CHAPTER 13

VGA CONTROLLER I: GRAPHIC

13.1 INTRODUCTION

VGA (video graphics array) is a video display standard introduced in the late 1980s in IBM PCs and is widely supported by PC graphics hardware and monitors. We discuss the design of a basic eight-color 640-by-480 resolution interface for CRT (cathode ray tube) monitors in this book. CRT synchronization and basic graphic processing are examined in this chapter, and text generation is discussed in Chapter 14.

13.1.1 Basic operation of a CRT

The conceptual sketch of a monochrome CRT monitor is shown in Figure 13.1. The electron gun (cathode) generates a focused electron beam, which traverses a vacuum tube and eventually hits the phosphorescent screen. Light is emitted at the instant that electrons hit a phosphor dot on the screen. The intensity of the electron beam and the brightness of the dot are determined by the voltage level of the external video input signal, labeled `mono` in Figure 13.1. The `mono` signal is an analog signal whose voltage level is between 0 and 0.7 V.

A vertical deflection coil and a horizontal deflection coil outside the tube produce magnetic fields to control how the electron beam travels and to determine where on the screen the electrons hit. In today's monitors, the electron beam traverses (i.e., scans) the screen systematically in a fixed pattern, from left to right and from top to bottom, as shown in Figure 13.2.

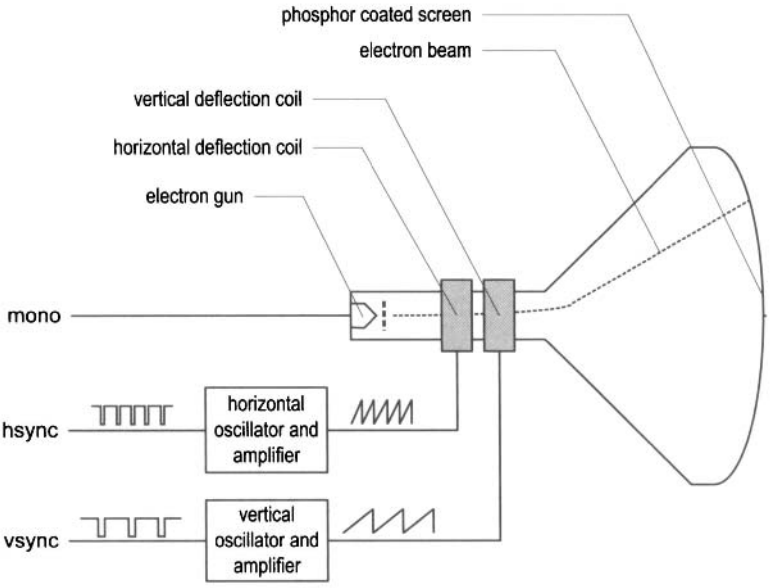


Figure 13.1 Conceptual diagram of a CRT monitor.

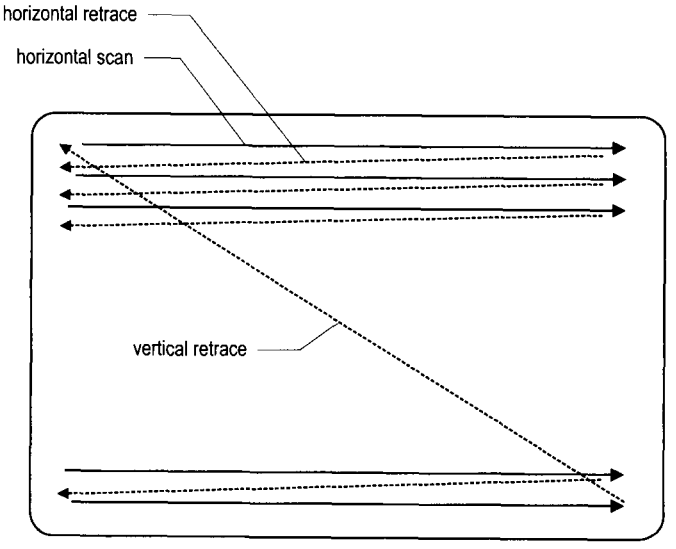


Figure 13.2 CRT scanning pattern.

Table 13.1 Three-bit VGA color combinations

Red (R)	Green (G)	Blue (B)	Resulting color
0	0	0	black
0	0	1	blue
0	1	0	green
0	1	1	cyan
1	0	0	red
1	0	1	magenta
1	1	0	yellow
1	1	1	white

The monitor's internal oscillators and amplifiers generate sawtooth waveforms to control the two deflection coils. For example, the electron beam moves from the left edge to the right edge as the voltage applied to the horizontal deflection coil gradually increases. After reaching the right edge, the beam returns rapidly to the left edge (i.e., *retraces*) when the voltage changes to 0. The relationship between the sawtooth waveform and the scan is shown in Figure 13.4. Two external synchronization signals, `hsync` and `vsync`, control generation of the sawtooth waveforms. These signals are digital signals. The relationship between the `hsync` signal and the horizontal sawtooth is also shown in Figure 13.4. Note that the "1" and "0" periods of the `hsync` signal correspond to the rising and falling ramps of the sawtooth waveform.

The basic operation of a color CRT is similar except that it has three electron beams, which are projected to the red, green, and blue phosphor dots on the screen. The three dots are combined to form a pixel. We can adjust the voltage levels of the three video input signals to obtain the desired pixel color.

13.1.2 VGA port of the S3 board

The VGA port has five active signals, including the horizontal and vertical synchronization signals, `hsync` and `vsync`, and three video signals for the red, green, and blue beams. It is physically connected to a 15-pin D-subminiature connector. A video signal is an analog signal and the video controller uses a digital-to-analog converter to convert the digital output to the desired analog level. If a video signal is represented by an N-bit word, it can be converted to 2^N analog levels. The three video signals can generate 2^{3N} different colors. This is also known as *3N-bit color* since a color is defined by $3N$ bits. In the S3 board, a 1-bit word is used for each video signal, and this leads to only eight (i.e., 2^3) possible colors. The possible color combinations are shown in Table 13.1. If we use the same 1-bit signal to drive the video signals, they become either "000" or "111" and the monitor functions as a black-and-white monochrome monitor.

13.1.3 Video controller

A video controller generates the synchronization signals and outputs data pixels serially. A simplified block diagram of a VGA controller is shown in Figure 13.3. It contains a synchronization circuit, labeled `vga_sync`, and a pixel generation circuit.

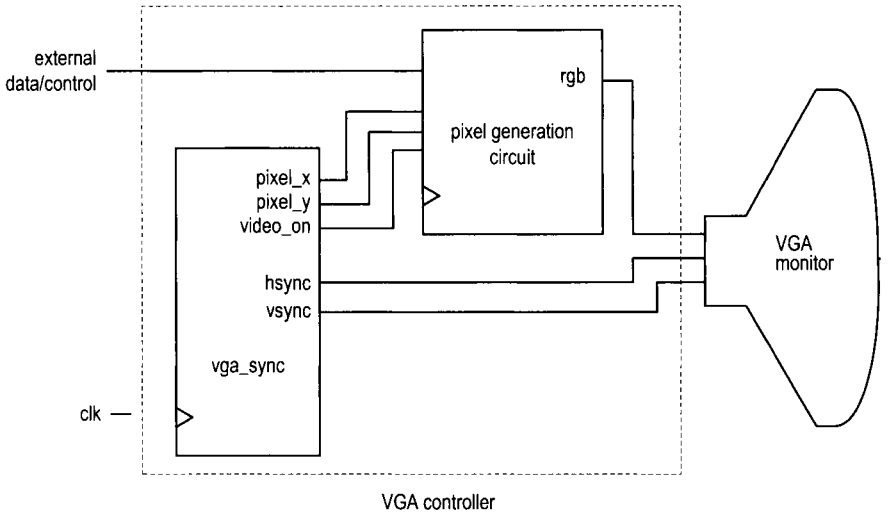


Figure 13.3 Simplified block diagram of a VGA controller.

The vga_sync circuit generates timing and synchronization signals. The hsync and vsync signals are connected to the VGA port to control the horizontal and vertical scans of the monitor. The two signals are decoded from the internal counters, whose outputs are the pixel_x and pixel_y signals. The pixel_x and pixel_y signals indicate the relative positions of the scans and essentially specify the location of the current pixel. The vga_sync circuit also generates the video_on signal to indicate whether to enable or disable the display. The design of this circuit is discussed in Section 13.2.

The pixel generation circuit generates the three video signals, which are collectively referred to as the rgb signal. A color value is obtained according to the current coordinates of the pixel (the pixel_x and pixel_y signals) and the external control and data signals. This circuit is more involved and is discussed in the second half of this chapter and Chapter 14.

13.2 VGA SYNCHRONIZATION

The video synchronization circuit generates the hsync signal, which specifies the required time to traverse (scan) a row, and the vsync signal, which specifies the required time to traverse (scan) the entire screen. Subsequent discussions are based on a 640-by-480 VGA screen with a 25-MHz *pixel rate*, which means that 25M pixels are processed in a second. Note that this resolution is also know as the *VGA mode*.

The screen of a CRT monitor usually includes a small black border, as shown at the top of Figure 13.4. The middle rectangle is the visible portion. Note that the coordinate of the vertical axis increases downward. The coordinates of the top-left and bottom-right corners are (0,0) and (639,479), respectively.

