5 MEMORIES

Many digital systems use memories for storing information. Memory in general-purpose computers takes several forms, including semiconductor memory chips, magnetic disks (hard disks), and optical disks (CDs and DVDs). In this chapter, we describe the various types of semiconductor memories, since other forms of memory are much less frequently used in application-specific digital systems. We start by introducing the general concepts that are common to all kinds of semiconductor memory, and then focus on the particular features of each type. We complete the chapter with a discussion of techniques for dealing with errors in the stored data.

5.1 GENERAL CONCEPTS

In Chapter 4 we introduced registers as components for storing binary-coded information. We generally use separate registers when the number of items of information to store is small, or when we need to use many of the items concurrently. When there are numerous items that we can use one after another, we use *memory* components instead to store the information. In this section, we will discuss some of the general concepts that apply to all kinds of memory components. Then, in the next section, we will identify some of the specific kinds of memory that are used in different design scenarios.

A memory is conceptually an array of storage registers, or *locations*, each of which has a distinct *address*, which is a number identifying the location. Addresses for a memory typically start at 0 and increase by one for each location, up to one less than the number of locations. For most memory components, the number of locations is a power of 2. Thus, a memory with 2^n locations would have addresses ranging from 0 to $2^n - 1$, requiring an n-bit address. If each location stores m bits of encoded information, the total number of bits in the memory component is $2^n \times m$.

EXAMPLE 5.1 If a memory has 32,768 locations, each of 32 bits, what is the total capacity of the memory, and how many address bits does it require?

SOLUTION The capacity is 1,048,576 bits, that is 2^{20} bits. Since $32{,}768 = 2^{15}$, the memory requires 15 address bits.

When referring to memory sizes, we usually use the following multiplier prefixes denoting powers of 2:

- Kilo (K): $2^{10} = 1{,}024$
- Mega (M): $2^{20} = 1{,}024 \times 2^{10} = 1{,}048{,}576$
- Giga (G): $2^{30} = 1{,}024 \times 2^{20} = 1{,}073{,}741{,}824$

Thus, the memory referred to in Example 5.1 has a capacity of 1M bit. Note that the multiplier values are close to, but slightly greater than, the decimal multiplier values with the same names. Note also that we use an uppercase "K" for the binary multiplier 2^{10}, compared with the lowercase "k" for the decimal multiplier 10^3. The context of referring to a memory size is usually assumed to indicate use of the binary multipliers rather than the decimal multipliers.

Given a memory of a certain capacity, we can organize it in different ways, varying the number of locations and the number of bits per location. For example, a 1M bit memory might be organized as a $32K \times 32$-bit memory, as shown in Example 5.1, or as a $16K \times 64$-bit memory, $64K \times 16$-bit memory, and so on. In practice, the number of locations and the size of each location are determined by the application requirements, dictating the memory capacity required.

The two basic operations performed by a memory are writing binary data to a location and reading the content of a location. For both operations, we need to provide the address of the location to be written or read on a set of input signals to the memory component. For a write operation, we provide the data to write as a further set of input signals, and for a read operation, the memory component provides the data as a set of output signals. We control the write operation using control signals generated by a control section of the digital system that contains the memory component. We will describe the particular control signals used by different kinds of memories in a later section. For now, we will just assume a simple form of memory component with simple control signals. The input and output signals are shown on a symbol for a memory component in Figure 5.1. The signal a is the address, encoded as an unsigned binary number. The signals d_in and d_out carry the data to be written and the data read, respectively. The encoding for these signals depends on the application. The control signals are en (enable) and wr (write). When en is 0, the memory simply maintains all of the stored

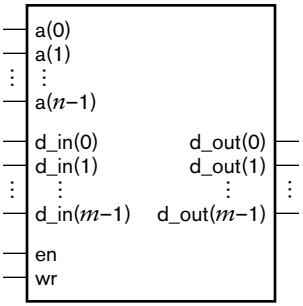

FIGURE 5.1 Symbol for a basic memory component.

data. When en is 1 and wr is 1, the memory writes data present on the d_in inputs at the location whose address is present on the a inputs. When en is 1 and wr is 0, the memory reads the content of the location whose address is present on the a inputs and drives the data value on the d_out outputs.

EXAMPLE 5.2 Design an audio echo effects unit that operates by delaying samples of an audio signal represented as a stream of 16-bit 2s-complement binary-coded values. The sample rate is 50kHz. Arrival of a new input sample is indicated by a control input, audio_in_en, being 1 for the clock cycle in which the sample arrives. The unit should indicate availability of an output sample using an output control signal, audio_out_en, in the same way. The delay time is determined by an 8-bit unsigned input representing the number of milliseconds of delay. The system clock frequency is 1MHz.

SOLUTION We can delay the arriving audio sample values by storing them in a memory until they are required at the output. The maximum delay expressed by the 8-bit unsigned input is 255ms. Since samples arrive at a rate of 50kHz (that is, 50 per millisecond), we need to store up to $255 \times 50 = 12{,}750$ samples. A $16K \times 16$-bit memory, with 14-bit addresses (since $16K = 2^{14}$), will suffice. A diagram of the datapath including the memory and other components to compute addresses is shown in Figure 5.2. The figure shows the widths of each of the multibit signals.

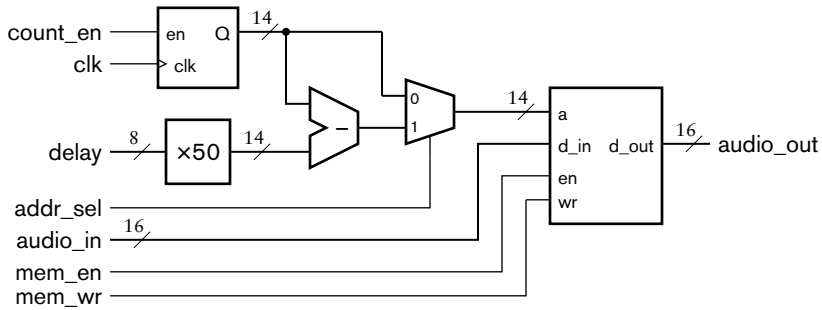

FIGURE 5.2 Datapath for an audio echo effects unit.

We need to use a 14-bit counter to keep track of where samples are stored in the memory. As each input sample arrives, we store it at the next available memory location, whose address is given by the counter. We next read from the memory the value written *d* milliseconds in the past (where *d* is the value of the delay input) and provide it at the output, then increment the counter to refer to the next location in memory. This behavior is illustrated in the timing diagram of Figure 5.3. The value written *d* milliseconds previously is stored $50 \times d$ locations prior to the current location given by the address counter. Thus, we can compute its address by multiplying *d* by 50 and subtracting the result from the value of the address counter. The counter will increment to the maximum address value

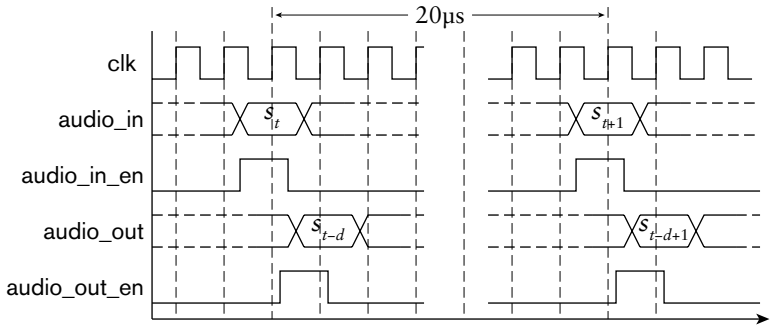

FIGURE 5.3 Timing diagram for the audio echo effects unit.

then wrap around to 0, effectively incrementing modulo 16K. Thus, once the memory is filled, old locations will be overwritten with newly arriving samples. However, they will have been written more than the maximum delay in the past, so they will no longer be needed. When we perform the subtraction, we can ignore the borrow output of the subtracter. The subtracter will yield the difference modulo 16K, and so give the correct address of the required delayed sample.

The control sequence for the unit involves two steps:

1. When a sample arrives (indicated by audio_in_en being 1), set the multiplexer to use the counter value as the memory address and enable the memory to perform a write.
2. Set the multiplexer to use the subtracter output as the memory address, enable the memory to perform a read, set audio_out_en to 1, and enable the counter to increment on the next clock edge.

We can use step 1 as the idle state for a state machine that controls this sequence, provided we use the audio_in_en signal to gate the write control signal to the memory. The transition and output functions are specified in Table 5.1.

TABLE 5.1 Transition and output functions for the echo unit control section.

STATE	audio_in_en	NEXT STATE	addr_sel	mem_en	mem_wr	count_en	audio_out_en
step 1	0	step 1	0	0	0	0	0
step 1	1	step 2	0	1	1	0	0
step 2	–	step 1	1	1	0	1	1

The mem_en and mem_wr signals are Mealy-style outputs, since they depend on both the state and the audio_in_en input, whereas the remaining control signals are all Moore-style outputs.

