CHAPTER 13

VGA CONTROLLER II: TEXT

13.1 INTRODUCTION

A tile-mapped pixel generation scheme is discussed in Section 12.3. 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.

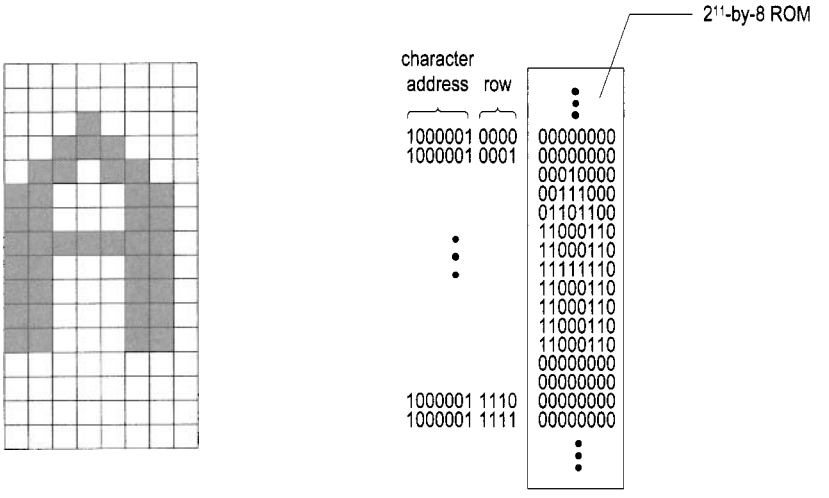
13.2 TEXT GENERATION

13.2.1 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 PC. In this font, each character is represented as an 8-by-16 pixel pattern. The pattern for the letter "A" is shown in Figure 13.1(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 13.1 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.

13.2.2 Font ROM

Our font set implements the 128 characters of the ASCII code, listed in Table 7.1. 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 13.1(b).

In the ASCII table, the first column (ASCII codes 00_{16} to $1F_{16}$) are 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 11.6 to ensure that a block RAM will be inferred during synthesis. Part of the HDL code is shown in Listing 13.1. The complete code has 2^{11} rows in constant definition and the file can be downloaded from the companion Web site.

Listing 13.1 Partial code of the font ROM

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity font_rom is
    port(
        clk: in std_logic;
        addr: in std_logic_vector(10 downto 0);
        data: out std_logic_vector(7 downto 0)
    );
end font_rom;

architecture arch of font_rom is
    constant ADDR_WIDTH: integer:=11;
    constant DATA_WIDTH: integer:=8;
    signal addr_reg: std_logic_vector(ADDR_WIDTH-1 downto 0);
    type rom_type is array (0 to 2**ADDR_WIDTH-1)
         of std_logic_vector(DATA_WIDTH-1 downto 0);
    -- ROM definition
    constant ROM: rom_type:=(    -- 2^11-by-8
    -- code x00 (blank space)
    "00000000", -- 0
    "00000000", -- 1
    "00000000", -- 2
    "00000000", -- 3
    "00000000", -- 4
    "00000000", -- 5
    "00000000", -- 6
    "00000000", -- 7
    "00000000", -- 8
    "00000000", -- 9
    "00000000", -- a
    "00000000", -- b
    "00000000", -- c
    "00000000", -- d
    "00000000", -- e
    "00000000", -- f
    -- code x01 (smiley face)
    "00000000", -- 0
    "00000000", -- 1
    "01111110", -- 2   ******
    "10000001", -- 3 *       *
    "10100101", -- 4 * *   * *
    "10000001", -- 5 *       *
    "10000001", -- 6 *       *
    "10111101", -- 7 * **** *
    "10011001", -- 8 *   **  *
    "10000001", -- 9 *       *
    "10000001", -- a *       *
    "01111110", -- b   ******
    "00000000", -- c
    "00000000", -- d
    "00000000", -- e
```

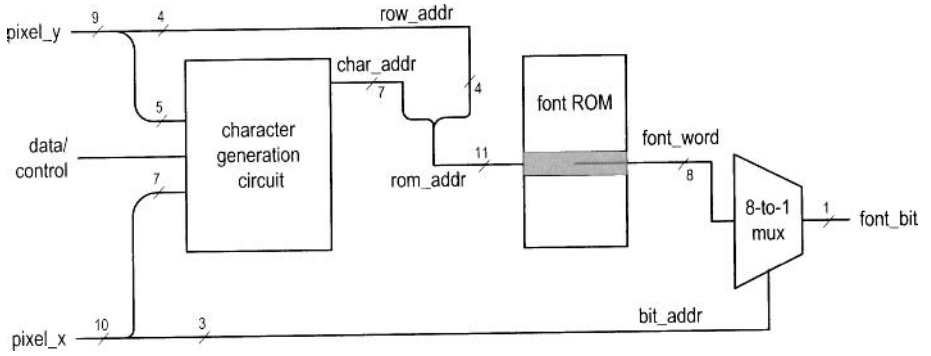


Figure 13.2 Two-stage text generation circuit.

