 14

VGA CONTROLLER II: TEXT

14.1 INTRODUCTION

A tile-mapped pixel generation scheme is discussed in Section 13.3. A tile can be considered as a "super pixel." Whereas a pixel is defined by a 3-bit word in a bit-mapped scheme, a tile is mapped to a predesigned pattern. One method of constructing a text display is to treat the characters as tiles and design the pixel generation circuit with the tile-mapped scheme. We discuss this method in this chapter and apply it to add scores and rules to the pong game.

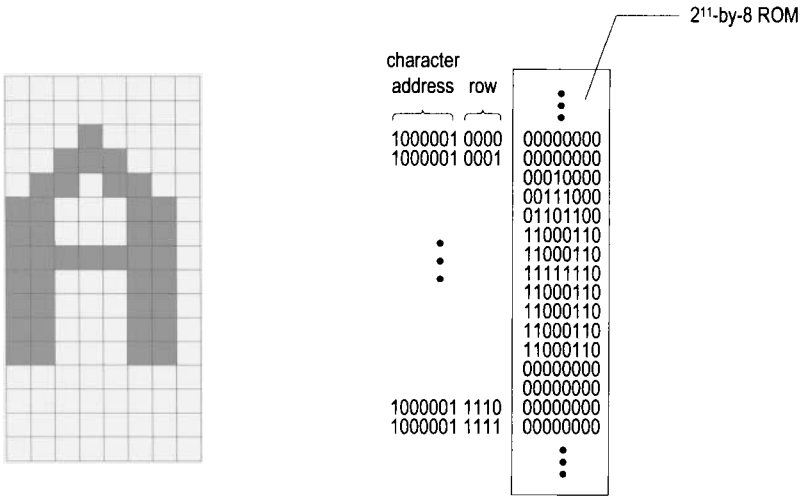
14.2 TEXT GENERATION

14.2.1 Character as a tile

When applying a tile-mapped scheme, we treat each character as a tile. In a bit-mapped scheme, the value of a pixel represents a 3-bit color. On the other hand, the value of a tile represents the code of a specific pattern. For the text display, we use the 7-bit ASCII code for the character tiles.

The patterns of the tiles constitute the *font* of the character set. A variety of fonts are available. We choose an 8-by-16 (i.e., 8-column-by-16-row) font similar to the one used in early IBM PCs. In this font, each character is represented as an 8-by-16 pixel pattern. The pattern for the letter "A" is shown in Figure 14.1(a).

The character patterns are stored in a ROM and each pattern requires $2^4 * 8$ bits. The pattern memory is known as *font ROM*. The original font set consists of 256 patterns,



(a) Pixel pattern (b) ROM content

Figure 14.1 Font pattern for the letter A.

including digits, upper- and lowercase letters, punctuation symbols, and many special-purpose graphic symbols. We implement only the first half [i.e., 128 (2^7)] of the patterns and exclude most graphic symbols. To accommodate this set, $2^7 * 2^4 * 8$ ROM bits are needed. It is usually configured as a 2^{11}-by-8 ROM.

When we use these 8-by-16 characters (i.e., tiles) in a 640-by-480 resolution screen, 80 (i.e., $\frac{640}{8}$) tiles can be fitted into a horizontal line and 30 (i.e., $\frac{480}{16}$) tiles can be fitted into a vertical line. In other words, the screen can be treated as an 80-by-25 tile screen. We can put characters on the screen using these scaled coordinates.

14.2.2 Font ROM

Our font set implements the 128 characters of the ASCII code, listed in Table 8.1. The 128 (2^7) character patterns can be accommodated by a 2^{11}-by-8 font ROM. In this ROM, the seven MSBs of the 11-bit address are used to identify the character, and the four LSBs of the address are used to identify the row within a character pattern. The address and ROM content for the letter "A" are shown in Figure 14.1(b).

In the ASCII table, the first column (ASCII codes 00_{16} to $1F_{16}$) consists of nonprintable control characters. The font ROM uses these codes to implement special graphic symbols. For example, the 06_{16} code will generate a spade pattern, ♠, on the screen. Note that the 00_{16} code is reserved for a blank tile.

The 2^{11}-by-8 font ROM can fit neatly into a single block RAM of the Spartan-3 device. We use the ROM template of Listing 12.6 to ensure that a block RAM will be inferred during synthesis. Part of the HDL code is shown in Listing 14.1. The complete code has 2^{11} rows in constant definition and the file can be downloaded from the companion Web site.

Listing 14.1 Partial code of the font ROM

```
module font_rom
   (
    input wire clk,
    input wire [10:0] addr,
    output reg [7:0] data
   );

   // signal declaration
   reg [10:0] addr_reg;

   // body
   always @(posedge clk)
      addr_reg <= addr;

   always @*
      case (addr_reg)
         //code x00 blank
         11'h000: data = 8'b00000000; //
         11'h001: data = 8'b00000000; //
         11'h002: data = 8'b00000000; //
         11'h003: data = 8'b00000000; //
         11'h004: data = 8'b00000000; //
         11'h005: data = 8'b00000000; //
         11'h006: data = 8'b00000000; //
         11'h007: data = 8'b00000000; //
         11'h008: data = 8'b00000000; //
         11'h009: data = 8'b00000000; //
         11'h00a: data = 8'b00000000; //
         11'h00b: data = 8'b00000000; //
         11'h00c: data = 8'b00000000; //
         11'h00d: data = 8'b00000000; //
         11'h00e: data = 8'b00000000; //
         11'h00f: data = 8'b00000000; //
         //code x01 smiley face
         11'h010: data = 8'b00000000; //
         11'h011: data = 8'b00000000; //
         11'h012: data = 8'b01111110; //  ******
         11'h013: data = 8'b10000001; // *      *
         11'h014: data = 8'b10100101; // * *  * *
         11'h015: data = 8'b10000001; // *      *
         11'h016: data = 8'b10000001; // *      *
         11'h017: data = 8'b10111101; // * **** *
         11'h018: data = 8'b10011001; // *  **  *
         11'h019: data = 8'b10000001; // *      *
         11'h01a: data = 8'b10000001; // *      *
         11'h01b: data = 8'b01111110; //  ******
         11'h01c: data = 8'b00000000; //
         11'h01d: data = 8'b00000000; //
         11'h01e: data = 8'b00000000; //
         11'h01f: data = 8'b00000000; //
         . . .
         //code x7f
```

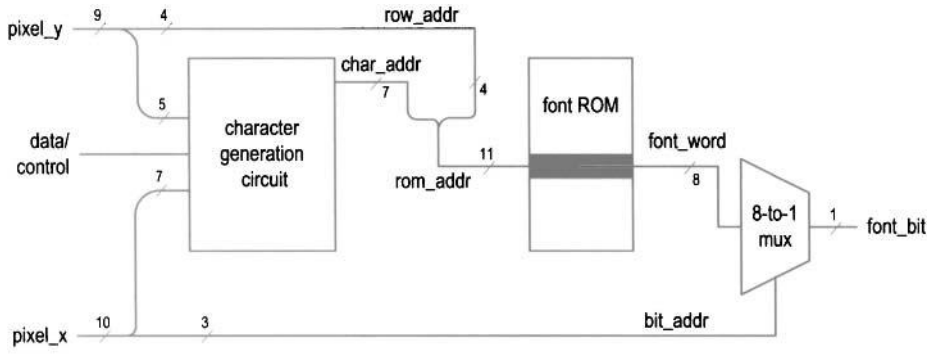


Figure 14.2 Two-stage text generation circuit.

```
            11'h7f0: data = 8'b00000000; //
            11'h7f1: data = 8'b00000000; //
            11'h7f2: data = 8'b00000000; //
            11'h7f3: data = 8'b00000000; //
            11'h7f4: data = 8'b00010000; //    *
            11'h7f5: data = 8'b00111000; //   ***
            11'h7f6: data = 8'b01101100; //  ** **
            11'h7f7: data = 8'b11000110; // **   **
            11'h7f8: data = 8'b11000110; // **   **
            11'h7f9: data = 8'b11000110; // **   **
            11'h7fa: data = 8'b11111110; // *******
            11'h7fb: data = 8'b00000000; //
            11'h7fc: data = 8'b00000000; //
            11'h7fd: data = 8'b00000000; //
            11'h7fe: data = 8'b00000000; //
            11'h7ff: data = 8'b00000000; //
        endcase

endmodule
```

Note that the block RAM-based ROM implementation introduces a one-clock-cycle delay, as discussed in Section 12.4.3.