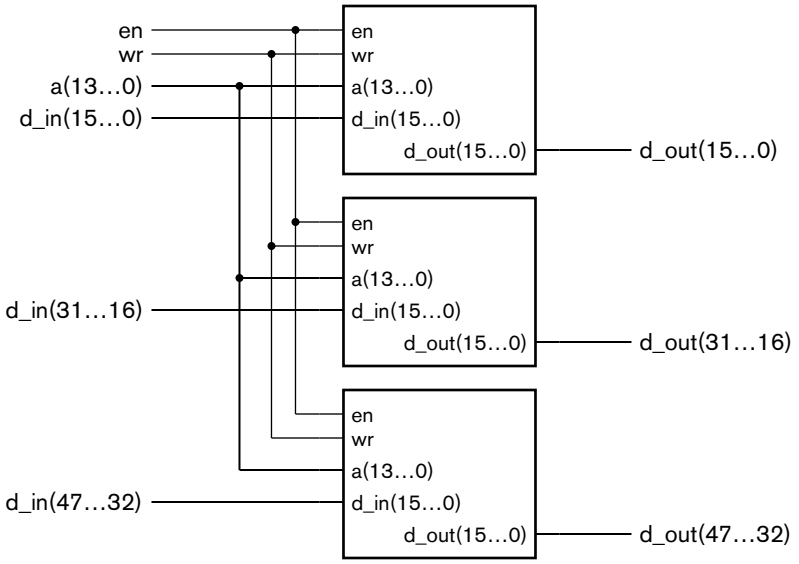

FIGURE 5.4 Connection of memory components in parallel to form a wider memory.

Manufacturers provide semiconductor memory components in a range of capacities, varying from a few Kbits through several Mbits and, at time of writing, up to 2G bits for separately package memory components. Typically, for a given capacity, a manufacturer provides components organized with differing widths (1, 4, 8 or 16 bits per location). If an application for which we are designing a system needs a memory of some other width, we need to use a number of memory components in parallel. For example, if we need a 16K × 48-bit memory for an application, we could construct it using three 16K × 16-bit memory components. We would connect the address and control signals together, as shown in Figure 5.4, and use the data input and output signals of each component for a slice of the overall data input and output signals.

Connecting multiple memory components together to construct a memory with more locations is somewhat more involved. We need to partition the total number of locations among the memory components. For each read and write operation we need to arrange for the component containing the required location to perform the operation, and for other components to remain passive. In many applications, the total number of locations is a power of 2, say 2^n, and each memory component has a smaller number of locations, 2^k. The number of memory components is $2^n/2^k$. The simplest approach to partitioning is to place the first 2^k locations in the first component, the second 2^k in the second component, and so on. If we number the individual memory components 0, 1, 2, and so on up to $(2^n/2^k) - 1$, the component containing a location with address A is $\lfloor A/2^k \rfloor$. This is represented by the most significant $n - k$ bits of the

address. We can decode these bits to derive select signals to activate the required memory component. The address of the location A within the selected memory component is $A \bmod 2^k$. This is represented by the least significant k bits of the address. We simply connect these bits of the address to each of the memory components. The data input signals are also connected to each of the memory components. The data output signals need to be driven by the memory component that is selected, so we use a multiplexer to choose the appropriate data value based on the most significant address bits.

EXAMPLE 5.3 Design a 64K × 8-bit composite memory using four 16K × 8-bit components.

SOLUTION The complete composite memory is shown in Figure 5.5. Address bits 15 and 14 are decoded to select which of the four memory components is enabled for read and write operations. Those bits also control the multiplexer to select the output data from the enabled component during a read operation.

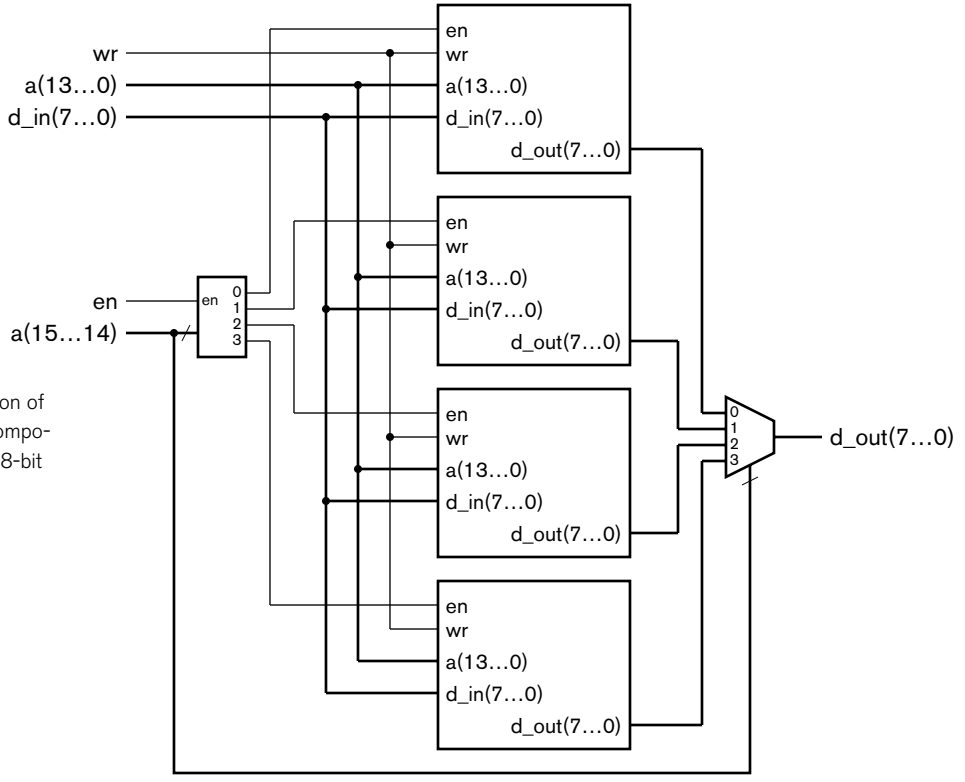

FIGURE 5.5 Connection of four 16K × 8-bit memory components to construct a 64K × 8-bit memory.

Many manufacturers simplify the connection of memory components to form larger memories by using a special kind of output driver, called a *tristate* driver, for each of the data outputs. Tristate drivers are also used for buses that allow multiple data sources to provide data in a system. We will discuss tristate and other bus structures in more detail in Chapter 8 as part of our discussion of embedded computer systems. For now, we will focus on their use in memory components.

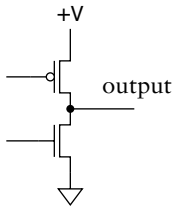

FIGURE 5.6 Output stage circuit.

Unlike ordinary component outputs, which always drive either a low or high logic level, the output of a tristate driver can be turned off by placing it in a *high-impedance*, or *hi-Z*, state. ("Z" is commonly used as the symbol for impedance in a circuit.) Thus, a tristate driver has three output states: logic low, logic high and high impedance; hence the name. The output circuit of a CMOS digital component involves two transistor switches as shown in Figure 5.6. To drive the output with a low logic level, the component turns the bottom transistor on and the top transistor off, and to drive a high logic level, the component turns the top transistor on and the bottom transistor off. A tristate driver has the same output stage, but can turn both transistors off, effectively isolating the component from the output.

If we use memory components with tristate data outputs to construct a larger memory, we can omit the output multiplexer shown in Figure 5.5. Instead, we simply connect the data outputs of the memory components together. When a read operation is performed, only the selected memory component enables its data outputs; all of the disabled components leave their outputs in the high-impedance state.

Many memory components that have tristate data outputs also combine the data inputs and outputs into a single set of *bidirectional* connections, illustrated in Figure 5.7. This allows a composite memory to be constructed as shown in Figure 5.8. For memory components implemented as separate integrated circuits for use on printed circuit boards, the use of bidirectional connections results in significant cost savings, since there are fewer package

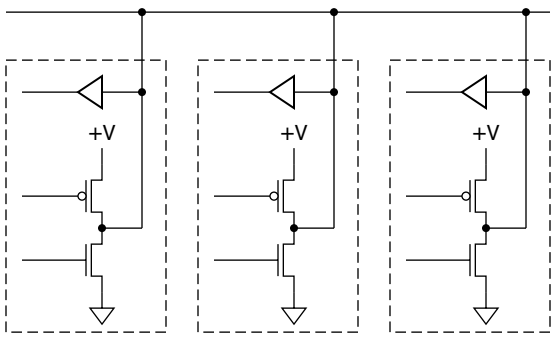

FIGURE 5.7 Bidirectional tristate data connections.

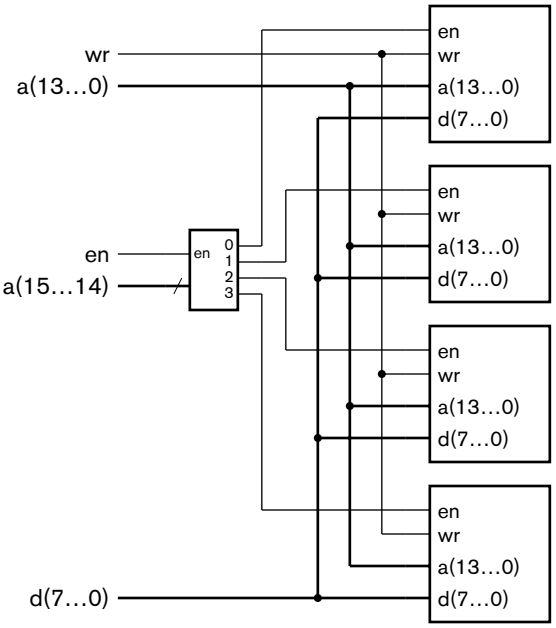

FIGURE 5.8 A composite memory constructed using components with common data inputs and outputs.

pins and interconnecting wires. As we shall see when we study embedded processors in more detail, this type of memory works well as part of an embedded computer system, since memory write and read operations are performed independently. When we perform a write operation, we drive the data signals with the data to be written. The selected memory component treats the data connections as inputs and accepts the data to be written. It keeps its tristate drivers disabled so as not to interfere with the logic levels in the data signals. When we perform a read operation, we ensure that all other drivers connected to the data signals are in the high-impedance state and allow the selected memory component to enable its tristate drivers. It drives the data signals with the data read from memory.

