Chapter 5: Introduction State Machines

Objectives

When you are finished with this chapter you will be able to:

- Describe the operation of State Machines
- Design a simple state machine
- Explain how a state machine can be used to control the instruction sequence of a simple microprocessor

The topic of state machines in computer systems deserves much more discussion than we can give it in this book. However, having said that, let's press forward anyway and try to gain some insight into the topic. Although you have probably never thought about this, there is a symmetry that exists between hardware design implementations and software design implementations. As software developers, you are already familiar with what an algorithm is. You've probably written many different algorithms in your other programming classes, and perhaps, you may have even studied the properties of algorithms in a separate class. In this lesson, we'll se how you might create an algorithm using dedicated hardware, rather than a set of C or C++ instructions.

It is a fact that an algorithm may be solved in hardware as readily as in software. For example, you might be a "gamer" and enjoy playing video games on your PC's. You know that to really get into the game, you need a potent video card, one capable of high-speed image creation and manipulation. If you don't have such a card, your game will still play, but it will be slower and less realistic than the game played with the high-performance card.

Another good example can be found in the current crop of low-cost laser printers entering the market. Prior generations of laser printers contained high-performance hardware to convert the input data stream into a pattern of laser imprints on a photosensitive drum. The process of data conversion is called ***rasterization***, and is similar to the method used to create an image on a CRT screen. Rasterization was handled in dedicated hardware by the laser printer, but as the computing power of PC's became more powerful it made economic sense to move the rasterization back to the PC as part of the printer's driver code. The printer then simply transferred the raster data coming in into the pulses of laser light on the photosensitized drum of the printer. Of course, you could only use the printer with the appropriate software driver, and there was no way that the printer could be used as a stand-alone device.

In general, an algorithm solved in hardware is much faster, sometimes many orders of magnitude faster, than the same algorithm solved in software. Today, we have evolved a method of creating ***Application Specific Integrated Circuits***, or ***ASIC's***, as a way to create a hardware implementation of an algorithm. The video processor in your PC, the sound chip and the modem chip are all examples of ASIC's in your PC, but they are even more prevalent throughout industry. In many of today's industrial and consumer applications the ASICs handle the data processing and data manipulation, and the microprocessor handles the error processing and initialization.

In the previous chapter we were introduced to the concept of the state of a system. State-based systems are a natural consequence of the introduction of the flip-flop. That is, a device whose outputs are not only dependent upon its inputs, but upon the current state of its outputs as well. Also, we introduced the concept of systems that can only change state when a synchronizing clock signal is present.

What we are leading to is that there is a class of digital system behaviors that are fundamentally sequential in their behavior. We'll soon look at how a computer steps its way through a finite sequence of steps as it fetches an instruction from memory, decodes the instruction, executes the instruction, writes back to memory any results and begins the sequence over again. This is a perfect example of such a sequential process.

How the computer is led through this sequence of steps constitutes the operation of its state machine engine. That engine can be controlled by data stored in memory, as we'll see in an example in this chapter, or it can be controlled by combinatorial logic, as we've previously learned to design. We'll also do an example of this type of design as well. Any sequential process that we can conceptually describe and implement as a synchronous digital system is called a ***finite state machine***[1]**.** We can define a sequential (finite state) machine as a model that conforms to the following set of requirements:

- It can exist in a finite set S of states s
- There is an initial state s_0 that is a member of set S
- There is a finite set I, of inputs i
- A next state function $d = d(s,i)$ that maps the present state values and the inputs into the next state in S
- An output function, $f = f(s,i)$

Does the above formalism describe what we intuitively already know? Figure 4.20 shows the state transition diagram for a 4-bit binary counter. Does it conform to our definition of a finite state machine (FSM)?

- It can only exist in a set of 16 states, labeled 0000 through 1111. These 16 states define the set, S
- There is an initial state, 0000, that is part of the set and it can be reached through initialization of the device, or as part of the sequence of steps
- There are no inputs in this example, but that does not diminish the correctness of the analysis
- The next state function is dependent upon the present state
- The output of the counter is dependent upon the current state of the counter

If f is a function only of its present state $f = f(s)$, as is the case with the 4-bit binary counter, we call that a ***Moore machine***. If f is a function of both its current state and its inputs, $f = f(s,i)$, then it is called a ***Mealy machine.*** If the machine is in state s and receives the input i, the next state that it will go to is determined by $d = d(s,i)$

We can take the principles of FSMs and apply them to the solution of process problems, such as the video card or modem examples. When we use these methods we are describing an ***algorithmic state machine***. That is, we are applying the methods and hardware implementation techniques of FSM design to the solution of an algorithm.

