are often classified into **logic families**. Each family has its advantages and disadvantages, and each differs from the others in its capabilities and limitations. The logic families currently of interest include TTL, NMOS/PMOS, CMOS, and ECL.

TTL (transistor–transistor logic) replaces all the diodes originally found in integrated circuits with bipolar transistors. (See the sidebar on transistors in Chapter 1 for more information.) TTL defines binary values as follows: 0 to 0.8 volts is logic 0, and 2 to 5 volts is logic 1. Virtually any gate can be implemented using TTL. Not only does TTL offer the largest number of logic gates (from the standard combinational and sequential logic gates to memory), but this technology also offers superior speed of operation. The problem with these relatively inexpensive integrated circuits is that they draw considerable power.

TTL was used in the first integrated circuits that were widely marketed. However, the most commonly used type of transistor used in integrated circuits today is called a **MOSFET (metal oxide semiconductor field effect transistor)**. **Field effect transistors (FETs)** are simply transistors whose output fields are controlled by a variable electric field. The phrase *metal-oxide semiconductor* is actually a reference to the process used to make the chip, and even though polysilicon is used today instead of metal, the name continues to be used.

NMOS (N-type metal-oxide semiconductors) and **PMOS (P-type metal-oxide semiconductors)** are the two basic types of MOS transistors. NMOS transistors are faster than PMOS transistors, but the real advantage of NMOS over PMOS is that of higher component density (more NMOS transistors can be put on a single chip). NMOS circuits have lower power consumption than their bipolar relatives. The main disadvantage of NMOS technology is its sensitivity to damage from electrical discharge. In addition, not as many gate implementations are available with NMOS as with TTL. Despite NMOS circuits using less power than TTL, increased NMOs circuit densities caused a resurgence in power consumption problems.

CMOS (complementary metal-oxide semiconductor) chips were designed as low-power alternatives to TTL and NMOS circuits, providing more TTL equivalents than NMOS in addition to addressing the power issues. Instead of using bipolar transistors, this technology uses a complementary pair of FETs, an NMOS and a PMOS FET (hence the name "complementary"). CMOS differs from NMOS because when the gate is in a static state, CMOS uses virtually no power. Only when the gate switches states does the circuit draw power. Lower power consumption translates to reduced heat dissipation.

For this reason, CMOS is extensively used in a wide variety of computer systems. In addition to low power consumption, CMOS chips operate within a wide range of supply voltages (typically from 3 to 15 volts)—unlike TTL, which requires a power supply voltage of plus or minus 0.5 volts. However, CMOS technology is extremely sensitive to static electricity, so extreme care must be taken when handling circuits. Although CMOS technology provides a larger selection of gates than NMOS, it still does not match that of its bipolar relative, TTL.

ECL (emitter coupled logic) gates are used in situations that require extremely high speeds. Whereas TTL and MOS use transistors as digital switches (the transistor is either saturated or cut off), ECL uses transistors to guide current through gates, resulting in transistors that are never completely turned off nor completely saturated. Because they are always in an active status, the transistors can change states very quickly. However, the trade-off for this high speed is substantial power requirements. Therefore, ECL is used only rarely, in very specialized applications.

A newcomer to the logic family scene, **BiCMOS (bipolar CMOS)** integrated circuits use both the bipolar and CMOS technologies. Despite the fact that BiCMOS logic consumes more power than TTL, it is considerably faster. Although not currently used in manufacturing, BiCMOS appears to have great potential.

3.6.6 An Application of Sequential Logic: Convolutional Coding and Viterbi Detection

The special section that follows Chapter 2 describes several coding methods employed in data storage and communication. One of them is the partial response maximum likelihood (PRML) encoding method. Our previous discussion (which isn't prerequisite for understanding this section) concerned the "partial response" component of PRML. The "maximum likelihood" component derives from the way that bits are encoded and decoded. The salient feature of the decoding process is that only certain bit patterns are valid. These patterns are produced using a convolutional code. A **Viterbi decoder** reads the bits that have been output by a convolutional encoder and compares the symbol stream read with a set of "probable" symbol streams. The one with the least error is selected for output. We present this discussion because it brings together a number of concepts from this chapter as well as Chapter 2. We begin with the encoding process.