Of course, whether we can use tristate data connections in a memory depends on whether the implementation fabric provides them. Memory components implemented as packaged integrated circuits, for use in a larger system implemented on a printed circuit board, typically do have tristate data outputs or tristate bidirectional data input/outputs. On the other hand, memory blocks provided within ASICs and FPGAs typically do not have tristate data connections, since tristate buses present some design and verification challenges in those fabrics. (We will return to this in Chapter 8.) Instead, data from individual memory blocks must be combined using multiplexers.

In this section, we have looked at ways of connecting multiple memory components together to form a memory with wider or more storage

locations than provided by a single chip. In each of these schemes, the memory performs just one operation at a time. In high performance systems, we can connect multiple memory components together in ways that permit multiple operations to proceed concurrently, thus increasing the total number of operations completed per second. These schemes usually involve organizing the memory into a number of banks, each of which can perform an operation in parallel with other banks. Successive addresses are assigned to different banks, since, in many systems, locations are often accessed in order. As an example, a system with four banks would assign locations 0, 4, 8, … to bank 0; locations 1, 5, 9, … to bank 1; 2, 6, 10, … to bank 2; and 3, 7, 11, … to bank 3. When a read operation is required for location 4, bank 0 would read that location. Moreover, the other banks would start a read, *prefetching* locations 5, 6 and 7. By the time a read operation is required for these locations (assuming access in order), the data would already be available from the memory. We will not describe these advanced memory organizations in any further detail in this book. Books on computer organization, particularly those concentrating on high-performance computers, are a good source of further information. (See Section 5.5, Further Reading.)

KNOWLEDGE TEST QUIZ

1. What is the capacity in bits of a memory with 4096 locations, each of 24 bits? How many address bits are required?
2. What is the effect of a write operation? What is the effect of a read operation?
3. How would we connect four 256M × 4-bit memory components to make a 256M × 16-bit memory?
4. How would we connect four 256M × 8-bit memory components to make a 1G × 8-bit memory?
5. Which memory component in Question 4 would contain the location with address $5FC0000_{16}$?
6. What are the three states of a tristate driver?
7. How do memory components with tristate data outputs simplify construction of large memories?

5.2 MEMORY TYPES

In this section, we will introduce the various types of memory provided by manufacturers, either as individual integrated circuits or as resources within ASIC or FPGA fabrics. We will discuss the distinguishing properties of each kind of memory, including their timing characteristics and costs,

and describe how to model some of them in Verilog. We will distinguish between memory that can be both read and written, called *random access memory* (RAM), and memory that can only be read, called *read-only memory* (ROM). We use the term RAM instead of read/write memory largely for historical reasons. Memories in very early computers enforced sequential access, that is, access to locations in increasing order of address, due to the physical medium on which the data was stored. The invention of memories in which locations could be read and written with equal facility in any order (that is, randomly) was a significant milestone, and so the term RAM has stuck.

5.2.1 ASYNCHRONOUS STATIC RAM

One of the simplest forms of memory is asynchronous static RAM. It is *asynchronous* because it does not rely on a clock for its timing. The term *static* means that the stored data persists indefinitely so long as power is applied to the memory component. Compare this with dynamic RAM, which we will describe later and which loses stored data if it is not periodically rewritten. Static RAM is *volatile*, meaning that it requires power to maintain the stored data, and loses data if power is removed. Since engineers are fond of abbreviations, the term static RAM is usually further shortened to SRAM.

Asynchronous SRAM internally uses 1-bit storage cells that are similar to the D-latch circuit that we described in Chapter 4. Within the memory component, the address is decoded to select a particular group of cells that comprise one location. For a write operation, the selected latch cells are enabled and the input data is stored. For a read operation, the address activates a multiplexer that routes the outputs of the selected latch cells to the data outputs of the memory component.

The external interface of an asynchronous SRAM is very close to our general description of a memory component in Section 5.1. For largely historical reasons, most manufacturers use active-low logic for the control signals. Further, since asynchronous SRAMs are usually only available as packaged integrated circuits, and not as blocks in ASIC libraries or FPGAs, they usually have bidirectional tristate data input/output pins. Figure 5.9 shows a symbol for a typical asynchronous SRAM. The address input and the data input/output are as we described in Section 5.1. The chip-enable input ($\overline{CE}$) is used to enable or disable the memory chip. We usually drive this input from a select control signal, for example, from an address decoder in a composite memory. The write-enable input ($\overline{WE}$) controls whether the memory, if enabled, performs a write or read operation. The output-enable input ($\overline{OE}$) controls the tristate data drivers during a read operation. When $\overline{OE}$ is low during a read, the drivers are enabled and can drive the read data onto the data pins. When $\overline{OE}$ is high, the drivers are in the high-impedance state.

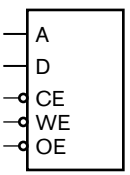

FIGURE 5.9 Symbol for an asynchronous SRAM.

Given that the storage cells in an asynchronous SRAM are basically latches, it is not surprising that the timing is similar to that of a D-latch. The sequencing of signals to perform a write operation is shown at the left of Figure 5.10. The control section that sequences the datapath containing the memory must ensure that the address is stable before commencing the write operation and is held stable during the entire operation. Otherwise, locations other than the one to be updated may be affected. The control section selects the particular memory chip by driving $\overline{CE}$ low, activates the write operation by driving $\overline{WE}$ low, and ensures that the chip's tristate drivers are disabled by driving $\overline{OE}$ high. It also sets control signals to the datapath to provide data on the data signals. The data is stored transparently in the latch cells for the addressed location. The final data to be stored must be stable on the data signals a setup time before the rising edge of the $\overline{WE}$ signal or the $\overline{CE}$ signal, whichever occurs first. The data and the address must also remain stable for a hold time after the $\overline{WE}$ or $\overline{CE}$ signal goes high.

The typical sequencing of signals for a read operation is similar, and is shown at the right of Figure 5.10. The difference is that the $\overline{WE}$ signal is held high, and the $\overline{OE}$ signal is driven low to enable the memory chip's tristate drivers. While this sequence is typical for a read operation done in isolation, we can also perform back-to-back read operations simply by changing the address value. The read operation is essentially a combinational operation, involving decoding the address and multiplexing the selected latch-cell's value onto the data outputs. Changing the address simply causes a different cell's value to appear on the outputs after a propagation delay.

Manufacturers of asynchronous SRAM chips publish the timing parameters for write and read operations in data sheets. The parameters typically include setup and hold times for address and data values, and delays for turning tristate drivers on and off. One of the figures of merit of a memory chip is its *access time*, which is the delay from the start of a read

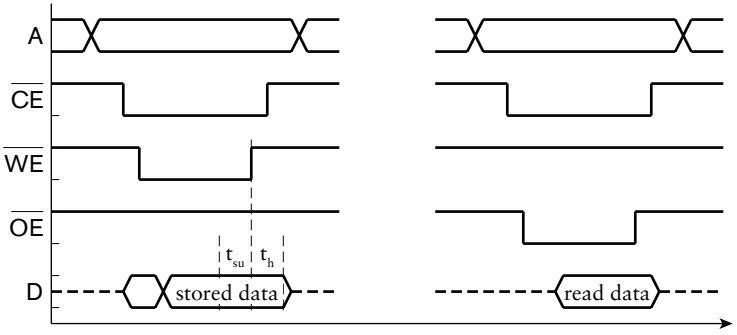

FIGURE 5.10 Timing for write and read operations in an asynchronous SRAM.

operation to having valid data at the outputs. Other performance-related parameters are the *write cycle time* and the *read cycle time*, which are the times taken to complete write and read operations, respectively. Manufacturers offer chips in different speed grades, with faster chips usually costing more. This allows us, as designers, to make cost/performance trade-offs in our designs.

While asynchronous SRAMs are conceptually simple and have simple timing behavior, the fact that they are asynchronous can make them difficult to use in clocked synchronous systems. The need to set up and hold address and data values before and after activation of the control signals and to keep the values stable during the entire cycle means that we must either perform operations over multiple clock cycles, or use delay elements to ensure correct timing within a clock cycle. The former approach reduces performance, and the latter approach violates assumptions inherent in the clocked synchronous methodology, and so complicates timing design and analysis. For these reasons, asynchronous SRAMs are usually used only in systems with low performance requirements, where their low cost is a benefit.