```
    "00000000", -- f
    -- code x02
    . . .
    );
begin
    -- addr register to infer block RAM
    process (clk)
    begin
       if (clk'event and clk = '1') then
         addr_reg <= addr;
       end if;
    end process;
    data <= ROM(to_integer(unsigned(addr_reg)));
end arch;
```

Note that the block RAM-based ROM implementation introduces one-clock-cycle delay, as discussed in Section 11.4.3.

13.2.3 Basic text generation circuit

The pixel generation circuit generates the 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 13.2. 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 downto 3)` and `pixel_y(8 downto 4)` signals provides 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 downto 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 downto 0)`, labeled `bit_addr`] specify the desired pixel location, and an 8-to-1 multiplexer routes the pixel to the output.

13.2.4 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 7.1. 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

```
char_addr <= pixel_y(5 downto 4) & pixel_x(7 downto 3);
```

The `pixel_x(7 downto 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 downto 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 13.2.

Listing 13.2 Pixel generation of a font display circuit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity font_test_gen is
   port(
      clk: in std_logic;
      video_on: in std_logic;
      pixel_x, pixel_y: std_logic_vector(9 downto 0);
      rgb_text: out std_logic_vector(2 downto 0)
   );
end font_test_gen;

architecture arch of font_test_gen is
   signal rom_addr: std_logic_vector(10 downto 0);
   signal char_addr: std_logic_vector(6 downto 0);
   signal row_addr: std_logic_vector(3 downto 0);
   signal bit_addr: std_logic_vector(2 downto 0);
   signal font_word: std_logic_vector(7 downto 0);
   signal font_bit, text_bit_on: std_logic;
begin
   -- instantiate font ROM
   font_unit: entity work.font_rom
      port map(clk=>clk, addr=>rom_addr, data=>font_word);
   -- font ROM interface
   char_addr<=pixel_y(5 downto 4) & pixel_x(7 downto 3);
   row_addr<=pixel_y(3 downto 0);
   rom_addr <= char_addr & row_addr;
   bit_addr<=pixel_x(2 downto 0);
   font_bit <= font_word(to_integer(unsigned(not bit_addr)));
   -- "on" region limited to top-left corner
   text_bit_on <=
```

```
      font_bit when pixel_x(9 downto 8)="00" and
                    pixel_y(9 downto 6)="0000" else
      '0';
   -- rgb multiplexing circuit
   process(video_on,font_bit,text_bit_on)
   begin
      if video_on='0' then
         rgb_text <= "000"; --blank
      else
        if text_bit_on='1' then
           rgb_text <= "010"; -- green
        else
           rgb_text <= "000"; -- black
        end if;
      end if;
   end process;
end arch;
```

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 13.2:

- `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 13.2. The routing of the `font_bit` signal is done by a multiplexer, coded as an array with dynamic index:

```
font_bit <= font_word(to_integer(unsigned(not bit_addr)));
```

Note that a row (i.e., a word) in the font ROM is defined with a descending order [i.e., `(7 downto 0)`]. Since the screen's x-coordinate is defined in an ascending fashion, in which the numbers increases from left to right, the order of the retrieved bits must be reversed. This is achieved by the **not** operator in the expression.

We need to combine the synchronization circuit and create the top-level description. The HDL code is shown in Listing 13.3.

Listing 13.3 Top-level description of a font display circuit

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

architecture arch of font_test_top 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);
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,
               pixel_x=>pixel_x, pixel_y=>pixel_y,
               p_tick=>pixel_tick);
   -- instantiate font ROM
   font_gen_unit: entity work.font_test_gen
      port map(clk=>clk, video_on=>video_on,
               pixel_x=>pixel_x, pixel_y=>pixel_y,
               rgb_text=>rgb_next);
   -- rgb buffer
   process (clk)
   begin
      if (clk'event and clk='1') then
         if (pixel_tick='1') then
            rgb_reg <= rgb_next;
         end if;
      end if;
   end process;
   rgb <= rgb_reg;
end arch;
```

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 is remained 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.

13.2.5 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 the 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

```
row_addr<=pixel_y(4 downto 1);
bit_addr<=pixel_x(3 downto 1);
char_addr<=pixel_y(6 downto 5) & pixel_x(8 downto 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:

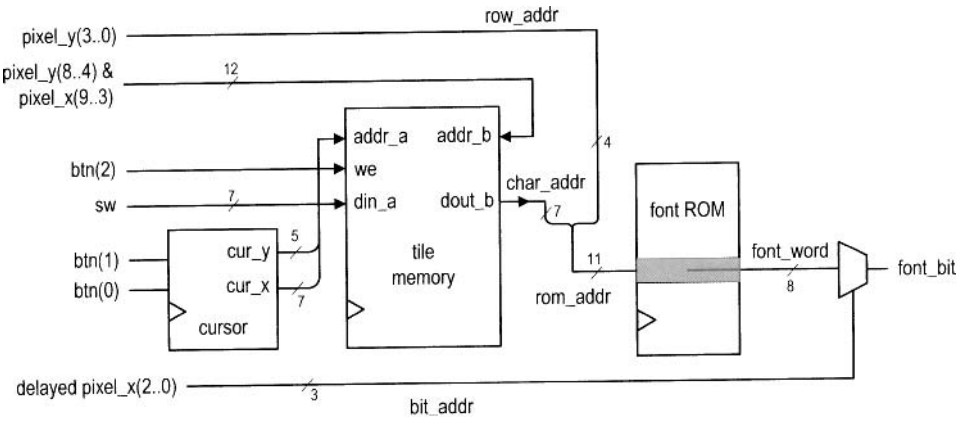


Figure 13.3 Text generation circuit with tile memory.

```
text_bit_on <=
   font_bit when pixel_x(9)="0" and
                 pixel_y(9 downto 7)="000" else
   '0';
```

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.

13.3 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 12.5. 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 13.3.

