CHAPTER 12

VGA CONTROLLER I: GRAPHIC

12.1 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 13.

12.1.1 Basic operation of a CRT

The conceptual sketch of a monochrome CRT monitor is shown in Figure 12.1. 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 12.1. 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 12.2.
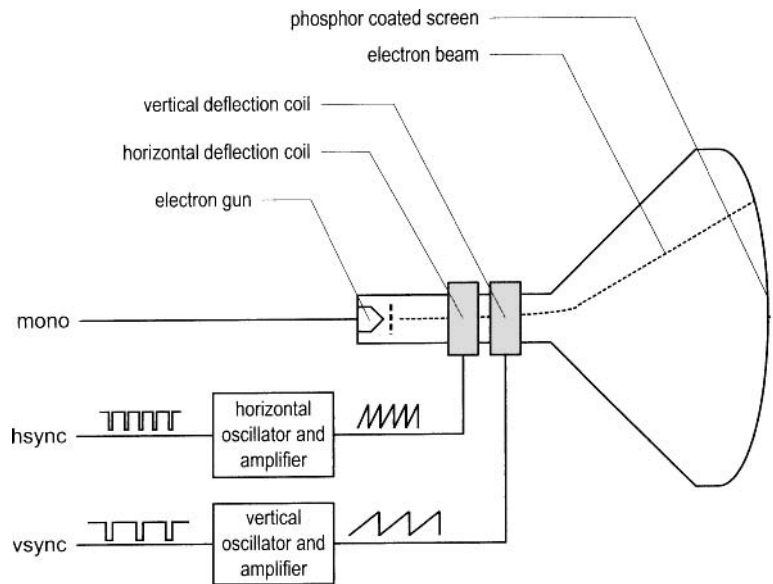
FPGA Prototyping by VHDL Examples. By Pong P. Chu
Copyright © 2008 John Wiley & Sons, Inc.

257

Figure 12.1 Conceptual diagram of a CRT monitor.

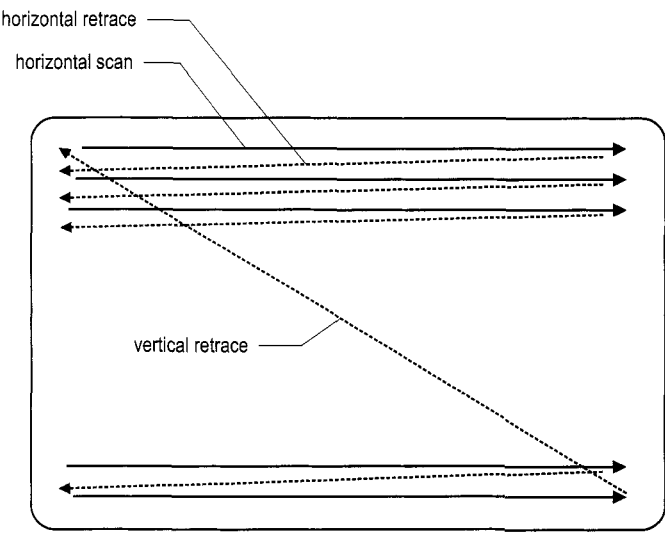**Figure 12.2** CRT scanning pattern.

Table 12.1   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 12.4. 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 12.4. 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.

12.1.2   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, 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 12.1. 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.

12.1.3   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 12.3. It contains a synchronization circuit, labeled vga_sync, and a pixel generation circuit.
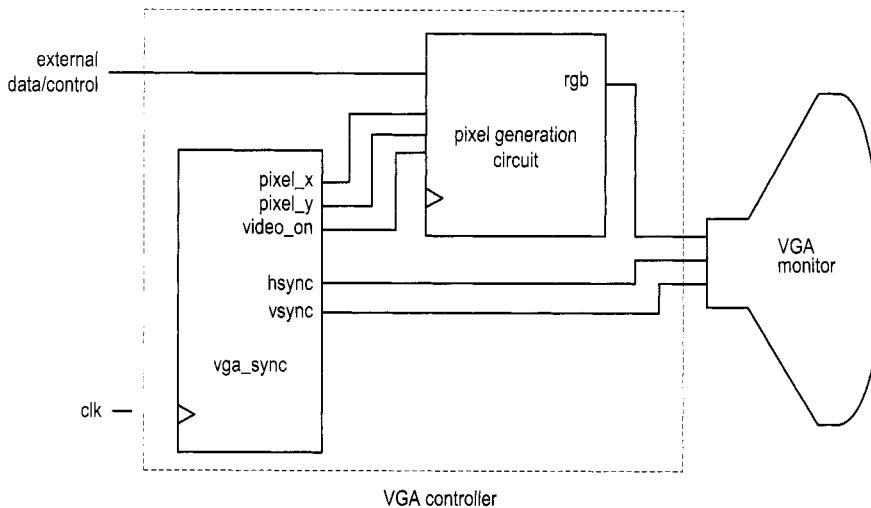
Figure 12.3   Simplified block diagram of a VGA controller.

The vga_sync circuit generates the 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 12.2.

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 13.

12.2 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 12.4. 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.