While this all seems highly structured and mathematical (and it is) perhaps we can gain some insight into the process by implementing a FSM for a simple sequential process.

Figure 5.1 is a state diagram for an arbitrary hardware algorithm. After a reset pulse, the system is initialized to state 000. The blue arrows show the state transitions that will occur when the input variable X = 1 and the red arrows show the state transitions that will occur when X = 0. Thus, if the system is in state 011 and X = 0, then the next state will be 100. However, if X = 1 when the clock arrives then the next state would be 010.

Figure 5.2 is the implementation method for the algorithm. The outputs of the circuit are state variables Aout, Bout and Cout, respectively. Note that state 000 means Aout = 0 , Bout = 0 and Cout = 0. These variables also feed back to the truth table to become the input variables for the next state transition that will occur on the next clock. In this example we will transfer the requirements of the state diagram to the truth table and then create a hardware implementation in combinatorial logic for the truth table. When the combinatorial logic and the D-FF sequencer are combined, we should have a FSM that replicates our state diagram. Therefore, we will first complete the truth table, and then simplify it using K-Map techniques. Finally, we will implement the logical equations in hardware.

Figure 5.3 is the completed truth table. The first question to ask is, “Does this make sense?” Let’s see. According to figure 5.1 when the system is in its initial state, just after ~RESET, it can go to either state 101 if X = 1 or to state 011 if X = 0. Referring to the truth table in figure 5.3, we see that when Ain, Bin, Cin =000 and X = 0, Anext, Bnext ,Cnext = 011 and when X = 1, Anext, Bnext ,Cnext = 101, which does, indeed, agree with the state diagram.

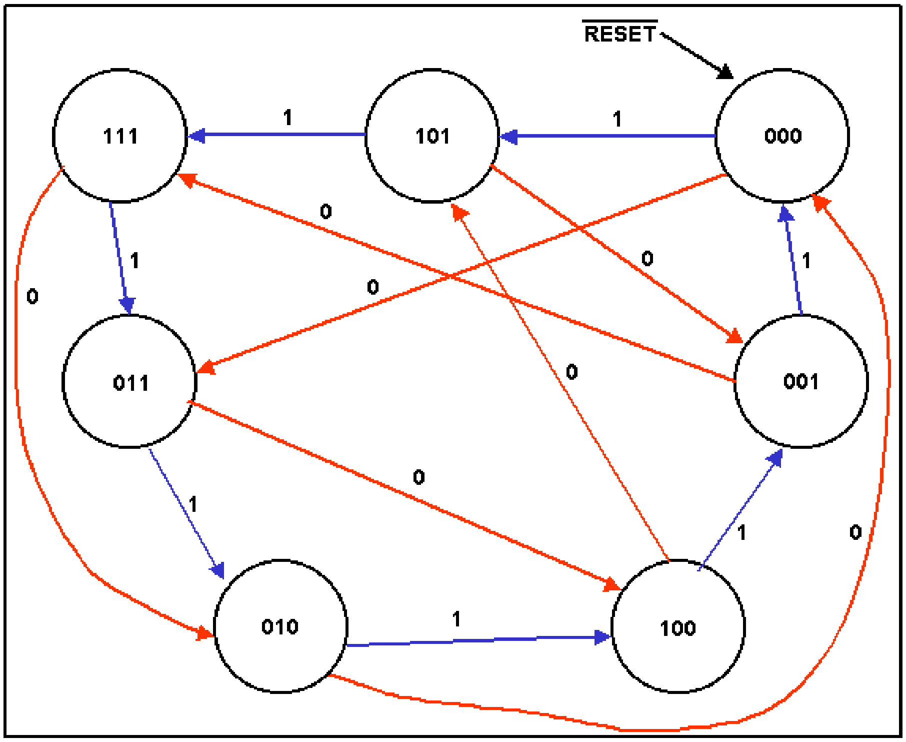


Figure 5.1: State diagram for an arbitrary sequential process.