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

```
bit_addr<=pixel_x(2 downto 0);
font_bit <= font_word(to_integer(unsigned(not 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 13.3. 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 13.4.

Listing 13.4 Pixel generation of a full-screen text display

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity text_screen_gen is
   port(
      clk, reset: std_logic;
      btn: std_logic_vector(2 downto 0);
      sw: std_logic_vector(6 downto 0);
      video_on: in std_logic;
      pixel_x, pixel_y: in std_logic_vector(9 downto 0);
      text_rgb: out std_logic_vector(2 downto 0)
   );
end text_screen_gen;

architecture arch of text_screen_gen is
   -- font ROM
   signal char_addr: std_logic_vector(6 downto 0);
   signal rom_addr: std_logic_vector(10 downto 0);
   signal row_addr: std_logic_vector(3 downto 0);
   signal bit_addr: unsigned(2 downto 0);
   signal font_word: std_logic_vector(7 downto 0);
   signal font_bit: std_logic;
   -- tile RAM
   signal we: std_logic;
   signal addr_r, addr_w: std_logic_vector(11 downto 0);
   signal din, dout: std_logic_vector(6 downto 0);
   -- 80-by-30 tile map
   constant MAX_X: integer:=80;
   constant MAX_Y: integer:=30;
   -- cursor
   signal cur_x_reg, cur_x_next: unsigned(6 downto 0);
   signal cur_y_reg, cur_y_next: unsigned(4 downto 0);
   signal move_x_tick, move_y_tick: std_logic;
   signal cursor_on: std_logic;
   -- delayed pixel count
   signal pix_x1_reg, pix_y1_reg: unsigned(9 downto 0);
   signal pix_x2_reg, pix_y2_reg: unsigned(9 downto 0);
   -- object output signals
   signal font_rgb, font_rev_rgb:
            std_logic_vector(2 downto 0);
begin
   -- instantiate debounce circuit for two buttons
   debounce_unit0: entity work.debounce
      port map(clk=>clk, reset=>reset, sw=>btn(0),
               db_level=>open, db_tick=>move_x_tick);
```

```
    debounce_unit1: entity work.debounce
       port map(clk=>clk, reset=>reset, sw=>btn(1),
                db_level=>open, db_tick=>move_y_tick);
  -- instantiate font ROM
  font_unit: entity work.font_rom
     port map(clk=>clk, addr=>rom_addr, data=>font_word);
  -- instantiate dual-port tile RAM (2^12-by-7)
  video_ram: entity work.xilinx_dual_port_ram_sync
     generic map(ADDR_WIDTH=>12, DATA_WIDTH=>7)
     port map(clk=>clk, we=>we,
              addr_a=>addr_w, addr_b=>addr_r,
              din_a=>din, dout_a=>open, dout_b=>dout);
  -- registers
  process (clk)
  begin
    if (clk'event and clk='1') then
        cur_x_reg <= cur_x_next;
        cur_y_reg <= cur_y_next;
        pix_x1_reg <= unsigned(pixel_x); -- 2-clock delay
        pix_x2_reg <= pix_x1_reg;
        pix_y1_reg <= unsigned(pixel_y);
        pix_y2_reg <= pix_y1_reg;
    end if;
  end process;
  -- tile RAM write
  addr_w <=std_logic_vector(cur_y_reg & cur_x_reg);
  we <= btn(2);
  din <= sw;
  -- tile RAM read
  -- use undelayed coordinates to form tile RAM address
  addr_r <=pixel_y(8 downto 4) & pixel_x(9 downto 3);
  char_addr <= dout;
  -- font ROM
  row_addr<=pixel_y(3 downto 0);
  rom_addr <= char_addr & row_addr;
  -- use delayed coordinate to select a bit
  bit_addr<=pix_x2_reg(2 downto 0);
  font_bit <= font_word(to_integer(not bit_addr));
  -- new cursor position
  cur_x_next <=
     (others=>'0') when move_x_tick='1' and -- wrap around
                        cur_x_reg=MAX_X-1 else
     cur_x_reg + 1  when move_x_tick='1' else
     cur_x_reg ;
  cur_y_next <=
     (others=>'0') when move_y_tick='1' and -- wrap around
                        cur_y_reg=MAX_Y-1 else
     cur_y_reg + 1  when move_y_tick='1' else
     cur_y_reg;
  -- object signals
  -- green over black and reversed video for curser
  font_rgb <="010" when font_bit='1' else "000";
  font_rev_rgb <="000" when font_bit='1' else "010";
```

```
   -- use delayed coordinates for comparison
100 cursor_on <='1' when pix_y2_reg(8 downto 4)=cur_y_reg and
                         pix_x2_reg(9 downto 3)=cur_x_reg else
              '0';
   -- rgb multiplexing circuit
   process(video_on,cursor_on,font_rgb,font_rev_rgb)
105 begin
      if video_on='0' then
         text_rgb <= "000"; --blank
      else
         if cursor_on='1' then
110         text_rgb <= font_rev_rgb;
         else
            text_rgb <= font_rgb;
         end if;
      end if;
115 end process;
end arch;
```

The font ROM interface signals are similar to those in Listing 13.2 except that the char_addr is obtained from the read port of the tile memory. To facilitate the font ROM access delay, we creat 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 is similar to those of the video RAM in Listing 12.7.

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 the overall visible portion of the VGA monitor is shifted to 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 13.5.