12.2.1   Horizontal synchronization

A detailed timing diagram of one horizontal scan is shown in Figure 12.4. A period of the hsync signal contains 800 pixels and can be divided into four regions:
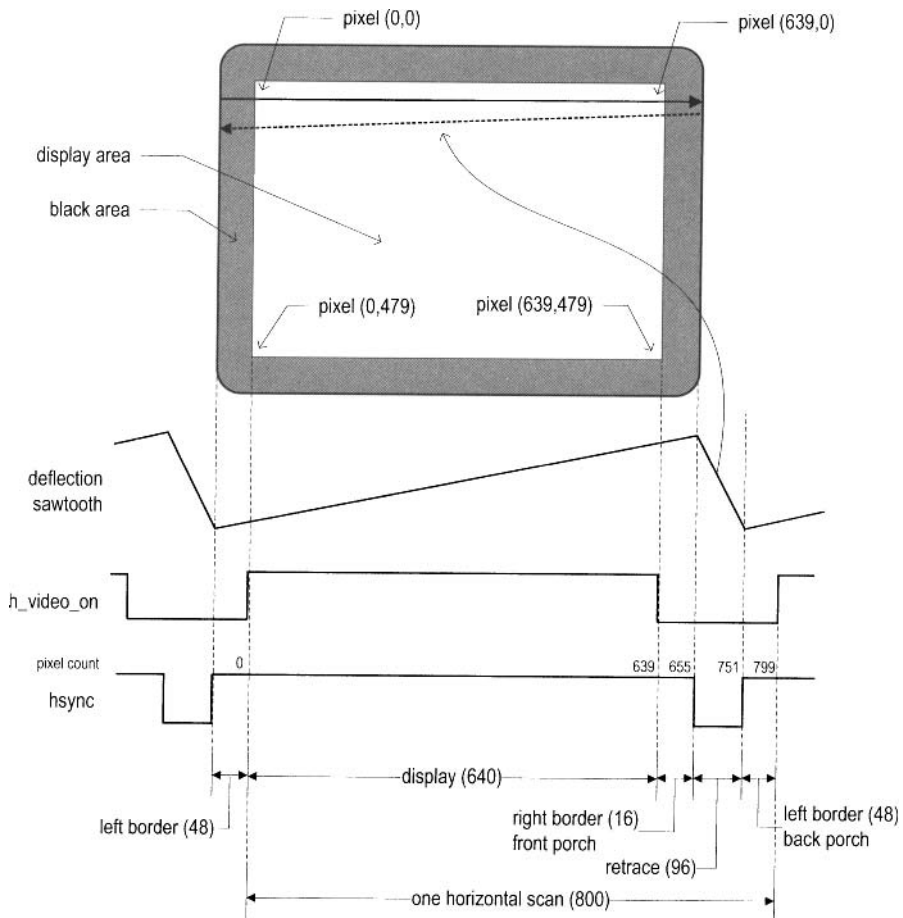
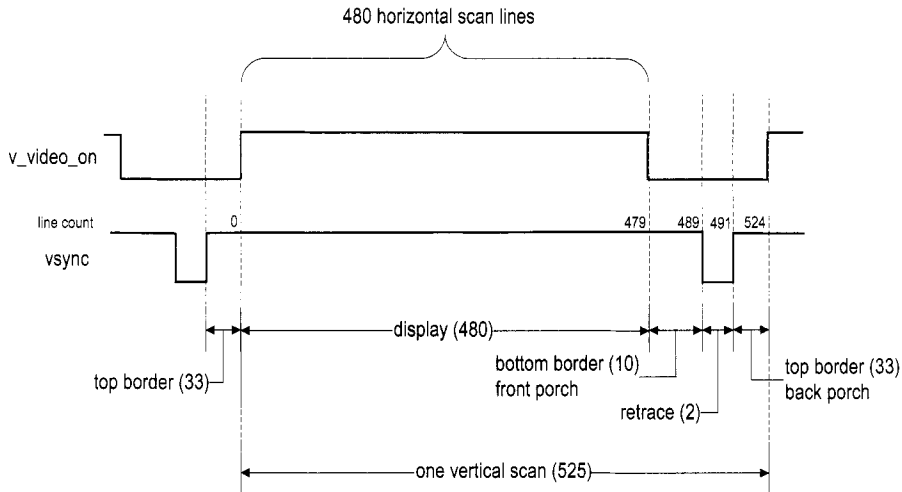**Figure 12.4**   Timing diagram of a horizontal scan.

Figure 12.5   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 12.4. 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.

12.2.2   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 12.5. 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.

12.2.3   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.

12.2.4   HDL implementation

The function of the vga_sync circuit is discussed in Section 12.1.3. 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 12.1. 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 be easily 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 12.1   VGA synchronization circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity vga_sync is
5    port(
         clk, reset: in std_logic;
         hsync, vsync: out std_logic;
         video_on, p_tick: out std_logic;
         pixel_x, pixel_y: out std_logic_vector (9 downto 0)
10       );
   end vga_sync;


   architecture arch of vga_sync is
       -- VGA 640-by-480 sync parameters
15     constant HD: integer:=640; --horizontal display area
       constant HF: integer:=16 ; --h. front porch
       constant HB: integer:=48 ; --h. back porch
       constant HR: integer:=96 ; --h. retrace
       constant VD: integer:=480; --vertical display area
20     constant VF: integer:=10;  --v. front porch
       constant VB: integer:=33;  --v. back porch
       constant VR: integer:=2;   --v. retrace
       -- mod-2 counter
       signal mod2_reg, mod2_next: std_logic;
25     -- sync counters
       signal v_count_reg, v_count_next: unsigned(9 downto 0);
       signal h_count_reg, h_count_next: unsigned(9 downto 0);
       -- output buffer
       signal v_sync_reg, h_sync_reg: std_logic;
30     signal v_sync_next, h_sync_next: std_logic;
       -- status signal
       signal h_end, v_end, pixel_tick: std_logic;
   begin
       -- registers
35     process (clk,reset)
```

```vhdl
begin
   if reset='1' then
      mod2_reg <= '0';
      v_count_reg <= (others=>'0');
      h_count_reg <= (others=>'0');
      v_sync_reg <= '0';
      h_sync_reg <= '0';
   elsif (clk'event and clk='1') then
      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 if;
end process;
-- mod-2 circuit to generate 25 MHz enable tick
mod2_next <= not mod2_reg;
-- 25 MHz pixel tick
pixel_tick <= '1' when mod2_reg='1' else '0';
-- status
h_end <=   -- end of horizontal counter
   '1' when h_count_reg=(HD+HF+HB+HR-1) else --799
   '0';
v_end <=   -- end of vertical counter
   '1' when v_count_reg=(VD+VF+VB+VR-1) else --524
   '0';
-- mod-800 horizontal sync counter
process (h_count_reg,h_end,pixel_tick)
begin
   if pixel_tick='1' then   -- 25 MHz tick
      if h_end='1' then
         h_count_next <= (others=>'0');
      else
         h_count_next <= h_count_reg + 1;
      end if;
   else
      h_count_next <= h_count_reg;
   end if;
end process;
-- mod-525 vertical sync counter
process (v_count_reg,h_end,v_end,pixel_tick)
begin
   if pixel_tick='1' and h_end='1' then
      if (v_end='1') then
         v_count_next <= (others=>'0');
      else
         v_count_next <= v_count_reg + 1;
      end if;
   else
      v_count_next <= v_count_reg;
   end if;
end process;
-- horizontal and vertical sync, buffered to avoid glitch
```

