without having to provide the address for each, other than the first, and the ability to transfer on both rising and falling clock edges (double-data rate, or DDR, and its successors, DDR2 and DDR3). These features are mainly motivated by the need to provide high-speed bursts of data in computer systems, but they can also be of benefit in noncomputer digital systems.

Because of the relative complexity of controlling DRAMs, we will not go into detail of the control signals required and their sequencing. For most implementation fabrics, we can incorporate a DRAM control block from a library, allowing us to connect external DRAMs to the sequential circuits in our chip. An example is the SDRAM controller, described in Xilinx Application Note XAPP134, that allows an FPGA-based system to connect to and control an external SDRAM memory (see Section 5.5, Further Reading).

5.2.5 READ-ONLY MEMORIES

The memories that we have looked at so far can both read the stored data and update it arbitrarily. In contrast, a *read-only memory*, or ROM, can only read the stored data. This is useful in cases where the data is constant, so there is no need to update it. It does, of course, beg the question of how the constant data is placed in the ROM in the first place. The answer is that the data is either incorporated into the circuit during its manufacture, or is programmed into the ROM subsequently. We will describe a number of kinds of ROM that take one or other of these approaches.

Combinational ROMs

A simple ROM is a combinational circuit that maps from an input address to a constant data value. We could specify the ROM contents in tabular form, with a row for each address and an entry showing the data value for that address. Such a table is essentially a truth table, so we could, in principle, implement the mapping using the combinational circuit design techniques we described in Chapter 2. However, ROM circuit structures are generally much denser than arbitrary gate-based circuits, since each ROM cell needs at most one transistor. Indeed, for a complex combinational function with multiple outputs, it may be better to use a ROM to implement the function than a gate-based circuit. For example, a ROM might be a good candidate for the next-state logic or the output logic of a complex finite-state machine.

EXAMPLE 5.9 Design a 7-segment decoder with blanking input, as described in Example 2.16 on page 67, using a ROM.

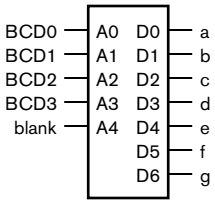

FIGURE 5.21 A 32×7-bit ROM used as a 7-segment decoder.

TABLE 5.2 ROM contents for the 7-segment decoder.

SOLUTION The decoder has five input bits: four for the BCD code and one for the blanking control. It has seven output bits: one for each segment. Thus, we need a 32 × 7-bit ROM, as shown in Figure 5.21. The contents of the ROM are given in Table 5.2.

ADDRESS	CONTENT	ADDRESS	CONTENT
0	0111111	6	1111101
1	0000110	7	0000111
2	1011011	8	1111111
3	1001111	9	1101111
4	1100110	10–15	1000000
5	1101101	16–31	0000000

EXAMPLE 5.10 Develop a Verilog model of the 7-segment decoder of Example 5.9.

SOLUTION The module definition is

```
module seven_seg_decoder ( output reg [7:1] seg,
                           input      [3:0] bcd,
                           input            blank );

  always @*
  case ({blank, bcd})
    5'b00000: seg = 7'b0111111;   // 0
    5'b00001: seg = 7'b0000110;   // 1
    5'b00010: seg = 7'b1011011;   // 2
    5'b00011: seg = 7'b1001111;   // 3
    5'b00100: seg = 7'b1100110;   // 4
    5'b00101: seg = 7'b1101101;   // 5
    5'b00110: seg = 7'b1111101;   // 6
    5'b00111: seg = 7'b0000111;   // 7
    5'b01000: seg = 7'b1111111;   // 8
    5'b01001: seg = 7'b1101111;   // 9
    5'b01010, 5'b01011,
    5'b01100, 5'b01101,
    5'b01110, 5'b01111:
              seg = 7'b1000000;   // "-" for invalid code
    default:  seg = 7'b0000000;   // blank
  endcase

endmodule
```

As in Example 2.16, we use a case statement in a combinational always block to implement a truth-table form of the mapping. In this example, however, we form the address from the concatenation of the blank and bcd inputs. The case statement then specifies the outputs for all possible combinations of value for the address. A synthesis tool could then infer a ROM to implement the mapping.

In FPGA fabrics that provide SSRAM blocks, we can use an SSRAM block as a ROM. We simply modify the always-block template for the memory to omit the part that updates the memory content. We could include a case statement to determine the data output, as in Example 5.10. For example,

```
always @(posedge clk)
  if (en)
    case (a)
      9'h0: d_out <= 20'h00000;
      9'h1: d_out <= 20'h0126F;
      ...
    endcase
```

The content of the memory is loaded into the FPGA as part of its programming when the system is turned on. Thereafter, since the data is not updated, it is constant. Note, in passing, that we have used the Verilog notation for hexadecimal values in this model. The notation 9'h1 means a 9-bit vector zero-extended from the value 1_{16}, and the notation 20'h0126F means a 20-bit vector with the value $0126F_{16}$.

For large ROMs, writing the data directly in the Verilog code like this is very cumbersome. Fortunately, Verilog provides a way of writing the data in a separate file that can be loaded into the ROM during simulation or synthesis. We use the $readmemh or $readmemb system task, as follows:

```
reg [19:0] data_ROM [0:511];
...
initial $readmemh("rom.data", data_ROM);

always @posedge clk)
  if (en)
    d_out <= data_ROM[a];
```

The $readmemh system task expects the content of the named file to be a sequence of hexadecimal numbers, separated by spaces or line breaks. Similarly, $readmemb expects the file to contain a sequence of binary