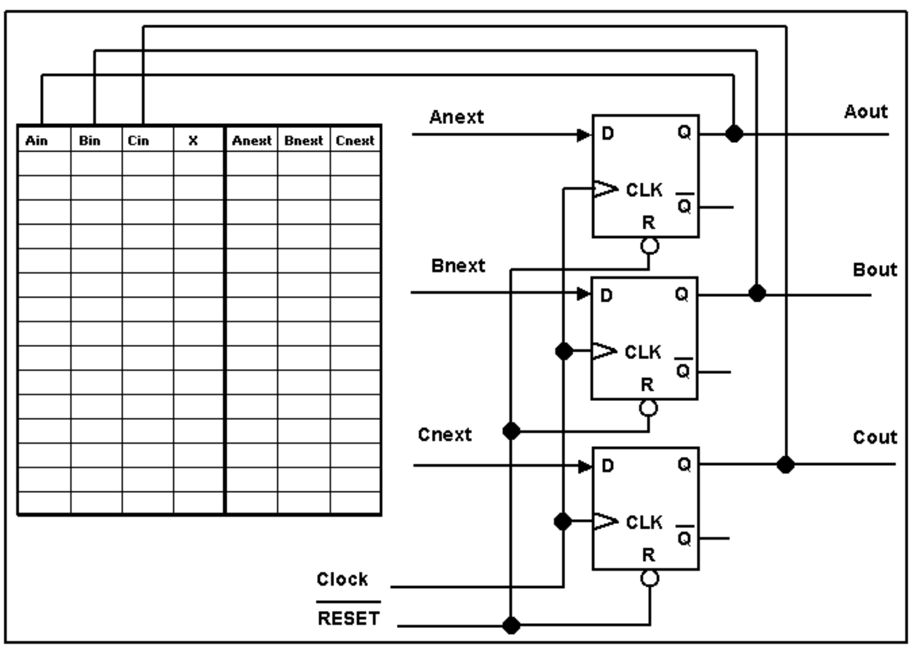


Figure 5.2: Implementation method for the sequential process of figure 5.1.
The truth table will be converted to combinatorial logic

One important point to note is that there is one state, 110, which is not part of the sequence process. In the truth table of figure 5.3, we see this state represented as XXX. This is a shorthand way of saying that this state will never be reached (if the circuit is operating properly). What we can do is reserve the right to replace the X with either a 1 or 0 if it represents a possible simplification of the K-MAP. We can do this because, as you can see from the state transition diagram in Figure 5.1, no other state leads us to the 110 state. Before we begin working through the logic of the simplification process we should stop and recall that the truth table is, in effect, a map of the contents of a memory device that could provide us with a circuit design for the sequential process. Figure 5.4 shows this alternative view of the logical design. Here, the inputs to the truth table, Ain, Bin, Cin, and X, become the 4 address inputs to the memory. The data out bits, DATA0 through DATA2, correspond to the output variables of the truth table, or the next state for the process.

Ain	Bin	Cin	X	Anext	Bnext	Cnext
0	0	0	0	0	1	1
1	0	0	0	1	0	1
0	1	0	0	0	0	0
1	1	0	0	X	X	X
0	0	1	0	1	1	1
1	0	1	0	0	0	1
0	1	1	0	1	0	0
1	1	1	0	0	1	0
0	0	0	1	1	0	1
1	0	0	1	0	0	1
0	1	0	1	1	0	0
1	1	0	1	X	X	X
0	0	1	1	0	0	0
1	0	1	1	1	1	1
0	1	1	1	0	1	0
1	1	1	1	0	1	1

Figure 5.3 Truth table for the sequential process example of figure 5.1

The K-Maps for the state variables, and the simplified equations are shown in figure 5.5 In figure 5.5 the simplified equations for Anext includes an XOR term. This was added to further simplify the term A*~X + ~A*X. It is fair to ask if any further simplifications could be made. Clearly, using the laws of Boolean algebra, there could be additional regroupings of some terms. However, from a gate complexity point of view, grouping terms may actually increase complexity because it could introduce an additional AND gate into the hardware implementation. Thus, it isn't always clear that the best algebraic solution leads us to the minimal hardware solution. There may be additional factors, such as available gates with the correct number of inputs, extra gates in the package needed elsewhere in the circuit, etc.

The hardware implementation is shown in figure 5.6. Note how the ~Q outputs of the 'D' flip-flops were used as inverter gates for the truth table. Rather than simply using NOT gates to create the complements of Anext, Bnext, Cnext and X, the ~Q outputs were convenient sources of the complemented signals.