```
      h_sync_next <=
90       '1' when (h_count_reg >= (HD+HF))              ——656
                and (h_count_reg <= (HD+HF+HR-1)) else ——751
         '0';
      v_sync_next <=
         '1' when (v_count_reg >= (VD+VF))              ——490
95               and (v_count_reg <= (VD+VF+VR-1)) else ——491
         '0';
      -- video on/off
      video_on <=
         '1' when (h_count_reg < HD) and (v_count_reg < VD) else
100      '0';
      -- output signal
      hsync <= h_sync_reg;
      vsync <= v_sync_reg;
      pixel_x <= std_logic_vector(h_count_reg);
105   pixel_y <= std_logic_vector(v_count_reg);
      p_tick <= pixel_tick;
   end arch;
```

12.2.5 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 12.1. The HDL code is shown in Listing 12.2. As mentioned in Section 12.2.4, an output buffer is added for the rgb signal.

Listing 12.2 VGA synchronization testing circuit

```
   library ieee;
   use ieee.std_logic_1164.all;
   entity vga_test is
      port (
5        clk, reset: in std_logic;
         sw: in std_logic_vector(2 downto 0);
         hsync, vsync: out  std_logic;
         rgb: out std_logic_vector(2 downto 0)
      );
10 end vga_test;

   architecture arch of vga_test is
      signal rgb_reg: std_logic_vector(2 downto 0);
      signal video_on: std_logic;
15 begin
      -- instantiate VGA sync circuit
      vga_sync_unit: entity work.vga_sync
         port map(clk=>clk, reset=>reset, hsync=>hsync,
                  vsync=>vsync, video_on=>video_on,
20                p_tick=>open, pixel_x=>open, pixel_y=>open);
      -- rgb buffer
      process (clk,reset)
      begin
```

```
         if reset='1' then
25           rgb_reg <= (others=>'0');
         elsif (clk'event and clk='1') then
             rgb_reg <= sw;
         end if;
      end process;
30    rgb <= rgb_reg when video_on='1' else "000";
   end arch;
```

12.3 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 12.5.

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 13 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 12.4.

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.
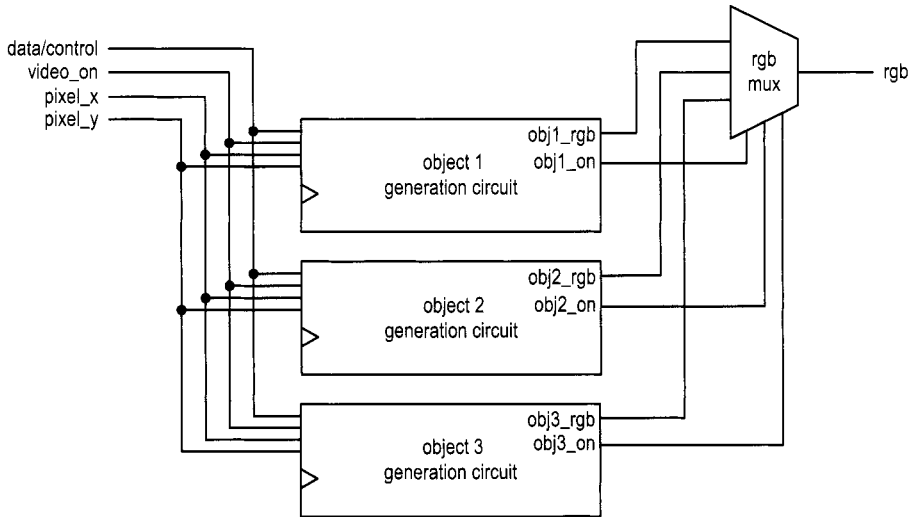
Figure 12.6 Conceptual diagram of object-mapped pixel generation.

12.4 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 12.6. 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 it with the current scan location provided by the pixel_x and pixel_y signals.
- 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 13.
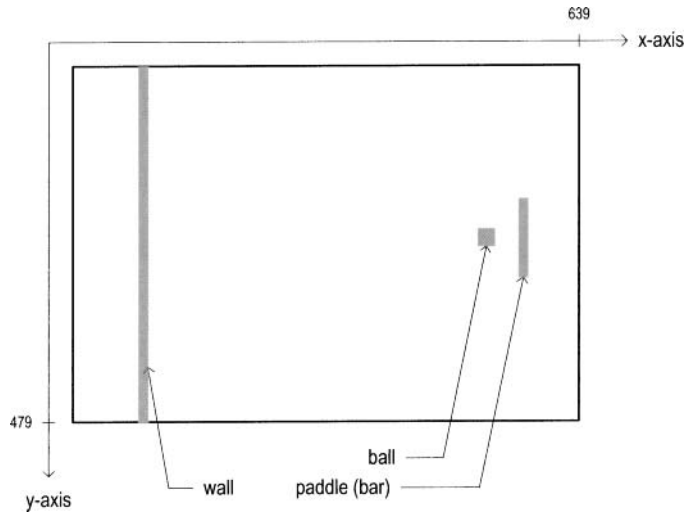
Figure 12.7   Still screen of the pong game.

12.4.1   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 12.7. 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 is 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

```
constant WALL_X_L: integer:=32;
constant WALL_X_R: integer:=35;
...
-- pixel within wall
wall_on <=
    '1' when (WALL_X_L<=pix_x) and (pix_x<=WALL_X_R) else
    '0';
-- wall rgb output
wall_rgb <= "001"; -- 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
constant BAR_X_L: integer:=600;
constant BAR_X_R: integer:=603;
```

```
—— bar  top ,  bottom  boundary
constant BAR_Y_SIZE: integer:=72;
constant BAR_Y_T: integer:=MAX_Y/2-BAR_Y_SIZE/2; ——204
constant BAR_Y_B: integer:=BAR_Y_T+BAR_Y_SIZE-1;
...
—— pixel  within  bar
bar_on <=
    '1' when (BAR_X_L<=pix_x) and (pix_x<=BAR_X_R) and
             (BAR_Y_T<=pix_y) and (pix_y<=BAR_Y_B) else
    '0';
—— bar  rgb  output
bar_rgb <= "010"; ——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

```
process (video_on ,wall_on ,bar_on ,sq_ball_on ,
         wall_rgb ,bar_rgb ,ball_rgb )
begin
    if video_on='0' then
        graph_rgb <= "000"; ——blank
    else
        if wall_on='1' then
            graph_rgb <= wall_rgb;
        elsif bar_on='1' then
            graph_rgb <= bar_rgb;
        elsif sq_ball_on='1' then
            graph_rgb <= ball_rgb;
        else
            graph_rgb <= "110"; —— yellow  background
        end if;
    end if;
end process;
```

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 12.3.