5.2.2 SYNCHRONOUS STATIC RAM

Given the difficulties associated with asynchronous SRAMs, many memory component vendors and implementation fabrics provide *synchronous SRAMs*, otherwise known as SSRAMs. The internal storage cells of SSRAMs are the same as those of asynchronous SRAMs. However, the interface includes clocked registers for storing the address, input data and control signal values, and in some cases, output data. In this section, we will describe two forms of SSRAMs in general terms. The details of control signals and timing will vary between SSRAMs provided by different component vendors and implementation fabrics. As always, we need to read and understand the data sheets before using a component in a design.

The simplest kind of SSRAM is often called a *flow-through* SSRAM. It includes registers on the inputs, but not on the data outputs. The term flow-through refers to the fact that data read from the memory cells flows through directly to the data outputs. Having registers on the inputs allows us to generate the address, data and control signal values according to our clocked synchronous design methodology, ensuring that they are stable in time for a clock edge. Figure 5.11 illustrates the timing for a flow-through SSRAM. During the first clock cycle, we set up the address (a_1), control signals and input data (xx) in preparation for a write operation. These values are stored in the input registers on the next clock edge, causing the SSRAM to start the write operation. The data is stored and flows through to the output during the second clock cycle. While that happens, we set up the address (a_2) and control signals in preparation for a read operation.

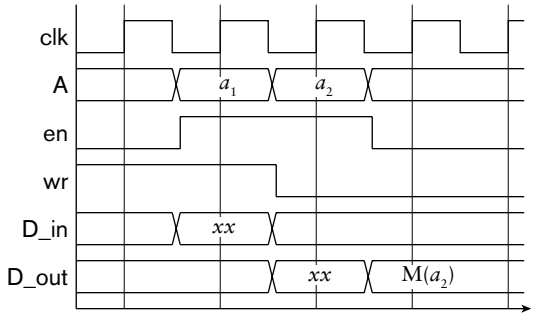

FIGURE 5.11 Timing for a flow-through SSRAM.

Again, these values are stored on the next clock edge, and during the third cycle the SSRAM performs the read operation. The data, denoted by M(a_2), flows through from the memory to the output. Now, in the third cycle, we set the enable signal to 0. This prevents the input registers from being updated on the next clock edge, so the previously read data is maintained at the output.

EXAMPLE 5.4 Design a circuit that computes the function $y = c_i \times x^2$, where x is a binary-coded input value and c_i is a coefficient stored in a flow-through SSRAM. x, c_i and y are all signed fixed-point values with 8 pre-binary-point and 12 post-binary-point bits. The index i is also an input to the circuit, encoded as a 12-bit unsigned integer. Values for x and i arrive at the input during the cycle when a control input, start, is 1. The circuit should minimize area by using a single multiplier to multiply c_i by x and then by x again.

SOLUTION A datapath for the circuit is shown in Figure 5.12. The 4K × 20-bit flow-through SSRAM stores the coefficients. A computation starts with the index value, i, being stored in the SSRAM address register, and the data

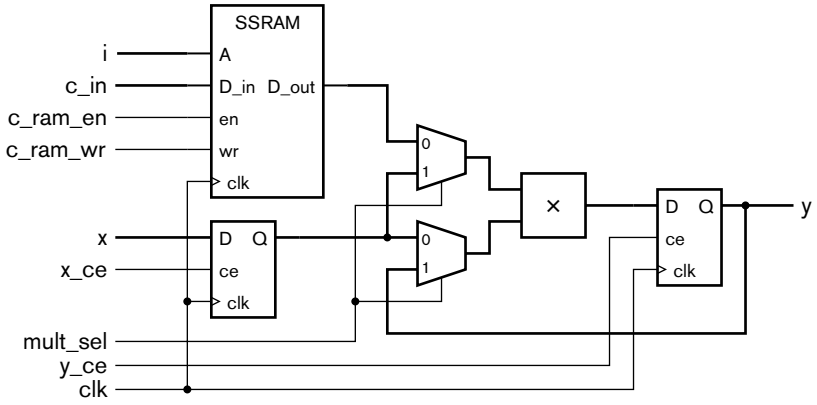

FIGURE 5.12 Datapath for a circuit to multiply the square of an input by an indexed coefficient.

input, x, being stored in the register shown below the SSRAM. On the second clock cycle, the SSRAM performs a read operation. The coefficient read from the SSRAM and the stored x value are multiplied, and the result is stored in the output register. On the third cycle, the multiplexer select inputs are changed so that the value in the output register is further multiplied by the stored x value, with the result again being stored in the output register.

For the control section, we need to develop a finite state machine that sequences the control signals. It is helpful to draw a timing diagram showing progress of the computation in the datapath and when each of the control signals needs to be activated. The timing diagram is shown in Figure 5.13, and includes state names for each clock cycle. An FSM transition diagram for the control section is

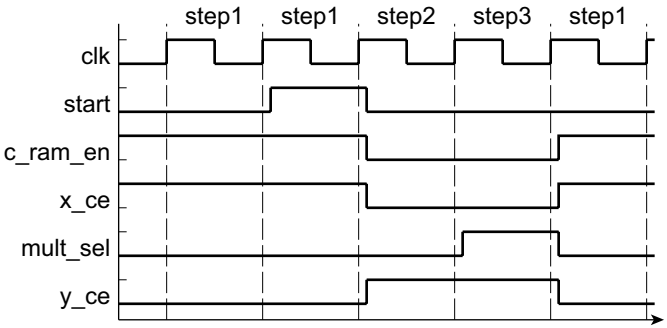

FIGURE 5.13 Timing diagram for the computation circuit.

shown in Figure 5.14. The FSM is a Moore machine, with the outputs shown in each state in the order c_ram_en, x_ce, mult_sel and y_ce. In the step1 state, we maintain c_ram_en and x_ce at 1 in order to capture input values. When start changes to 1, we change c_ram_en and x_ce to 0 and transition to the step2 state to start computation. The y_ce control signal is set to 1 to allow the product of the coefficient read from the SSRAM and the x value to be stored in the y output register. In the next cycle, the FSM transitions to the step3 state, changing the mult_sel control signal to multiply the intermediate result by the x value again and storing the final result in the y output register. The FSM then transitions back to the step1 state on the next cycle.

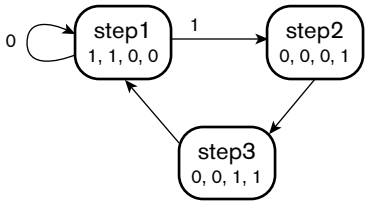

FIGURE 5.14 Transition diagram for the circuit control section.

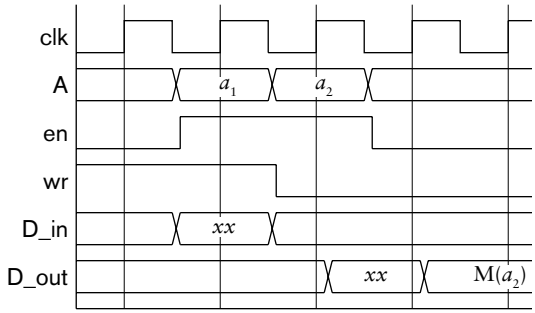

FIGURE 5.15 Timing for a pipelined SSRAM.

Another form of SSRAM is called a *pipelined* SSRAM. It includes a register on the data output, as well as registers on the inputs. A pipelined SSRAM is useful in higher-speed systems where the access time of the memory is a significant proportion of the clock cycle time. If there is no time in which to perform combinational operations on the read data before the next clock edge, it needs to be stored in an output register and used in the subsequent clock cycle. A pipelined SSRAM provides that output register. The timing for a pipelined SSRAM is illustrated in Figure 5.15. Timing for the inputs is the same as that for a flow-through SSRAM. The difference is that the data output does not reflect the result of a read or write operation until one clock cycle later, albeit immediately after the clock edge marking the beginning of that cycle.

EXAMPLE 5.5 Suppose we discover that, in the datapath of Example 5.4, the combination of the SSRAM access time plus the delays through the multiplexer and multiplier is too long. This causes the clock frequency to be too slow to meet our performance constraint. We change the memory from a flow-through to a pipelined SSRAM. How is the circuit design affected?

SOLUTION As a consequence of the SSRAM change, the coefficient value is available at the SSRAM output one cycle later. To accommodate this, we could insert a cycle into the control sequence to wait for the value to be available. Rather than wasting this time, we can use it to multiply the value of x by itself, and perform the multiplication by the coefficient in the third cycle. This change requires us to swap the input to the top multiplexer in Figure 5.12, so that it selects the stored x value when mult_sel is 0 in state step2 and the SSRAM output when mult_sel is 1 in step3. The FSM control sequence is otherwise unchanged.

Verilog Models of Synchronous Static Memories

In this section, we will describe how to model SSRAMs in such a way that synthesis CAD tools can infer a RAM and use the appropriate memory

resources provided in the target implementation fabric. We saw in Chapter 4 that to model a register, we declare a variable to represent the stored register value and assign a new value to it on a rising clock edge. We can extend this approach to model an SSRAM in Verilog. We need to declare a variable that represents all of the locations in the memory. The way to do this is to declare an *array variable*, which represents a collection of values, each with an index that corresponds to its location in the array. For example, to model a 4K × 16-bit memory, we would write the following declaration:

```
reg [15:0] data_RAM [0:4095];
```

The declaration specifies a variable named data_RAM that is an array with elements index from 0 to 4095. Each element is a 16-bit vector.