The Hamming code introduced in Chapter 2 is a type of forward error correction that uses blocks of data (or block coding) to compute the necessary redundant bits. Some applications require a coding technique suitable for a continuous stream of data, such as that from a satellite television transmitter. **Convolutional coding** is a method that operates on an incoming serial bit stream, generating an encoded serial output stream (including redundant bits) that enables it to correct errors continuously. A **convolutional code** is an encoding process whereby the output is a function of the input and some number of bits previously received. Thus, the input is overlapped, or *convoluted*, over itself to form a stream of output symbols. In a sense, a convolutional code builds a context for accurate decoding of its output. Convolutional encoding combined with Viterbi decoding has become an accepted industry standard for encoding and decoding data stored or transmitted over imperfect (*noisy*) media.

The convolutional coding mechanism used in PRML is illustrated in Figure 3.33. Careful examination of this circuit reveals that two output bits are written for each input bit. The first output bit is a function of the input bit and the second

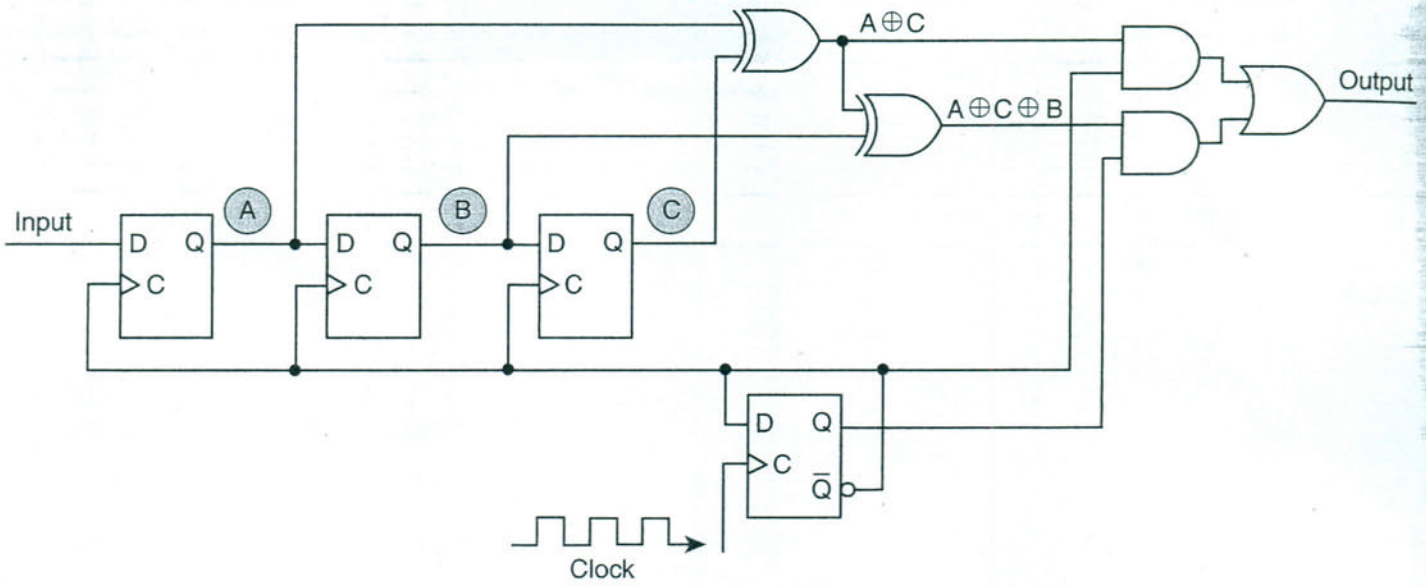

FIGURE 3.33 Convolutional Encoder for PRML

previous input bit: *A* XOR *C*. The second bit is a function of the input bit and the two previous bits: *A* XOR *C* XOR *B*. The two AND gates at the right-hand side of the diagram alternatively select one of these functions during each pulse of the clock. The input is shifted through the D flip-flops on every second clock pulse. We note that the leftmost flip-flop serves only as a buffer for the input and isn't strictly necessary.

At first glance, it may not be easy to see how the encoder produces two output bits for every input bit. The trick has to do with the flip-flop situated between the clock and the other components of the circuit. When the complemented output of this flip-flop is fed back to its input, the flip-flop alternately stores 0s and 1s. Thus, the output goes high on every other clock cycle, enabling and disabling the correct AND gate with each cycle.