Listing 12.3 Pixel-generation circuit for the pong game screen

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

```vhdl
    port(
        video_on: in std_logic;
        pixel_x,pixel_y: in std_logic_vector(9 downto 0);
        graph_rgb: out std_logic_vector(2 downto 0)
    );
end pong_graph_st;

architecture sq_ball_arch of pong_graph_st is
    -- x, y coordinates (0,0) to (639,479)
    signal pix_x, pix_y: unsigned(9 downto 0);
    constant MAX_X: integer:=640;
    constant MAX_Y: integer:=480;

    ----------------------------------------------
    -- vertical stripe as a wall
    ----------------------------------------------
    -- wall left, right boundary
    constant WALL_X_L: integer:=32;
    constant WALL_X_R: integer:=35;

    ----------------------------------------------
    -- right vertical bar
    ----------------------------------------------
    -- bar left, right boundary
    constant BAR_X_L: integer:=600;
    constant BAR_X_R: integer:=603;
    -- bar top, bottom boundary
    constant BAR_Y_SIZE: integer:=72;
    constant BAR_Y_T: integer:=MAX_Y/2-BAR_Y_SIZE/2;  --204
    constant BAR_Y_B: integer:=BAR_Y_T+BAR_Y_SIZE-1;

    ----------------------------------------------
    -- square ball
    ----------------------------------------------
    constant BALL_SIZE: integer:=8;
    -- ball left, right boundary
    constant BALL_X_L: integer:=580;
    constant BALL_X_R: integer:=BALL_X_L+BALL_SIZE-1;
    -- ball top, bottom boundary
    constant BALL_Y_T: integer:=238;
    constant BALL_Y_B: integer:=BALL_Y_T+BALL_SIZE-1;

    ----------------------------------------------
    -- object output signals
    ----------------------------------------------
    signal wall_on, bar_on, sq_ball_on: std_logic;
    signal wall_rgb, bar_rgb, ball_rgb:
            std_logic_vector(2 downto 0);

begin
    pix_x <= unsigned(pixel_x);
    pix_y <= unsigned(pixel_y);

    ----------------------------------------------
    -- (wall) left vertical stripe
    ----------------------------------------------
    -- pixel within wall
    wall_on <=
```

```
                    '1' when (WALL_X_L<=pix_x) and (pix_x<=WALL_X_R) else
                    '0';
60  -- wall rgb output
    wall_rgb <= "001"; -- blue


    -- right vertical bar


65  -- pixel within bar
    bar_on <=
        '1' when (BAR_X_L<=pix_x) and (pix_x<=BAR_X_R) and
                 (BAR_Y_T<=pix_y) and (pix_y<=BAR_Y_B) else
        '0';
70  -- bar rgb output
    bar_rgb <= "010"; --green


    -- square ball


75  -- pixel within squared ball
    sq_ball_on <=
        '1' when (BALL_X_L<=pix_x) and (pix_x<=BALL_X_R) and
                 (BALL_Y_T<=pix_y) and (pix_y<=BALL_Y_B) else
        '0';
80  ball_rgb <= "100";    -- red


    -- rgb multiplexing circuit


    process(video_on,wall_on,bar_on,sq_ball_on,
85          wall_rgb, bar_rgb, ball_rgb)
    begin
        if video_on='0' then
            graph_rgb <= "000"; --blank
        else
90          if wall_on='1' then
                graph_rgb <= wall_rgb;
            elsif bar_on='1' then
                graph_rgb <= bar_rgb;
            elsif sq_ball_on='1' then
95              graph_rgb <= ball_rgb;
            else
                graph_rgb <= "110"; -- yellow background
            end if;
        end if;
100 end process;
    end sq_ball_arch;
```

After deriving the pixel generation circuit, we can combine it with the VGA synchro-nization circuit to construct the complete video interface. The top-level HDL code is shown in Listing 12.4. 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 12.4 Complete circuit for a still pong game screen

```
library ieee;
```

```
    use ieee.std_logic_1164.all;
    entity pong_top_st is
       port (
5          clk,reset: in std_logic;
           hsync, vsync: out  std_logic;
           rgb: out std_logic_vector(2 downto 0)
       );
    end pong_top_st;
10
    architecture arch of pong_top_st is
       signal pixel_x, pixel_y: std_logic_vector (9 downto 0);
       signal video_on, pixel_tick: std_logic;
       signal rgb_reg, rgb_next: std_logic_vector(2 downto 0);
15  begin
       -- instantiate VGA sync
       vga_sync_unit: entity work.vga_sync
          port map(clk=>clk, reset=>reset,
                   video_on=>video_on, p_tick=>pixel_tick,
20                 hsync=>hsync, vsync=>vsync,
                   pixel_x=>pixel_x, pixel_y=>pixel_y);
       -- instantiate graphic generator
       pong_grf_st_unit: entity work.pong_graph_st(sq_ball_arch)
          port map (video_on=>video_on,
25                  pixel_x=>pixel_x, pixel_y=>pixel_y,
                    graph_rgb=>rgb_next);
       -- rgb buffer
       process (clk)
       begin
30        if (clk'event and clk='1') then
             if (pixel_tick='1') then
                rgb_reg <= rgb_next;
             end if;
          end if;
35     end process;
       rgb <= rgb_reg;
    end arch;
```

12.4.2   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 12.8. 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 12.3 must be modified. First, we define a pattern ROM for the circle. It can be done by declaring a two-dimensional
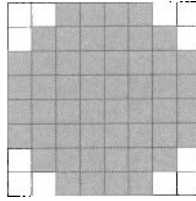
Figure 12.8 Bit map of a circle.

constant, as in the ROM template of Listing 11.5. To facilitate future animation, we also use signals to replace constants for the square ball boundaries. The revised architecture declaration portion becomes

```
constant BALL_SIZE: integer:=8;
-- ball left, right boundary
signal ball_x_l, ball_x_r: unsigned(9 downto 0);
-- ball top, bottom boundary
signal ball_y_t, ball_y_b: unsigned(9 downto 0);
-- =========================================
-- round ball image ROM
-- =========================================
type rom_type is array (0 to 7) of std_logic_vector(0 to 7);
-- ROM definition
constant BALL_ROM: rom_type :=
(
    "00111100", --    ****
    "01111110", --   ******
    "11111111", -- ********
    "11111111", -- ********
    "11111111", -- ********
    "11111111", -- ********
    "01111110", --   ******
    "00111100"  --    ****
);
signal rom_addr, rom_col: unsigned(2 downto 0);
signal rom_data: std_logic_vector(7 downto 0);
signal rom_bit: std_logic;
-- new signal to indicate whether the scan coordinates
-- are within the round ball region
signal rd_ball_on: std_logic;
```