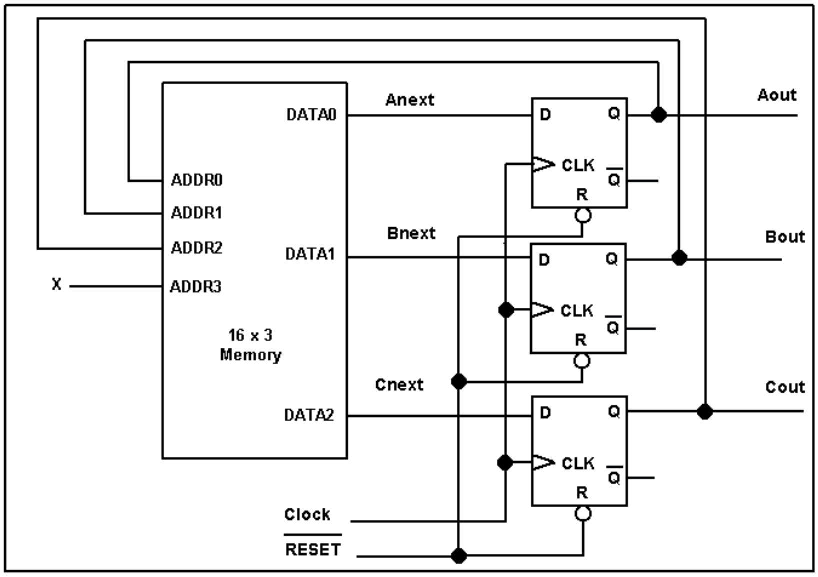


Figure 5.4 A memory-based solution to the sequential process of a state machine.

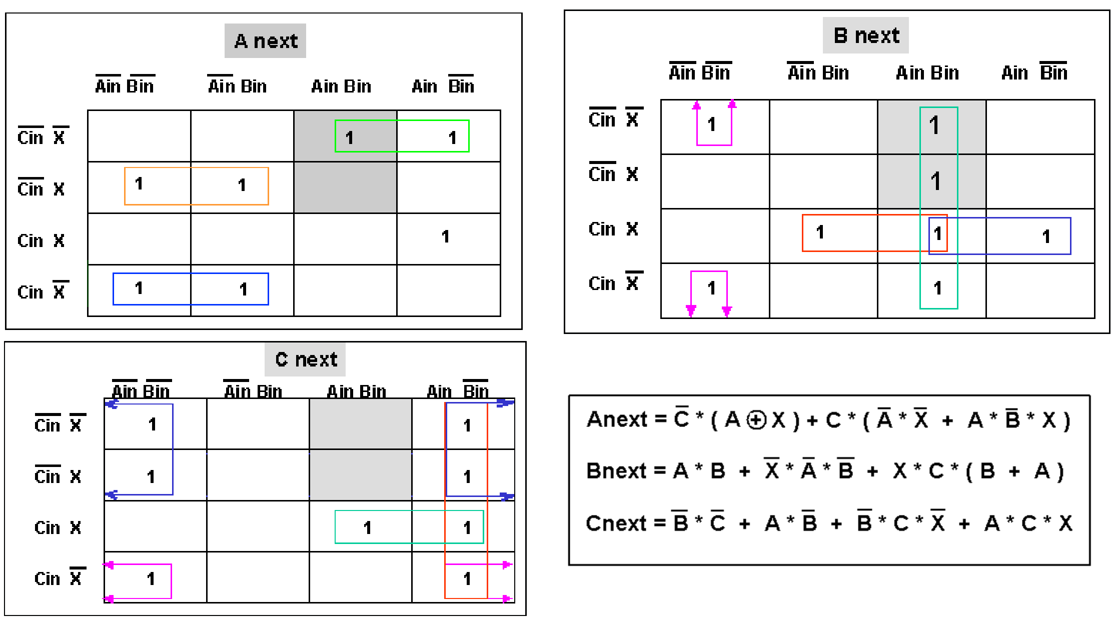


Figure 5.5: K-Maps and simplified equations for variables Anext, Bnext and Cnext

The circuit of figure 5.6 meets the requirements of a finite state machine. That is:

- There is a set S of seven states that we sequence through in series, the transitions between the states occurs on the rising edge of the clock input to the D-FF's.
- We can establish an initial state, 000, with the RESET pulse, and the initial state is part of the set, S, of states.
- There is an input to the system, X, and three outputs, Aout, Bout and Cout.
- The next state function, ***d = d(s,i)*** and the output function ***f = f(s,i)***, are the same. In a later example we'll see how these can be different functions entirely.

Obviously, this example was completely made up and doesn't really represent a real-world digital design (although I'm sure you could buy some parts at a surplus electronics store and, with some simple instructions, build the circuit and watch it sequence through the states ***ad nauseum***).