Listing 13.5 Top-level system of a full-screen text display

```
library ieee;
use ieee.std_logic_1164.all;
entity text_screen_top is
   port(
5     clk,reset: in std_logic;
      btn: in std_logic_vector (2 downto 0);
      sw: in std_logic_vector (6 downto 0);
      hsync, vsync: out  std_logic;
      rgb: out std_logic_vector(2 downto 0)
10 );
end text_screen_top;

architecture arch of text_screen_top 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);
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, p_tick=>pixel_tick,
               pixel_x=>pixel_x, pixel_y=>pixel_y);
   -- instantiate full-screen text generator
   text_gen_unit: entity work.text_screen_gen
      port map(clk=>clk, reset=>reset, btn=>btn, sw=>sw,
               video_on=>video_on, pixel_x=>pixel_x,
               pixel_y=>pixel_y, text_rgb=>rgb_next);
   -- rgb buffer
   process (clk)
   begin
      if (clk'event and clk='1') then
         if (pixel_tick='1') then
            rgb_reg <= rgb_next;
         end if;
      end if;
   end process;
   rgb <= rgb_reg;
end arch;
```

13.4 THE COMPLETE PONG GAME

We create a free-running graphic circuit for the pong game in Section 12.4.3. 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 13.4.

13.4.1 Text subsystem

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

- Display the score as "`Scores: DD`" and the number of remaining balls as"`Ball: D`" in 16-by-32 font on top of the screen.

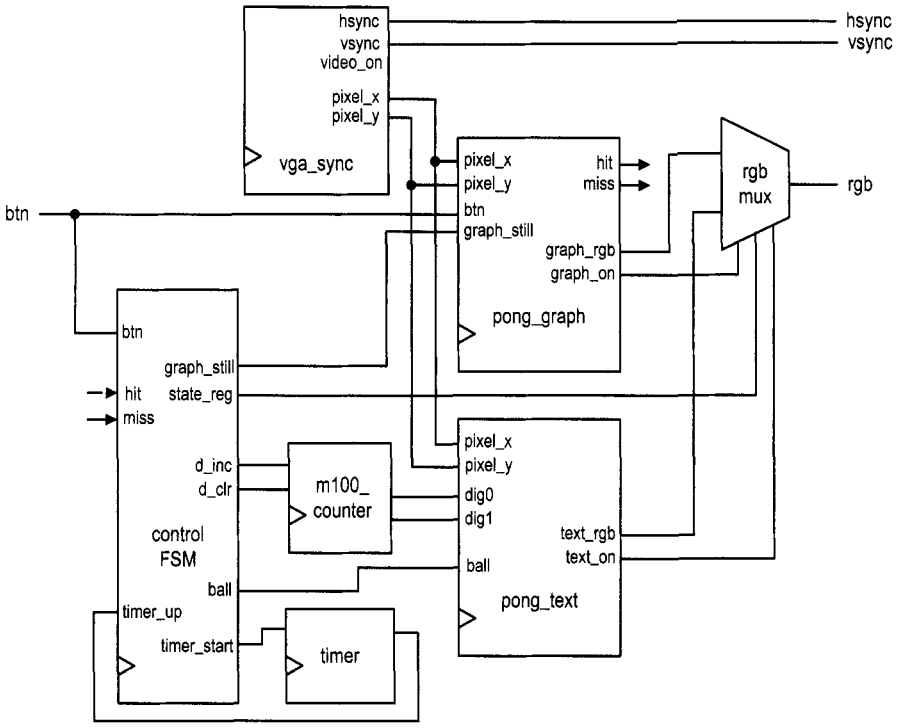


Figure 13.4 Top-level block diagram of the complete pong game.

- Display the rule message "`Rules: Use two buttons to move paddle up or down.`" in regular font at the beginning of the game.
- Display the "`PONG`" logo in 64-by-128 font on the background.
- Display the end-of-game message "`Game Over`" in 32-by-64 font at the end of the game.

A sketch of the first three messages is shown in Figure 13.5. 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

```
logo_on <=
   '1' when pix_y(9 downto 7)=2 and
      (3<= pix_x(9 downto 6) and pix_x(9 downto 6)<=6) else
   '0';
row_addr_l <= std_logic_vector(pix_y(6 downto 3));
bit_addr_l <= std_logic_vector(pix_x(5 downto 3));
with pix_x(8 downto 6) select
  char_addr_l <=
     "1010000" when "011", -- P x50
     "1001111" when "100", -- O x4f
     "1001110" when "101", -- N x4e
```

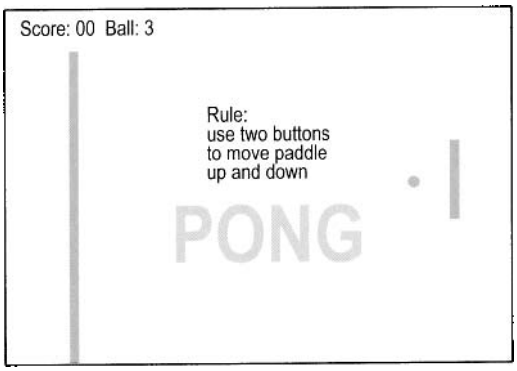


Figure 13.5 Text of the pong game.

```
"1000111" when others; --G x47
```

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 13.6.

Listing 13.6 Text subsystem for the pong game

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pong_text is
   port(
      clk, reset: in std_logic;
      pixel_x, pixel_y: in std_logic_vector(9 downto 0);
      dig0, dig1: in std_logic_vector(3 downto 0);
      ball: in std_logic_vector(1 downto 0);
      text_on: out std_logic_vector(3 downto 0);
      text_rgb: out std_logic_vector(2 downto 0)
   );
end pong_text;

architecture arch of pong_text is
   signal pix_x, pix_y: unsigned(9 downto 0);
   signal rom_addr: std_logic_vector(10 downto 0);
   signal char_addr, char_addr_s, char_addr_l, char_addr_r,
          char_addr_o: std_logic_vector(6 downto 0);
   signal row_addr, row_addr_s, row_addr_l,row_addr_r,
          row_addr_o: std_logic_vector(3 downto 0);
   signal bit_addr, bit_addr_s, bit_addr_l,bit_addr_r,
          bit_addr_o: std_logic_vector(2 downto 0);
```