13.2.1 Horizontal synchronization

A detailed timing diagram of one horizontal scan is shown in Figure 13.4. A period of the hsync signal contains 800 pixels and can be divided into four regions:

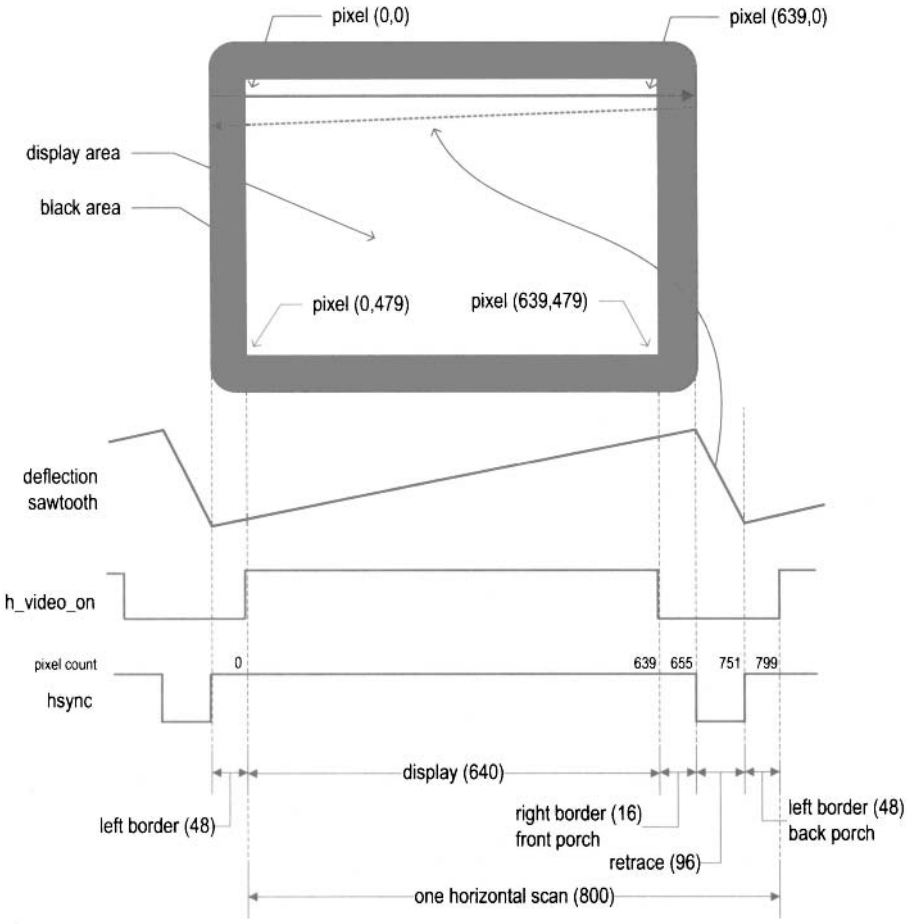


Figure 13.4 Timing diagram of a horizontal scan.

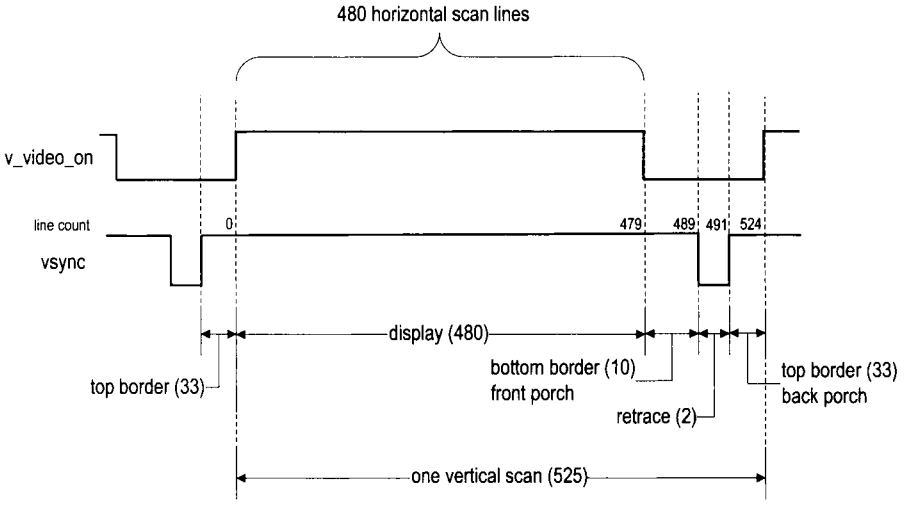


Figure 13.5 Timing diagram of a vertical scan.

- *Display*: region where the pixels are actually displayed on the screen. The length of this region is 640 pixels.
- *Retrace*: region in which the electron beams return to the left edge. The video signal should be disabled (i.e., black), and the length of this region is 96 pixels.
- *Right border*: region that forms the right border of the display region. It is also know as the *front porch* (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 16 pixels.
- *Left border*: region that forms the left border of the display region. It is also know as the *back porch* (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 48 pixels.

Note that the lengths of the right and left borders may vary for different brands of monitors.

The `hsync` signal can be obtained by a special mod-800 counter and a decoding circuit. The counts are marked on the top of the `hsync` signal in Figure 13.4. We intentionally start the counting from the beginning of the display region. This allows us to use the counter output as the horizontal (x-axis) coordinate. This output constitutes the `pixel_x` signal. The `hsync` signal goes low when the counter's output is between 656 and 751.

Note that the CRT monitor should be black in the right and left borders and during retrace. We use the `h_video_on` signal to indicate whether the current horizontal coordinate is in the displayable region. It is asserted only when the pixel count is smaller than 640.

13.2.2 Vertical synchronization

During the vertical scan, the electron beams move gradually from top to bottom and then return to the top. This corresponds to the time required to refresh the entire screen. The format of the `vsync` signal is similar to that of the `hsync` signal, as shown in Figure 13.5. The time unit of the movement is represented in terms of horizontal scan lines. A period of the `vsync` signal is 525 lines and can be divided into four regions:

- *Display*: region where the horizontal lines are actually displayed on the screen. The length of this region is 480 lines.

- *Retrace*: region that the electron beams return to the top of the screen. The video signal should be disabled, and the length of this region is 2 lines.
- *Bottom border*: region that forms the bottom border of the display region. It is also know as the *front porch* (i.e., porch before retrace). The video signal should be disabled, and the length of this region is 10 lines.
- *Top border*: region that forms the top border of the display region. It is also know as the *back porch* (i.e., porch after retrace). The video signal should be disabled, and the length of this region is 33 lines.

As in the horizontal scan, the lengths of the top and bottom borders may vary for different brands of monitors.

The `vsync` signal can be obtained by a special mod-525 counter and a decoding circuit. Again, we intentionally start counting from the beginning of the display region. This allows us to use the counter output as the vertical (y-axis) coordinate. This output constitutes the `pixel_y` signal. The `vsync` signal goes low when the line count is 490 or 491.

As in the horizontal scan, we use the `v_video_on` signal to indicate whether the current vertical coordinate is in the displayable region. It is asserted only when the line count is smaller than 480.

13.2.3 Timing calculation of VGA synchronization signals

As mentioned earlier, we assume that the pixel rate is 25 MHz. It is determined by three parameters:

- p: the number of pixels in a horizontal scan line. For 640-by-480 resolution, it is

$$p = 800 \frac{pixels}{line}$$

- l: the number of lines in a screen (i.e., a vertical scan). For 640-by-480 resolution, it is

$$l = 525 \frac{lines}{screen}$$

- s: the number of screens per second. For flickering-free operation, we can set it to

$$s = 60 \frac{screens}{second}$$

The s parameter specifies how fast the screen should be refreshed. For a human eye, the refresh rate must be at least 30 screens per second to make the motion appear to be continuous. To reduce flickering, the monitor usually has a much higher rate, such as the 60 screens per second specification above. The pixel rate can be calculated by the three parameters:

$$\text{pixel rate} = p * l * s \approx 25M \frac{pixels}{second}$$

The pixel rate for other resolutions and refresh rates can be calculated in a similar fashion. Clearly, the rate increases as the resolution and refresh rate grow.

13.2.4 HDL implementation

The function of the `vga_sync` circuit is discussed in Section 13.1.3. If the frequency of the system clock is 25 MHz, the circuit can be implemented by two special counters: a

mod-800 counter to keep track of the horizontal scan and a mod-525 counter to keep track of the vertical scan.

Since our designs generally use the 50-MHz oscillator of the prototyping board, the system clock rate is twice the pixel rate. Instead of creating a separate 25-MHz clock domain and violating the synchronous design methodology, we can generate a 25-MHz enable tick to enable or pause the counting. The tick is also routed to the p_tick port as an output signal to coordinate operation of the pixel generation circuit.

The HDL code is shown in Listing 13.1. It consists of a mod-2 counter to generate the 25-MHz enable tick and two counters for the horizontal and vertical scans. We use two status signals, h_end and v_end, to indicate completion of the horizontal and vertical scans. The values of various regions of the horizontal and vertical scans are defined as constants. They can easily be modified if a different resolution or refresh rate is used. To remove potential glitches, output buffers are inserted for the hsync and vsync signals. This leads to a one-clock-cycle delay. We should add a similar buffer for the rgb signal in the pixel generation circuit to compensate for the delay.

