microwave oven. The oven will be in the "on" state only when the door is closed, the control dial is set to "cook," or "defrost," and there is time on the timer. The "on" state means that the magnetron is producing microwaves, the light in the oven compartment is lit, and the carousel is rotating. If the time expires, the door opens, or the control is turned from "cook," to "off," the oven moves to the "off" state. The dimension provided by the timer along with the numerous signals that define a state are hard to capture in the Moore and Mealy models. For this reason, Christopher R. Clare invented the **algorithmic state machine** (**ASM**). As its name implies, an algorithmic state machine is directed at expressing the algorithms that advance an FSM from one state to another.

An algorithmic state machine consists of blocks that contain a state box, a label, and optionally condition and output boxes (Figure 3.28). Each ASM block has exactly one entry point and at least one exit point. Moore type outputs (the circuit signals) are indicated inside the state block; Mealy-type outputs are indicated in the oval output "box." If a signal is asserted when "high," it is prefixed with an *H*, otherwise, it is prefixed with an *L*. If the signal is asserted immediately, it is also prefixed with an *I*; otherwise, the signal asserts at the next clock cycle. The input conditions that cause changes in state (this is the algorithmic part) are expressed by elongated, six-sided polygons called condition boxes. Any number of condition boxes can be placed inside an ASM block, and the order in which they are shown is unimportant. An ASM for our microwave oven example is shown in Figure 3.29.

As implied, ASMs can express the behavior of either a Moore or Mealy machine. Moore and Mealy machines are probably equivalent and can be used interchangeably. However, it is sometimes easier to use one rather than the other, depending on the application. In most cases, Moore machines require more states (memory) but result in simpler implementations than Mealy machines, because there are fewer transitions to account for in Moore machines.