```
    signal font_word: std_logic_vector(7 downto 0);
    signal font_bit: std_logic;
    signal score_on, logo_on, rule_on, over_on: std_logic;
    signal rule_rom_addr: unsigned(5 downto 0);
    type rule_rom_type is array (0 to 63) of
        std_logic_vector (6 downto 0);
    -- rule text ROM definition
    constant RULE_ROM: rule_rom_type :=
    (
       -- row 1
       "1010010", -- R
       "1010101", -- U
       "1001100", -- L
       "1000101", -- E
       "0111010", -- :
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       "0000000", --
       -- row 2
       "1010101", -- U
       "1110011", -- s
       "1100101", -- e
       "0000000", --
       "1110100", -- t
       "1110111", -- w
       "1101111", -- o
       "0000000", --
       "1100010", -- b
       "1110101", -- u
       "1110100", -- t
       "1110100", -- t
       "1101111", -- o
       "1101110", -- n
       "1110011", -- s
       "0000000", --
       -- row 3
       "1110100", -- t
       "1101111", -- o
       "0000000", --
       "1101101", -- m
       "1101111", -- o
       "1110110", -- v
       "1100101", -- e
       "0000000", --
       "1110000", -- p
```

```
        "1100001", -- a
        "1100100", -- d
        "1100100", -- d
        "1101100", -- l
        "1100101", -- e
        "0000000", --
        "0000000", --
        -- row 4
        "1110101", -- u
        "1110000", -- p
        "0000000", --
        "1100001", -- a
        "1101110", -- n
        "1100100", -- d
        "0000000", --
        "1100100", -- d
        "1101111", -- o
        "1110111", -- w
        "1101110", -- n
        "0101110", -- .
        "0000000", --
        "0000000", --
        "0000000", --
        "0000000"  --
    );
begin
    pix_x <= unsigned(pixel_x);
    pix_y <= unsigned(pixel_y);
    -- instantiate font ROM
    font_unit: entity work.font_rom
        port map(clk=>clk, addr=>rom_addr, data=>font_word);

    ------------------------------------------------------------
    -- score region
    --  - display score and ball at top left
    --  - text: "Score:DD Ball:D"
    --  - scale to 16-by-32 font
    ------------------------------------------------------------
    score_on <=
        '1' when pix_y(9 downto 5)=0 and
                 pix_x(9 downto 4)<16 else
        '0';
    row_addr_s <= std_logic_vector(pix_y(4 downto 1));
    bit_addr_s <= std_logic_vector(pix_x(3 downto 1));
    with pix_x(7 downto 4) select
      char_addr_s <=
        "1010011" when "0000", -- S x53
        "1100011" when "0001", -- c x63
        "1101111" when "0010", -- o x6f
        "1110010" when "0011", -- r x72
        "1100101" when "0100", -- e x65
        "0111010" when "0101", -- : x3a
        "011" & dig1 when "0110", -- digit 10
```

```
        "011" & dig0 when "0111", -- digit 1
        "0000000" when "1000",
        "0000000" when "1001",
        "1000010" when "1010", -- B x42
        "1100001" when "1011", -- a x61
        "1101100" when "1100", -- l x6c
        "1101100" when "1101", -- l x6c
        "0111010" when "1110", -- :
        "01100" & ball when others;

-- logo region:
--    - display logo "PONG" at top center
--    - used as background
--    - scale to 64-by-128 font
logo_on <=
   '1' when pix_y(9 downto 7)=2 and
      (3<= pix_x(9 downto 6) and pix_x(9 downto 6)<=6) else
   '0';
row_addr_l <= std_logic_vector(pix_y(6 downto 3));
bit_addr_l <= std_logic_vector(pix_x(5 downto 3));
with pix_x(8 downto 6) select
  char_addr_l <=
     "1010000" when "011", -- P x50
     "1001111" when "100", -- O x4f
     "1001110" when "101", -- N x4e
     "1000111" when others; --G x47

-- rule region
--    - display rule at center
--    - 4 lines, 16 characters each line
--    - rule text:
--         Rule:
--         Use two buttons
--         to move paddle
--         up and down
rule_on <= '1' when pix_x(9 downto 7) = "010" and
                    pix_y(9 downto 6)=  "0010"  else
           '0';
row_addr_r <= std_logic_vector(pix_y(3 downto 0));
bit_addr_r <= std_logic_vector(pix_x(2 downto 0));
rule_rom_addr <= pix_y(5 downto 4) & pix_x(6 downto 3);
char_addr_r <= RULE_ROM(to_integer(rule_rom_addr));

-- game over region
--  - display "Game Over" at center
--  - scale to 32-by-64 fonts
over_on <=
   '1' when pix_y(9 downto 6)=3 and
      5<= pix_x(9 downto 5) and pix_x(9 downto 5)<=13 else
```