We step through a series of clock cycles in Figure 3.34. The initial state of the encoder is assumed to contain all 0s in the flip-flops labeled *A*, *B*, and *C*. A couple of clock cycles are required to move the first input into the *A* flip-flop (buffer), and the encoder outputs two zeros. Figure 3.34a shows the encoder with the first input (1) after it has passed to the output of flip-flop *A*. We see that the clock on flip-flops *A*, *B*, and *C* is enabled, as is the upper AND gate. Thus, the function *A* XOR *C* is routed to the output. At the next clock cycle (Figure 3.34b), the lower AND gate is enabled, which routes the function *A* XOR *C* XOR *B* to the output. However, because the clock on flip-flops *A*, *B*, and *C* is disabled, the input bit does not propagate from flip-flop *A* to flip-flop *B*. This prevents the next input bit from being consumed while the second output bit is written. At clock cycle 3 (Figure 3.34c), the input has propagated through flip-flop *A*, and the bit that was in flip-flop *A* has propagated to flip-flop *B*. The upper AND gate on the output is enabled and the function *A* XOR *C* is routed to the output.

The characteristic table for this circuit is given in Table 3.13. As an example, consider the stream of input bits, 11010010. The encoder initially contains all 0s,

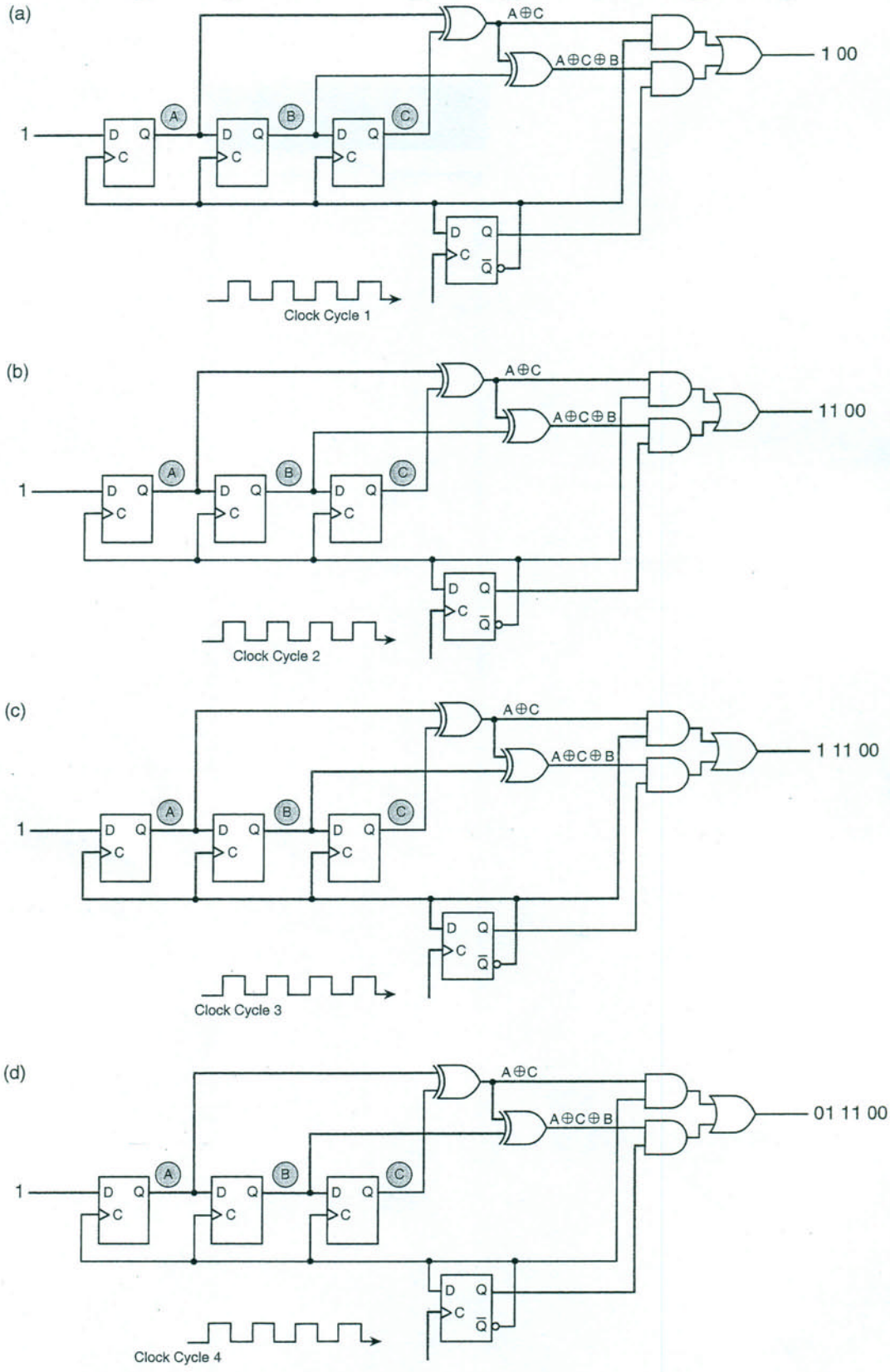

FIGURE 3.34 Stepping Through Four Clock Cycles of a Convolutional Encoder

Input A	Current State B C	Next State B C	Output
0	00	00	00
1	00	10	11
0	01	00	11
1	01	10	00
0	10	01	10
1	10	11	01
0	11	01	01