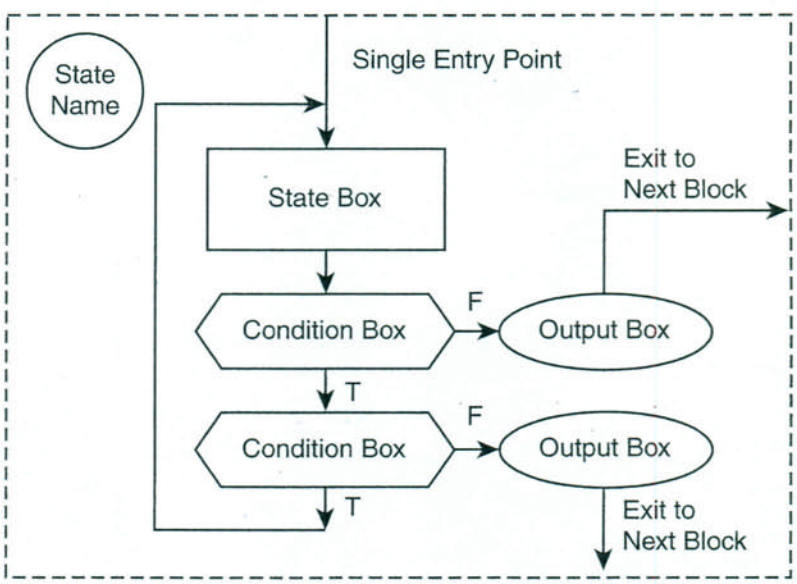


FIGURE 3.28 Components of an Algorithmic State Machine

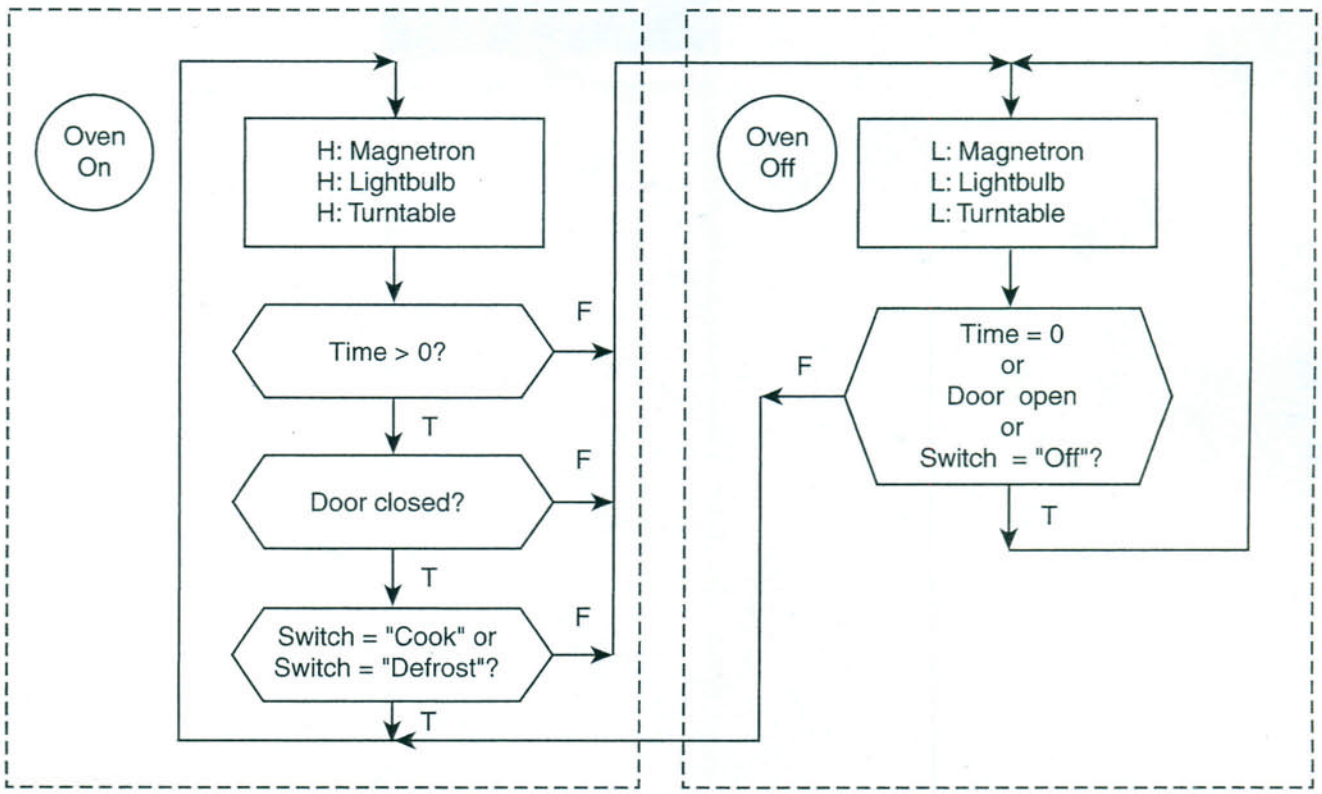


FIGURE 3.29 Algorithmic State Machine for a Microwave Oven

Hardware-free Machines

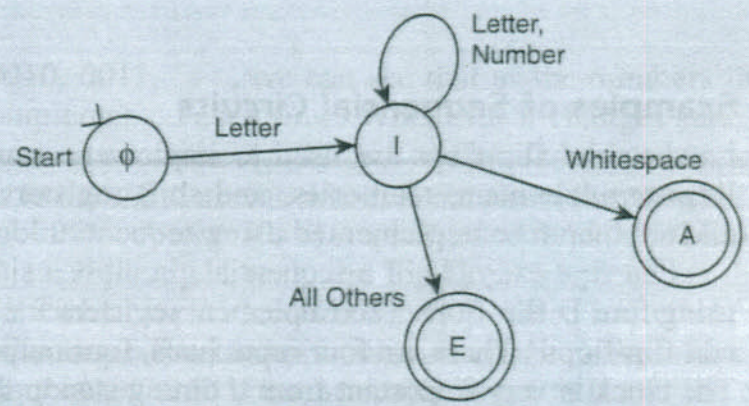
Moore and Mealy machines are only two of many different types of finite state machines that you will encounter in computer science literature. An understanding of FSMs is essential in the study of programming languages, compilers, the theory of computation, and automata theory. We refer to these abstractions as *machines* because machines are devices that respond to a set of stimuli (events) by generating predictable responses (actions) based on a history of prior events (current state). One of the most important of these is the **deterministic finite automata** (**DFA**) computational model. Formally speaking, a DFA, M, is completely described by the quintuple $M = (Q, S, \Sigma, \delta, F)$ where:

- Q is a finite set of states that represents every configuration the machine can assume;
- S is an element of Q that represents the start state, which is the initial state of the machine before it receives any inputs;
- Σ is the input alphabet or set of events that the machine will recognize;
- δ is a transition function that maps a state in Q and a letter from the input alphabet to another (possibly the same) state in Q; and

- F is a set of states (elements of Q) designated as the final (or accepting) states.