14.2.3 Basic text generation circuit

The pixel generation circuit generates pixel values according to the current pixel coordinates (provided by the `pixel_x` and `pixel_y` signals) and the external data and control signals. Pixel generation based on a tile-mapped scheme involves two stages. The first stage uses the upper bits of the `pixel_x` and `pixel_y` signals to generate a tile's code, and the second stage uses this code and lower bits to generate the pixel's value.

The text generation circuit follows this method, and the basic diagram is shown in Figure 14.2. The screen is treated as a grid of 80-by-30 tiles, each containing an 8-by-16 font pattern. In the first stage, the `pixel_x[9:3]` and `pixel_y[8:4]` signals provide the x- and y-coordinates of the current tile location. The character generation circuit uses these coordinates, combined with other external data, to generate the value of this tile (labeled `char_addr`), which corresponds to a character's ASCII code. In the second stage, the ASCII

code becomes the seven MSBs of the address of the font ROM and specifies the location of the current pattern. It is concatenated with the four LSBs of the screen's y-coordinate (i.e., `pixel_y[3:0]`, labeled `row_addr`) to form the complete address (labeled `rom_addr`) of the font ROM. The output of the font ROM (labeled `font_word`) corresponds to an 8-bit row in the pattern. The three LSBs of the screen's x-coordinate (i.e., `pixel_x[2:0]`, labeled `bit_addr`) specify the desired pixel location, and an 8-to-1 multiplexer routes the pixel to the output.

14.2.4 Font display circuit

We use a simple font display circuit to verify operation of the font ROM and display all font patterns on the screen. The 128 patterns are arranged in four rows, which correspond to the four columns of the ASCII table in Table 8.1. We can obtain each pattern by using the proper x- and y-coordinates to generate the desired ASCII code, which is labeled the `char_addr` signal. The code segment is

```
assign char_addr = {pixel_y[5:4], pixel_x[7:3]};
```

The `pixel_x[7:3]` signal forms the five LSBs of the ASCII code, and thus 32 (2^5) consecutive font patterns will be displayed in a row. The `pixel_y[5:4]` signal forms the two MSBs of the ASCII code, and thus four consecutive rows will be displayed. Since the upper bits of the `pixel_x` and `pixel_y` signals are left unspecified, the 32-by-4 region will be displayed repetitively on the screen. An additional code segment is included to turn on the display for the top-left portion of the screen only. The complete code is shown in Listing 14.2.

Listing 14.2 Pixel generation of a font display circuit

```
module font_test_gen
   (
    input wire clk,
    input wire video_on,
    input wire [9:0] pixel_x, pixel_y,
    output reg [2:0] rgb_text
   );

   // signal declaration
   wire [10:0] rom_addr;
   wire [6:0] char_addr;
   wire [3:0] row_addr;
   wire [2:0] bit_addr;
   wire [7:0] font_word;
   wire font_bit, text_bit_on;

   // body
   // instantiate font ROM
   font_rom font_unit
      (.clk(clk), .addr(rom_addr), .data(font_word));
   // font ROM interface
   assign char_addr = {pixel_y[5:4], pixel_x[7:3]};
   assign row_addr = pixel_y[3:0];
   assign rom_addr = {char_addr, row_addr};
   assign bit_addr = pixel_x[2:0];
```

```
   assign font_bit = font_word[~bit_addr];
   // "on" region limited to top-left corner
   assign text_bit_on = (pixel_x[9:8]==0 && pixel_y[9:6]==0) ?
                        font_bit : 1'b0;
   // rgb multiplexing circuit
   always @*
      if (~video_on)
         rgb_text = 3'b000; // blank
      else
         if (text_bit_on)
            rgb_text = 3'b010;  // green
          else
            rgb_text = 3'b000;  // black

endmodule
```

The key part of the code is the font ROM interface. For clarity, we define the following signals for the font ROM, as shown in Figure 14.2:

- `char_addr`: 7 bits, the ASCII code of the character
- `row_addr`: 4 bits, the row number in a particular font pattern
- `rom_addr`: 11 bits, the address of the font ROM; the concatenation of `char_addr` and `row_addr`
- `bit_addr`: 3 bits, the column number in a particular font pattern
- `font_word`: 8 bits, a row of pixels of the font pattern specified by `rom_addr`
- `font_bit`: 1 bit, one pixel of `font_word` specified by `bit_addr`

The connection of these signals follows the diagram in Figure 14.2. The routing of the `font_bit` signal is done by a multiplexer, coded as an array with a dynamic index:

```
assign font_bit = font_word[~bit_addr];
```

Note that a row (i.e., a word) in the font ROM is defined in descending order (i.e., `[7:0]`). Since the screen's x-coordinate is defined in ascending fashion, in which the number increases from left to right, the order of the retrieved bits must be reversed. This is achieved by the ~ operator in the expression.

We need to combine the synchronization circuit and create the top-level description. The HDL code is shown in Listing 14.3.

Listing 14.3 Top-level description of a font display circuit

```
module font_test_top
   (
    input wire clk, reset,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;

   // body
   // instantiate vga sync circuit
```

```
    vga_sync vsync_unit
       (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
        .video_on(video_on), .p_tick(pixel_tick),
        .pixel_x(pixel_x), .pixel_y(pixel_y));
    // font generation circuit
    font_test_gen font_gen_unit
       (.clk(clk), .video_on(video_on), .pixel_x(pixel_x),
        .pixel_y(pixel_y), .rgb_text(rgb_next));
    // rgb buffer
    always @(posedge clk)
       if (pixel_tick)
          rgb_reg <= rgb_next;
    // output
    assign rgb = rgb_reg;

endmodule
```

There is subtle timing issue in this circuit. Because of the block RAM implementation, the font ROM's output suffers a one-clock-cycle delay. However, since the `pixel_tick` signal is asserted every two clock cycles, the `pixel_x` signal remains unchanged within this interval and the corresponding bit (i.e., `font_bit`) can be retrieved properly. The rgb multiplexing circuit can use this data, and the desired value is stored to the `rgb_reg` register in a timely manner.