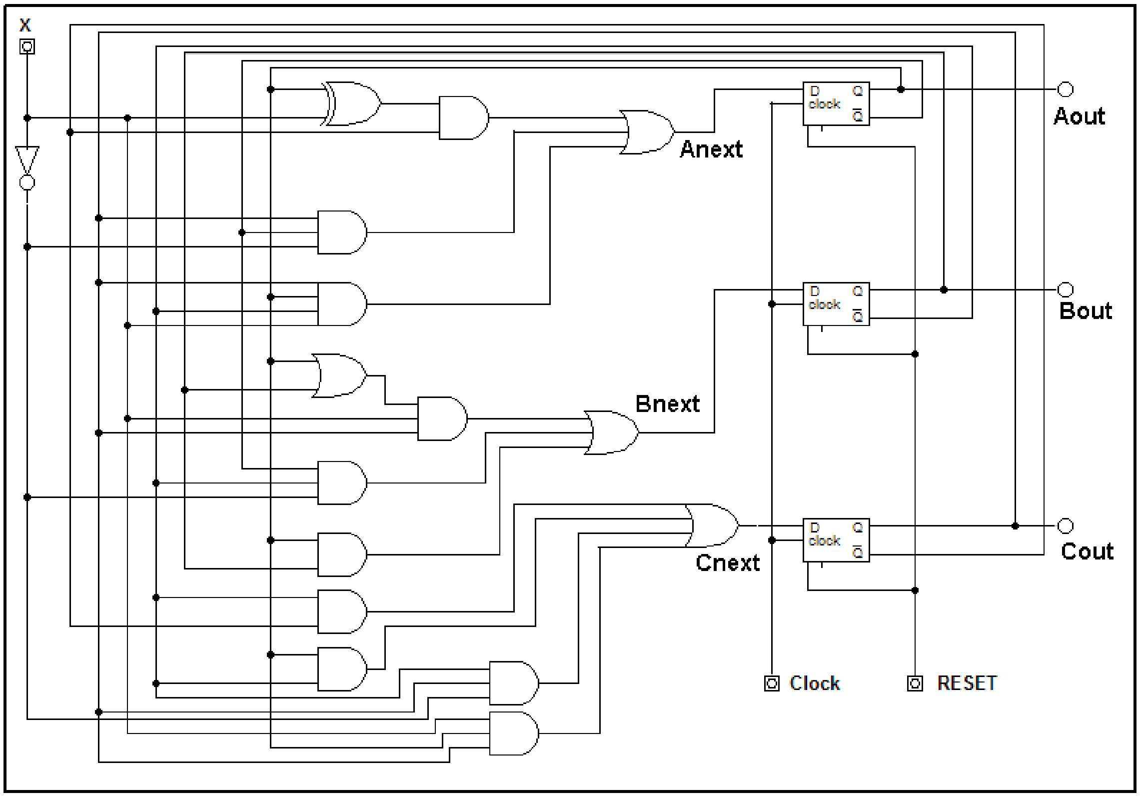


Figure 5.6: Hardware implementation of the sequence process from figure 5.1

This is fun! Let's do another example. This time we'll do one that might be actually useful. Let's consider that we have a serial bit stream coming into our system. Suppose that we need to detect every time 3 or more successive 1 bits are received. Offhand, I don't know why you would want to do this, but given some time, I'm sure we could figure out a circuit need for such a circuit. Figure 5.7 is the state diagram for this circuit.

Here we have 3 states, 00, 10, and 01, respectively. The arrows represent the state transitions on the clock edge. Associated with each arrow are two numbers separated by a forward slash. The first number is the state of the input bit at the time of the rising edge of the clock and the second number is the output signal, T, which goes TRUE if three or more successive 1's are detected in the bit stream.

In this case, the clock signal that we would use is likely be the clock that the system is using to synchronize the data transmission with the data bit stream. Thus, a new data bit would appear on every rising edge of the clock.

Every time that a 0 bit appears, the circuit returns to the 00 state waiting for the arrival of a 1 bit. With the arrival of the first 1 bit, the circuit transitions to state 10. If a second 1 bit arrives it transitions to state 01. If the second bit to arrive is a 0 bit, the circuit returns to state 00. However, if a third 1 bit arrives, the circuit asserts output T and returns to state 01 with the T bit set to 1. As soon as the next 0 bit arrives, the circuit returns to state 00. What is new in this example is the addition of an explicit output bit, T.