DFAs are particularly important in the study of programming languages; they are used to recognize grammars or languages. To use a DFA, you begin in the Start state and process an input string, one character at a time, changing states as you go. Upon processing the entire string, if you are in a final accepting state, a legal string is "accepted" by that DFA. Otherwise, the string is rejected.

We can use this DFA definition to describe a machine—as in a compiler—that extracts variable names (character strings) from a source code file. Suppose our computer language accepts variable names that must start with a letter, can contain an infinite stream of letters or numbers following the initial letter, and is terminated by a whitespace character (tab, space, linefeed, etc.). The initial state of the variable name is the null string, because no input has been read. We indicate this starting state in the figure below with an exaggerated arrowhead (there are several other notations). When the machine recognizes an alphabetic character, it transitions to State *I*, where it stays as long as a letter or number is input. Upon accepting a whitespace character, the machine transitions to State *A*, its final accepting state, which we indicate with a double circle. If a character other than a number, letter, or whitespace is entered, the machine enters its error state, which is a final state that rejects the string.



Finite State Machine for Accepting a Variable Name

Of more interest to us (because we are discussing hardware), are Moore and Mealy FSMs that have output states. The basic difference between these FSMs and DFAs is that—in addition to the transition function moving us from state to state—Moore and Mealy machines also generate an output symbol. Furthermore, no set of final states is defined because circuits have no concept of halting or accepting strings, they instead generate output. Both the Moore

and Mealy machines, M, can be completely described by the quintuple $M = (Q, S, \Sigma, \Gamma, \delta)$ where:

- Q is a finite set of states that represents each configuration of the machine;
- S is an element of Q that represents the Start state, the state of the machine before it has received any inputs;
- Σ is the input alphabet or set of events that the machine will recognize;
- Γ is the finite output alphabet; and
- δ is a transition function that maps a state from Q and a letter from the input alphabet to a state from Q.

We note that the input and output alphabets are usually identical, but they don't have to be. The way in which output is produced is the distinguishing element between the Moore and Mealy machines. Hence, the output function of the Moore machine is embedded in its definition of S, and the output function for the Mealy machine is embedded in the transition function, δ.

If any of this seems too abstract, just remember that a computer can be thought of as a universal finite state machine. It takes the description of one machine plus its input and then produces output that is as (usually) expected. Finite state machines are just a different way of thinking about the computer and computation.

3.6.5 Examples of Sequential Circuits

Latches and flip-flops are used to implement more complex sequential circuit Registers, counters, memories, and shift registers all require the use of storag and are therefore implemented using sequential logic.

Our first example of a sequential circuit is a simple 4-bit register implemente using four D flip-flops. (To implement registers for larger words, we would need add flip-flops.) There are four input lines, four output lines, and a clock signal lin The clock is very important from a timing standpoint; the registers must all acce their new input values and change their storage elements at the same time. Remem ber that a synchronous sequential circuit cannot change state unless the clock pulse The same clock signal is tied into all four D flip-flops, so they change in unisor Figure 3.30 depicts the logic diagram for our 4-bit register, as well as a block di gram for the register. In reality, physical components have additional lines for pow and for ground, as well as a clear line (which gives the ability to reset the entire re ister to all zeros). However, in this text, we are willing to leave those concepts to th computer engineers and focus on the actual digital logic present in these circuits.