Second, we expand the ball generation segment to include the mapping of the circle bit map:

```
-- pixel within square ball
sq_ball_on <=
    '1' when (ball_x_l<=pix_x) and (pix_x<=ball_x_r) and
             (ball_y_t<=pix_y) and (pix_y<=ball_y_b) else
    '0';
-- map current pixel location to ROM addr/col
rom_addr <= pix_y(2 downto 0) - ball_y_t(2 downto 0);
rom_col <= pix_x(2 downto 0) - ball_x_l(2 downto 0);
```

```
rom_data <= BALL_ROM(to_integer(rom_addr));
rom_bit <= rom_data(to_integer(rom_col));
rd_ball_on <=
    '1' when (sq_ball_on='1') and (rom_bit='1') else
    '0';
-- ball rgb output
ball_rgb <= "100";    -- 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 12.3 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 bit can then be retrieved by two indexing operations. 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:

```
process ...
    . . .
        elsif rd_ball_on='1' then
            graph_rgb <= ball_rgb;
    . . .
end process;
```

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

12.4.3 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
process(bar_y_reg,bar_y_b,bar_y_t,refr_tick,btn)
begin
    bar_y_next <= bar_y_reg; -- default, no move
    if refr_tick='1' then
        if btn(1)='1' and bar_y_b<(MAX_Y-1-BAR_V) then
        -- button 1 asserted and bar not reach bottom yet
            bar_y_next <= bar_y_reg + BAR_V; -- move down
        elsif btn(0)='1' and bar_y_t > BAR_V then
        -- button 0 asserted and bar not reach top yet
            bar_y_next <= bar_y_reg - BAR_V; -- move up
        end if;
    end if;
end process;
```

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
ball_x_next <=
    ball_x_reg + x_delta_reg when refr_tick='1' else
    ball_x_reg ;
ball_y_next <=
    ball_y_reg + y_delta_reg when refr_tick='1' else
    ball_y_reg ;
```

and the code segment for updating x_delta_reg and y_delta_reg is

```
-- new ball velocity
process(x_delta_reg,y_delta_reg,ball_y_t,ball_x_l,ball_x_r,
        ball_y_t,ball_y_b,bar_y_t,bar_y_b)
begin
    x_delta_next <= x_delta_reg; --default, no change
    y_delta_next <= y_delta_reg; --default, no change
    if ball_y_t < 1 then -- reach top
        y_delta_next <= BALL_V_P; --down
    elsif ball_y_b > (MAX_Y-1) then --reach bottom
        y_delta_next <= BALL_V_N; --up
    elsif ball_x_l <= WALL_X_R  then --reach wall
        x_delta_next <= BALL_V_P; --bounce back (to right)
    elsif (BAR_X_L<=ball_x_r) and (ball_x_r<=BAR_X_R) then
        -- reach x-coordinate of bar
        if (bar_y_t<=ball_y_b) and (ball_y_t<=bar_y_b) then
            -- within y-range of bar, hit
            x_delta_next <= BALL_V_N; --bounce back (to left)
        end if;
    end if;
end process;
```

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

The complete code is shown in Listing 12.5.

Listing 12.5 Pixel-generation circuit for the animated pong game

```
   library ieee;
   use ieee.std_logic_1164.all;
   use ieee.numeric_std.all;
   entity pong_graph_animate is
5      port(
            clk, reset: std_logic;
            btn: std_logic_vector(1 downto 0);
            video_on: in std_logic;
            pixel_x,pixel_y: in std_logic_vector(9 downto 0);
10           graph_rgb: out std_logic_vector(2 downto 0)
       );
   end pong_graph_animate;

   architecture arch of pong_graph_animate is
15     signal refr_tick: std_logic;
       -- x, y coordinates (0,0) to (639,479)
       signal pix_x, pix_y: unsigned(9 downto 0);
       constant MAX_X: integer:=640;
       constant MAX_Y: integer:=480;
20     ------------------------------------------------
       -- vertical stripe as a wall
       ------------------------------------------------
       -- wall left, right boundary
       constant WALL_X_L: integer:=32;
25     constant WALL_X_R: integer:=35;
       ------------------------------------------------
       -- right paddle bar
       ------------------------------------------------
       -- bar left, right boundary
30     constant BAR_X_L: integer:=600;
       constant BAR_X_R: integer:=603;
       -- bar top, bottom boundary
       signal bar_y_t, bar_y_b: unsigned(9 downto 0);
       constant BAR_Y_SIZE: integer:=72;
35     -- reg to track top boundary (x position is fixed)
       signal bar_y_reg, bar_y_next: unsigned(9 downto 0);
       -- bar moving velocity when a button is pressed
       constant BAR_V: integer:=4;
       ------------------------------------------------
40     -- square ball
       ------------------------------------------------
       constant BALL_SIZE: integer:=8; -- 8
       -- ball left, right boundary
       signal ball_x_l, ball_x_r: unsigned(9 downto 0);
45     -- ball top, bottom boundary
       signal ball_y_t, ball_y_b: unsigned(9 downto 0);
       -- reg to track left, top boundary
       signal ball_x_reg, ball_x_next: unsigned(9 downto 0);
```

```vhdl
   signal ball_y_reg, ball_y_next: unsigned(9 downto 0);
50 -- reg to track ball speed
   signal x_delta_reg, x_delta_next: unsigned(9 downto 0);
   signal y_delta_reg, y_delta_next: unsigned(9 downto 0);
   -- ball velocity can be pos or neg
   constant BALL_V_P: unsigned(9 downto 0)
55             :=to_unsigned(2,10);
   constant BALL_V_N: unsigned(9 downto 0)
               :=unsigned(to_signed(-2,10));
```

```vhdl
   -- round ball image ROM
```

```vhdl
60
   type rom_type is array (0 to 7)
         of std_logic_vector(0 to 7);
   -- ROM definition
   constant BALL_ROM: rom_type :=
65 (
      "00111100", --    ****
      "01111110", --   ******
      "11111111", -- ********
      "11111111", -- ********
70    "11111111", -- ********
      "11111111", -- ********
      "01111110", --   ******
      "00111100"  --    ****
   );
75 signal rom_addr, rom_col: unsigned(2 downto 0);
   signal rom_data: std_logic_vector(7 downto 0);
   signal rom_bit: std_logic;
```