14.2.5 Font scaling

In the tile-mapped scheme, we can scale a tile pattern to larger sizes by "enlarging" the screen pixels. For example, we can scale the 8-by-16 font to a 16-by-32 font by enlarging the original pixel four times (i.e., expanding one pixel to four pixels). To perform the scaling, we just need to shift pixel coordinates to the right 1 bit and discard the LSBs of the `pixel_x` and `pixel_y` signals. This can best be explained by an example. Let us repeat the previous font displaying circuit with enlarged 16-by-32 fonts. The screen can now be treated as a grid of 40-by-15 tiles. The new font addresses become

```
assign row_addr = pixel_y[4:1];
assign bit_addr = pixel_x[3:1];
assign char_addr = {pixel_y[6:5], pixel_x[8:4]};
```

The first two statements imply that the same `font_bit` value will be obtained when `pixel_x[0]` and `pixel_y[0]` are "00", "01", "10", and "11", and this effectively enlarges the original pixel to four pixels. The `text_bit_on` condition also needs to be modified to accommodate a larger region:

```
assign text_bit_on = (pixel_x[9]==0 && pixel_y[9:7]==0) ?
                     font_bit : 1'b0;
```

We can apply this scheme to scale up the font even further. Note that the enlarged fonts may appear jagged because they simply magnify the original pattern and introduce no new detail.

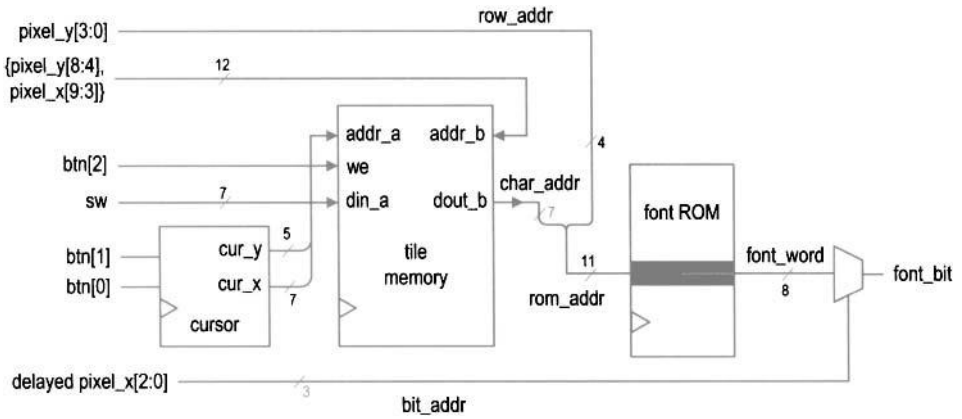


Figure 14.3 Text generation circuit with tile memory.

14.3 FULL-SCREEN TEXT DISPLAY

A full-screen text display, as the name indicates, uses the entire screen to display text characters. The character generation circuit now contains a *tile memory* that stores the ASCII code of each tile. The design of the tile memory is similar to the video memory of the bit-mapped circuit in Section 13.5. For easy memory access, we can concatenate the x- and y-coordinates of a tile to form the address. This translates to 12 bits for the 80-by-30 (i.e., 2^7-by-2^5) tile screen. Since each tile contains a 7-bit ASCII code, a 2^{12}-by-7 memory module is required. A synchronous dual-port RAM can be used for this purpose. A circuit with tile memory is shown in Figure 14.3.

Because accessing tile memory requires another clock cycle, retrieving a font pattern is now increased to two clock cycles. This prolonged delay introduces a subtle timing problem. Because the `pixel_x` signal is updated every two clock cycles, its value has incremented when the `font_word` value becomes available. Thus, when the bit is retrieved by the statements

```
assign bit_addr = pix_x2_reg[2:0];
assign font_bit = font_word[~bit_addr];
```

the incremented `bit_addr` is used and an incorrect font bit will be selected and routed to the output. One way to overcome the problem is to pass the `pixel_x` signal through two buffers and use this delayed signal in place of the `pixel_x` signal.

We use a simple circuit to demonstrate the design of the full-screen tile-mapped scheme. The circuit reads an ASCII code from a 7-bit switch and places it in the marked location of the 80-by-30 tile screen. The conceptual diagram is shown in Figure 14.3. A cursor is included to mark the current location of entry, where the color is reversed. The `cursor` block keeps track of the current location of the cursor. The circuit uses three pushbutton switches for control. Two buttons move the cursor right and down, respectively. The third button is for the write operation. When it is pressed, the current value of the 7-bit switch is written to the tile memory. The HDL code is shown in Listing 14.4.

Listing 14.4 Pixel generation of a full-screen text display

```
module text_screen_gen
   (
    input wire clk, reset,
    input wire video_on,
    input wire [2:0] btn,
    input wire [6:0] sw,
    input wire [9:0] pixel_x, pixel_y,
    output reg [2:0] text_rgb
   );

   // signal declaration
   // font ROM
   wire [10:0] rom_addr;
   wire [6:0] char_addr;
   wire [3:0] row_addr;
   wire [2:0] bit_addr;
   wire [7:0] font_word;
   wire font_bit;
   // tile RAM
   wire we;
   wire [11:0] addr_r, addr_w;
   wire [6:0] din, dout;
   // 80-by-30 tile map
   localparam MAX_X = 80;
   localparam MAX_Y = 30;
   // cursor
   reg [6:0] cur_x_reg;
   wire [6:0] cur_x_next;
   reg [4:0] cur_y_reg;
   wire [4:0]cur_y_next;
   wire move_x_tick, move_y_tick, cursor_on;
   // delayed pixel count
   reg [9:0] pix_x1_reg, pix_y1_reg;
   reg [9:0] pix_x2_reg, pix_y2_reg;
   // object output signals
   wire [2:0] font_rgb, font_rev_rgb;

   // body
   // instantiate debounce circuit for two buttons
   debounce deb_unit0
      (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(move_x_tick));
   debounce deb_unit1
      (.clk(clk), .reset(reset), .sw(btn[1]),
       .db_level(), .db_tick(move_y_tick));
   // instantiate font ROM
   font_rom font_unit
      (.clk(clk), .addr(rom_addr), .data(font_word));
   // instantiate dual-port video RAM (2^12-by-7)
   xilinx_dual_port_ram_sync
      #(.ADDR_WIDTH(12), .DATA_WIDTH(7)) video_ram
      (.clk(clk), .we(we), .addr_a(addr_w), .addr_b(addr_r),
```

```
        .din_a(din), .dout_a(), .dout_b(dout));

    // registers
    always @(posedge clk)
       begin
          cur_x_reg <= cur_x_next;
          cur_y_reg <= cur_y_next;
          pix_x1_reg <= pixel_x;
          pix_x2_reg <= pix_x1_reg;
          pix_y1_reg <= pixel_y;
          pix_y2_reg <= pix_y1_reg;
       end
    // tile RAM write
    assign addr_w = {cur_y_reg, cur_x_reg};
    assign we = btn[2];
    assign din = sw;
    // tile RAM read
    // use nondelayed coordinates to form tile RAM address
    assign addr_r = {pixel_y[8:4], pixel_x[9:3]};
    assign char_addr = dout;
    // font ROM
    assign row_addr = pixel_y[3:0];
    assign rom_addr = {char_addr, row_addr};
    // use delayed coordinate to select a bit
    assign bit_addr = pix_x2_reg[2:0];
    assign font_bit = font_word[~bit_addr];
    // new cursor position
    assign cur_x_next =
       (move_x_tick && (cur_x_reg==MAX_X-1)) ? 0 : // wrap
       (move_x_tick) ? cur_x_reg + 1 :
                       cur_x_reg;
    assign cur_y_next =
       (move_y_tick && (cur_x_reg==MAX_Y-1)) ? 0 : // wrap
       (move_y_tick) ? cur_y_reg + 1 :
                       cur_y_reg;
    // object signals
    // green over black and reversed video for cursor
    assign font_rgb = (font_bit) ? 3'b010 : 3'b000;
    assign font_rev_rgb = (font_bit) ? 3'b000 : 3'b010;
    // use delayed coordinates for comparison
    assign cursor_on = (pix_y2_reg[8:4]==cur_y_reg) &&
                       (pix_x2_reg[9:3]==cur_x_reg);
    // rgb multiplexing circuit
    always @*
       if (~video_on)
          text_rgb = 3'b000; // blank
       else
          if (cursor_on)
             text_rgb = font_rev_rgb;
           else
             text_rgb = font_rgb;
endmodule
```

The font ROM interface signals are similar to those in Listing 14.2 except that the char_addr is obtained from the read port of the tile memory. To facilitate the font ROM access delay, we create two delayed signals, pix_x2_reg and pix_y2_reg, from the current x- and y-coordinates, pixel_x and pixel_y. Note that the undelayed signals, pixel_x and pixel_y, are used to form the address to access the font ROM, but the delayed signal, pix_x2_reg, is used to obtain the font bit. The instantiation and interface of the dual-port tile RAM are similar to those of the video RAM in Listing 13.7.