Once we have declared the variable representing the storage, we write an always block that performs the write and read operations. The block is similar in form to that for a register. For example, an always block to model a flow-through SSRAM based on the variable declaration above is

```
always @(posedge clk)
  if (en)
    if (wr) begin
      data_RAM[a] <= d_in; d_out <= d_in;
    end
    else
      d_out <= data_RAM[a];
```

On a rising clock edge, the block checks the enable input, and only performs an operation if it is 1. If the write control input is 1, the block updates the element of the data_RAM signal indexed by the address using the data input. The block also assigns the data input to the data output, representing the flow-through that occurs during a write operation. If the write control input is 0, the block performs a read operation by assigning the value of the indexed data_RAM element to the data output.

EXAMPLE 5.6 Develop a Verilog model of the circuit using flow-through SSRAMs, as described in Example 5.4.

SOLUTION The module definition includes the address, data and control ports, as follows:

```
module scaled_square ( output reg signed [7:-12]  y,
                       input      signed [7:-12]  c_in, x,
```

(*continued*)

```
                        input          [11:0]  i,
                        input                  start,
                        input                  clk, reset );

  wire               c_ram_wr;
  reg                c_ram_en, x_ce, mult_sel, y_ce;
  reg signed [7:-12] c_out, x_out;

  reg signed [7:-12] c_RAM [0:4095];

  reg signed [7:-12] operand1, operand2;

  parameter [1:0] step1 = 2'b00, step2 = 2'b01, step3 = 2'b10;
  reg       [1:0] current_state, next_state;

  assign c_ram_wr = 1'b0;

  always @(posedge clk)  // c RAM - flow through
    if (c_ram_en)
      if (c_ram_wr) begin
        c_RAM[i] <= c_in;
        c_out    <= c_in;
      end
      else
        c_out <= c_RAM[i];

  always @(posedge clk)  // y register
    if (y_ce) begin
      if (!mult_sel) begin
        operand1 = c_out;
        operand2 = x_out;
      end
      else begin
        operand1 = x_out;
        operand2 = y;
      end
      y <= operand1 * operand2;
    end

  always @(posedge clk)  // State register
    ...

  always @*  // Next-state logic
    ...

  always @* begin  // Output logic
    ...

endmodule
```

The module declares nets and variables for the internal datapath connections and control signals. It declares an array variable to represent the coefficient memory (c_RAM). It also declares parameters for the state of the control section finite-state machine, and variables for the current and next state.

After the declarations, we include always blocks and assignments for the datapath and control section. We omit the details of the finite-state machine. They are based on the template we described in Chapter 4, and are available on the companion website. The first block represents the coefficient SSRAM. It uses the i input as its address. The second block represents both the combinational circuits of the datapath and the output register. If the y_ce variable is 1, the register is updated with the value computed by the combinational circuits. We use intermediate variables to divide the computation into two parts, corresponding to the multiplexers and the multiplier, respectively. Note that we use blocking assignments to these intermediate variables, rather than nonblocking assignments, since they do not represent outputs of storage registers.

Modeling a pipelined SSRAM in Verilog is somewhat more involved, as we must represent the internal connection from the memory storage to the output register and ensure that the pipeline timing is correctly represented. One approach, extending our previous always block for a 16-bit-wide memory, is

```
reg        pipelined_en;
reg [15:0] pipelined_d_out;
...

always @(posedge clk) begin
  if (pipelined_en) d_out <= pipelined_d_out;
  pipelined_en <= en;
  if (en)
    if (wr) begin
      data_RAM[a] <= d_in;  pipelined_d_out <= d_in;
    end
    else
      pipelined_d_out <= data_RAM[a];
end
```

In this block, the variable pipelined_en saves the value of the enable input on a clock edge so that it can be used on the next clock edge to control the output register. Similarly, the variable pipelined_d_out saves the value read or written through the memory on one clock edge for assignment to the output on the next clock edge if the output register is enabled. Since there are many minor variations on the general concept of a pipelined SSRAM, it is difficult to present a general template, especially one that can be recognized by synthesis tools. A common alternative approach is to use a CAD tool that generates a memory circuit and a Verilog model of

that circuit. We can then instantiate the generated model as a component in a larger system.

5.2.3 MULTIPORT MEMORIES

Each of the memories that we have looked at, both in Section 5.1 and previously in this section, is a *single-port* memory, with just one port for writing and reading data. It has only one address input, even though the data connections may be separated into input and output connections. Thus, a single-port memory can perform only one access (a write or a read operation) at a time. In contrast, a *multiport* memory has multiple address inputs, with corresponding data inputs and outputs. It can perform as many operations concurrently as there are address inputs. The most common form of multiport memory is a *dual-port* memory, illustrated in Figure 5.16, which can perform two operations concurrently. (Note that in this context, we are using the term "port" to refer to a combination of address, data and control connections used to access a memory, as distinct from a Verilog port.)

A multiport memory typically consumes more circuit area than a single-port memory with the same number of bits of storage, since it has separate address decoders and data multiplexers for each access port. Only the internal storage cells of the memory are shared between the multiple ports, though additional wiring is needed to connect the cells to the access ports. However, the cost of the extra circuit area is warranted in some applications, such as high performance graphics processing and high-speed network connections. Suppose we have one subsystem producing data to store in the memory, and another subsystem accessing the data to process it in some way. If we use a single-port memory, we would need to multiplex the addresses and input data from the subsystems into the memory, and we would have to arrange the control sections of the subsystems so that they take turns to access the memory. There are two potential problems here. First, if the combined rate at which the subsystems need to move data in and out of the memory exceeds the rate at which a single access port can operate, the memory becomes a bottleneck. Second, even if the average rates don't exceed the capacity of a single access port, if the two subsystems need to access the memory at the same time, one must wait, possibly causing it to lose data. Having separate access ports for the subsystems obviates both of these problems.

The only remaining difficulty is the case of both subsystems accessing the same memory location at the same time. If both accesses are reads, they can proceed. If one or both is a write, the effect depends on the characteristics of the particular dual-port memory. In an asynchronous dual-port memory, a write operation performed concurrently with a read of the same location will result in the written data being reflected on the read port after some delay. Two write operations performed concurrently to the same location result in an unpredictable value being stored. In the case of a synchronous

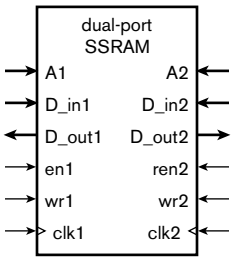

FIGURE 5.16 A dual-port memory.

dual-port memory, the effect of concurrent write operations depends on when the operations are performed internally by the memory. We should consult the data sheet for the memory component to understand the effect.

Some multiport memories, particularly those manufactured as packaged components, provide additional circuits that compare the addresses on the access ports and indicate when contention arises. They may also provide circuits to arbitrate between conflicting accesses, ensuring that one proceeds only after the other has completed. If we are using multiport memory components or circuit blocks that do not provide such features and our application may result in conflicting accesses, we need to include some form of arbitration as a separate part of the control section in our design. An alternative is to ensure that the subsystems accessing the memory through separate ports always access separate locations, for example, by ensuring that they always operate on different blocks of data stored in different parts of the memory. We will discuss block processing of data in more detail in Chapter 9.

EXAMPLE 5.7 Develop a Verilog model of a dual-port, 4K × 16-bit flow-through SSRAM. One port allows data to be written and read, while the other port only allows data to be read.

SOLUTION In the following module definition, the clk input is common to both memory ports. The inputs and outputs with names ending in "1" are the connections for the read/write memory port, and the inputs and outputs with names ending in "2" are the connection for the read-only memory port.

```
module dual_port_SSRAM ( output reg [15:0] d_out1,
                         input      [15:0] d_in1,
                         input      [11:0] a1,
                         input             en1, wr1,
                         output reg [15:0] d_out2,
                         input      [11:0] a2,
                         input      [11:0] en2,
                         input             clk );

  reg [15:0] data_RAM [0:4095];

  always @(posedge clk) // read/write port
    if (en1)
      if (wr1) begin
        data_RAM[a1] <= d_in1; d_out1 <= d_in1;
      end
      else
        d_out1 <= data_RAM[a1];

  always @(posedge clk) // read-only port
    if (en2) d_out2 <= data_RAM[a2];

endmodule
```

This is much like our earlier model of a flow-through SSRAM, except that there are two always blocks, one for each memory port. The declaration of the variable for the memory storage is the same, with the variable being shared between the two blocks. The block for the read/write port is identical in form to the block we introduced earlier. The block for the read-only port is a simplified version, since it does not need to deal with updating the storage variable.

In this model, we make no special provision for the possibility of concurrent write and read accesses to the same address. During simulation of the model, one or other block would be activated first. If the block for the read/write port is activated first, it updates the memory location, and the read operation yields the updated value. On the other hand, if the block for the read-only port is activated first, it reads the old value before the location is updated. When the model is synthesized, the synthesis tool chooses a dual-port memory component from its library. The effect of a concurrent write and read would depend on the behavior of the chosen component.