Listing 13.1 VGA synchronization circuit

```
module vga_sync
   (
    input wire clk, reset,
    output wire hsync, vsync, video_on, p_tick,
    output wire [9:0] pixel_x, pixel_y
   );

   // constant declaration
   // VGA 640-by-480 sync parameters
   localparam HD = 640; // horizontal display area
   localparam HF = 48 ; // h. front (left) border
   localparam HB = 16 ; // h. back (right) border
   localparam HR = 96 ; // h. retrace
   localparam VD = 480; // vertical display area
   localparam VF = 10;  // v. front (top) border
   localparam VB = 33;  // v. back (bottom) border
   localparam VR = 2;   // v. retrace

   // mod-2 counter
   reg mod2_reg;
   wire mod2_next;
   // sync counters
   reg [9:0] h_count_reg, h_count_next;
   reg [9:0] v_count_reg, v_count_next;
   // output buffer
   reg v_sync_reg, h_sync_reg;
   wire v_sync_next, h_sync_next;
   // status signal
   wire h_end, v_end, pixel_tick;

   // body
   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
```

```
            mod2_reg <= 1'b0;
            v_count_reg <= 0;
            h_count_reg <= 0;
            v_sync_reg <= 1'b0;
            h_sync_reg <= 1'b0;
         end
      else
         begin
            mod2_reg <= mod2_next;
            v_count_reg <= v_count_next;
            h_count_reg <= h_count_next;
            v_sync_reg <= v_sync_next;
            h_sync_reg <= h_sync_next;
         end

   // mod-2 circuit to generate 25 MHz enable tick
   assign mod2_next = ~mod2_reg;
   assign pixel_tick = mod2_reg;

   // status signals
   // end of horizontal counter (799)
   assign h_end = (h_count_reg==(HD+HF+HB+HR-1));
   // end of vertical counter (524)
   assign v_end = (v_count_reg==(VD+VF+VB+VR-1));

   // next-state logic of mod-800 horizontal sync counter
   always @*
      if (pixel_tick)  // 25 MHz pulse
         if (h_end)
            h_count_next = 0;
         else
            h_count_next = h_count_reg + 1;
      else
         h_count_next = h_count_reg;

   // next-state logic of mod-525 vertical sync counter
   always @*
      if (pixel_tick & h_end)
         if (v_end)
            v_count_next = 0;
         else
            v_count_next = v_count_reg + 1;
      else
         v_count_next = v_count_reg;

   // horizontal and vertical sync, buffered to avoid glitch
   // h_sync_next asserted between 656 and 751
   assign h_sync_next = (h_count_reg>=(HD+HB) &&
                         h_count_reg<=(HD+HB+HR-1));
   // vh_sync_next asserted between 490 and 491
   assign v_sync_next = (v_count_reg>=(VD+VB) &&
                         v_count_reg<=(VD+VB+VR-1));
```

```
   // video on/off
90 assign video_on = (h_count_reg<HD) && (v_count_reg<VD);

   // output
   assign hsync = h_sync_reg;
   assign vsync = v_sync_reg;
95 assign pixel_x = h_count_reg;
   assign pixel_y = v_count_reg;
   assign p_tick = pixel_tick;

endmodule
```

13.2.5 Testing circuit

To verify operation of the synchronization circuit, we can connect the rgb signal to three switches. The entire visible region should be turned on with a single color. We can go through the eight possible combinations and check the colors defined in Table 13.1. The HDL code is shown in Listing 13.2. As mentioned in Section 13.2.4, an output buffer is added for the rgb signal.

Listing 13.2 VGA synchronization testing circuit

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

   //signal declaration
10 reg [2:0] rgb_reg;
   wire video_on;

   // instantiate vga sync circuit
   vga_sync vsync_unit
15    (.clk(clk), .reset(reset), .hsync(hsync), .vsync(vsync),
       .video_on(video_on), .p_tick(), .pixel_x(), .pixel_y());
   // rgb buffer
   always @(posedge clk, posedge reset)
      if (reset)
20       rgb_reg <= 0;
      else
         rgb_reg <= sw;
   // output
   assign rgb = (video_on) ? rgb_reg : 3'b0;
25
endmodule
```

13.3 OVERVIEW OF THE PIXEL GENERATION CIRCUIT

The pixel generation circuit generates the 3-bit `rgb` signal for the VGA port. The external control and data signals specify the content of the screen, and the `pixel_x` and `pixel_y` signals from the `vga_sync` circuit provide the current coordinates of the pixel. For our discussion purposes, we divided this circuit into three broad categories:

- Bit-mapped scheme
- Tile-mapped scheme
- Object-mapped scheme

In a *bit-mapped scheme*, a *video memory* is used to store the data to be displayed on the screen. Each pixel of the screen is mapped directly to a memory word, and the `pixel_x` and `pixel_y` signals form the address. A graphics processing circuit continuously updates the screen and writes relevant data to the video memory. A retrieval circuit continuously reads the video memory and routes the data to the `rgb` signal. This is the scheme used in today's high-performance video controller. For 640-by-480 resolution, there are about 310k (i.e., 640*480) pixels on a screen. This translates to 310k memory bits for a monochrome display and 930k memory bits (i.e., 3 bits per pixel) for a 3-bit color display. A bit-mapped example is discussed in Section 13.5.

To reduce the memory requirement, one alternative is to use a *tile-mapped scheme*. In this scheme, we group a collection of bits to form a *tile* and treat each tile as a display unit. For example, we can define an 8-by-8 square of pixels (i.e., 64 pixels) as a tile. The 640-by-480 pixel-oriented screen becomes an 80-by-60 tile-oriented screen. Only 4800 (i.e., 80*60) words are needed for the *tile memory*. The number of bits in a word depends on the number of tile patterns. For example, if there are 32 tile patterns, each word should contain 5 bits, and the size of the tile memory is about 24k bits (i.e., 5*4800). The tile-mapped scheme usually requires a ROM to store the tile patterns. We call it *pattern memory*. Assume that monochrome patterns are used in the previous example. Each 8-by-8 tile pattern requires 64 bits, and the entire 32 patterns need 2K (i.e., 8*8*32) bits. The overall memory requirement is about 26k bits, which is much smaller than the 310k bits of the bit-mapped scheme. The text display discussed in Chapter 14 is based on this scheme.

For some applications, the video display can be very simple and contains only a few objects. Instead of wasting memory to store a mostly blank screen, we can generate these objects using simple object generation circuits. We call this approach an *object-mapped scheme*. An object-mapped example is discussed in Section 13.4.

The three schemes can be mixed together to generate a full screen. For example, we can use a bit-mapped scheme to generate the background and use an object-mapped scheme to produce the main objects. We can also use a bit-mapped scheme for one portion of a screen and tile-mapped text for another part of the screen.

13.4 GRAPHIC GENERATION WITH AN OBJECT-MAPPED SCHEME

The conceptual diagram of an object-mapped pixel generation circuit that contains three objects is shown in Figure 13.6. The diagram consists of three object generation circuits and a special selecting and routing circuit, labeled `rgb mux`. An object generation circuit performs the following tasks:

- It keeps the coordinates of the current object and compares them with the current scan location provided by the `pixel_x` and `pixel_y` signals.

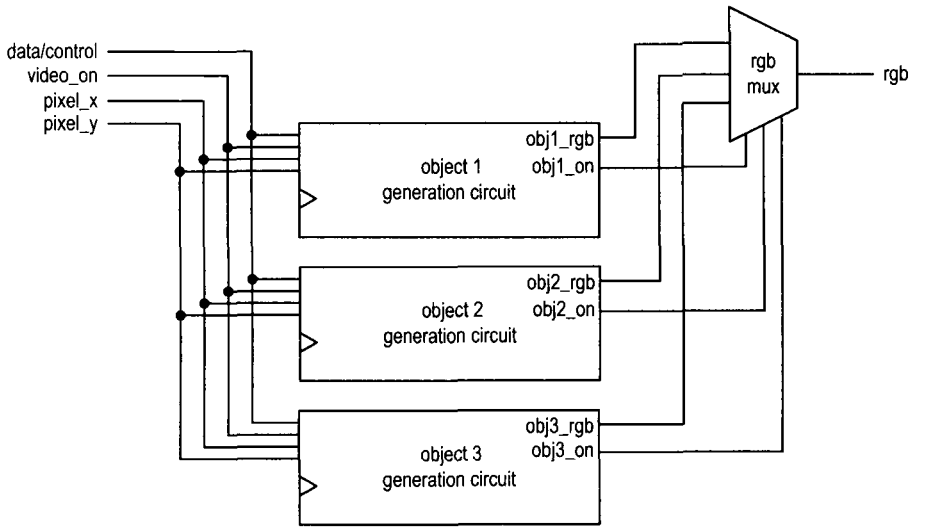


Figure 13.6 Conceptual diagram of object-mapped pixel generation.

- If the current scan location falls within the region, it asserts the `obj_i_on` signal to indicate that the current scan location is within the region of the ith object and the object should be "turned on."
- It specifies the desired color in the `obj_i_rgb` signal.

The `rgb mux` circuit performs multiplexing according to an internal prioritizing scheme. It examines various `obj_i_on` signals and determines which `obj_i_rgb` signal is to be routed to the `rgb` output. The prioritizing scheme prioritizes the order of the displays when multiple `obj_i_on` signals are asserted at the same time. It corresponds to selecting an object for the foreground.

We use a simplified ping-pong-like game to illustrate the various graphic generation schemes. The design is constructed as follows:

1. Create a simple still screen with rectangular objects.
2. Add a round object.
3. Introduce animation.
4. Add text for scores and information.
5. Create a top-level control circuit.

The first three steps are discussed in this section, and the last two steps are discussed in Chapter 14.

13.4.1 Rectangular objects

A rectangular object can be described by its boundary coordinates on the screen. The still screen of the game is shown in Figure 13.7. It has three objects: a wall, which is shown as a narrow stripe on the left; a paddle, which is shown as a short vertical bar on the right; and a square ball. The coordinates of the displayable area of the screen are also shown. Note that the y-axis increases downward.

Let us first examine generation of the wall stripe. For clarity, we define constants for the relevant boundaries and sizes in code. The code segment for the wall is

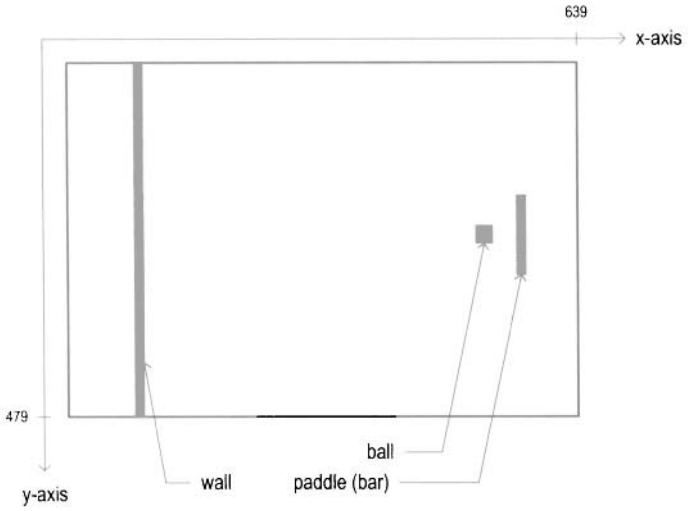


Figure 13.7 Still screen of the pong game.