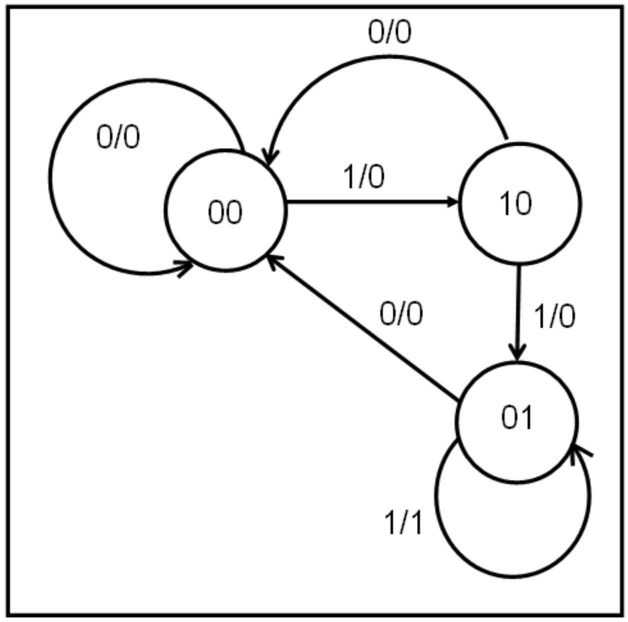


Figure 5.7 State transition diagram for a circuit that detects 3 or more successive 1's in a serial bit stream.

You might be wondering why we don't have a fourth state, 11, which represents the arrival of 3 successive 1's. This would also work, but it would be redundant, since the arrival of the third successive 1 bit also means that there has been two successive 1's as well. Thus, we may remain in state 01 as long as 1's are being received. We could, however, change the problem specification so that each group of three 1's is a unique event. This would imply that a fourth state might be needed, depending upon what we intend to do after each group of three is received.

Figure 5.8 shows the truth table, K-map and simplified logical equations for the circuit.

A	B	X	Anext	Bnext	T
0	0	0	0	0	0
0	0	1	1	0	0
1	0	0	0	0	0
1	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	1

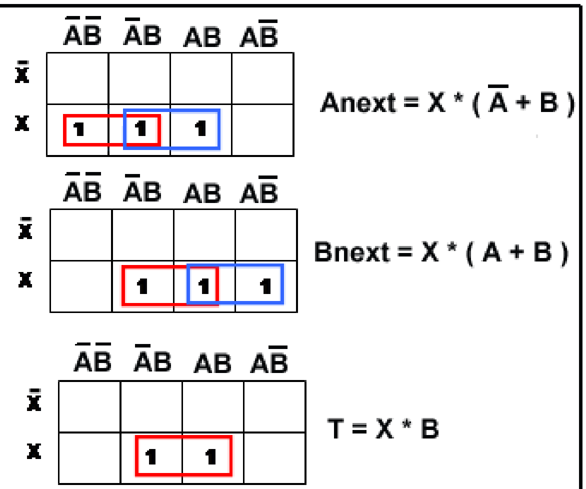


Figure 5.8 Truth table, Karnaugh Maps and simplified logical equations for the state diagram

of figure 5.7

The circuit diagram is shown in figure 5.9. This problem is also an example of a FSM where the output bit, T, is a separate function of the input, X, and the combinatorial logic of the problem.