1	11	11	10

TABLE 3.13 Characteristic Table for the Convolutional Encoder in Figure 3.33

so $B = 0$ and $C = 0$. We say that the encoder is in State 0 (00_2). When the leading 1 of the input stream exits the buffer, A, $B = 0$ and $C = 0$, giving (A XOR C XOR B) = 1 and (A XOR C) = 1. The output is 11 and the encoder transitions to State 2 (10_2). The next input bit is 1, and we have $B = 1$ and $C = 0$ (in State 2), giving (A XOR C XOR B) = 0 and (A XOR C) = 1. The output is 01 and the encoder transitions to State 1 (01_2). Following this process over the remaining six bits, the completed function is:

F(1101 0010) = 11 01 01 00 10 11 11 10

The encoding process is made a little clearer using the Mealy machine (Figure 3.35). This diagram informs us at a glance as to which transitions are possible and which are not. You can see the correspondence between Figure 3.35 machine and the characteristic table by reading the table and tracing the arcs or vice versa. The fact that there is a limited set of allowable transitions is crucial to the error correcting properties of this code and to the operation of the Viterbi decoder,

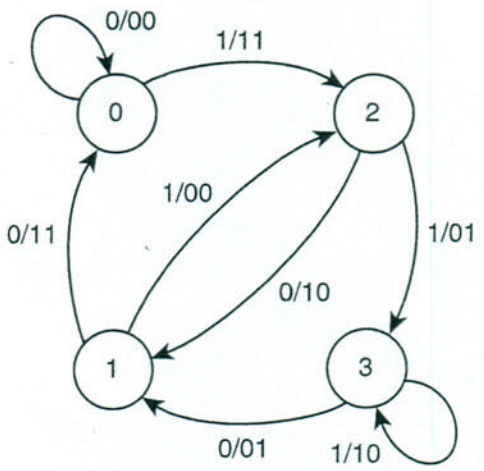

FIGURE 3.35 Mealy Machine for the Convolutional Encoder in Figure 3.33

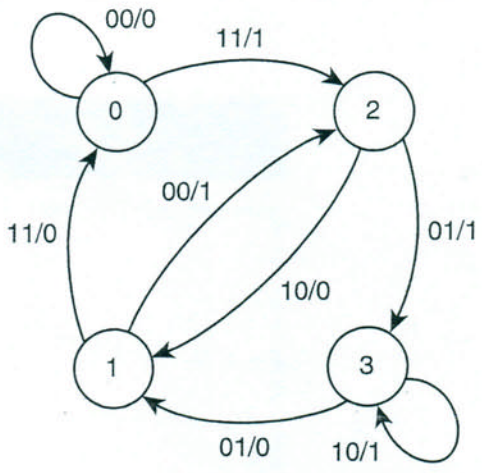

FIGURE 3.36 Mealy Machine for a Convolutional Decoder

which is responsible for decoding the stream of bits correctly. By reversing the inputs with the outputs on the transition arcs, as shown in Figure 3.36, we place bounds around the set of possible decoding inputs.