```
// wall left, right boundary
localparam WALL_X_L = 32;
localparam WALL_X_R = 35;
...
// pixel within wall
assign wall_on = (WALL_X_L<=pix_x) && (pix_x<=WALL_X_R);
// wall rgb output
assign wall_rgb = 3'b001; // blue
```

The wall is a four-pixel-wide vertical stripe between columns 32 and 35, which as defined as `WALL_X_L` and `WALL_X_R`, representing the left and right x-coordinates of the wall, respectively. The object has two output signals, `wall_on` and `wall_rgb`. The `wall_on` signal, which indicates that the wall object should be turned on, is asserted when the current horizontal scan is within its region. Since the stripe covers the entire vertical column, there is no need for the y-axis boundaries. The `wall_rgb` signal indicates that the color of the wall is "001" (blue).

The code segment for the bar (paddle) is

```
// bar left, right boundary
localparam BAR_X_L = 600;
localparam BAR_X_R = 603;
// bar top, bottom boundary
localparam BAR_Y_SIZE = 72;
localparam BAR_Y_T = MAX_Y/2-BAR_Y_SIZE/2; //204
localparam BAR_Y_B = BAR_Y_T+BAR_Y_SIZE-1;
...
// pixel within bar
assign bar_on = (BAR_X_L<=pix_x) && (pix_x<=BAR_X_R) &&
                (BAR_Y_T<=pix_y) && (pix_y<=BAR_Y_B);
// bar rgb output
assign bar_rgb = 3'b010; // green
```

The code is similar to that of the wall segment except that it includes the y-axis boundaries. The desired vertical length of the bar is 72 pixels, which is defined by BAR_Y_SIZE. Since we wish to place the bar in the middle, the top boundary of the bar, which is BAR_Y_T, is one half of the maximal y-value (i.e., 480/2) minus one half of the bar length. The bottom boundary of the bar is the top boundary plus the bar length. Generation of the bar_on signal is similar to that of the wall_on signal except that the vertical scan must be within the bar's y-axis boundaries as well.

The code for the ball can be constructed in a similar fashion. The final code segment is the selection and multiplexing circuit, which examines the on signals of three objects and routes the corresponding rgb signal to output. The code is

```
always @*
   if (~video_on)
      graph_rgb = 3'b000; // blank
   else
      if (wall_on)
         graph_rgb = wall_rgb;
      else if (bar_on)
         graph_rgb = bar_rgb;
      else if (sq_ball_on)
         graph_rgb = ball_rgb;
      else
         graph_rgb = 3'b110; // yellow background
```

The circuit first checks whether the video_on is asserted, and if this is the case, examines the three on signals in turn. When an on signal is asserted, it indicates that the scan is within its region, and the corresponding rgb signal is passed to the output. If no signal is asserted, the scan is in the "background" and the output is assigned to be "110" (yellow).

The complete HDL code is shown in Listing 13.3.

Listing 13.3 Pixel-generation circuit for the pong game screen

```
module pong_graph_st
   (
    input wire video_on,
    input wire [9:0] pix_x, pix_y,
    output reg [2:0] graph_rgb
   );

   // constant and signal declaration
   // x, y coordinates (0,0) to (639,479)
   localparam MAX_X = 640;
   localparam MAX_Y = 480;
   //-----------------------------------------------
   // vertical stripe as a wall
   //-----------------------------------------------
   // wall left, right boundary
   localparam WALL_X_L = 32;
   localparam WALL_X_R = 35;
   //-----------------------------------------------
   // right vertical bar
   //-----------------------------------------------
   // bar left, right boundary
   localparam BAR_X_L = 600;
```

```
   localparam BAR_X_R = 603;
   // bar top, bottom boundary
25 localparam BAR_Y_SIZE = 72;
   localparam BAR_Y_T = MAX_Y/2-BAR_Y_SIZE/2; //204
   localparam BAR_Y_B = BAR_Y_T+BAR_Y_SIZE-1;
   //-----------------------------------------------
   // square ball
30 //-----------------------------------------------
   localparam BALL_SIZE = 8;
   // ball left, right boundary
   localparam BALL_X_L = 580;
   localparam BALL_X_R = BALL_X_L+BALL_SIZE-1;
35 // ball top, bottom boundary
   localparam BALL_Y_T = 238;
   localparam BALL_Y_B = BALL_Y_T+BALL_SIZE-1;
   //-----------------------------------------------
   // object output signals
40 //-----------------------------------------------
   wire wall_on, bar_on, sq_ball_on;
   wire [2:0] wall_rgb, bar_rgb, ball_rgb;

   // body
45 //-----------------------------------------------
   // (wall) left vertical strip
   //-----------------------------------------------
   // pixel within wall
   assign wall_on = (WALL_X_L<=pix_x) && (pix_x<=WALL_X_R);
50 // wall rgb output
   assign wall_rgb = 3'b001; // blue
   //-----------------------------------------------
   // right vertical bar
   //-----------------------------------------------
55 // pixel within bar
   assign bar_on = (BAR_X_L<=pix_x) && (pix_x<=BAR_X_R) &&
                   (BAR_Y_T<=pix_y) && (pix_y<=BAR_Y_B);
   // bar rgb output
   assign bar_rgb = 3'b010; // green
60 //-----------------------------------------------
   // square ball
   //-----------------------------------------------
   // pixel within squared ball
   assign sq_ball_on =
65           (BALL_X_L<=pix_x) && (pix_x<=BALL_X_R) &&
             (BALL_Y_T<=pix_y) && (pix_y<=BALL_Y_B);
   assign ball_rgb = 3'b100;    // red
   //-----------------------------------------------
   // rgb multiplexing circuit
70 //-----------------------------------------------
   always @*
      if (~video_on)
         graph_rgb = 3'b000; // blank
      else
75       if (wall_on)
```

```
            graph_rgb = wall_rgb;
         else if (bar_on)
            graph_rgb = bar_rgb;
         else if (sq_ball_on)
            graph_rgb = ball_rgb;
         else
            graph_rgb = 3'b110; // yellow background

endmodule
```

After deriving the pixel generation circuit, we can combine it with the VGA synchronization circuit to construct the complete video interface. The top-level HDL code is shown in Listing 13.4. Note that the graph_rgb signal is routed to output through an output buffer. It is loaded when the pixel_tick signal is asserted. This synchronizes the rgb output with the buffered hsync and vsync signals.

Listing 13.4 Complete circuit for a still pong game screen

```
module pong_top_st
   (
    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));
   // instantiate graphic generator
   pong_graph_st pong_grf_unit
      (.video_on(video_on), .pix_x(pixel_x), .pix_y(pixel_y),
       .graph_rgb(rgb_next));
   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;

endmodule
```

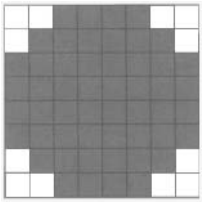


Figure 13.8 Bit map of a circle.

13.4.2 Non-rectangular object

Direct checking of the boundaries of a non-rectangular object is very difficult. An alternative is to specify the object pattern in a bit map and generate the `rgb` and on signals according to the map. This can best be explained by an example. Assume that we want to have a round ball in the pong game screen. The bit map of a circle within an 8-by-8 pixel square is shown in Figure 13.8. The circle object can be generated as follows:

- Check whether the scan coordinates are within the 8-by-8 pixel square.
- If this is the case, obtain the corresponding pixel from the bit map.
- Use the retrieved bit to generate the `rgb` and on signals for the circle object.

To implement this scheme, we need to include a *pattern ROM* to store the bit map and an address mapping circuit to convert the scan coordinates to the ROM's row and column.

To accommodate the change, the ball portion from Listing 13.3 must be modified. First, we define a pattern ROM for the circle using a case statement, as in the ROM template of Listing 12.5:

```
wire [2:0] rom_addr;
reg [7:0] rom_data;
...
// round ball image ROM
always @*
case (rom_addr)
   3'h0: rom_data = 8'b00111100; //   ****
   3'h1: rom_data = 8'b01111110; //  ******
   3'h2: rom_data = 8'b11111111; // ********
   3'h3: rom_data = 8'b11111111; // ********
   3'h4: rom_data = 8'b11111111; // ********
   3'h5: rom_data = 8'b11111111; // ********
   3'h6: rom_data = 8'b01111110; //  ******
   3'h7: rom_data = 8'b00111100; //   ****
endcase
```

Second, we expand the ball generation segment to include the mapping of the circle bit map. To facilitate future animation, we also use signals to replace constants for the square ball boundaries. The revised code becomes

```
// pixel within ball
assign sq_ball_on =
         (ball_x_l<=pix_x) && (pix_x<=ball_x_r) &&
         (ball_y_t<=pix_y) && (pix_y<=ball_y_b);
// map current pixel location to ROM addr/col
assign rom_addr = pix_y[2:0] - ball_y_t[2:0];
```

```
assign rom_col = pix_x[2:0] - ball_x_l[2:0];
assign rom_bit = rom_data[rom_col];
// pixel within ball
assign rd_ball_on = sq_ball_on & rom_bit;
// ball rgb output
assign ball_rgb = 3'b100;    // red
```

The first statement checks whether the current scan coordinates are within the square ball region and asserts the `sq_ball_on` signal accordingly. This part is the same as Listing 13.3 except that signals are used for boundaries. The second part obtains the corresponding ROM bit according to the current scan coordinates. If the scan coordinates are within the square ball region, subtracting the three LSBs from the top boundary (i.e., `ball_y_t`) provides the corresponding ROM row (i.e., `rom_addr`), and subtracting the three LSBs from the left boundary (i.e., `ball_x_l`) provides the corresponding ROM column (i.e., `rom_col`). The final bit is retrieved by an indexing operation. It is then combined with the `sq_ball_on` signal to generate the `rd_ball_on` signal. This design just assigns a monochrome color (i.e., 100, red) for the round ball region. We can duplicate the pattern ROM three times to store the `rgb` value for each pixel and generate a multiple-color ball.

Finally, we need to make a minor modification in the multiplexing circuit to substitute the `sq_ball_on` signal with the `rd_ball_on` signal:

```
...
else if (rd_ball_on)
   graph_rgb = ball_rgb;
...
```

These modifications are incorporated into the animated graph in the next subsection.

13.4.3 Animated object

When an object changes its location gradually in each scan, it creates the illusion of motion and becomes *animated*. To achieve this, we can use registers to store the boundaries of an object and update its value in each scan. In the pong game, the paddle is controlled by two pushbuttons and can move up and down, and the ball can move and bounce in all directions. We illustrate how to create animation for these two objects in this subsection.

While the VGA controller is driven by a 25-MHz pixel rate, the screen of the VGA monitor is refreshed only 60 times per second. The boundary registers only need to be updated at this rate. We create a 60-Hz enable tick, `refr_tick`, which is asserted one clock cycle every $\frac{1}{60}$ second.

Let us first examine the design of the paddle. To accommodate the changing y-axis coordinates, we replace the constants with two signals, `bar_y_t` and `bar_y_b`, to represent the top and bottom boundaries, and create a register, `bar_y_reg`, to store the current y-axis location of the top boundary. If one of the pushbuttons is pressed, `bar_y_reg` either increases or decreases a fixed amount when the `refr_tick` signal is asserted. The amount is defined by a constant, `BAR_V`, which stands for the bar velocity. We assume that assertion of the `btn[1]` and `btn[0]` signals causes the paddle to move up and down, respectively, and that the paddle stops moving when it reaches the top or the bottom of the screen. The code segment for updating `bar_y_reg` is

```
// new bar y-position
always @*
begin
```

```
      bar_y_next = bar_y_reg; // no move
      if (refr_tick)
         if (btn[1] & (bar_y_b < (MAX_Y-1-BAR_V)))
            bar_y_next = bar_y_reg + BAR_V; // move down
         else if (btn[0] & (bar_y_t > BAR_V))
            bar_y_next = bar_y_reg - BAR_V; // move up
   end
```

The design of the ball is more involved. We have to replace the four boundary constants with four signals and create two registers, `ball_x_reg` and `ball_y_reg`, to store the current x- and y-axis coordinates of the left and top boundaries. The ball usually moves at a constant velocity (i.e., at a constant speed and in the same direction). It may change direction when hitting the wall, the paddle, or the bottom or top of the screen. We decompose the velocity into an x-component and a y-component, whose values can be either a positive constant value, `BALL_V_P`, or a negative constant value, `BALL_V_N`. The current values of the two components are stored in the `x_delta_reg` and `y_delta_reg` registers. The code segment for updating `ball_x_reg` and `ball_y_reg` is

```
   // new ball position
   assign ball_x_next = (refr_tick) ? ball_x_reg+x_delta_reg :
                        ball_x_reg ;
   assign ball_y_next = (refr_tick) ? ball_y_reg+y_delta_reg :
                        ball_y_reg ;
```

and the code segment for updating `x_delta_reg` and `y_delta_reg` is

```
   // new ball velocity
   always @*
   begin
      x_delta_next = x_delta_reg;
      y_delta_next = y_delta_reg;
      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))
         // reach x of right bar and hit, ball bounce back
         x_delta_next = BALL_V_N;
   end
```

Note that if the paddle bar misses the ball, the ball continues moving to the right and eventually wraps around.

The complete code is shown in Listing 13.5.

Listing 13.5 Pixel-generation circuit for the animated pong game

```
module pong_graph_animate
   (
    input wire clk, reset,
    input wire video_on,
    input wire [1:0] btn,
    input wire [9:0] pix_x, pix_y,
```

```
    output reg [2:0] graph_rgb
);

// constant and signal declaration
// x, y coordinates (0,0) to (639,479)
localparam MAX_X = 640;
localparam MAX_Y = 480;
wire refr_tick;
//-----------------------------------------------
// vertical stripe as a wall
//-----------------------------------------------
// wall left, right boundary
localparam WALL_X_L = 32;
localparam WALL_X_R = 35;
//-----------------------------------------------
// right vertical bar
//-----------------------------------------------
// bar left, right boundary
localparam BAR_X_L = 600;
localparam BAR_X_R = 603;
// bar top, bottom boundary
wire [9:0] bar_y_t, bar_y_b;
localparam BAR_Y_SIZE = 72;
// register to track top boundary (x position is fixed)
reg [9:0] bar_y_reg, bar_y_next;
// bar moving velocity when a button is pressed
localparam BAR_V = 4;
//-----------------------------------------------
// square ball
//-----------------------------------------------
localparam BALL_SIZE = 8;
// ball left, right boundary
wire [9:0] ball_x_l, ball_x_r;
// ball top, bottom boundary
wire [9:0] ball_y_t, ball_y_b;
// reg to track left, top position
reg [9:0] ball_x_reg, ball_y_reg;
wire [9:0] ball_x_next, ball_y_next;
// reg to track ball speed
reg [9:0] x_delta_reg, x_delta_next;
reg [9:0] y_delta_reg, y_delta_next;
// ball velocity can be pos or neg)
localparam BALL_V_P = 2;
localparam BALL_V_N = -2;
//-----------------------------------------------
// round ball
//-----------------------------------------------
wire [2:0] rom_addr, rom_col;
reg [7:0] rom_data;
wire rom_bit;
//-----------------------------------------------
// object output signals
//-----------------------------------------------
```

```
60    wire wall_on, bar_on, sq_ball_on, rd_ball_on;
      wire [2:0] wall_rgb, bar_rgb, ball_rgb;

      // body
      //------------------------------------------------------------
65    // round ball image ROM
      //------------------------------------------------------------
      always @*
      case (rom_addr)
         3'h0: rom_data = 8'b00111100; //   ****
70       3'h1: rom_data = 8'b01111110; //  ******
         3'h2: rom_data = 8'b11111111; // ********
         3'h3: rom_data = 8'b11111111; // ********
         3'h4: rom_data = 8'b11111111; // ********
         3'h5: rom_data = 8'b11111111; // ********
75       3'h6: rom_data = 8'b01111110; //  ******
         3'h7: rom_data = 8'b00111100; //   ****
      endcase

      // registers
80    always @(posedge clk, posedge reset)
         if (reset)
            begin
               bar_y_reg <= 0;
               ball_x_reg <= 0;
85             ball_y_reg <= 0;
               x_delta_reg <= 10'h004;
               y_delta_reg <= 10'h004;
            end
         else
90          begin
               bar_y_reg <= bar_y_next;
               ball_x_reg <= ball_x_next;
               ball_y_reg <= ball_y_next;
               x_delta_reg <= x_delta_next;
95             y_delta_reg <= y_delta_next;
            end

      // refr_tick: 1-clock tick asserted at start of v-sync
      //            i.e., when the screen is refreshed (60 Hz)
100   assign refr_tick = (pix_y==481) && (pix_x==0);

      //------------------------------------------------------------
      // (wall) left vertical strip
      //------------------------------------------------------------
105   // pixel within wall
      assign wall_on = (WALL_X_L<=pix_x) && (pix_x<=WALL_X_R);
      // wall rgb output
      assign wall_rgb = 3'b001; // blue
      //------------------------------------------------------------
110   // right vertical bar
      //------------------------------------------------------------
      // boundary
```

```
   assign bar_y_t = bar_y_reg;
   assign bar_y_b = bar_y_t + BAR_Y_SIZE - 1;
   // pixel within bar
   assign bar_on = (BAR_X_L<=pix_x) && (pix_x<=BAR_X_R) &&
                   (bar_y_t<=pix_y) && (pix_y<=bar_y_b);
   // bar rgb output
   assign bar_rgb = 3'b010; // green
   // new bar y-position
   always @*
   begin
      bar_y_next = bar_y_reg; // no move
      if (refr_tick)
         if (btn[1] & (bar_y_b < (MAX_Y-1-BAR_V)))
            bar_y_next = bar_y_reg + BAR_V; // move down
         else if (btn[0] & (bar_y_t > BAR_V))
            bar_y_next = bar_y_reg - BAR_V; // move up
   end

   //-----------------------------------------------
   // square ball
   //-----------------------------------------------
   // boundary
   assign ball_x_l = ball_x_reg;
   assign ball_y_t = ball_y_reg;
   assign ball_x_r = ball_x_l + BALL_SIZE - 1;
   assign ball_y_b = ball_y_t + BALL_SIZE - 1;
   // pixel within ball
   assign sq_ball_on =
            (ball_x_l<=pix_x) && (pix_x<=ball_x_r) &&
            (ball_y_t<=pix_y) && (pix_y<=ball_y_b);
   // map current pixel location to ROM addr/col
   assign rom_addr = pix_y[2:0] - ball_y_t[2:0];
   assign rom_col = pix_x[2:0] - ball_x_l[2:0];
   assign rom_bit = rom_data[rom_col];
   // pixel within ball
   assign rd_ball_on = sq_ball_on & rom_bit;
   // ball rgb output
   assign ball_rgb = 3'b100;    // red
   // new ball position
   assign ball_x_next = (refr_tick) ? ball_x_reg+x_delta_reg :
                        ball_x_reg ;
   assign ball_y_next = (refr_tick) ? ball_y_reg+y_delta_reg :
                        ball_y_reg ;
   // new ball velocity
   always @*
   begin
      x_delta_next = x_delta_reg;
      y_delta_next = y_delta_reg;
      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))
            // reach x of right bar and hit, ball bounce back
            x_delta_next = BALL_V_N;
      end
      //-----------------------------------------------
      // rgb multiplexing circuit
      //-----------------------------------------------
      always @*
         if (~video_on)
            graph_rgb = 3'b000; // blank
         else
            if (wall_on)
               graph_rgb = wall_rgb;
            else if (bar_on)
               graph_rgb = bar_rgb;
            else if (rd_ball_on)
               graph_rgb = ball_rgb;
            else
               graph_rgb = 3'b110; // yellow background

   endmodule
```

As in the still screen, we can combine the synchronization circuit and create the top-level description. The HDL code is shown in Listing 13.6.

Listing 13.6 Complete circuit for the animated pong game screen