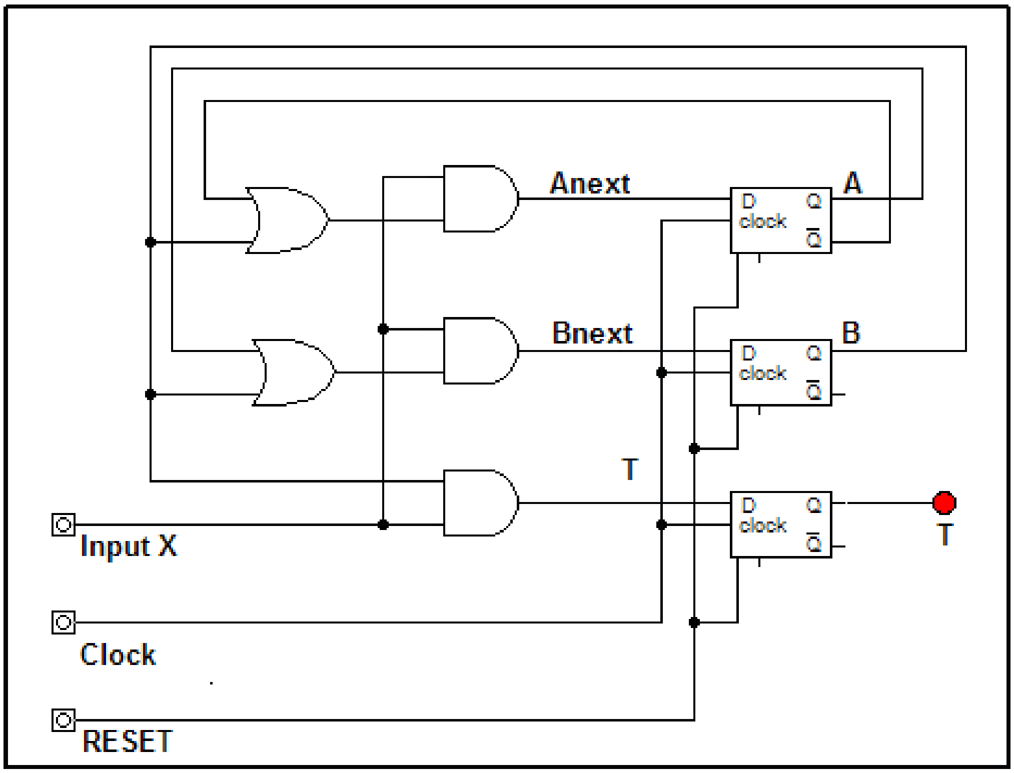


Figure 5.9 Hardware implementation of the state transition diagram of figure 5.7

Another interesting observation that we could make is that this FSM is actually implementing an algorithm. It probably wouldn't take you very long to write an algorithm in C, C++ or Java that would mimic this hardware circuit in software. In fact, we have implemented an ***algorithmic state machine,*** or ***ASM.***

Algorithmic State Machine Example: A traffic intersection controller

Let's walk through the gory details (at least once) of a real problem that might be typical of a range of problems that could be solved using Algorithmic State Machines[2,3]. In this case we'll consider the problem of designing the control circuitry for the traffic lights in a typical traffic intersection. Figure 5.10 shows the intersection. Here:

- W = Signal light for westbound traffic
- E = Signal light for eastbound traffic
- N = Signal light for northbound traffic
- S = Signal light for southbound traffic

Since the road for the N/S traffic is four lanes wide and the road for the E/W traffic is only two lanes wide, we can reasonably assume that the volume of traffic on the N/S road is considerably higher than the traffic volume on the E/W road. Therefore, we don't want to stop the N/S traffic at this intersection unless there is waiting E/W traffic. For example, how many times have you sat in your car at a red light at an intersection with perhaps 20 or 30 other cars, waiting for the light to change and there is no cross traffic in the intersection? But I digress….

Here is a pseudo-code specification for our algorithm:

```
While (1)
{
 timer = 20 seconds;
 NS = green;
 EW = red;
 while ( timer != 0)
   wait;
 if (EW sensor != 0 )
 {
    timer = 5 seconds;
    NS = yellow;
    while ( timer != 0)
      wait;
    timer = 20 seconds;
    NS = red ;
    EW = green;
    while ( timer != 0)
      wait;
    timer = 5 seconds;
    EW = yellow;
    while ( timer != 0)
      wait;
  }
}
```

Notice that there is a new wrinkle in this problem. The times are different. We are nominally in the NS state for 20 seconds, and then, if there is an EW car waiting, we have to transition for 5 seconds through a yellow light before turning on EW green for 20 seconds.

The statement, while(1), is often used in embedded systems to identify the portion of the program code that will run forever. Once the system, such as a printer, is initialized, the printer runs forever (or until you turn off the power).