```vhdl
   -- object output signals
```

```vhdl
80
   signal wall_on, bar_on, sq_ball_on, rd_ball_on: std_logic;
   signal wall_rgb, bar_rgb, ball_rgb:
         std_logic_vector(2 downto 0);
begin
85 -- registers
   process (clk,reset)
   begin
      if reset='1' then
         bar_y_reg <= (others=>'0');
90       ball_x_reg <= (others=>'0');
         ball_y_reg <= (others=>'0');
         x_delta_reg <= ("0000000100");
         y_delta_reg <= ("0000000100");
      elsif (clk'event and clk='1') then
95       bar_y_reg <= bar_y_next;
         ball_x_reg <= ball_x_next;
         ball_y_reg <= ball_y_next;
         x_delta_reg <= x_delta_next;
         y_delta_reg <= y_delta_next;
100      end if;
   end process;
```

```
      pix_x <= unsigned(pixel_x);
      pix_y <= unsigned(pixel_y);
      -- refr_tick: 1-clock tick asserted at start of v-sync
105   --          i.e., when the screen is refreshed (60 Hz)
      refr_tick <= '1' when (pix_y=481) and (pix_x=0) else
                    '0';

      -- (wall) left vertical stripe
110
      -- pixel within wall
      wall_on <=
          '1' when (WALL_X_L<=pix_x) and (pix_x<=WALL_X_R) else
          '0';
115   -- wall rgb output
      wall_rgb <= "001"; -- blue

      -- right vertical bar

120   -- boundary
      bar_y_t <= bar_y_reg;
      bar_y_b <= bar_y_t + BAR_Y_SIZE - 1;
      -- pixel within bar
      bar_on <=
125       '1' when (BAR_X_L<=pix_x) and (pix_x<=BAR_X_R) and
                   (bar_y_t<=pix_y) and (pix_y<=bar_y_b) else
          '0';
      -- bar rgb output
      bar_rgb <= "010"; --green
130   -- new bar y-position
      process(bar_y_reg,bar_y_b,bar_y_t,refr_tick,btn)
      begin
          bar_y_next <= bar_y_reg; -- no move
          if refr_tick='1' then
135           if btn(1)='1' and bar_y_b<(MAX_Y-1-BAR_V) then
                  bar_y_next <= bar_y_reg + BAR_V; -- move down
              elsif btn(0)='1' and bar_y_t > BAR_V then
                  bar_y_next <= bar_y_reg - BAR_V; -- move up
              end if;
140       end if;
      end process;


      -- square ball
145
      -- boundary
      ball_x_l <= ball_x_reg;
      ball_y_t <= ball_y_reg;
      ball_x_r <= ball_x_l + BALL_SIZE - 1;
150   ball_y_b <= ball_y_t + BALL_SIZE - 1;
      -- pixel within ball
      sq_ball_on <=
          '1' when (ball_x_l<=pix_x) and (pix_x<=ball_x_r) and
                   (ball_y_t<=pix_y) and (pix_y<=ball_y_b) else
```

```
155        '0';
        -- map current pixel location to ROM addr/col
        rom_addr <= pix_y(2 downto 0) - ball_y_t(2 downto 0);
        rom_col <= pix_x(2 downto 0) - ball_x_l(2 downto 0);
        rom_data <= BALL_ROM(to_integer(rom_addr));
160     rom_bit <= rom_data(to_integer(rom_col));
        -- pixel within ball
        rd_ball_on <=
            '1' when (sq_ball_on='1') and (rom_bit='1') else
            '0';
165     -- ball rgb output
        ball_rgb <= "100";    -- red
        -- new ball position
        ball_x_next <= ball_x_reg + x_delta_reg
                        when refr_tick='1' else
170                     ball_x_reg ;
        ball_y_next <= ball_y_reg + y_delta_reg
                        when refr_tick='1' else
                        ball_y_reg ;
        -- new ball velocity
175     process(x_delta_reg,y_delta_reg,ball_y_t,ball_x_l,ball_x_r,
                ball_y_t,ball_y_b,bar_y_t,bar_y_b)
        begin
            x_delta_next <= x_delta_reg;
            y_delta_next <= y_delta_reg;
180         if ball_y_t < 1 then -- reach top
                y_delta_next <= BALL_V_P;
            elsif ball_y_b > (MAX_Y-1) then    -- reach bottom
                y_delta_next <= BALL_V_N;
            elsif ball_x_l <= WALL_X_R  then -- reach wall
185             x_delta_next <= BALL_V_P;      -- bounce back
            elsif (BAR_X_L<=ball_x_r) and (ball_x_r<=BAR_X_R) then
                -- reach x of right bar
                if (bar_y_t<=ball_y_b) and (ball_y_t<=bar_y_b) then
                    x_delta_next <= BALL_V_N; --hit, bounce back
190             end if;
            end if;
        end process;

        -- rgb multiplexing circuit
195
        process(video_on,wall_on,bar_on,rd_ball_on,
                wall_rgb, bar_rgb, ball_rgb)
        begin
            if video_on='0' then
200             graph_rgb <= "000"; --blank
            else
                if wall_on='1' then
                    graph_rgb <= wall_rgb;
                elsif bar_on='1' then
205                 graph_rgb <= bar_rgb;
                elsif rd_ball_on='1' then
                    graph_rgb <= ball_rgb;
```

```
                  else
                      graph_rgb <= "110";  —— yellow background
210                end if;
              end if;
        end process;
    end arch;
```

As in the still screen, we can combine the synchronization circuit and create the top-level description. The HDL code is shown in Listing 12.6.

Listing 12.6 Complete circuit for the animated pong game screen