The cursor_on signal is used to identify the current cursor location. The colors of the font pattern are reversed in this location. Because the font bits are delayed by two clocks, we use the delayed coordinates, pix_x2_reg and pix_y2_reg, for comparison.

The delayed font bits also introduce one pixel delay for the final rgb signal. This implies that the overall visible portion of the VGA monitor is shifted to the right by one pixel. To correct the problem, we should revise the vga_sync circuit and use the delayed pix_x2_reg and pix_y2_reg signals to generate the hsync and vsync signals. Since the shift has little effect on the overall video quality, we do not make this modification.

The top-level code combines the text pixel generation circuit and the synchronization circuit and is shown in Listing 14.5.

Listing 14.5 Top-level system of a full-screen text display

```
module text_screen_top
   (
    input wire clk, reset,
    input wire [2:0] btn,
    input wire [6:0] sw,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // signal declaration
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick;
   reg [2:0] rgb_reg;
   wire [2:0] rgb_next;
   // body
   // instantiate vga sync circuit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));
   // font generation circuit
   text_screen_gen text_gen_unit
      (.clk(clk), .reset(reset), .video_on(video_on),
       .btn(btn), .sw(sw), .pixel_x(pixel_x),
       .pixel_y(pixel_y), .text_rgb(rgb_next));
   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;
endmodule
```

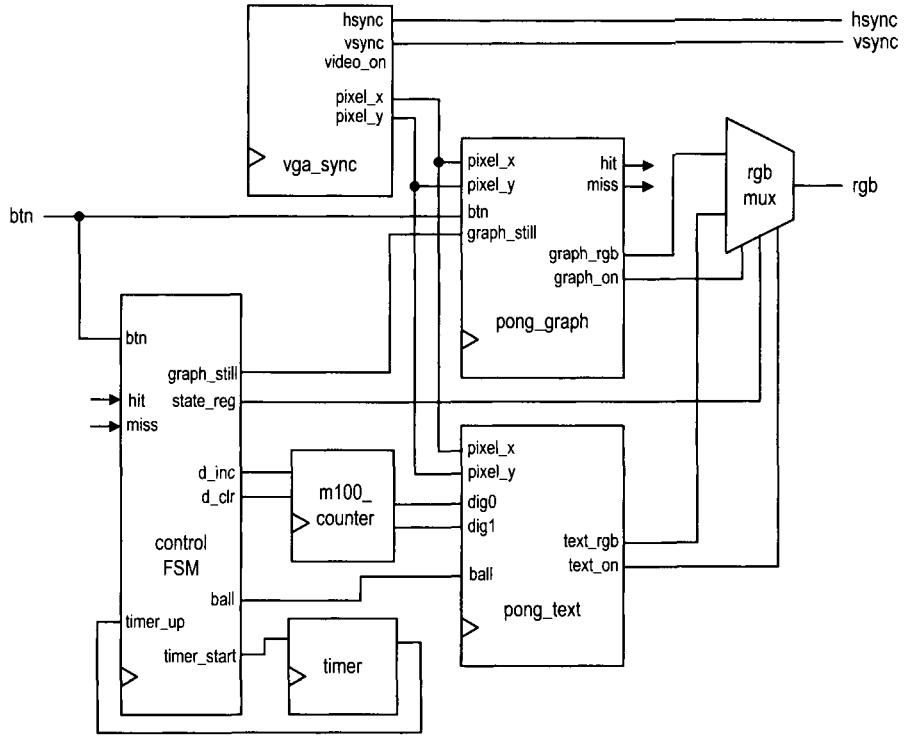


Figure 14.4 Top-level block diagram of the complete pong game.

14.4 THE COMPLETE PONG GAME

We create a free-running graphic circuit for the pong game in Section 13.4.3. In this section, we add a text interface to display scores and messages, and design a top-level control FSM that integrates the graphic and text subsystems and coordinates the overall circuit operation. The rules and operations of the complete game are:

- When the game starts, it displays the text of the rule.
- After a player presses a button, the game starts.
- The player scores a point each time hitting the ball with the paddle.
- When the player misses the ball, the game pauses and a new ball is provided. Three balls are provided in each session.
- The score and the number of remaining balls are displayed on the top of the screen.
- After three misses, the game is ended and displays the end-of-game message.

In the following subsections, we first discuss the text subsystem, graphic subsystem, and auxiliary counters, and then derive a top-level FSM to coordinate and control the overall operation. The conceptual diagram is shown in Figure 14.4.

14.4.1 Text subsystem

The text subsystem of the pong game consists of four text messages:

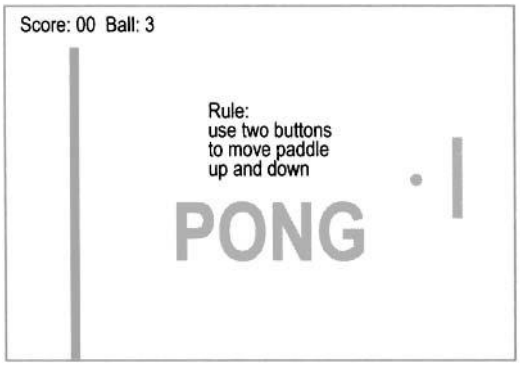


Figure 14.5 Text of the pong game.

- Display the score as "`Scores: DD`" and the number of remaining balls as"`Ball: D`" in the 16-by-32 font on top of the screen.
- Display the rule message "`Rules: Use two buttons to move paddle up or down.`" in the regular font at the beginning of the game.
- Display the "`PONG`" logo in the 64-by-128 font on the background.
- Display the end-of-game message "`Game Over`" in the 32-by-64 font at the end of the game.

A sketch of the first three messages is shown in Figure 14.5. The end-of-game message is overlapped with the rule message and not included.

Since these messages use different font sizes and are displayed at different occasions, they cannot be treated as a single screen. We treat each text message as an individual object and generate the on status signal and the font ROM address. For example, the logo message segment is

```
assign logo_on = (pix_y[9:7]==2) &&
                 (3<=pix_x[9:6]) && (pix_x[9:6]<=6);
assign row_addr_l = pix_y[6:3];
assign bit_addr_l = pix_x[5:3];
always @*
   case (pix_x[8:6])
      3'o3: char_addr_l = 7'h50;     // P
      3'o4: char_addr_l = 7'h4f;     // O
      3'o5: char_addr_l = 7'h4e;     // N
      default: char_addr_l = 7'h47; // G
   endcase
```

The `logo_on` signal indicates that the current scan is in the logo region and the corresponding text should be "turned on." The other statements specify the message content and the font ROM connections to generate the scaled 32-by-64 characters. The other three segments are similar. A separate multiplexing circuit examines various on signals and routes one set of addresses to the font ROM.