```
module pong_top_an
   (
    input wire clk, reset,
    input wire [1:0] btn,
    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));

   // instantiate graphic generator
   pong_graph_animate pong_graph_an_unit
      (.clk(clk), .reset(reset), .btn(btn),
       .video_on(video_on), .pix_x(pixel_x),
```

```
      .pix_y(pixel_y), .graph_rgb(rgb_next));

   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;

endmodule
```

Note that there is no other control mechanism is this code. The ball simply moves and bounces continuously. A top-level control circuit is discussed in Chapter 14.

13.5 GRAPHIC GENERATION WITH A BIT-MAPPED SCHEME

The bit-mapped scheme maps each pixel to a word in video memory. There are about 310k pixels in a 640-by-480 screen. This translates to 310k and 930k bits for monochrome and color displays, respectively. The actual size of the video memory can be much larger since the memory address must be properly aligned for fast access. For example, to map the pixel's current coordinates to a memory location, we can concatenate the pixel's x-coordinate, which is 10 bits (i.e., $\lceil \log_2(640) \rceil$), and the pixel's y-coordinate, which is 9 bits (i.e., $\lceil \log_2(480) \rceil$). This approach requires no additional circuit to translate the pixel's coordinates to a memory address but introduces some unused "holes" in memory. The memory size is increased from 310k words to 512K (i.e., 2^{10+9}) words.

For the S3 board, memory is available from the external SRAM chips and FPGA's embedded block RAMs, as discussed in Chapters 11 and 12. Recall that the total capacity of the Spartan 3S200 device's block RAM is only about 192K bits. It is not large enough for a full-screen bit-mapped display. We must use the external SRAM, which is 8M bits, for this purpose.

In this section, we use a small 128-by-128 (2^7-by-2^7) area of the screen to illustrate the design of the bit-mapped scheme. The screen has 16K (2^{14}) pixels in this area and requires a 16K-by-3 video memory for color display. This can be implemented by three embedded block RAMs. The small area is at the top-left corner of the screen and displays the trace of a bouncing one-pixel dot, as shown in Figure 13.9. The circuit uses a 3-bit switch to specify the color of the trace and a pushbutton switch to randomly select the origin of the trace. When the pushbutton switch is pressed, the dot starts to move, like the bouncing ball in Section 13.4.3. The trace forms a rectangle after the dot hits the four sides of the small area. A new trace is generated each time the pushbutton switch is pressed.

13.5.1 Dual-port RAM implementation

A conceptual block diagram of this circuit is shown in Figure 13.10. The video memory is a synchronous 16K-by-3 (i.e., 2^{14}-by-3) dual-port RAM. The dual-port module discussed in Listing 12.4 can be used for this purpose. The seven LSBs of the pixel's y-coordinate form the seven MSBs of the memory address, and the seven LSBs of the pixel's x-coordinate form the seven LSBs of the memory address. The `dot_xy` circuit keeps track of the current location of the dot and generates its current y- and x-coordinates, which are concatenated as the write address. The 3-bit external switch input, `sw`, is the `rgb` value, which is connected

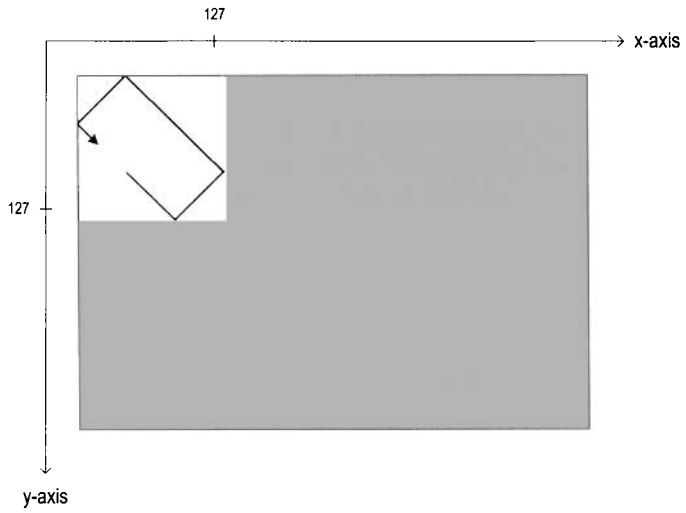


Figure 13.9 Dot trace shown in a 128-by-128 bit map.

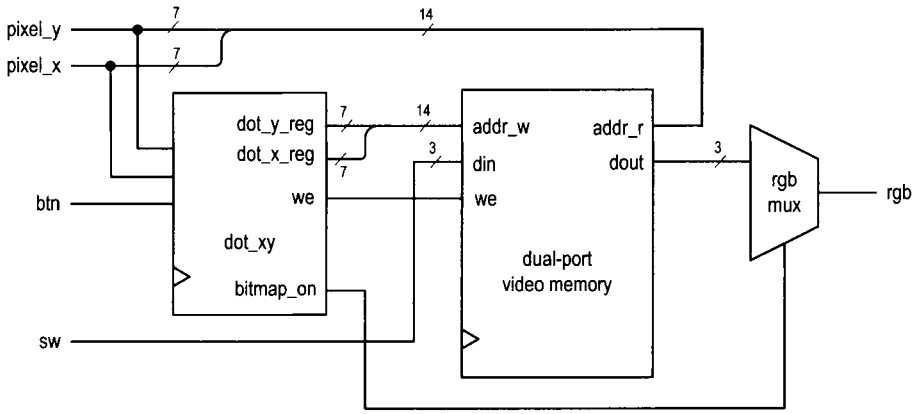


Figure 13.10 Conceptual block diagram of a dot trace circuit.

to the memory's din_a port. The seven LSBs of pixel_y and the seven LSBs of pixel_x form the read address. The data is retrieved continuously and the corresponding readout is routed to the rgb multiplexing circuit.

The complete code of the dot trace pixel generation circuit is shown in Listing 13.7. We use two registers, dot_x_reg and dot_y_reg, to keep track of the dot's current x- and y-coordinates and use two registers, v_x_reg and v_y_reg, to keep track of the current horizontal and vertical velocities. Computation of the dot's coordinates and velocities is similar to that of the bouncing ball in Section 13.4.3. In addition to regular updates, the dot_x_next and dot_y_next signals obtain the values of the seven LSBs of pix_x and pix_y when the pushbutton switch is pressed. Since these signals change much faster than a human's perception, the new origin appears to be random.

Listing 13.7 Pixel-generation circuit for a 128-by-128 bit map

```
module bitmap_gen
   (
    input wire clk, reset,
    input wire video_on,
    input [1:0] btn,
    input [2:0] sw,
    input wire [9:0] pix_x, pix_y,
    output reg [2:0] bit_rgb
   );

   // constant and signal declaration
   wire refr_tick, load_tick;
   //-----------------------------------------------
   // video sram
   //-----------------------------------------------
   wire we;
   wire [13:0] addr_r, addr_w;
   wire [2:0] din, dout;
   //-----------------------------------------------
   // dot location and velocity
   //-----------------------------------------------
   localparam MAX_X = 128;
   localparam MAX_Y = 128;
   // dot velocity can be pos or neg
   localparam DOT_V_P = 1;
   localparam DOT_V_N = -1;
   // reg to keep track of dot location
   reg [6:0] dot_x_reg, dot_y_reg;
   wire [6:0] dot_x_next, dot_y_next;
   // reg to keep track of dot velocity
   reg [6:0] v_x_reg, v_y_reg;
   wire [6:0] v_x_next, v_y_next;
   //-----------------------------------------------
   // object output signals
   //-----------------------------------------------
   wire bitmap_on;
   wire [2:0] bitmap_rgb;

   // body
```

```
   // instantiate debounce circuit for a button
   debounce deb_unit
      (.clk(clk), .reset(reset), .sw(btn[0]),
       .db_level(), .db_tick(load_tick));
   // instantiate dual-port video RAM (2^12-by-7)
   xilinx_dual_port_ram_sync
      #(.ADDR_WIDTH(14), .DATA_WIDTH(3)) video_ram
      (.clk(clk), .we(we), .addr_a(addr_w), .addr_b(addr_r),
       .din_a(din), .dout_a(), .dout_b(dout));
   // video ram interface
   assign addr_w = {dot_y_reg, dot_x_reg};
   assign addr_r = {pix_y[6:0], pix_x[6:0]};
   assign we = 1'b1;
   assign din = sw;
   assign bitmap_rgb = dout;
   // registers
   always @(posedge clk, posedge reset)
      if (reset)
         begin
            dot_x_reg <= 0;
            dot_y_reg <= 0;
            v_x_reg <= DOT_V_P;
            v_y_reg <= DOT_V_P;
         end
      else
         begin
            dot_x_reg <= dot_x_next;
            dot_y_reg <= dot_y_next;
            v_x_reg <= v_x_next;
            v_y_reg <= v_y_next;
         end

   // refr_tick: 1-clock tick asserted at start of v-sync
   assign refr_tick = (pix_y==481) && (pix_x==0);

   // pixel within bit map area
   assign bitmap_on =(pix_x<=127) & (pix_y<=127);
   // dot position
   // "randomly" load dot location when btn[0] pressed
   assign dot_x_next = (load_tick) ? pix_x[6:0] :
                       (refr_tick) ? dot_x_reg + v_x_reg :
                       dot_x_reg ;
   assign dot_y_next = (load_tick) ? pix_y[6:0] :
                       (refr_tick) ? dot_y_reg + v_y_reg :
                       dot_y_reg ;
   // dot x velocity
   assign v_x_next =
        (dot_x_reg==1) ? DOT_V_P :          // reach left
        (dot_x_reg==(MAX_X-2)) ? DOT_V_N : // reach right
         v_x_reg;
   // dot y velocity
   assign v_y_next =
        (dot_y_reg==1) ? DOT_V_P :          // reach top
```