```
    library ieee;
    use ieee.std_logic_1164.all;
    entity pong_top_an is
        port (
5          clk,reset: in std_logic;
           btn: in std_logic_vector (1 downto 0);
           hsync, vsync: out  std_logic;
           rgb: out std_logic_vector(2 downto 0)
        );
10  end pong_top_an;

    architecture arch of pong_top_an is
        signal pixel_x, pixel_y: std_logic_vector (9 downto 0);
        signal video_on, pixel_tick: std_logic;
15      signal rgb_reg, rgb_next: std_logic_vector(2 downto 0);
    begin
        —— instantiate VGA sync
        vga_sync_unit: entity work.vga_sync
            port map(clk=>clk, reset=>reset,
20                   video_on=>video_on, p_tick=>pixel_tick,
                     hsync=>hsync, vsync=>vsync,
                     pixel_x=>pixel_x, pixel_y=>pixel_y);
        —— instantiate graphic generator
        pong_graph_an_unit: entity work.pong_graph_animate
25          port map (clk=>clk, reset=>reset,
                      btn=>btn, video_on=>video_on,
                      pixel_x=>pixel_x, pixel_y=>pixel_y,
                      graph_rgb=>rgb_next);
        —— rgb buffer
30      process (clk)
        begin
            if (clk'event and clk='1') then
                if (pixel_tick='1') then
                    rgb_reg <= rgb_next;
35              end if;
            end if;
        end process;
        rgb <= rgb_reg;
    end arch;
```

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 13.
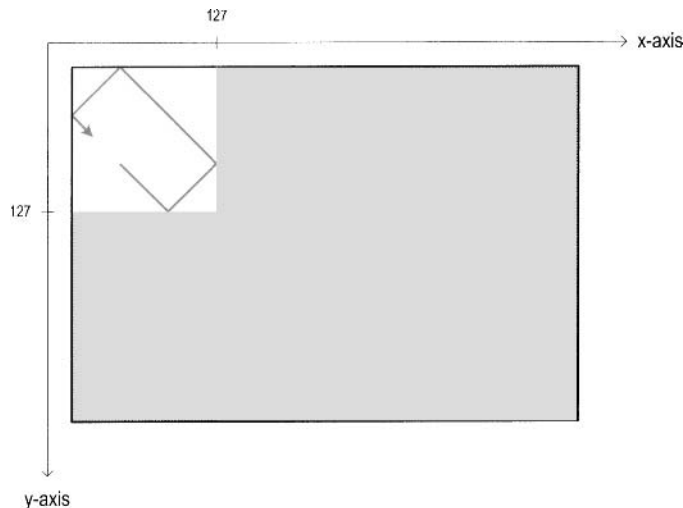
Figure 12.9   Dot trace shown in a 128-by-128 bit map.

12.5   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 translating 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 10 and 11. 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 12.9. 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 12.4.3. 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.

12.5.1   Dual-port RAM implementation

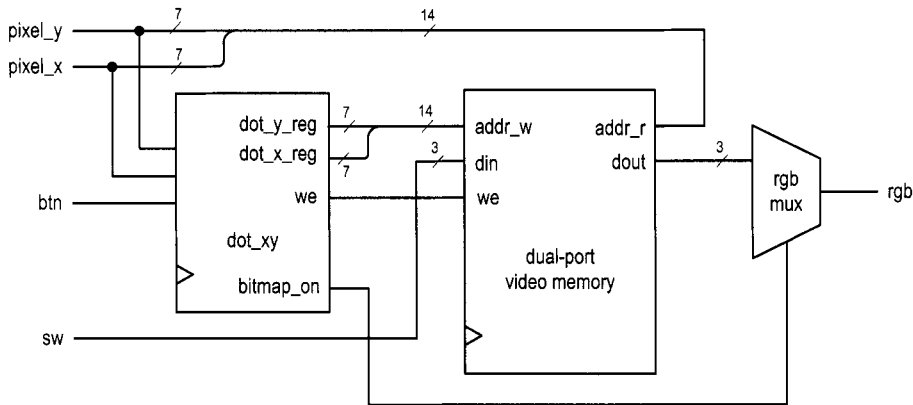A conceptual block diagram of this circuit is shown in Figure 12.10. The video memory is a

Figure 12.10   Conceptual block diagram of a dot trace circuit.

synchronous 16K-by-3 (i.e., 2^{14}-by-3) dual-port RAM. The dual-port module discussed in Listing 11.4 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 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 12.7. 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 12.4.3. 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 12.7   Pixel-generation circuit for a 128-by-128 bit map

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity bitmap_gen is
    port(
        clk, reset: std_logic;
        btn: std_logic_vector(1 downto 0);
        sw: std_logic_vector(2 downto 0);
        video_on: in std_logic;
        pixel_x,pixel_y: in std_logic_vector(9 downto 0);
        bit_rgb: out std_logic_vector(2 downto 0)
    );
end bitmap_gen;
```

```vhdl
15 architecture dual_port_ram_arch of bitmap_gen is
      signal pix_x, pix_y: unsigned(9 downto 0);
      signal refr_tick: std_logic;
      signal load_tick: std_logic;

20    -- video sram

      signal we: std_logic;
      signal addr_r, addr_w: std_logic_vector(13 downto 0);
      signal din, dout: std_logic_vector(2 downto 0);
25
      -- dot location and velocity

      constant MAX_X: integer:=128;
      constant MAX_Y: integer:=128;
30    -- dot velocity can be pos or neg
      constant DOT_V_P: unsigned(6 downto 0)
              :=to_unsigned(1,7);
      constant DOT_V_N: unsigned(6 downto 0)
              :=unsigned(to_signed(-1,7));
35    -- reg to keep track of dot location
      signal dot_x_reg, dot_x_next: unsigned(6 downto 0);
      signal dot_y_reg, dot_y_next: unsigned(6 downto 0);
      -- reg to keep track of dot velocity
      signal v_x_reg, v_x_next: unsigned(6 downto 0);
40    signal v_y_reg, v_y_next: unsigned(6 downto 0);

      -- object output signals

      signal bitmap_on: std_logic;
45    signal bitmap_rgb: std_logic_vector(2 downto 0);
  begin
      -- instantiate debounce circuit for a button
      debounce_unit: entity work.debounce
        port map(clk=>clk, reset=>reset, sw=>btn(0),
50              db_level=>open, db_tick=>load_tick);
      -- instantiate dual-port video RAM (2^12-by-7)
      video_ram: entity work.xilinx_dual_port_ram_sync
        generic map(ADDR_WIDTH=>14, DATA_WIDTH=>3)
        port map(clk=>clk, we=>we,
55              addr_a=>addr_w, addr_b=>addr_r,
                din_a=>din, dout_a=>open, dout_b=>dout);
      -- video ram interface
      addr_w <= std_logic_vector(dot_y_reg & dot_x_reg);
      addr_r <=
60        std_logic_vector(pix_y(6 downto 0) & pix_x(6 downto 0));
      we <= '1';
      din <= sw;
      bitmap_rgb <= dout;
      -- registers
65    process (clk,reset)
      begin
          if reset='1' then
```

```
            dot_x_reg <= (others=>'0');
            dot_y_reg <= (others=>'0');
70          v_x_reg <= DOT_V_P;
            v_y_reg <= DOT_V_P;
         elsif (clk'event and clk='1') then
            dot_x_reg <= dot_x_next;
            dot_y_reg <= dot_y_next;
75          v_x_reg <= v_x_next;
            v_y_reg <= v_y_next;
         end if;
      end process;
      -- misc. signals
80    pix_x <= unsigned(pixel_x);
      pix_y <= unsigned(pixel_y);
      refr_tick <= '1' when (pix_y=481) and (pix_x=0) else
                    '0';
      -- pixel within bit map area
85    bitmap_on <=
         '1' when (pix_x<=127) and (pix_y<=127) else
         '0';
      -- dot position
      -- "randomly" load dot location when btn(0) pressed
90    dot_x_next <=
         pix_x(6 downto 0) when load_tick='1' else
         dot_x_reg + v_x_reg when refr_tick='1' else
         dot_x_reg ;
      dot_y_next <=
95       pix_y(6 downto 0) when load_tick='1' else
         dot_y_reg + v_y_reg when refr_tick='1' else
      dot_y_reg ;
      -- dot x velocity
      process(v_x_reg,dot_x_reg)
100   begin
         v_x_next <= v_x_reg;
         if dot_x_reg =1 then              -- reach left
            v_x_next <= DOT_V_P;           -- bounce back
         elsif dot_x_reg=(MAX_X-2) then -- reach right
105         v_x_next <= DOT_V_N;           -- bounce back
         end if;
      end process;
      -- dot y velocity
      process(v_y_reg,dot_y_reg)
110   begin
         v_y_next <= v_y_reg;
         if dot_y_reg =1 then              -- reach top
            v_y_next <= DOT_V_P;
         elsif dot_y_reg = (MAX_Y-2) then -- reach bottom
115         v_y_next <= DOT_V_N;
         end if;
      end process;
      -- rgb multiplexing circuit
      process(video_on,bitmap_on,bitmap_rgb)
120   begin
```

```
         if video_on='0' then
             bit_rgb <= "000"; ──blank
         else
             if bitmap_on='1' then
125              bit_rgb <= bitmap_rgb;
             else
                 bit_rgb <= "110"; ── yellow background
             end if;
         end if;
130     end process;
    end dual_port_ram_arch;
```