Another useful sequential circuit is a binary counter, which goes through predetermined sequence of states as the clock pulses. In a straight binary counte these states reflect the binary number sequence. If we begin counting in binary

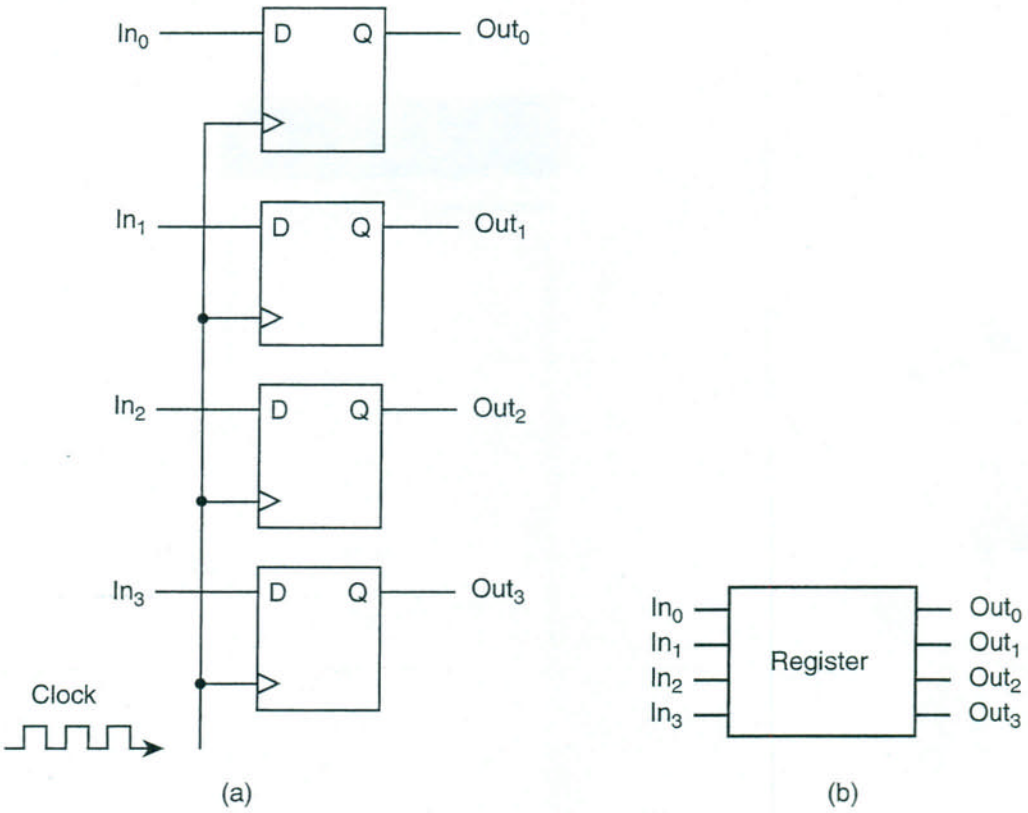


FIGURE 3.30 a) 4-Bit Register
b) Block Diagram for a 4-Bit Register

0000, 0001, 0010, 0011, . . . , we can see that as the numbers increase, the low-order bit is complemented each time. Whenever it changes state from 1 to 0, the bit to the left is then complemented. Each of the other bits changes state from 0 to 1 when all bits to the right are equal to 1. Because of this concept of complementing states, our binary counter is best implemented using a JK flip-flop (recall that when J and K are both equal to 1, the flip-flop complements the present state). Instead of independent inputs to each flip-flop, there is a **count enable line** that runs to each flip-flop. The circuit counts only when the clock pulses and this count enable line is set to 1. If count enable is set to 0 and the clock pulses, the circuit does not change state. Examine Figure 3.31 very carefully, tracing the circuit with various inputs to make sure you understand how this circuit outputs the binary numbers from 0000 to 1111. Also check to see which state the circuit enters if the current state is 1111 and the clock is pulsed.

We have looked at a simple register and a binary counter. We are now ready to examine a very simple memory circuit.