The text subsystem receives the score and the number of remaining balls via the `ball`, `dig0`, and `dig1` ports. It outputs the `rgb` information via the `rgb_text` port and outputs the on status information via the 4-bit `text_on` port, which is the concatenation of four individual on signals. The complete code is shown in Listing 14.6.

Listing 14.6 Text subsystem for the pong game

```
module pong_text
   (
    input wire clk,
    input wire [1:0] ball,
    input wire [3:0] dig0, dig1,
    input wire [9:0] pix_x, pix_y,
    output wire [3:0] text_on,
    output reg [2:0] text_rgb
   );

   // signal declaration
   wire [10:0] rom_addr;
   reg [6:0] char_addr, char_addr_s, char_addr_l,
             char_addr_r, char_addr_o;
   reg [3:0] row_addr;
   wire [3:0] row_addr_s, row_addr_l, row_addr_r, row_addr_o;
   reg [2:0] bit_addr;
   wire [2:0] bit_addr_s, bit_addr_l,bit_addr_r, bit_addr_o;
   wire [7:0] font_word;
   wire font_bit, score_on, logo_on, rule_on, over_on;
   wire [7:0] rule_rom_addr;

   // instantiate font ROM
   font_rom font_unit
      (.clk(clk), .addr(rom_addr), .data(font_word));

   //-------------------------------------------------
   // score region
   //  - display two-digit score, ball on top left
   //  - scale to 16-by-32 font
   //  - line 1, 16 chars: "Score:DD Ball:D"
   //-------------------------------------------------
   assign score_on = (pix_y[9:5]==0) && (pix_x[9:4]<16);
   assign row_addr_s = pix_y[4:1];
   assign bit_addr_s = pix_x[3:1];
   always @*
      case (pix_x[7:4])
         4'h0: char_addr_s = 7'h53; // S
         4'h1: char_addr_s = 7'h63; // c
         4'h2: char_addr_s = 7'h6f; // o
         4'h3: char_addr_s = 7'h72; // r
         4'h4: char_addr_s = 7'h65; // e
         4'h5: char_addr_s = 7'h3a; // :
          4'h6: char_addr_s = {3'b011, dig1}; // digit 10
          4'h7: char_addr_s = {3'b011, dig0}; // digit 1
          4'h8: char_addr_s = 7'h00; //
          4'h9: char_addr_s = 7'h00; //
          4'ha: char_addr_s = 7'h42; // B
          4'hb: char_addr_s = 7'h61; // a
          4'hc: char_addr_s = 7'h6c; // l
          4'hd: char_addr_s = 7'h6c; // l
          4'he: char_addr_s = 7'h3a; // :
```

```
            4'hf: char_addr_s = {5'b01100, ball};
        endcase
    //-------------------------------------------------
    // logo region:
    //   - display logo "PONG" at top center
    //   - used as background
    //   - scale to 64-by-128 font
    //-------------------------------------------------
    assign logo_on = (pix_y[9:7]==2) &&
                     (3<=pix_x[9:6]) && (pix_x[9:6]<=6);
    assign row_addr_l = pix_y[6:3];
    assign bit_addr_l = pix_x[5:3];
    always @*
        case (pix_x[8:6])
            3'o3: char_addr_l = 7'h50; // P
            3'o4: char_addr_l = 7'h4f; // O
            3'o5: char_addr_l = 7'h4e; // N
            default: char_addr_l = 7'h47; // G
        endcase
    //-------------------------------------------------
    // rule region
    //   - display rule (4-by-16 tiles)on center
    //   - rule text:
    //       Rule:
    //         Use two buttons
    //         to move paddle
    //         up and down
    //-------------------------------------------------
    assign rule_on = (pix_x[9:7]==2) && (pix_y[9:6]==2);
    assign row_addr_r = pix_y[3:0];
    assign bit_addr_r = pix_x[2:0];
    assign rule_rom_addr = {pix_y[5:4], pix_x[6:3]};
    always @*
        case (rule_rom_addr)
            // row 1
            6'h00: char_addr_r = 7'h52; // R
            6'h01: char_addr_r = 7'h55; // U
            6'h02: char_addr_r = 7'h4c; // L
            6'h03: char_addr_r = 7'h45; // E
            6'h04: char_addr_r = 7'h3a; // :
            6'h05: char_addr_r = 7'h00; //
            6'h06: char_addr_r = 7'h00; //
            6'h07: char_addr_r = 7'h00; //
            6'h08: char_addr_r = 7'h00; //
            6'h09: char_addr_r = 7'h00; //
            6'h0a: char_addr_r = 7'h00; //
            6'h0b: char_addr_r = 7'h00; //
            6'h0c: char_addr_r = 7'h00; //
            6'h0d: char_addr_r = 7'h00; //
            6'h0e: char_addr_r = 7'h00; //
            6'h0f: char_addr_r = 7'h00; //
            // row 2
            6'h10: char_addr_r = 7'h55; // U
```

```
            6'h11: char_addr_r = 7'h73; // s
            6'h12: char_addr_r = 7'h65; // e
            6'h13: char_addr_r = 7'h00; //
            6'h14: char_addr_r = 7'h74; // t
            6'h15: char_addr_r = 7'h77; // w
            6'h16: char_addr_r = 7'h6f; // o
            6'h17: char_addr_r = 7'h00; //
            6'h18: char_addr_r = 7'h62; // b
            6'h19: char_addr_r = 7'h75; // u
            6'h1a: char_addr_r = 7'h74; // t
            6'h1b: char_addr_r = 7'h74; // t
            6'h1c: char_addr_r = 7'h6f; // o
            6'h1d: char_addr_r = 7'h6e; // n
            6'h1e: char_addr_r = 7'h73; // s
            6'h1f: char_addr_r = 7'h00; //
            // row 3
            6'h20: char_addr_r = 7'h74; // t
            6'h21: char_addr_r = 7'h6f; // o
            6'h22: char_addr_r = 7'h00; //
            6'h23: char_addr_r = 7'h6d; // m
            6'h24: char_addr_r = 7'h6f; // o
            6'h25: char_addr_r = 7'h76; // v
            6'h26: char_addr_r = 7'h65; // e
            6'h27: char_addr_r = 7'h00; //
            6'h28: char_addr_r = 7'h70; // p
            6'h29: char_addr_r = 7'h61; // a
            6'h2a: char_addr_r = 7'h64; // d
            6'h2b: char_addr_r = 7'h64; // d
            6'h2c: char_addr_r = 7'h6c; // l
            6'h2d: char_addr_r = 7'h65; // e
            6'h2e: char_addr_r = 7'h00; //
            6'h2f: char_addr_r = 7'h00; //
            // row 4
            6'h30: char_addr_r = 7'h75; // u
            6'h31: char_addr_r = 7'h70; // p
            6'h32: char_addr_r = 7'h00; //
            6'h33: char_addr_r = 7'h61; // a
            6'h34: char_addr_r = 7'h6e; // n
            6'h35: char_addr_r = 7'h64; // d
            6'h36: char_addr_r = 7'h00; //
            6'h37: char_addr_r = 7'h64; // d
            6'h38: char_addr_r = 7'h6f; // o
            6'h39: char_addr_r = 7'h77; // w
            6'h3a: char_addr_r = 7'h6e; // n
            6'h3b: char_addr_r = 7'h2e; // .
            6'h3c: char_addr_r = 7'h00; //
            6'h3d: char_addr_r = 7'h00; //
            6'h3e: char_addr_r = 7'h00; //
            6'h3f: char_addr_r = 7'h00; //
        endcase
    //-------------------------------------------------------------
    // game over region
    //   - display "Game Over" at center
```

```
   // - scale to 32-by-64 fonts
160 //------------------------------------------------
   assign over_on = (pix_y[9:6]==3) &&
                    (5<=pix_x[9:5]) && (pix_x[9:5]<=13);
   assign row_addr_o = pix_y[5:2];
   assign bit_addr_o = pix_x[4:2];
165 always @*
      case(pix_x[8:5])
         4'h5: char_addr_o = 7'h47; // G
         4'h6: char_addr_o = 7'h61; // a
         4'h7: char_addr_o = 7'h6d; // m
170      4'h8: char_addr_o = 7'h65; // e
         4'h9: char_addr_o = 7'h00; //
         4'ha: char_addr_o = 7'h4f; // O
         4'hb: char_addr_o = 7'h76; // v
         4'hc: char_addr_o = 7'h65; // e
175      default: char_addr_o = 7'h72; // r
      endcase
   //------------------------------------------------
   // mux for font ROM addresses and rgb
   //------------------------------------------------
180 always @*
   begin
      text_rgb = 3'b110;  // background, yellow
      if (score_on)
         begin
185         char_addr = char_addr_s;
            row_addr = row_addr_s;
            bit_addr = bit_addr_s;
            if (font_bit)
               text_rgb = 3'b001;
190      end
      else if (rule_on)
         begin
            char_addr = char_addr_r;
            row_addr = row_addr_r;
195         bit_addr = bit_addr_r;
            if (font_bit)
               text_rgb = 3'b001;
         end
      else if (logo_on)
200      begin
            char_addr = char_addr_l;
            row_addr = row_addr_l;
            bit_addr = bit_addr_l;
            if (font_bit)
205            text_rgb = 3'b011;
         end
      else // game over
         begin
            char_addr = char_addr_o;
210         row_addr = row_addr_o;
            bit_addr = bit_addr_o;
```

```
            if (font_bit)
               text_rgb = 3'b001;
         end
   end

   assign text_on = {score_on, logo_on, rule_on, over_on};
   //-------------------------------------------------
   // font rom interface
   //-------------------------------------------------
   assign rom_addr = {char_addr, row_addr};
   assign font_bit = font_word[~bit_addr];

endmodule
```