```
      (dot_y_reg==(MAX_Y-2)) ? DOT_V_N : // reach bottom
       v_y_reg;
//-----------------------------------------------
// rgb multiplexing circuit
//-----------------------------------------------
always @*
   if (~video_on)
      bit_rgb = 3'b000; // blank
   else
      if (bitmap_on)
         bit_rgb = bitmap_rgb;
      else
         bit_rgb = 3'b110; // yellow background

endmodule
```

The HDL code for the top-level system is shown in Listing 13.8.

Listing 13.8 Complete circuit for a bit-mapped screen

```
module dot_top
   (
    input wire clk, reset,
    input wire [1:0] btn,
    input wire [2:0] 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));

   // instantiate graphic generator
   bitmap_gen bitmap_unit
      (.clk(clk), .reset(reset), .btn(btn), .sw(sw),
       .video_on(video_on), .pix_x(pixel_x),
       .pix_y(pixel_y), .bit_rgb(rgb_next));

   // rgb buffer
   always @(posedge clk)
      if (pixel_tick)
         rgb_reg <= rgb_next;
   // output
   assign rgb = rgb_reg;
```

```
35
endmodule
```

13.5.2 Single-port RAM implementation

Although a dual-port memory is ideal, it is not always available. Using regular single-port memory, such as the S3 board's external SRAM, for the video memory requires careful coordination between the write and read operations to avoid interruption in data retrieval. For demonstration purposes, we configure the embedded block RAM as a single-port synchronous SRAM and redesign the previous dot trace circuit.

In the dot trace circuit, the dot's coordinates are updated once every screen scan. Thus, the video memory can be written at this rate as well. We can do this during the vertical retrace since the video is off in this period and writing video memory does not interfere with screen data retrieval. Note that the `refr_tick` signal is asserted when `pixel_y` is 481. The video is off in this location, and writing video memory will not interfere with the screen data retrieval. We use this signal as the write enable signal, `we`, for the single-port RAM. The single-port RAM module discussed in Listing 12.2 can be used for this purpose. The memory portion of Listing 13.7 now becomes

```
// instantiate dual-port video RAM (2^12-by-7)
xilinx_one_port_ram_sync
   #(.ADDR_WIDTH(14), .DATA_WIDTH(3)) video_ram
   (.clk(clk), .we(we), .addr(addr),
    .din(din), .dout(dout));
// video ram interface
assign addr_w = {dot_y_reg, dot_x_reg};
assign addr_r = {pix_y[6:0], pix_x[6:0]};
assign addr  = (refr_tick) ? addr_w : addr_r;
assign we = refr_tick;
assign din = sw;
assign bitmap_rgb = dout;
```

The dot trace circuit updates one pixel in a screen scan. The required memory bandwidth for writing is 60*3 bits per second, which is rather low. Thus, the previous design is fairly straightforward. The design of memory interface becomes much more difficult when a large memory bandwidth is required (i.e., when a large portion of the screen is updated at a rapid rate).

13.6 BIBLIOGRAPHIC NOTES

Rapid Prototyping of Digital Systems by James O. Hamblen et al. contains timing information for monitors with different resolutions and refresh rates.

13.7 SUGGESTED EXPERIMENTS

13.7.1 VGA test pattern generator

A VGA test pattern generator produces two simple patterns to verify operation of a VGA monitor. The first pattern divides the screen evenly into eight vertical stripes, each displaying

a unique color. The second pattern is similar but the screen is divided into eight horizontal stripes. A 1-bit switch is used to select the pattern.

Design a pixel-generating circuit for this pattern generator and then combine it with the synchronization circuit in a top-level module. Synthesize and verify operation of the circuit.

13.7.2 SVGA mode synchronization circuit

The specification for the super VGA (SVGA) mode with 72-Hz refresh rate is

- *resolution*: 800-by-600 pixels
- *pixel rate*: 50 MHz
- *horizontal display region*: 800 pixels
- *horizontal right border*: 64 pixels
- *horizontal left border*: 56 pixels
- *horizontal retrace*: 120 pixels
- *vertical display region*: 600 lines
- *vertical bottom border*: 23 lines
- *vertical top border*: 37 lines
- *vertical retrace*: 6 lines

We wish to create a dual-mode synchronization circuit that can support both VGA and SVGA modes. The mode can be selected by a switch. Construct the circuit as follows:

1. Modify the horizontal and vertical synchronization counters of Listing 13.1 to accommodate both modes.
2. Design a pixel-generating circuit that draws a 100-pixel grid on the screen (i.e., draw a vertical line every 100 pixels and draw a horizontal line every 100 pixels).
3. Derive a top-level module. Synthesize and verify operation of the two modes.

13.7.3 Visible screen adjustment circuit

Due to the internal timing error of a monitor, the visible portion of the screen may not always be centered. We can adjust the location of the visible portion by slightly modifying the widths surrounding black border areas. In a horizontal scan line, there are 64 pixels for the right and left border regions. To move the visible portion horizontally, we can add a certain number of pixels to one border region and subtract the same number from the opposite border region. We can adjust the visible portion vertically in a similar fashion. Design a screen adjustment circuit as follows:

1. Expand the VGA synchronization circuit to include this feature. Use a switch to select the vertical or horizontal mode, and use two pushbuttons to move the visible screen to left/up and right/down.
2. Modify the testing circuit in Section 13.2.5 to incorporate the new synchronization circuit.
3. Synthesize and verify operation of the circuit.

13.7.4 Ball-in-a-box circuit

The ball-in-a-box circuit displays a bouncing ball inside a square box. The square box is centered on the screen and its size is 256-by-256 pixels. The ball is an 8-by-8 round ball. When the ball hits the wall, the ball bounces back and the wall flashes (i.e., changes color briefly). The ball can travel at four different speeds, which are selected by two slide

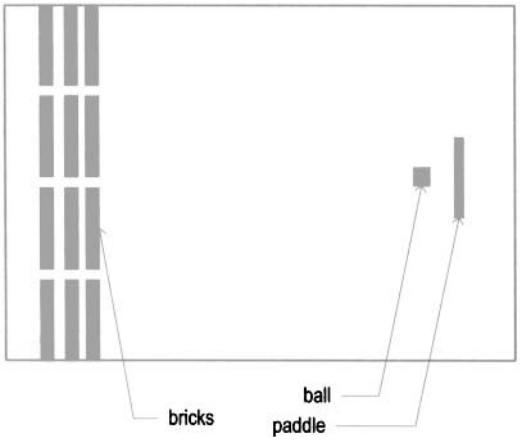


Figure 13.11 Screen of the breakout game.

switches, and its direction changes randomly when a pushbutton switch is pressed. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.5 Two-balls-in-a-box circuit

We can expand the circuit in Experiment 13.7.4 to include two balls inside the box. When two balls collide, the new directions of the two balls should follow the laws of physics. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.6 Two-player pong game

The two-player pong game replaces the left wall with another paddle, which is controlled by the second player. To better accommodate two players, we can use the keyboard interface of Section 9.4 as the input device. Four keys can be defined to control vertical movements of the two paddles. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.7 Breakout game

The breakout game is somewhat like the pong game. In this game, the left wall is replaced by several layers of "bricks." When the ball hits a brick, the ball bounces back and the brick disappears. The basic screen is shown in Figure 13.11. As in the code of Listing 13.5, we assume that the game runs continuously. Derive the HDL code and then synthesize and verify operation of the circuit.

13.7.8 Full-screen dot trace

We can implement the full-screen dot trace circuit of Section 13.5 using the external SRAM chip as follows:

1. Modify the SRAM controller in Chapter 11 to configure the SRAM chip as a 2^{19}-by-8 memory.

2. Follow the discussion in Section 13.5.2 to incorporate the new memory module in the circuit. Note that accessing the external memory requires two clock cycles.
3. Synthesize and verify operation of the circuit.

13.7.9 Mouse pointer circuit

The mouse interface is discussed in Section 10.5. The mouse pointer circuit uses a mouse to control the movement of a small 16-by-16 square on the screen. It functions as follows:

- The square moves according to the movement of the mouse.
- The pointer wraps around when it reaches a border.
- The pointer changes color when the left button of the mouse is pressed. It circulates through the eight colors defined in Table 13.1.

Synthesize and verify operation of the circuit.

13.7.10 Small-screen mouse scribble circuit

Mouse scribble circuit keeps track of the trace of the mouse movement in a 128-by-128 screen, somewhat similar to the dot trace circuit discussed in Section 13.5. Its specification is as follows:

- The 3-bit switch determines the color of the trace.
- Clicking the left button of the mouse turns on and off the trace alternately.
- Clicking the right button of the mouse clears the screen.

Synthesize and verify operation of the circuit.

13.7.11 Full-screen mouse scribble circuit

Repeat Experiment 13.7.10, but use the full screen. An external SRAM module similar to that in Experiment 13.7.8 is needed for this circuit.

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.

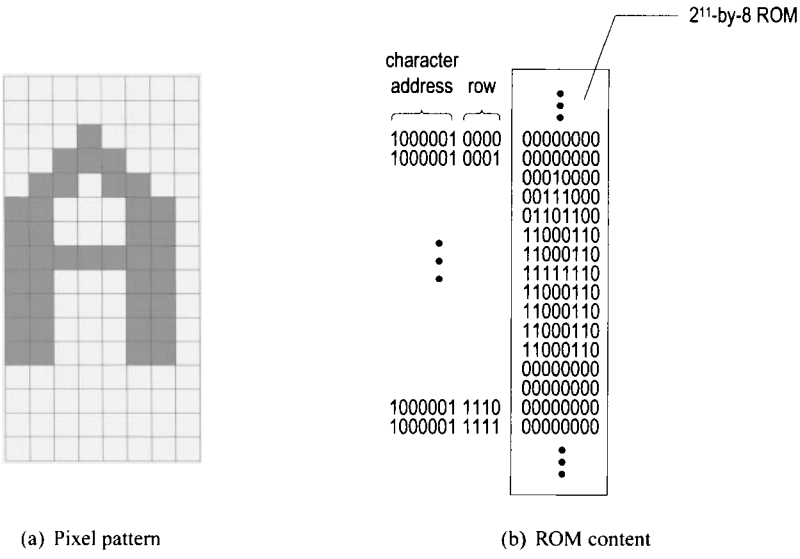
14.2 TEXT GENERATION

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
```