The memory depicted in Figure 3.32 holds four 3-bit words (this is typically denoted as a 4×3 memory). Each column in the circuit represents one 3-bit word. Notice that the flip-flops storing the bits for each word are synchronized via the clock signal, so a read or write operation always reads or writes a complete word. The inputs In_0, In_1, and In_2 are the lines used to store, or write, a 3-bit word

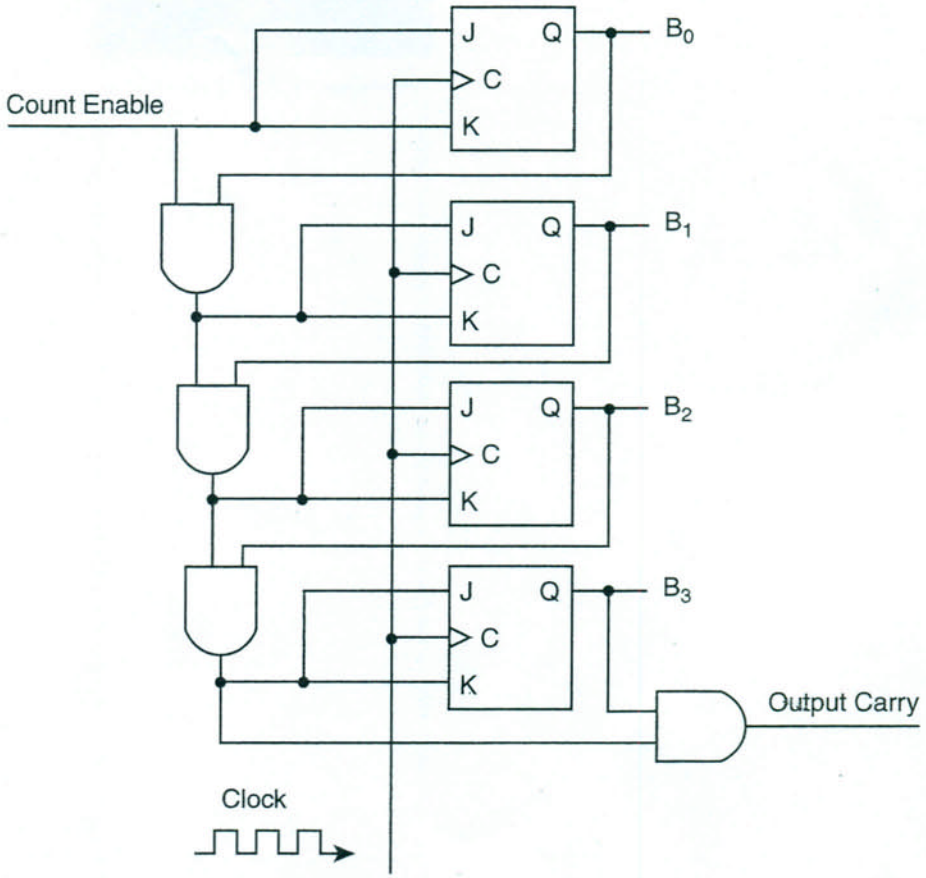


FIGURE 3.31 4-Bit Synchronous Counter Using JK Flip-Flops

to memory. The lines S_0 and S_1 are the address lines used to select which word in memory is being referenced. (Notice that S_0 and S_1 are the input lines to a 2-to-4 decoder that is responsible for selecting the correct memory word.) The three output lines (Out_0, Out_1, and Out_2) are used when reading words from memory.

You should notice another control line as well. The *write enable* control line indicates whether we are reading or writing. Note that in this chip, we have separated the input and output lines for ease of understanding. In practice, the input lines and output lines are the same lines.

To summarize our discussion of this memory circuit, here are the steps necessary to write a word to memory:

1. An address is asserted on S_0 and S_1.
2. WE is set to high.
3. The decoder using S_0 and S_1 enables only one AND gate, selecting a given word in memory.
4. The line selected in Step 3 combined with the clock and WE select only one word.
5. The write gate enabled in Step 4 drives the clock for the selected word.
6. When the clock pulses, the word on the input lines is loaded into the D flip-flops.