One specialized form of dual-port memory is a *first-in first-out* memory, or FIFO. It is used to queue data arriving from a source to be processed in order of arrival by another subsystem. The data that is first in to the FIFO is the first that comes out; hence, the name. The most common way of building a FIFO is to use a dual-port memory as a *circular buffer* for the data storage, with one port accepting data from the source and the other port reading data to provide to the processing subsystem. Each port has an address counter to keep track of where data is written or read. Data written to the FIFO is stored in successive free locations. When the write-address counter reaches the last location, it wraps to location 0. As data is read, the read-address counter is advanced to the next available location, also wrapping to 0 when the last location is reached. If the write address wraps around and catches up with the read address, the FIFO is full and can accept no more data. If the read address catches up with the write address, the FIFO is empty and can provide no more data. This scheme is similar to that used for the audio echo effects unit in Example 5.2, except that the distance between the write and read addresses is not fixed. Thus, a FIFO can store a variable amount of data, depending on the rates of writing and reading data. The size of memory needed in a FIFO depends on the maximum amount by which reading of data lags writing. Determining the maximum size may be difficult to do. We may need to evaluate worst-case scenarios for our application using mathematical or statistical models of data rates or using simulation.

EXAMPLE 5.8 Design a FIFO to store up to 256 data items of 16 bits each, using a 256×16-bit dual-port SSRAM for the data storage. The FIFO should provide status outputs, as shown in the symbol in Figure 5.17, to indicate when the FIFO is empty and full. Assume that the FIFO will not be read when it is

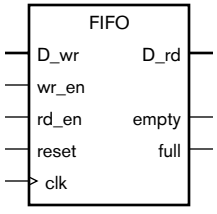

FIGURE 5.17 Symbol for a FIFO with empty and full status outputs.

empty, nor be written to when it is full, and that the write and read ports share a common clock.

SOLUTION The datapath for the FIFO, shown in Figure 5.18, uses 8-bit counters for the write and read addresses. The write address refers to the next free location in the memory, provided the FIFO is not full. The read address refers to the next location to be read, provided the FIFO is not empty. Both counters are cleared to 0 when the reset signal is active.

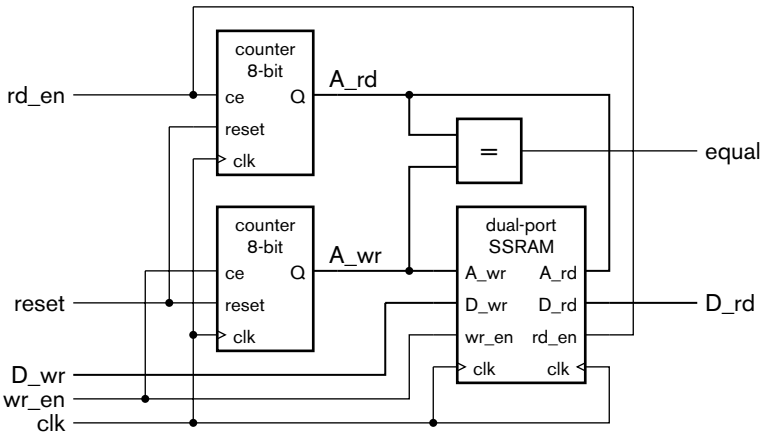

FIGURE 5.18 Datapath for a FIFO using a dual-port memory.

The FIFO being empty is indicated by the two address counters having the same value. The FIFO is full when the write counter wraps around and catches up with the read counter, in which case the counters have same value again. So equality of the counters is not sufficient to distinguish between the cases of the FIFO being empty or full. We could keep track of the number of items in the FIFO, for example, by using a separate up/down counter to count the number of items rather than trying to compare the addresses. However, a simpler way is to keep track of whether the FIFO is filling or emptying. A write operation without a concurrent read means the FIFO is filling. If the write address becomes equal to the read address as a consequence of the FIFO filling, the FIFO is full. A read operation without a concurrent write means the FIFO is emptying. If the write address becomes equal to the read address as a consequence of the FIFO emptying, the FIFO is empty. If a write and a read operation occur concurrently, the amount of data in the FIFO remains unchanged, so the filling or emptying state remains unchanged. We can describe this behavior using an FSM, as shown in Figure 5.19, in which the transitions are labeled with the values of the wr_en and rd_en control signals, respectively. The FSM starts in the emptying state. The empty status output is 1 if the current state is emptying and the equal signal is 1, and the full status output is 1 if the current state is filling and the equal signal is 1. Note that this control sequence relies on the assumption of a common clock between the two FIFO ports, since the FSM must have a single clock to operate.

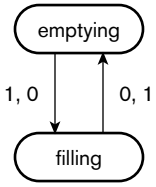

FIGURE 5.19 Transition diagram for the FIFO FSM.

One important use for FIFOs is to pass data between subsystems operating with different clock frequencies, that is, between different *clock domains*. As we discussed in Section 4.4.1, when data arrives asynchronously, we need to resynchronize it with the clock. If the clocks of two clock domains are not in phase, data arriving at one clock domain from the other could change at any time with respect to the receiving domain's clock, and so must be treated as an asynchronous input. Resynchronizing the data means passing it through two or more registers. If the sending domain's clock is faster than that of the receiving domain, the data being resynchronized may be overrun by further arriving data. A FIFO allows us to smooth out the flow of data between the domains. Data arriving is written into the FIFO synchronously with the sending domain's clock, and the receiving domain reads data synchronously with its clock. Control of such a FIFO is more involved than that for the FIFO with a single clock illustrated in Example 5.8. The Xilinx Application Note, XAPP 051 (see Section 5.5, Further Reading) describes a technique that can be used.

FIFOs are also used in applications such as computer networking, where data arrives from multiple network connections at unpredictable times and must be processed and forwarded at high speed. Several memory component vendors provide packaged FIFO circuits that include the dual-port memory and the address counting and control circuits. Some of the larger FPGA fabrics also provide FIFO address counting control circuits that can be used with built-in memory blocks. If we need a FIFO in a system implemented in other fabrics, we can either design one, as we did in Example 5.8, or use a FIFO block from a library or a generator tool.

5.2.4 DYNAMIC RAM

Dynamic RAM (DRAM) is another form of volatile memory that uses a different form of storage cell for storing data. We mentioned in Section 5.2.1 that static RAM uses storage cells that are similar to D-latches. In contrast, a storage cell for a dynamic RAM uses a single capacitor and a single transistor, illustrated in Figure 5.20. The DRAM cells are thus much smaller than SRAM cells, so we can fit many more of them on a chip, making the cost per bit of storage lower. However, the access times of DRAMs are longer than those of SRAMs, and the complexity of access and control is greater. Thus, there is a trade-off of cost, performance and complexity against memory capacity. DRAMs are most commonly used as the main memory in computer systems, since they satisfy the need for high capacity with relatively low cost. However, they can also be used in other digital systems. The choice between SRAM and DRAM depends on the requirements and constraints of each application.

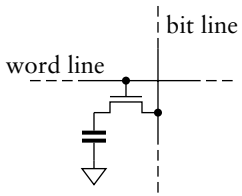

FIGURE 5.20 A DRAM storage cell.

A DRAM represents a stored 1 or 0 bit in a cell by the presence or absence of charge on the capacitor. When the transistor is turned

off, the capacitor is isolated from the bit line, thus storing the charge on the capacitor. To write to the cell, the DRAM control circuit pulls the bit line high or low and turns on the transistor, thus charging or discharging the capacitor. To read from the cell, the DRAM control circuit precharges the bit line to an intermediate level, then turns on the transistor. As the charges on the capacitor and the bit line equalize, the voltage on the bit line either increases slightly or decreases slightly, depending on whether the storage capacitor was charged or discharged. A sensor detects and amplifies the change, thus determining whether the cell stored a 1 or a 0. Unfortunately, this process destroys the stored value in the cell, so the control circuit must then restore the value by pulling the bit line high or low, as appropriate, before turning off the transistor. The time taken to complete the restoration is added to the access time, making the overall read cycle significantly longer than than that for an SRAM.

Another property of a DRAM cell is that, while the transistor is turned off, charge leaks from the capacitor. This is the meaning of the term "dynamic" applied to DRAMs. To compensate, the control circuit must read and restore the value in each cell in the DRAM before the charge decays too much. This process is called *refreshing* the DRAM. DRAM manufacturers typically specify a period of 64ms between refreshes for each cell. The cells in a DRAM are typically organized into several rectangular arrays, called banks, and the DRAM control circuit is organized to refresh one row of each bank at a time. Since the DRAM cannot perform a normal write or read operation while it is refreshing a row, the refresh operations must be interleaved between writes and reads. Depending on the application, it may be possible to refresh all rows in a burst once every 64ms. Alternatively, we may have to refresh one row at a time between writes and reads, making sure that all rows are refreshed within 64ms. The important thing is to avoid scheduling a refresh when a write or read is required and cannot be deferred.

Historically, timing of DRAM control signals used to be asynchronous, and management of refreshing was performed by control circuits external to the DRAM chips. More recently, manufacturers changed to synchronous DRAMs (SDRAMs) that use registers on inputs to sample address, data and control signals on clock edges. This is analogous to the difference between asynchronous and synchronous SRAMs, and makes it easier to incorporate DRAMs into systems that use a clocked synchronous timing methodology. Manufacturers have also incorporated refresh control circuits into the DRAM chips, also making use of DRAMs easier. Since applications with very high data transfer rate requirements may be limited by the relatively slow access times of DRAMs, manufacturers have more recently incorporated further features to improve performance. These include the ability to access a burst of data from successive locations

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.