```
      '0';
   row_addr_o <= std_logic_vector(pix_y(5 downto 2));
   bit_addr_o <= std_logic_vector(pix_x(4 downto 2));
   with pix_x(8 downto 5) select
     char_addr_o <=
        "1000111" when "0101", -- G x47
        "1100001" when "0110", -- a x61
        "1101101" when "0111", -- m x6d
        "1100101" when "1000", -- e x65
        "0000000" when "1001", --
        "1001111" when "1010", -- O x4f
        "1110110" when "1011", -- v x76
        "1100101" when "1100", -- e x65
        "1110010" when others; -- r x72
-----------------------------------------------------
-- mux for font ROM addresses and rgb
-----------------------------------------------------
   process(score_on,logo_on,rule_on,pix_x,pix_y,font_bit,
           char_addr_s,char_addr_l,char_addr_r,char_addr_o,
           row_addr_s,row_addr_l,row_addr_r,row_addr_o,
           bit_addr_s,bit_addr_l,bit_addr_r,bit_addr_o)
   begin
      text_rgb <= "110";  -- yellow background
      if score_on='1' then
         char_addr <= char_addr_s;
         row_addr <= row_addr_s;
         bit_addr <= bit_addr_s;
         if font_bit='1' then
            text_rgb <= "001";
         end if;
      elsif rule_on='1' then
         char_addr <= char_addr_r;
         row_addr <= row_addr_r;
         bit_addr <= bit_addr_r;
         if font_bit='1' then
            text_rgb <= "001";
         end if;
      elsif logo_on='1' then
         char_addr <= char_addr_l;
         row_addr <= row_addr_l;
         bit_addr <= bit_addr_l;
         if font_bit='1' then
            text_rgb <= "011";
         end if;
      else -- game over
         char_addr <= char_addr_o;
         row_addr <= row_addr_o;
         bit_addr <= bit_addr_o;
         if font_bit='1' then
            text_rgb <= "001";
         end if;
      end if;
   end process;
```

```
   text_on <= score_on & logo_on & rule_on & over_on;
   -- font ROM interface
   rom_addr <= char_addr & row_addr;
   font_bit <= font_word(to_integer(unsigned(not bit_addr)));
end arch;
```

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`.

13.4.2 Modified graphic subsystem

To accommodate the new top-level controller, the graphic circuit in Section 12.4.3 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 13.7.

Listing 13.7 Modified portion of a graph subsystem for the pong game

```
   . . .
   -- new ball position
   ball_x_next <=
      to_unsigned((MAX_X)/2,10) when gra_still='1' else
      ball_x_reg + ball_vx_reg when refr_tick='1' else
      ball_x_reg ;
   ball_y_next <=
      to_unsigned((MAX_Y)/2,10) when gra_still='1' else
      ball_y_reg + ball_vy_reg when refr_tick='1' else
      ball_y_reg ;
   -- new ball velocity
   process(ball_vx_reg,ball_vy_reg,ball_y_t,ball_x_l,ball_x_r,
           ball_y_t,ball_y_b,bar_y_t,bar_y_b,gra_still)
   begin
      hit <='0';
      miss <='0';
      ball_vx_next <= ball_vx_reg;
      ball_vy_next <= ball_vy_reg;
      if gra_still='1' then                    --initial velocity
         ball_vx_next <= BALL_V_N;
         ball_vy_next <= BALL_V_P;
      elsif ball_y_t < 1 then                  -- reach top
```

```
            ball_vy_next <= BALL_V_P;
         elsif ball_y_b > (MAX_Y-1) then  -- reach bottom
            ball_vy_next <= BALL_V_N;
         elsif ball_x_l <= WALL_X_R  then -- reach wall
            ball_vx_next <= BALL_V_P;     -- bounce back
         elsif (BAR_X_L<=ball_x_r) and (ball_x_r<=BAR_X_R) and
               (bar_y_t<=ball_y_b) and (ball_y_t<=bar_y_b) then
               -- reach x of right bar, a hit
               ball_vx_next <= BALL_V_N; -- bounce back
               hit <= '1';
         elsif (ball_x_r>MAX_X) then      -- reach right border
            miss <= '1';                  -- a miss
         end if;
      end process;
      . . .
      graph_on <= wall_on or bar_on or rd_ball_on;
      . . .
```

13.4.3 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 13.8.

Listing 13.8 Two-digit decade counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity m100_counter is
   port(
      clk, reset: in std_logic;
      d_inc, d_clr: in std_logic;
      dig0,dig1: out std_logic_vector (3 downto 0)
   );
end m100_counter;

architecture arch of m100_counter is
   signal dig0_reg, dig1_reg: unsigned(3 downto 0);
   signal dig0_next, dig1_next: unsigned(3 downto 0);
begin
   -- registers
   process (clk,reset)
   begin
      if reset='1' then
         dig1_reg <= (others=>'0');
         dig0_reg <= (others=>'0');
      elsif (clk'event and clk='1') then
         dig1_reg <= dig1_next;
         dig0_reg <= dig0_next;
      end if;
```

```
   end process;
   -- next-state logic for the decimal counter
   process(d_clr,d_inc,dig1_reg,dig0_reg)
   begin
      dig0_next <= dig0_reg;
      dig1_next <= dig1_reg;
      if (d_clr='1') then
         dig0_next <= (others=>'0');
         dig1_next <= (others=>'0');
      elsif (d_inc='1') then
         if dig0_reg=9 then
            dig0_next <= (others=>'0');
            if dig1_reg=9 then -- 10th digit
               dig1_next <= (others=>'0');
            else
               dig1_next <= dig1_reg + 1;
            end if;
         else -- dig0 not 9
            dig0_next <= dig0_reg + 1;
         end if;
      end if;
   end process;
   dig0 <= std_logic_vector(dig0_reg);
   dig1 <= std_logic_vector(dig1_reg);
end arch;
```

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 13.9.