The structure of each segment is similar. Because the messages are short, they are coded with the regular ROM template. Since no clock signal is used, a distributed RAM or combinational logic should be inferred. Generation of the two-digit score depends on the two 4-bit external signals, `dig0` and `dig1`. Note that the ASCII codes for the digits 0, 1, ..., 9, are 30_{16}, 31_{16}, ..., 39_{16}. We can generate the `char_addr` signal simply by concatenating `"011"` in front of `dig0` and `dig1`.

14.4.2 Modified graphic subsystem

To accommodate the new top-level controller, the graphic circuit in Section 13.4.3 requires several modifications:

- Add a `gra_still` (for "still graphics") control signal. When it is asserted, the vertical bar is placed in the middle and the ball is placed at the center of the screen without movement.
- Add the `hit` and `miss` status signals. The `hit` signal is asserted for one clock cycle when the paddle hits the ball. The `miss` signal is asserted when the paddle misses the ball and the ball reaches the right border.
- Add a `graph_on` signal to indicate the on status of the graph subsystem.

The modified portion of the code is shown in Listing 14.7.

Listing 14.7 Modified portion of a graph subsystem for the pong game

```
   . . .
   // new ball position
   assign ball_x_next = (gra_still) ? MAX_X/2 :
                        (refr_tick) ? ball_x_reg+x_delta_reg :
                        ball_x_reg ;
   assign ball_y_next = (gra_still) ? MAX_Y/2 :
                        (refr_tick) ? ball_y_reg+y_delta_reg :
                        ball_y_reg ;
   // new ball velocity
   always @*
   begin
      hit = 1'b0;
      miss = 1'b0;
      x_delta_next = x_delta_reg;
      y_delta_next = y_delta_reg;
      if (gra_still)     // initial velocity
```

```
         begin
            x_delta_next = BALL_V_N;
            y_delta_next = BALL_V_P;
         end
      else if (ball_y_t < 1) // reach top
         y_delta_next = BALL_V_P;
      else if (ball_y_b > (MAX_Y-1)) // reach bottom
         y_delta_next = BALL_V_N;
      else if (ball_x_l <= WALL_X_R) // reach wall
         x_delta_next = BALL_V_P;     // bounce back
      else if ((BAR_X_L<=ball_x_r) && (ball_x_r<=BAR_X_R) &&
               (bar_y_t<=ball_y_b) && (ball_y_t<=bar_y_b))
         begin
            // reach x of right bar and hit, ball bounce back
            x_delta_next = BALL_V_N;
            hit = 1'b1;
         end
      else if (ball_x_r>MAX_X)    // reach right border
         miss = 1'b1;             // a miss
   end
   . . .
   assign graph_on = wall_on | bar_on | rd_ball_on;
   . . .
```

14.4.3 Auxiliary counters

The top-level design requires two small utility modules, `m100_counter` and `timer`, to facilitate the counting. The `m100_counter` module is a two-digit decade counter that counts from 00 to 99 and is used to keep track of the scores of the game. Two control signals, `d_inc` and `d_clr`, increment and clear the counter, respectively. The code is shown in Listing 14.8.

Listing 14.8 Two-digit decade counter

```
module m100_counter
   (
    input wire clk, reset,
    input wire d_inc, d_clr,
    output wire [3:0] dig0, dig1
   );

   // signal declaration
   reg [3:0] dig0_reg, dig1_reg, dig0_next, dig1_next;

   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            dig1_reg <= 0;
            dig0_reg <= 0;
         end
      else
         begin
```

```
            dig1_reg <= dig1_next;
            dig0_reg <= dig0_next;
         end

   // next-state logic
   always @*
   begin
      dig0_next = dig0_reg;
      dig1_next = dig1_reg;
      if (d_clr)
         begin
            dig0_next = 0;
            dig1_next = 0;
         end
      else if (d_inc)
         if (dig0_reg==9)
            begin
               dig0_next = 0;
               if (dig1_reg==9)
                  dig1_next = 0;
               else
                  dig1_next = dig1_reg + 1;
             end
         else  // dig0 not 9
            dig0_next = dig0_reg + 1;
   end
   // output
   assign dig0 = dig0_reg;
   assign dig1 = dig1_reg;

endmodule
```

The `timer` module uses the 60-Hz tick, `timer_tick`, to generate a 2-second interval. Its purpose is to pause the video for a small interval between transitions of the screens. It starts counting when the `timer_start` signal is asserted and activates the `timer_up` signal when the 2-second interval is up. The code is shown in Listing 14.9.

Listing 14.9 Two-second timer

```
module timer
   (
    input wire clk, reset,
    input wire timer_start, timer_tick,
    output wire timer_up
   );

   // signal declaration
   reg [6:0] timer_reg, timer_next;

   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         timer_reg <= 7'b1111111;
      else
```

```
      timer_reg <= timer_next;

   // next-state logic
   always @*
      if (timer_start)
         timer_next = 7'b1111111;
      else if ((timer_tick) && (timer_reg != 0))
         timer_next = timer_reg - 1;
      else
         timer_next = timer_reg;
   // output
   assign timer_up = (timer_reg==0);

endmodule
```

14.4.4 Top-level system

The top-level system of the pong game consists of the previously designed modules, including a video synchronization circuit, graphic subsystem, text subsystem, and utility counters, as well as a control FSM and an rgb multiplexing circuit. The block diagram is shown in Figure 14.4.