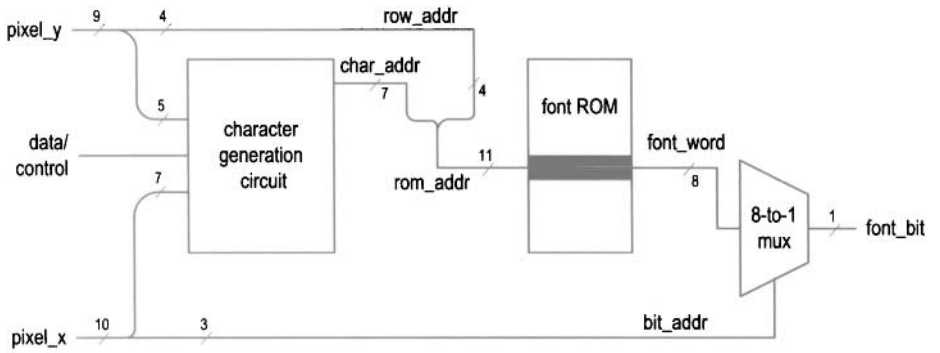


Figure 14.2 Two-stage text generation circuit.

```
            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.

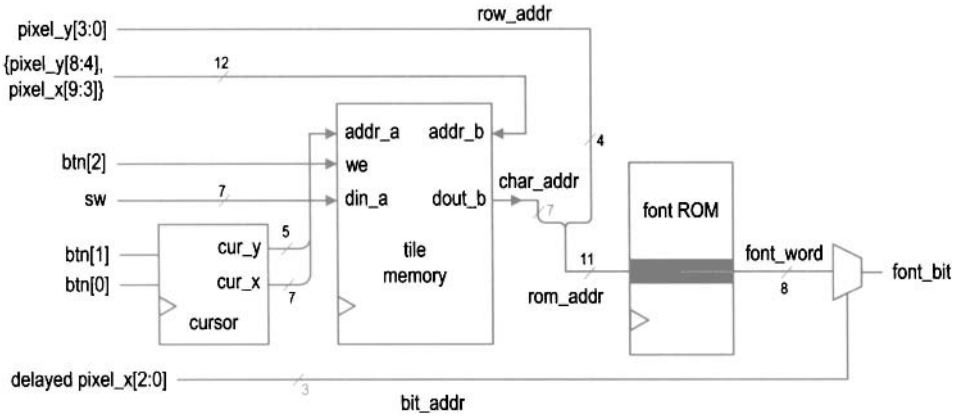


Figure 14.3 Text generation circuit with tile memory.

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
```

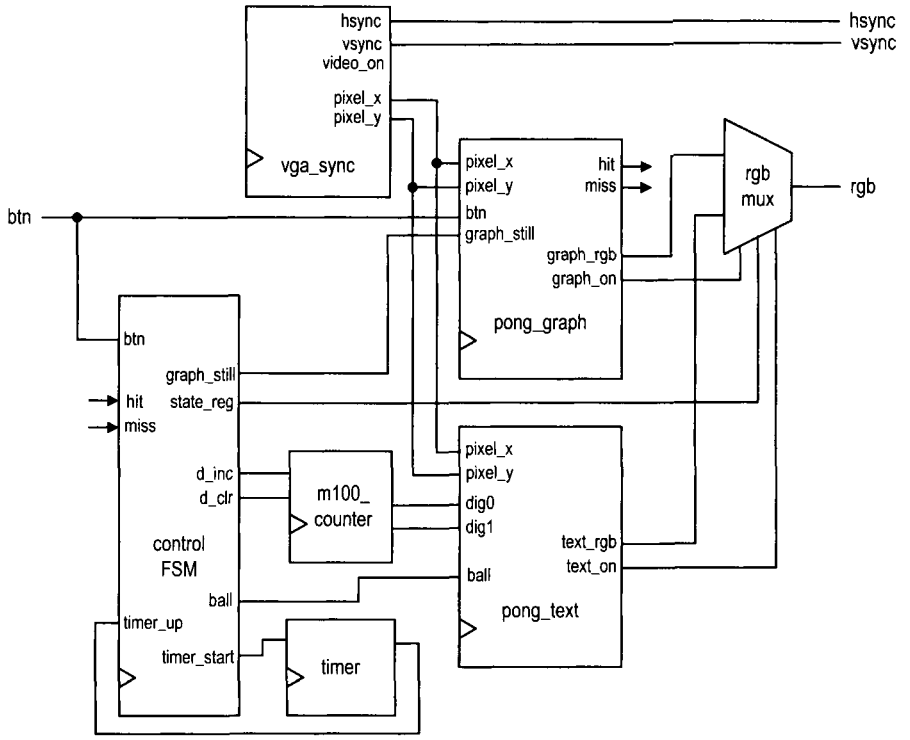


Figure 14.4 Top-level block diagram of the complete pong game.

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:

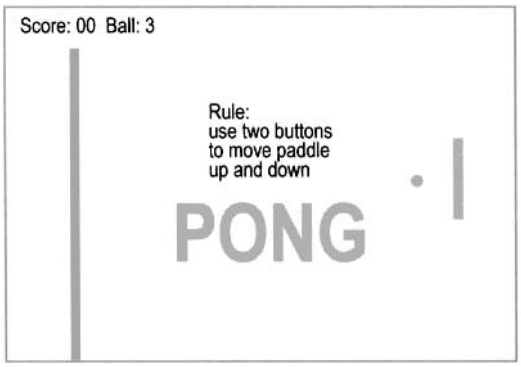


Figure 14.5 Text of the pong game.

- 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
215   end

      assign text_on = {score_on, logo_on, rule_on, over_on};
      //-------------------------------------------------
      // font rom interface
220   //-------------------------------------------------
      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 :
5                       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
10 always @*
   begin
      hit = 1'b0;
      miss = 1'b0;
      x_delta_next = x_delta_reg;
15    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
```

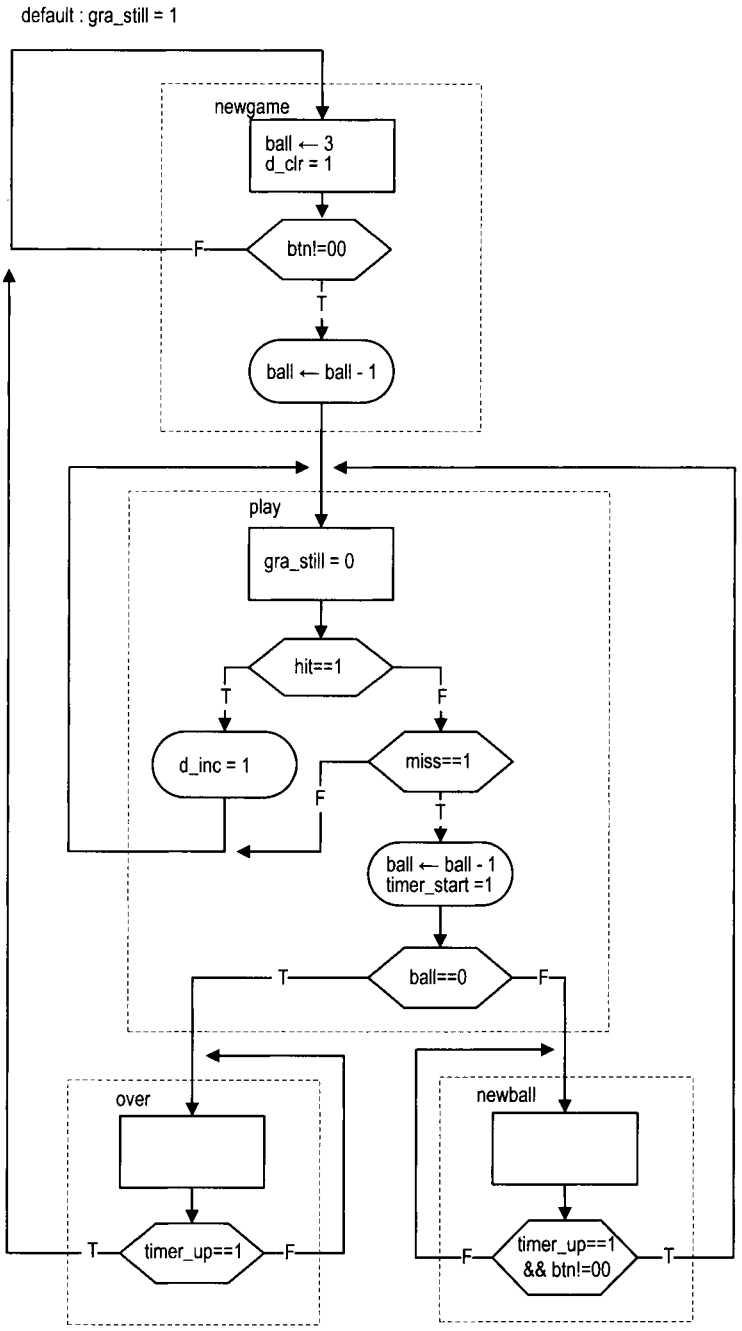


Figure 14.6 ASMD chart of the pong controller.

```
        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.

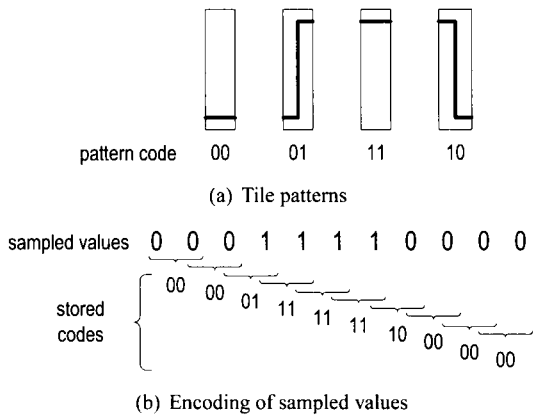


Figure 14.7 Tile patterns and encoding of a square wave.

- 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.