For example, suppose the decoder is in State 1 and sees the pattern, 00 01. The decoded bit values returned are 1 1, and the decoder ends up in State 3. (The path traversed is $1 \rightarrow 2 \rightarrow 3$.) If on the other hand, the decoder is in State 2 and sees the pattern, 00 11, an error has occurred because there is no outbound transition on State 2 for 00. The outbound transitions on State 2 are 01 and 10. Both of these have a Hamming distance of 1 from 00. If we follow both (equally likely) paths out of State 2, the decoder ends up either in State 1 or State 3. We see that there is no outbound transition on State 3 for the next pair of bits, 11. Each outbound transition from State 3 has a Hamming distance of 1 from 11. This gives an accumulated Hamming distance of 2 for both paths: $2 \rightarrow 3 \rightarrow 1$ and $2 \rightarrow 3 \rightarrow 2$. However, State 1 has a valid transition on 11. By taking the path, $2 \rightarrow 1 \rightarrow 0$, the accumulated error is only 1, so this is the most likely sequence. The input therefore decodes to 00 with **maximum likelihood**.

An equivalent (and probably clearer) way of expressing this idea is through the trellis diagram, shown in Figure 3.37. The four states are indicated on the left side of the diagram. The transition (or time) component reads from left to right. Every code word in a convolutional code is associated with a unique path in the trellis diagram. A Viterbi detector uses the logical equivalent of paths through this diagram to determine the most likely bit pattern. In Figure 3.37, we show the state transitions that occur when the input sequence 00 10 11 11 is encountered with the decoder starting in State 1. You can compare the transitions in the trellis diagram with the transitions in the Mealy diagram in Figure 3.36.

Suppose we introduce an error in the first pair of bits in our input, giving the erroneous string 10 10 11 11. With our decoder starting in State 1 as before, Figure 3.38 traces the possible paths through the trellis. The accumulated Hamming

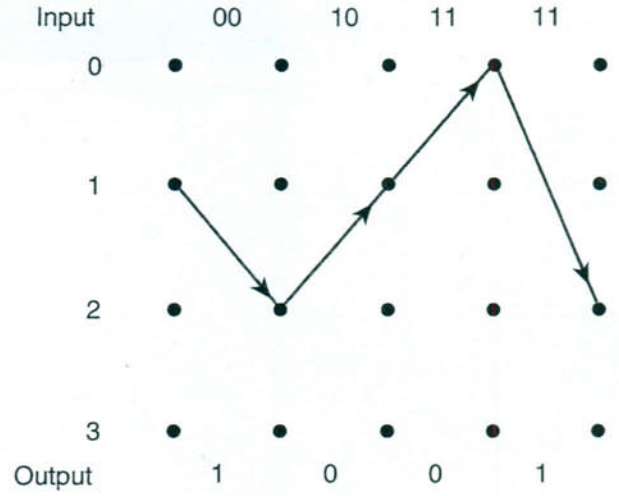

FIGURE 3.37 Trellis Diagram Illustrating State Transitions for the Sequence 00 10 11 11

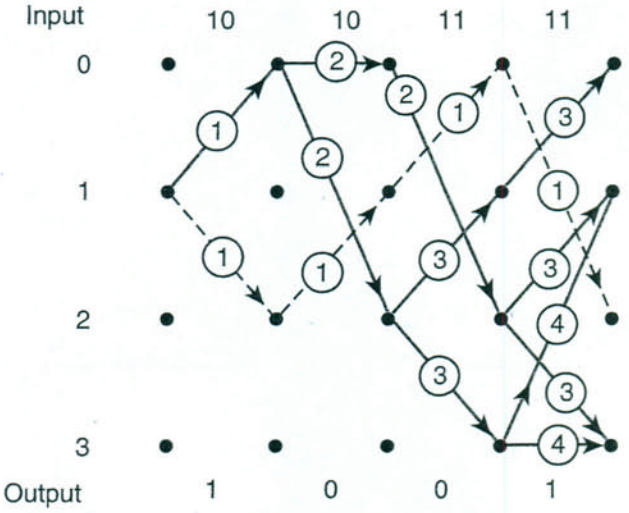

FIGURE 3.38 Trellis Diagram Illustrating Hamming Errors for the Sequence 10 10 11 11

distance is shown on each of the transition arcs. The correct path that correctly assumes that the string should be 00 10 11 11 is the one having the smallest accumulated error, so it is accepted as the correct sequence.

In most cases where it is applied, the Viterbi decoder provides only one level of error correction. Additional error correction mechanisms such as cyclic redundancy checking and Reed-Soloman coding (discussed in Chapter 2) are applied after the Viterbi algorithm has done what it can to produce a clean stream of symbols. All these algorithms are usually implemented in hardware for utmost speed using the digital building blocks described in this chapter.