The control FSM monitors overall system operation and coordinates the activities of the text and graphic subsystems. Its ASMD chart is shown in Figure 14.6. The FSM has four states and operates as follows:

- Initially, the FSM is in the newgame state. The game starts when a button is pressed and the FSM moves to the play state.
- In the play state, the FSM checks the hit and miss signals continuously. When the hit signal is activated, the d_inc signal is asserted for one clock cycle to increment the score counter. When the miss signal is asserted, the FSM activates the 2-second timer, decrements the number of the balls by 1, and examines the number of remaining balls. If it is zero, the game is ended and the FSM moves to the over state. Otherwise, the FSM moves to the newball state.
- The FSM waits in the newball state until the 2-second interval is up (i.e., when the timer_up signal is asserted) and a button is pressed. It then moves to the play state to continue the game.
- The FSM stays in the over state until the 2-second interval is up. It then moves to the newgame state for a new game.

The rgb multiplexing circuit routes the text_rgb or graph_rgb signals to output according to the text_on and graphic_on signals. The key segment is

```
always @*
   if (~video_on)
      rgb_next = "000"; // blank the edge/retrace
   else
      // display score, rule, or game over
      if ( text_on[3] ||
           ((state_reg==newgame) && text_on[1]) ||
           ((state_reg==over) && text_on[0]) )
         rgb_next = text_rgb;
      else if (graph_on)  // display graph
```

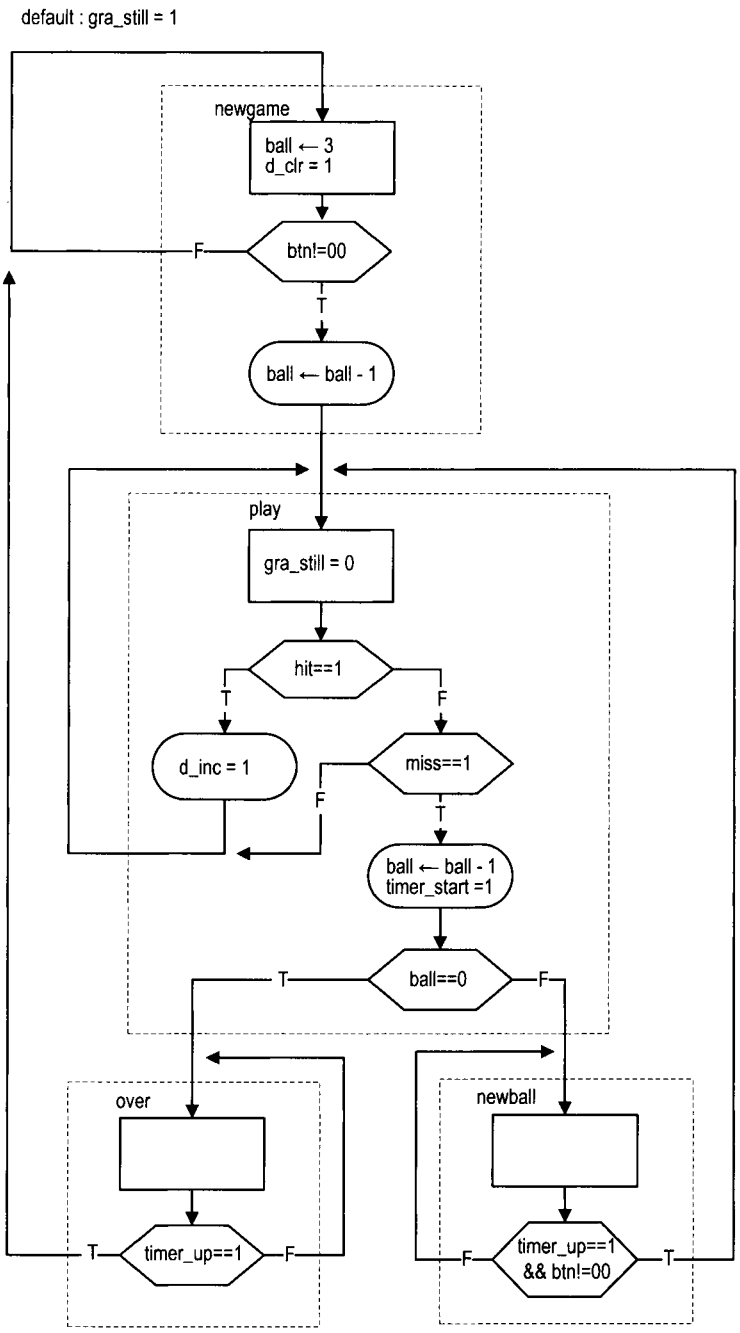


Figure 14.6 ASMD chart of the pong controller.

```
            rgb_next = graph_rgb;
         else if (text_on[2]) // display logo
            rgb_next = text_rgb;
         else
            rgb_next = 3'b110; // yellow background
   // output
   assign rgb = rgb_reg;
```

The `text_on[3]` signal is for the scores, which is always displayed. The `text_on[1]` signal is for the rule, which is displayed only when the FSM is in the `newgame` state. Similarly, the end-of-game message, whose status is indicated by the `text_on[0]` signal, is displayed only when the FSM is in the `over` state. The logo, whose status is indicated by the `text_on[2]` signal, is used as part of the background and is displayed only when no other on signal is asserted.

The complete code is shown in Listing 14.10.