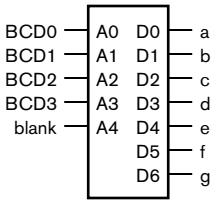

FIGURE 5.21 A 32×7-bit ROM used as a 7-segment decoder.

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

numbers. Thus, the file rom.data specified in the above example could contain the data

```
00000  0126F  017C0  A0018
10009  2667A  30115  00000
```

Values are read from the file into successive elements of the specified variable until either the end of the file is reached or all elements of the variable are loaded.

Programmable ROMs

ROMs in which the contents are manufactured into the memory are suitable for applications where the number of manufactured parts is high and where we are sure that the contents will not need to change over the lifetime of the product. In other applications, we would prefer to be able to revise the ROM contents from time to time, or to use a form of ROM with lower costs for low-volume production. A *programmable ROM* (PROM) meets these requirements. It is manufactured as a separately packaged chip with no content stored in its memory cells. The memory contents are programmed into the cells after manufacture, either using a special programming device before the chip is assembled into a system, or using special programming circuits when the chip is in the final system.

There are a number of forms of PROMs. Early PROMs used fusible links to program the memory cells. Once a link was fused, it could not be replaced, so programming could only be done once. These devices are now largely obsolete. They were replaced by PROMs that could be erased, either with ultraviolet light (so called EPROMs), or electrically using a higher-than-normal power-supply voltage (so-called electrically erasable PROMs, or EEPROMs).

Flash Memories

Most new designs use flash memory, which is a form of electrically erasable programmable ROM. It is organized so that blocks of storage can be erased at once, followed by programming of individual memory locations. A flash memory typically allows only a limited number of erasure and programming operations, typically hundreds of thousands, before the device "wears out." Thus, flash memories are not a suitable replacement for RAMs.

There are two kinds of flash memories, NOR and NAND flash, referring to the organization of the transistors that make up the memory cells. Both kinds are organized as blocks (commonly of 16, 64, 128, or 256 Kbytes) that must be erased in whole before being written. In a NOR flash memory, locations can then be written (once per erasure) and read (an arbitrary number of times) in random order. The IC has similar address,

data and control signals to an SRAM and can read data with a comparable access time, making it suitable for use as a program memory for an embedded processor, for storing configuration parameters to be used to control system operation, and for storing configuration information for FPGAs.

In a NAND flash memory, on the other hand, locations are written and read one page at a time, a page being typically 2 Kbytes. Read access to a given location would require reading the page containing the location, followed by selection of the required data, taking several microseconds. If all of the locations in a page are required, however, sequential reading is much faster, comparable in time to SRAM. Erasing a block and writing a page of data are significantly slower than SRAM access times. For example, the data sheet for the Micron Technology MT29F16G08FAA 16G bit IC specifies a random read time of 25μs, a sequential read time of 25ns, a block erase time of 1.5ms, and a page write time of 220μs. Given their different access behavior, NAND flash memories have a different interface than SRAMs, making control circuits more involved. The advantage of NAND flash memory is that the density of storage cells is greater than that of NOR flash. Thus, NAND flash chips are better suited to applications in which large amounts of data must be stored cheaply. One of the largest applications of NAND flash memories is in memory cards for consumer devices such as digital cameras. They are also used in USB memory sticks for general purpose computers.

KNOWLEDGE TEST QUIZ

1. What is the difference between RAM and ROM?
2. What is meant by the terms volatile and nonvolatile?
3. What is the difference between static and dynamic RAM?
4. What is meant by the access time of a RAM?
5. Why are asynchronous SRAMs difficult to use in high-speed clocked synchronous designs?
6. What is the difference between flow-through and pipelined SSRAMs?
7. What Verilog type is required for a variable to represent memory storage?
8. What benefit does a multiport memory have over a single-port memory with multiplexed address and data connections?
9. How can we work out what will happen if we perform concurrent writes to a given location in a synchronous dual-port memory?
10. What does FIFO stand for?
11. How does a FIFO facilitate communication of data between clock domains?

5.3 ERROR DETECTION AND CORRECTION

In most of our discussions, we have assumed that digital circuits store and process information correctly, though in Section 2.2.2 we did introduce the idea of bit errors and some approaches to dealing with them. Bit errors can occur in memories from a number of causes. Some errors are *transient*, also called *soft errors*, and involve a bit flip in a memory cell without a permanent effect on the cell's capacity to store data. In DRAMs, soft errors are typically caused by high-energy neutrons generated by collision of cosmic rays with atoms in the earth's atmosphere. The neutrons collide with silicon atoms in the DRAM chip, leaving a stream of charge that can disrupt the storage or reading of charge in a DRAM cell. The frequency of soft-error occurrence, the *soft-error rate*, depends on the way in which DRAMs are manufactured and the location in which they operate. Hence, soft-error rates are highly variable between systems. Soft errors can also occur in DRAMs and other memories from electrical interference, the effects of poor physical circuit design and other causes.

Errors that persist in a memory circuit are called *hard errors*. They can result from manufacturing defects or from electrical "wear" after prolonged use. A memory cell or chip affected by a hard error is no longer able to store data. A read operation would always yield a 0 or a 1 value, regardless of the bit value that was previously written.

Given that memories are more susceptible to bit errors than logic circuits using flip-flops and registers for storage, due to the storage density and the longevity of data in memories, it is more common to include some form of error detection in memory circuits than in logic circuits. A common approach is to use parity, described in Section 2.2.2. Recall that parity involves counting the number of 1 bits in a code word and setting a parity bit to 1 or 0 to ensure that the total number of 1 bits is even (if we choose even parity) or odd (if we choose odd parity). In the case of memories, use of parity involves adding an extra bit cell to each memory location. When we write to a location, we compute the parity bit and store it in the extra cell. When we read a location, we check that the data, together with the parity bit, have the correct parity. If so, we assume the data is uncorrupted. Otherwise, we take appropriate action to deal with the error in the stored data.

The problem with using parity to check for errors, as we discussed in Section 2.2.2, is that it only allows us to detect a single bit flip in a stored code word. It does not allow us to identify which bit flipped, nor does it allow us to detect an even number of bit flips. If we could identify the particular bit that flipped, we could correct the error by flipping the bit back to its original value, and then continue operating as normal. We could also write the corrected data back to the memory on the assumption that the bit flip was a soft error. In order to be able to identify which bit flipped, we need to consider the invalid code words that result from flipping each bit of each valid code word. Provided all of those invalid code words are distinct, we can use the value of the invalid code word to identify the flipped bit.

One scheme for doing this is to use a form of *error correcting code* (ECC) known as a *Hamming code*. We will start with a single-error correcting Hamming code, that is, a code that allows us to correct a single bit flip within a code word. If our code word has N bits, we need $\log_2 N + 1$ additional check bits for the ECC. For example, if we have 8 data bits, we need 4 check bits, giving a total of 12 bits. The check bits are computed from the values of the data bits during a write operation, and the entire ECC word is written to the memory location.

To illustrate how the check bits are computed, we will number the data bits of an 8-bit code word d_1 through d_8 and the ECC bits e_1 through e_{12}. (Normally, we've numbered bits starting from 0, but for this explanation, it's more convenient to number all index positions from 1.) The ECC bits whose indices are powers of 2 are used as check bits, and the remaining ECC bits are the data bits, in order, as shown in Figure 5.22. If we write the indices of the ECC bits in binary, the check bit with a 1 in position i of its index is the exclusive-OR (that is, the parity) of the data ECC bits that have a 1 in position i of their indices. For example, check bit e_2 (at index

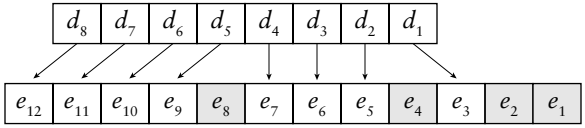

FIGURE 5.22 Distribution of data and check bits within an ECC word.

0010_2) is the exclusive-OR of data bits e_3, e_6, e_7, e_{10} and e_{11} (at indices 0011_2, 0110_2, 0111_2, 1010_2 and 1011_2). Since each data ECC bit has at least two 1 bits in its binary index (otherwise it would be a check bit), each data bit is included in the computation of at least two check bits.

When the memory location is read, again, the entire ECC word is read. We recompute the values of the check bits from the data ECC bits and compare them, using a bit-wise exclusive OR, with the check bits read from memory. If the comparison result is 0000, the recomputed check bits match the read check bits, so all is well. However, if one of the stored ECC bits (either a data bit or a check bit) is flipped from the original, the comparison result, called the *syndrome*, will be other than 0000. It turns out to be the binary index of the ECC bit that has flipped. Thus, we can use the syndrome value to correct the error by flipping the indexed bit back.

EXAMPLE 5.11 Compute the 12-bit ECC word corresponding to the 8-bit data word 01100001.