Listing 13.9 Two-second timer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity timer is
   port(
      clk, reset: in std_logic;
      timer_start, timer_tick: in std_logic;
      timer_up: out std_logic
   );
end timer;

architecture arch of timer is
   signal timer_reg, timer_next: unsigned(6 downto 0);
begin
   -- registers
   process (clk, reset)
   begin
      if reset='1' then
         timer_reg <= (others=>'1');
      elsif (clk'event and clk='1') then
         timer_reg <= timer_next;
```

```
      end if;
   end process;
   -- next-state logic
   process(timer_start,timer_reg,timer_tick)
   begin
      if (timer_start='1') then
         timer_next <= (others=>'1');
      elsif timer_tick='1' and timer_reg/=0 then
         timer_next <= timer_reg - 1;
      else
         timer_next <= timer_reg;
      end if;
   end process;
   -- output logic
   timer_up <='1' when timer_reg=0 else '0';
end arch;
```

13.4.4 Top-level system

The top-level system of the pong game consists of the previously designed modules, including 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 13.4.

The control FSM monitors overall system operation and coordinates the activities of the text and graphic subsystems. Its ASMD chart is shown in Figure 13.6. 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

```
if (text_on(3)='1') or
   (state_reg=newgame and text_on(1)='1') or
   (state_reg=over and text_on(0)='1') then
   rgb_next <= text_rgb;
elsif graph_on='1' then -- display graph
  rgb_next <= graph_rgb;
elsif text_on(2)='1' then -- display logo
  rgb_next <= text_rgb;
```

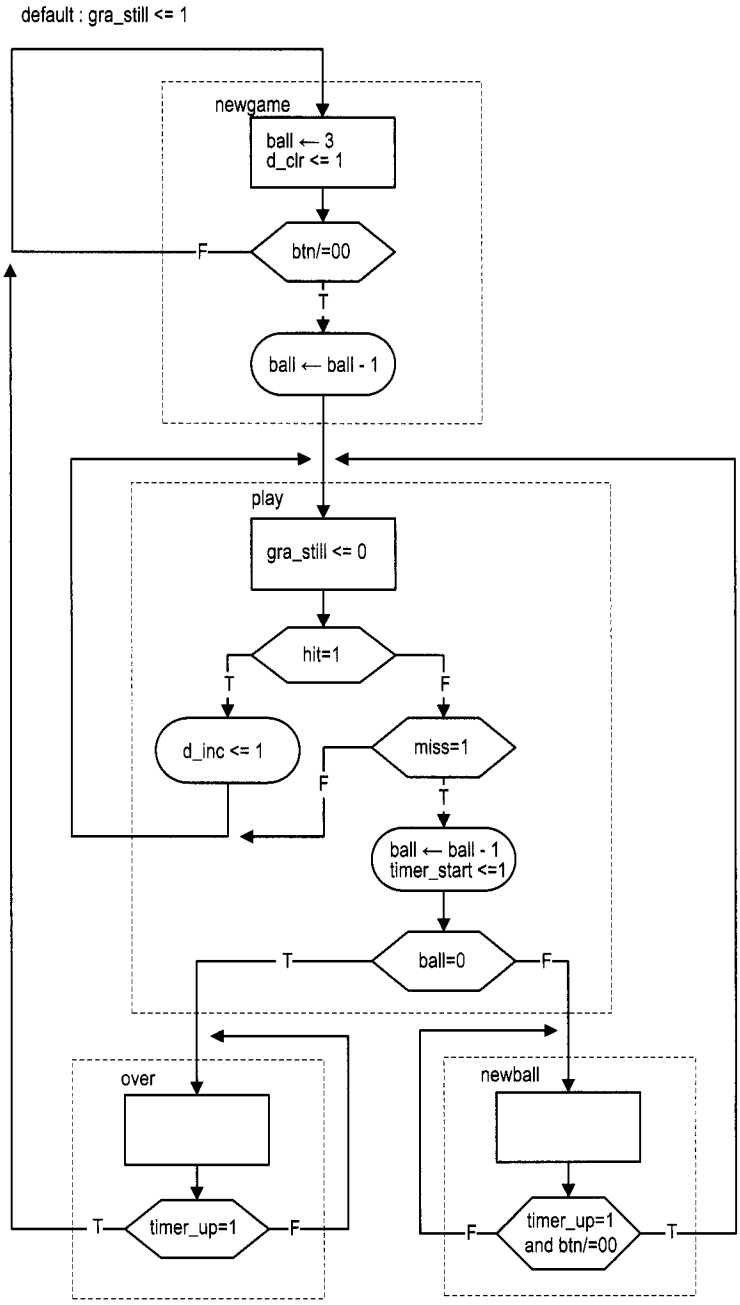


Figure 13.6 ASMD chart of the pong controller.