Listing 14.10 Top-level system for the pong game

```
module pong_top
   (
    input wire clk, reset,
    input wire [1:0] btn,
    output wire hsync, vsync,
    output wire [2:0] rgb
   );

   // symbolic state declaration
   localparam  [1:0]
      newgame = 2'b00,
      play    = 2'b01,
      newball = 2'b10,
      over    = 2'b11;

   // signal declaration
   reg [1:0] state_reg, state_next;
   wire [9:0] pixel_x, pixel_y;
   wire video_on, pixel_tick, graph_on, hit, miss;
   wire [3:0] text_on;
   wire [2:0] graph_rgb, text_rgb;
   reg [2:0] rgb_reg, rgb_next;
   wire [3:0] dig0, dig1;
   reg gra_still, d_inc, d_clr, timer_start;
   wire timer_tick, timer_up;
   reg [1:0] ball_reg, ball_next;

   //=============================================================
   // instantiation
   //=============================================================
   // instantiate video synchronization unit
   vga_sync vsync_unit
      (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(pixel_tick),
       .pixel_x(pixel_x), .pixel_y(pixel_y));
   // instantiate text module
```

```
pong_text text_unit
   (.clk(clk),
    .pix_x(pixel_x), .pix_y(pixel_y),
    .dig0(dig0), .dig1(dig1), .ball(ball_reg),
    .text_on(text_on), .text_rgb(text_rgb));
// instantiate graph module
pong_graph graph_unit
   (.clk(clk), .reset(reset), .btn(btn),
    .pix_x(pixel_x), .pix_y(pixel_y),
    .gra_still(gra_still), .hit(hit), .miss(miss),
    .graph_on(graph_on), .graph_rgb(graph_rgb));
// instantiate 2 sec timer
// 60 Hz tick
assign timer_tick = (pixel_x==0) && (pixel_y==0);
timer timer_unit
   (.clk(clk), .reset(reset), .timer_tick(timer_tick),
    .timer_start(timer_start), .timer_up(timer_up));
// instantiate 2-digit decade counter
m100_counter counter_unit
   (.clk(clk), .reset(reset), .d_inc(d_inc), .d_clr(d_clr),
    .dig0(dig0), .dig1(dig1));
//=============================================================
// FSMD
//=============================================================
// FSMD state & data registers
 always @(posedge clk, posedge reset)
    if (reset)
       begin
          state_reg <= newgame;
          ball_reg <= 0;
          rgb_reg <= 0;
       end
    else
       begin
         state_reg <= state_next;
         ball_reg <= ball_next;
         if (pixel_tick)
            rgb_reg <= rgb_next;
       end
// FSMD next-state logic
always @*
begin
   gra_still = 1'b1;
   timer_start = 1'b0;
   d_inc = 1'b0;
   d_clr = 1'b0;
   state_next = state_reg;
   ball_next = ball_reg;
   case (state_reg)
      newgame:
         begin
            ball_next = 2'b11; // three balls
            d_clr = 1'b1;      // clear score
```

```
90                if (btn != 2'b00)  // button pressed
                     begin
                        state_next = play;
                        ball_next = ball_reg - 1;
                     end
95             end
            play:
               begin
                  gra_still = 1'b0;  // animated screen
                  if (hit)
100                  d_inc = 1'b1;   // increment score
                  else if (miss)
                     begin
                        if (ball_reg==0)
                           state_next = over;
105                     else
                           state_next = newball;
                        timer_start = 1'b1;  // 2 sec timer
                        ball_next = ball_reg - 1;
                     end
110            end
            newball:
               // wait for 2 sec and until button pressed
               if (timer_up && (btn != 2'b00))
                    state_next = play;
115         over:
               // wait for 2 sec to display game over
               if (timer_up)
                    state_next = newgame;
         endcase
120   end
   //=======================================================
   // rgb multiplexing circuit
   //=======================================================
   always @*
125   if (~video_on)
         rgb_next = "000"; // blank the edge/retrace
      else
         // display score, rule, or game over
         if (text_on[3] ||
130            ((state_reg==newgame) && text_on[1]) || // rule
               ((state_reg==over) && text_on[0]))
            rgb_next = text_rgb;
         else if (graph_on)  // display graph
           rgb_next = graph_rgb;
135      else if (text_on[2]) // display logo
           rgb_next = text_rgb;
         else
           rgb_next = 3'b110; // yellow background
   // output
140 assign rgb = rgb_reg;
endmodule
```

14.5 BIBLIOGRAPHIC NOTES

Several other character fonts are available. *Rapid Prototyping of Digital Systems* by James O. Hamblen et al. uses a compact 64-character 8-by-8 font set. The tile-mapped scheme is not limited to the text display. It is widely used in the early video game. The article "Computer Graphics During the 8-Bit Computer Game Era" by Steven Collins (*ACM SIGGRAPH*, May 1998) provides a comprehensive review of the history and design techniques of the tile-based game.

14.6 SUGGESTED EXPERIMENTS

14.6.1 Rotating banner

A rotating banner on the monitor screen moves a line from right to left and then wraps around. It is similar to the Window's Marquee screen saver. Let the text on the banner be "Hello, FPGA World." The banner should be displayed in four different font sizes and can travel at four different speeds. The font size and speed are controlled by four switches. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.2 Underline for the cursor

The full-screen text display circuit in Section 14.3 uses reversed color to indicate the current cursor location. Modify the design to use an underline to indicate the cursor location. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.3 Dual-mode text display

It is sometimes better for text to be displayed on a "vertical" screen. This can be done by turning the monitor 90 degrees and resting it on its side. Design this circuit as follows:

1. Modify the full-screen text display circuit in Section 14.3 for a vertical screen.
2. Merge the normal and vertical designs to create a "dual-mode" text display. Use a switch to select the desired mode.
3. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.4 Keyboard text entry

Instead of switches and buttons, it is more natural to use a keyboard to enter text. We can use the four arrow keys to move the cursor and use the regular keys to enter the characters. Use the keyboard interface discussed in Section 9.4 to design the new circuit. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.5 UART terminal

The UART terminal receives input from the UART port and displays the received characters on a monitor. When connected to the PC's serial port, it should echo the text on Window's HypterTerminal. The detailed specifications are:

- A cursor is used to indicate the current location.
- The screen starts a new line when a "carriage return" code ($0d_{16}$) is received.

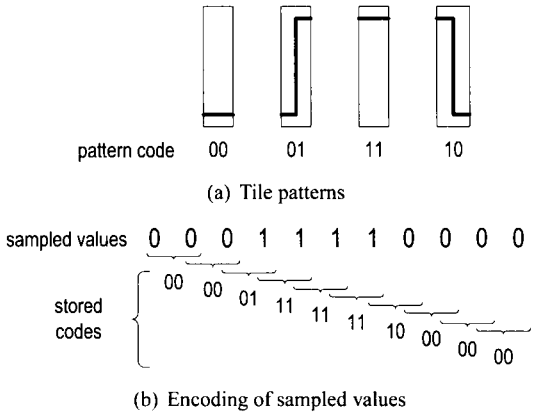


Figure 14.7 Tile patterns and encoding of a square wave.

- A line wraps around (i.e., starts a new line) after 80 characters.
- When the cursor reaches the bottom of the screen (i.e., the last line), the first line will be discarded and all other lines move up (i.e., scroll up) one position.

Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.6 Square-wave display

We can draw a square wave by using the four simple tile patterns shown in Figure 14.7(a). Follow the procedure of a full-screen text display in Section 14.3 to design a full-screen wave editor:

1. Let the tile size be 8 columns by 64 rows. Create a pattern ROM for the four patterns.
2. Calculate the number of tiles on a 640-by-480 resolution screen and derive the proper configuration for the tile memory.
3. Use three pushbuttons for control and a 2-bit switch to enter the pattern.
4. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.7 Simple four-trace logic analyzer

A logic analyzer displays the waveforms of a collection of digital signals. We want to design a simple logic analyzer that captures the waveforms of four input signals in "free-running" mode. Instead of using a trigger pattern, data capture is initiated with activation of a pushbutton switch. For simplicity, we assume that the frequencies of the input waveform are between 10 kHz and 100 kHz. The circuit can be designed as follows:

1. Use a sampling tick to sample the four input signals. Make sure to select a proper rate so that the desired input frequency range can be displayed properly on the screen.
2. For a point in the sampled signal, its value can be encoded as a tile pattern by including the value of the previous point. For example, if the sampled sequence of one signal is "00001111000", the tile patterns become "00 00 00 01 11 11 11 10 00 00", as shown in Figure 14.7(b).
3. Follow the procedure of the preceding square-wave experiment to design the tile memory and video interface to display the four waveforms being stored .
4. Derive the HDL description and then synthesize the circuit.

To verify operation of the circuit, we can connect four external signals via headers around the prototyping board. Alternatively, we can create a top-level test module that includes a 4-bit counter (say, a mod-10 counter around 50 kHz) and the logic analyzer, resynthesize the circuit, and verify its operation.

14.6.8 Complete two-player pong game

The free-running two-player pong game is described in Experiment 13.7.6. Follow the procedure of the pong game in Section 14.4 to derive the complete system. This should include the design of a new text display subsystem and the design of a top-level FSM controller. Derive the HDL description and then synthesize and verify operation of the circuit.

14.6.9 Complete breakout game

The free-running breakout game is described in Experiment 13.7.7. Follow the procedure of the pong game in Section 14.4 to derive the complete system. This should include the design of a new text display subsystem and the design of a top-level FSM controller. Derive the HDL description and then synthesize and verify operation of the circuit.