The HDL code for the top-level system is shown in Listing 12.8.

Listing 12.8   Complete circuit for a bit-mapped screen

```
library ieee;
use ieee.std_logic_1164.all;
entity dot_top is
    port (
5       clk,reset: in std_logic;
        btn: in std_logic_vector (1 downto 0);
        sw: in std_logic_vector (2 downto 0);
        hsync, vsync: out  std_logic;
        rgb: out std_logic_vector(2 downto 0)
10   );
    end dot_top;

    architecture arch of dot_top is
        signal pixel_x, pixel_y: std_logic_vector(9 downto 0);
15      signal video_on, pixel_tick: std_logic;
        signal rgb_reg, rgb_next: std_logic_vector(2 downto 0);
    begin
        ── instantiate VGA sync circuit
        vga_sync_unit: entity work.vga_sync
20          port map(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 bit-mapped pixel generator
25      bitmap_unit: entity work.bitmap_gen
            port map(clk=>clk, reset=>reset, btn=>btn, sw=>sw,
                    video_on=>video_on, pixel_x=>pixel_x,
                    pixel_y=>pixel_y, bit_rgb=>rgb_next);
        ── rgb buffer
30      process (clk)
        begin
            if (clk'event and clk='1') then
                if (pixel_tick='1') then
                    rgb_reg <= rgb_next;
35              end if;
            end if;
        end process;
        rgb <= rgb_reg;
```

```
end arch;
```

12.5.2   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 on 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 the 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 11.2 can be used for this purpose. The memory portion of Listing 12.7 now becomes

```
-- instantiate video sram
video_ram: entity work.xilinx_one_port_ram_sync
    generic map(ADDR_WIDTH=>14, DATA_WIDTH=>3)
    port map(clk=>clk, we=>we, addr=>addr,
             din=>din, dout=>dout);
-- video ram interface
addr_w <=std_logic_vector(dot_y_reg & dot_x_reg);
addr_r <=
    std_logic_vector(pix_y(6 downto 0) & pix_x(6 downto 0));
addr <= addr_w when refr_tick='1' else addr_r;
we <= refr_tick;
din <= sw;
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).

12.6   BIBLIOGRAPHIC NOTES

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

12.7   SUGGESTED EXPERIMENTS

12.7.1   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.

12.7.2   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 12.1 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.

12.7.3   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 12.2.5 to incorporate the new synchronization circuit.
3. Synthesize and verify operation of the circuit.

12.7.4   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
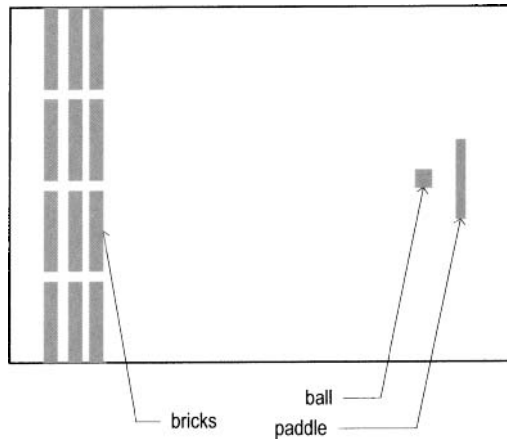
Figure 12.11   Screen of the breakout game.

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.

12.7.5   Two-balls-in-a-box circuit

We can expand the circuit in Experiment 12.7.4 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.

12.7.6   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 8.4 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.

12.7.7   Breakout game

The breakout game is a 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 12.11. As in the code of Listing 12.5, we assume that the game runs continuously. Derive the HDL code and then synthesize and verify operation of the circuit.

12.7.8   Full-screen dot trace

We can implement the full-screen dot trace circuit of Section 12.5 using the external SRAM chip as follows:

　　1. Modify the SRAM controller in Chapter 10 to configure the SRAM chip as a 2^{19}-by-8 memory.

2. Follow the discussion in Section 12.5.2 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.

12.7.9   Mouse pointer circuit

The mouse interface is discussed in Section 9.5. 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 12.1.

Synthesize and verify operation of the circuit.

12.7.10   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 12.5. 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.

12.7.11   Full-screen mouse scribble circuit

Repeat Experiment 12.7.10, but use the full screen. An external SRAM module similar to that in Experiment 12.7.8 is needed for this circuit.