SOLUTION The check bits are

$$e_1 = e_3 \oplus e_5 \oplus e_7 \oplus e_9 \oplus e_{11} = d_1 \oplus d_2 \oplus d_4 \oplus d_5 \oplus d_7 = 1 \oplus 0 \oplus 0 \oplus 0 \oplus 1 = 0$$

$$e_2 = e_3 \oplus e_6 \oplus e_7 \oplus e_{10} \oplus e_{11} = d_1 \oplus d_3 \oplus d_4 \oplus d_6 \oplus d_7 = 1 \oplus 0 \oplus 0 \oplus 1 \oplus 1 = 1$$

$$e_4 = e_5 \oplus e_6 \oplus e_7 \oplus e_{12} = d_2 \oplus d_3 \oplus d_4 \oplus d_8 = 0 \oplus 0 \oplus 0 \oplus 0 = 0$$

$e_8 = e_9 \oplus e_{10} \oplus e_{11} \oplus e_{12} = d_5 \oplus d_6 \oplus d_7 \oplus d_8 = 0 \oplus 1 \oplus 1 \oplus 0 = 0$

Thus the ECC word is 011000000110.

EXAMPLE 5.12 Determine whether there is an error in the ECC word 110111000110, and if so, correct it.

SOLUTION The check bits computed from the data bits of the ECC word are

$e_1 = e_3 \oplus e_5 \oplus e_7 \oplus e_9 \oplus e_{11} = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 1 = 0$

$e_2 = e_3 \oplus e_6 \oplus e_7 \oplus e_{10} \oplus e_{11} = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 = 1$

$e_4 = e_5 \oplus e_6 \oplus e_7 \oplus e_{12} = 0 \oplus 0 \oplus 1 \oplus 1 = 0$

$e_8 = e_9 \oplus e_{10} \oplus e_{11} \oplus e_{12} = 1 \oplus 1 \oplus 0 \oplus 1 = 1$

The syndrome is $1010 \oplus 1010 = 0000$. Thus, there is no error in the read ECC.

EXAMPLE 5.13 Determine whether there is an error in the ECC word 000111000100, and if so, correct it.

SOLUTION The check bits computed from the data bits of the ECC word are

$e_1 = e_3 \oplus e_5 \oplus e_7 \oplus e_9 \oplus e_{11} = 1 \oplus 0 \oplus 1 \oplus 1 \oplus 0 = 1$

$e_2 = e_3 \oplus e_6 \oplus e_7 \oplus e_{10} \oplus e_{11} = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 0$

$e_4 = e_5 \oplus e_6 \oplus e_7 \oplus e_{12} = 0 \oplus 0 \oplus 1 \oplus 0 = 1$

$e_8 = e_9 \oplus e_{10} \oplus e_{11} \oplus e_{12} = 0 \oplus 0 \oplus 0 \oplus 1 = 1$

The syndrome is $1101 \oplus 1000 = 0101$. Thus, there is an error in bit e_5 of the read ECC. That bit should be flipped back from 0 to 1, giving the corrected ECC word 000111010100.

Note that we have assumed that only one bit of the stored ECC word could be in error. If two or more bits flip, the checking process may incorrectly identify a single bit as having flipped, or it may yield an invalid syndrome. The problem arises from the fact that we have insufficient invalid code words to distinguish between single-bit errors and double-bit errors. A simple remedy is to add further check bits. If we add a check bit that is the exclusive-OR of all of the data bits, the resulting error-checking code allows us to correct any single-bit error and to detect (but not correct) any double-bit error. If we assume that errors are independent, the probability of a double-bit error is very low, so this scheme suffices in many applications. If extreme reliability and resilience to errors is required, we can further extend the error-checking code to enable correcting of multiple-bit errors. The details of how we might do this are beyond the scope of this book, but are described in Section 5.5, Further Reading.

	SINGLE-BIT CORRECTION		DOUBLE-BIT DETECTION	
N	CHECK BITS	OVERHEAD	CHECK BITS	OVERHEAD
8	4	50%	5	63%
16	5	31%	6	38%
32	6	19%	7	22%
64	7	11%	8	13%
128	8	6.3%	9	7.0%
256	9	3.5%	10	3.9%

TABLE 5.3 Number of check bits and relative storage overhead for single-bit correction and additional double-bit detection of errors.

A final consideration in our discussion of error checking and correcting for memories is the storage overhead required. In our illustration of ECCs for 8-bit code words, we saw that correcting single-bit errors requires 4 check bits (a 50% overhead) and detecting double-bit errors requires 5 check bits (a 63% overhead). This is clearly a significant storage overhead, especially when compared to the single parity bit required just to detect single-bit errors (a 13% overhead). However, we noted that single-bit correction using Hamming codes needs $\log_2 N + 1$ check bits for N bits of data. Double-bit error detection needs $\log_2 N + 2$ check bits. If we provide checking and correction over longer data words, the relative storage overhead is less, as shown in Table 5.3. For larger data words, provision of this form of error detection and correction is increasingly attractive.

There are other, more elaborate, error correction and detection codes that we can use as alternatives to Hamming codes. However, they also add check bits to the data, and so require extra storage capacity and extra circuitry to detect and correct errors. They differ in the storage overhead and the complexity of the additional circuitry, as well as in the number of simultaneous errors they can deal with. This range of techniques allows us to make design trade-offs, depending on the reliability requirements and other constraints of our application. Since Hamming codes are one of the simplest ECCs, they are most often used in applications requiring moderately high reliability, such as network server computers. More complex ECCs are used in specialized high-reliability applications, such as aerospace computers and communications systems.

KNOWLEDGE TEST QUIZ

1. What is the distinction between a soft error and a hard error?
2. What is a common cause of soft errors in DRAMs?
3. What corrective action can we take when a parity error is detected?
4. Using a Hamming code, how many check bits are required for single-error correction and double-error detection for 4-bit data words?

5.4 CHAPTER SUMMARY

- A memory contains an array of storage locations, each with a unique address. A $2^n \times m$-bit memory has n-bit addresses that run from 0 to $2^n - 1$.
- A write operation stores a data value at a given location. A read operation yields the data value stored at a given location. Control signals govern write and read operations.
- We can connect multiple memory components in parallel to store wider data values. We can connect multiple memory components in banks, with a decoder to select among the banks, to provide more locations.
- Memories with tristate drivers on the data outputs simplify bank connection. At most one component drives data outputs at a time; the rest place their outputs in the high-impedance (hi-Z) state.
- Volatile memory only retains data for as long as power is applied. Nonvolatile memory retains data without power. The term RAM refers to volatile memory that can be written and read with equal facility in any order. ROM refers to memory that can only be read once it is manufactured or programmed.
- Data in static RAM (SRAM) persists for as long as power is supplied, whereas data in dynamic RAM (DRAM) must be periodically refreshed. Asynchronous SRAM does not rely on a clock for its timing. Synchronous SRAM (SSRAM) uses a clock to sample control, address and data signals, thus simplifying their incorporation into clocked synchronous systems. SSRAMs include flow-through and pipelined variants.
- The access time is the delay from starting a read operation to having valid data. The cycle time is the total time taken for a read or write operation.
- Multiport memories allow concurrent operations by different parts of a digital system. A first-in first-out (FIFO) is a dual-port memory used as a queue for data. An important use of FIFOs is to pass data between different clock domains.
- A ROM is a combinational circuit that maps from an address to a data value. It can be used to implement an arbitrary Boolean function.
- Programmable ROMs (PROMs) are programmed with data after manufacture. Flash memories can be erased and reprogrammed during system operation, and are useful for storing configuration information.

- Atmospheric neutrons and other effects can cause bit errors in data stored in a memory. The error may be transient (a soft error) or permanent (a hard error).
- Check bits can be stored along with data to detect and correct errors. A single parity bit can detect a single-bit error but not a double-bit error. Error correcting codes, such as Hamming codes, can correct single-bit errors and detect double-bit errors.

5.5 FURTHER READING

Advanced Semiconductor Memories: Architectures, Designs, and Applications, Ashok K. Sharma, Wiley-IEEE Press, 2002. Describes a range of memory devices, including SRAMS, DRAMS and nonvolatile memories.

Computer Organization and Design: The Hardware/Software Interface, David A. Patterson and John L. Hennessy, Morgan Kaufmann Publishers, 2005. This book contains a chapter on memory system design for computers, describing how alternative organizations can improve memory system performance.

Memory Systems: Cache, DRAM, Disk—A Holistic Approach to Design, Bruce Jacob, Spencer Ng, and David Wang, Morgan Kaufmann Publishers, 2007. Includes an extensive description of DRAM technology and its place in computer memory systems. Also describes error-correcting codes, including Hamming codes and more elaborate schemes, and the causes and frequency of occurrence of memory errors.

Synchronous and Asynchronous FIFO Designs, Peter Alfke, Xilinx Application Note XAPP051, 1996, http://direct.xilinx.com/bvdocs/appnotes/xapp051.pdf. Describes a FIFO control scheme for an FPGA in which the write and read clocks are different.

Synthesizable High-Performance SDRAM Controllers, Xilinx Application Note XAPP134, 2005, http://www.xilinx.com/bvdocs/appnotes/xapp134.pdf. This application note gives an overview of SDRAM operation and describes a controller subsystem that can be implemented as part of an FPGA-based design.

A Nonvolatile Memory Overview, Jitu J. Makwana and Dieter K. Schroder, 2004, http://aplawrence.com/Makwana/nonvolmem.html. Describes the circuit structures and operation of nonvolatile memory devices.