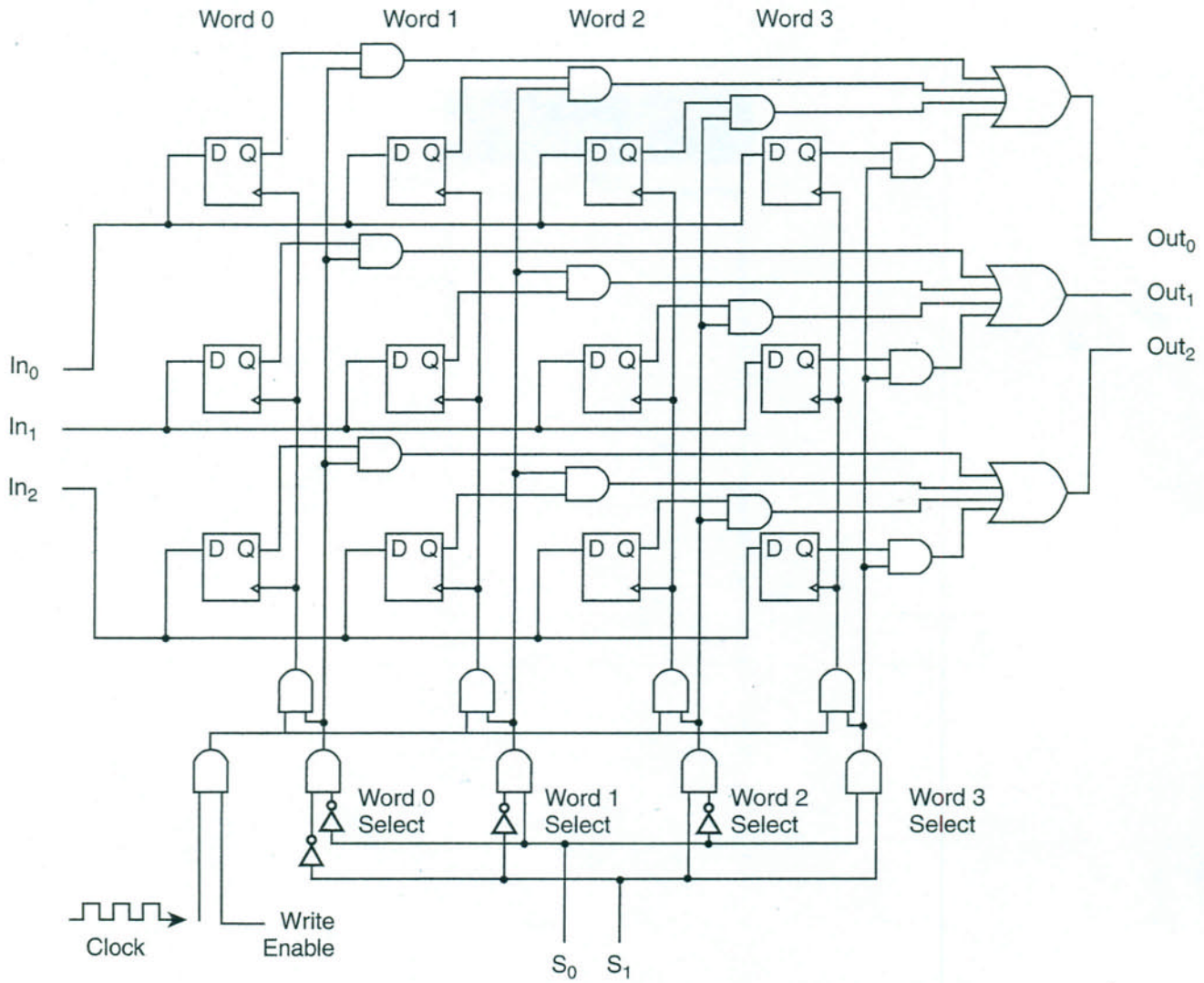


FIGURE 3.32 4 × 3 Memory

We leave it as an exercise to create a similar list of the steps necessary to read a word from this memory. Another interesting exercise is to analyze this circuit and determine what additional components would be necessary to extend the memory from, say, a 4 × 3 memory to an 8 × 3 memory or a 4 × 8 memory.

Logically Speaking, How'd They Do That?

In this chapter, we introduced logic gates. But exactly what goes on inside these gates to carry out the logic functions? How do these gates physically work? It's time to open the hood and take a peek at the internal composition of digital logic gates.

The implementation of the logic gates is accomplished using different types of logic devices belonging to different production technologies. These devices