```
   else
     rgb_next <= "110"; -- yellow background
   end if;
```

The `text_on(3)='1'` expression is the condition for the scores, which is always displayed. The `text_on(1)='1'` expression is the condition 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 13.10.

Listing 13.10 Top-level system for the pong game

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pong_top is
   port(
      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)
   );
end pong_top;

architecture arch of pong_top is
   type state_type is (newgame, play, newball, over);
   signal video_on, pixel_tick: std_logic;
   signal pixel_x, pixel_y: std_logic_vector (9 downto 0);
   signal graph_on, gra_still, hit, miss: std_logic;
   signal text_on: std_logic_vector(3 downto 0);
   signal graph_rgb, text_rgb: std_logic_vector(2 downto 0);
   signal rgb_reg, rgb_next: std_logic_vector(2 downto 0);
   signal state_reg, state_next: state_type;
   signal dig0, dig1: std_logic_vector(3 downto 0);
   signal d_inc, d_clr: std_logic;
   signal timer_tick, timer_start, timer_up: std_logic;
   signal ball_reg, ball_next: unsigned(1 downto 0);
   signal ball: std_logic_vector(1 downto 0);
begin
   -- instantiate video synchronization unit
   vga_sync_unit: entity work.vga_sync
      port map(clk=>clk, reset=>reset,
               hsync=>hsync, vsync=>vsync,
               pixel_x=>pixel_x, pixel_y=>pixel_y,
               video_on=>video_on, p_tick=>pixel_tick);
   -- instantiate text module
   ball <= std_logic_vector(ball_reg);  --type conversion
   text_unit: entity work.pong_text
      port map(clk=>clk, reset=>reset,
               pixel_x=>pixel_x, pixel_y=>pixel_y,
               dig0=>dig0, dig1=>dig1, ball=>ball,
               text_on=>text_on, text_rgb=>text_rgb);
```

```
-- instantiate graph module
graph_unit: entity work.pong_graph
   port map(clk=>clk, reset=>reset, btn=>btn,
            pixel_x=>pixel_x, pixel_y=>pixel_y,
            gra_still=>gra_still,hit=>hit, miss=>miss,
            graph_on=>graph_on,rgb=>graph_rgb);
-- instantiate 2-sec timer
timer_tick <=  -- 60-Hz tick
   '1' when pixel_x="0000000000" and
            pixel_y="0000000000" else
   '0';
timer_unit: entity work.timer
   port map(clk=>clk, reset=>reset,
            timer_tick=>timer_tick,
            timer_start=>timer_start,
            timer_up=>timer_up);
-- instantiate 2-digit decade counter
counter_unit: entity work.m100_counter
   port map(clk=>clk, reset=>reset,
            d_inc=>d_inc, d_clr=>d_clr,
            dig0=>dig0, dig1=>dig1);
-- registers
process (clk,reset)
begin
   if reset='1' then
      state_reg <= newgame;
      ball_reg <= (others=>'0');
      rgb_reg <= (others=>'0');
   elsif (clk'event and clk='1') then
      state_reg <= state_next;
      ball_reg <= ball_next;
      if (pixel_tick='1') then
        rgb_reg <= rgb_next;
      end if;
   end if;
end process;
-- fsmd next-state logic
process(btn,hit,miss,timer_up,state_reg,
        ball_reg,ball_next)
begin
   gra_still <= '1';
   timer_start <='0';
   d_inc <= '0';
   d_clr <= '0';
   state_next <= state_reg;
   ball_next <= ball_reg;
   case state_reg is
      when newgame =>
         ball_next <= "11";     -- three balls
         d_clr <= '1';          -- clear score
         if (btn /= "00") then -- button pressed
            state_next <= play;
            ball_next <= ball_reg - 1;
```

```
            end if;
95          when play =>
               gra_still <= '0';     -- animated screen
               if hit='1' then
                  d_inc <= '1';      -- increment score
               elsif miss='1' then
100               if (ball_reg=0) then
                     state_next <= over;
                  else
                     state_next <= newball;
                  end if;
                  timer_start <= '1'; -- 2-sec timer
105               ball_next <= ball_reg - 1;
               end if;
            when newball =>
               -- wait for 2 sec and until button pressed
110            if  timer_up='1' and (btn /= "00") then
                 state_next <= play;
               end if;
            when over =>
               -- wait for 2 sec to display game over
115            if timer_up='1' then
                  state_next <= newgame;
               end if;
         end case;
      end process;
120   -- rgb multiplexing circuit
      process(state_reg,video_on,graph_on,graph_rgb,
              text_on,text_rgb)
      begin
         if video_on='0' then
125         rgb_next <= "000"; -- blank the edge/retrace
         else
            -- display score, rule or game over
            if (text_on(3)='1') or
               (state_reg=newgame and text_on(1)='1') or -- rule
130            (state_reg=over and text_on(0)='1') then
               rgb_next <= text_rgb;
            elsif graph_on='1'  then -- display graph
              rgb_next <= graph_rgb;
            elsif text_on(2)='1'  then -- display logo
135           rgb_next <= text_rgb;
            else
              rgb_next <= "110"; -- yellow background
            end if;
         end if;
140   end process;
      rgb <= rgb_reg;
end arch;
```

13.5 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.

13.6 SUGGESTED EXPERIMENTS

13.6.1 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.

13.6.2 Underline for the cursor

The full-screen text display circuit in Section 13.3 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.

13.6.3 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 13.3 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.

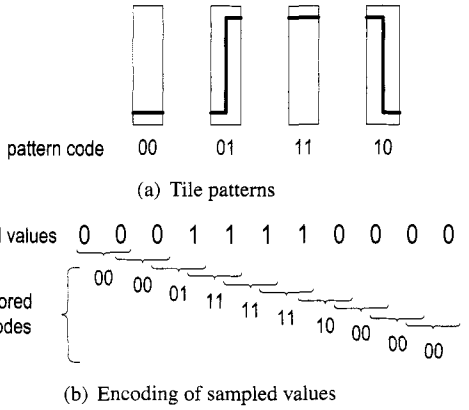
13.6.4 Keyboard text entry

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 8.4 to design the new circuit. Derive the HDL description and then synthesize and verify operation of the circuit.

13.6.5 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.



(a) Tile patterns

(b) Encoding of sampled values

Figure 13.7 Tile patterns and encoding of 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.

13.6.6 Square wave display

We can draw a square wave by using four simple tile patterns shown in Figure 13.7(a). Follow the procedure of a full-screen text display in Section 13.3 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.

13.6.7 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 13.7(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.

13.6.8 Complete two-player pong game

The free-running two-player pong game is described in Experiment 12.7.6. Follow the procedure of the pong game in Section 13.4 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.

13.6.9 Complete breakout game

The free-running breakout game is described in Experiment 12.7.7. Follow the procedure of the pong game in Section 13.4 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.