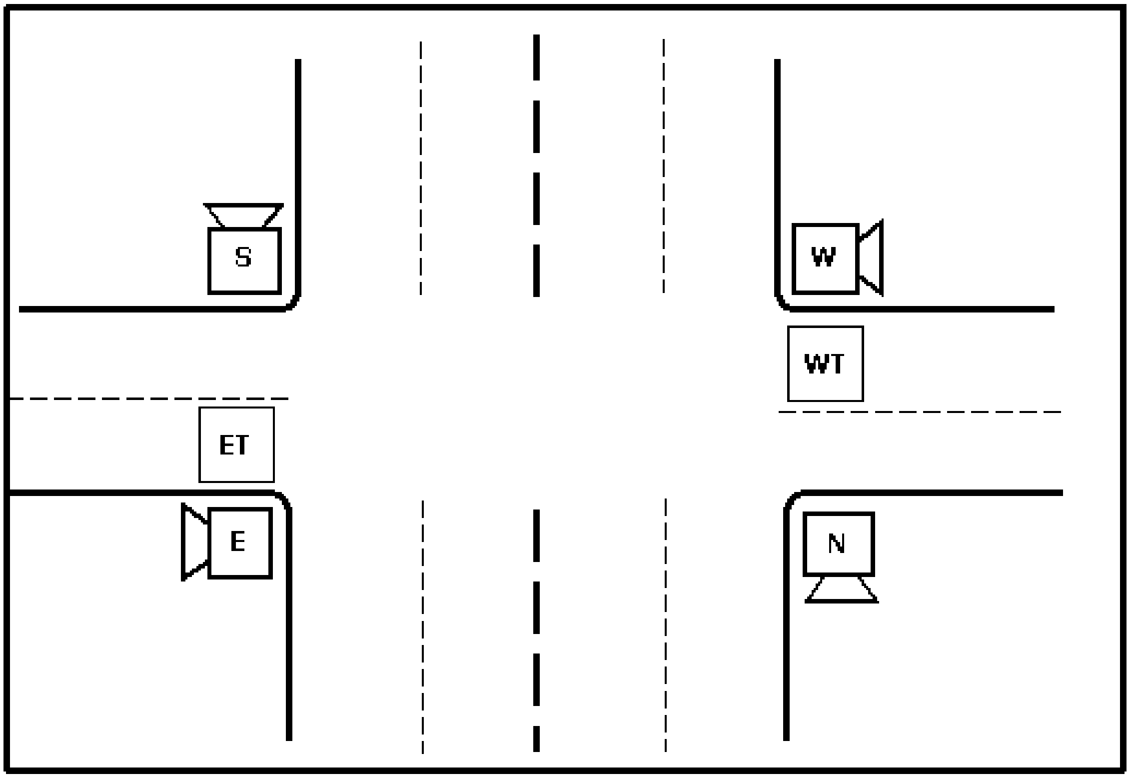


Figure 5.10: Traffic intersection with a four-lane NS roadway and a two-lane EW roadway. The EW roadway has sensors to detect waiting traffic.

Let's look at the as a state transition diagram, shown in figure 5.11.The state of the EW traffic sensor is an input to our algorithm and it can modify the behavior of the system. We can represent this condition by using the fact that there are two possible paths for the state machine to take once it is in the state NS GRN/EW RED. As long as neither of the EW sensors detect a waiting car, the traffic control algorithm will remain in this state. However, if a waiting car is detected at the end of the time interval for NS GRN, then the state machine transitions to the state NS YEL/EW RED. Once in this state, the algorithm must sequence through NS RED/EW GRN and NS RED/EW YEL before it re-enters the state NS GRN/EW RED. There are no possible inputs in these other states that can change the sequence.

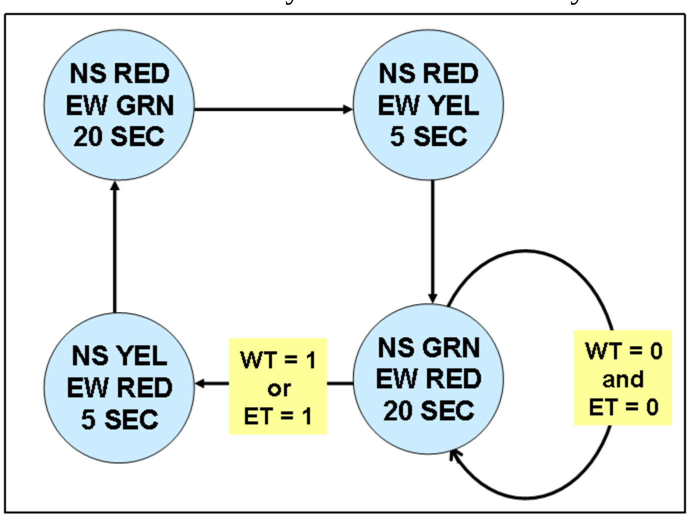


Figure 5.11: State transition diagram for the traffic intersection

Again, assuming we could take care of the time requirements, you could easily change the pseudo code of this algorithm to a real program in C or C++. Of course, we'd need to supply an I/O port to drive the lights and a timer mechanism, but these are simple implementation details. Let's take a brief diversion from our algorithmic state machine design and introduce a new concept (just when you thought it was safe to go back into the water, bang!). Figure 5.12 shows us the state of the outputs represented as a timing diagram. This timing diagram shows us the state of the input signals that control the lights versus time. Notice in figure 5.12 that our horizontal axis is calibrated in 5-second ticks. This just shows that the outputs can only change on the 5-second clock ticks.

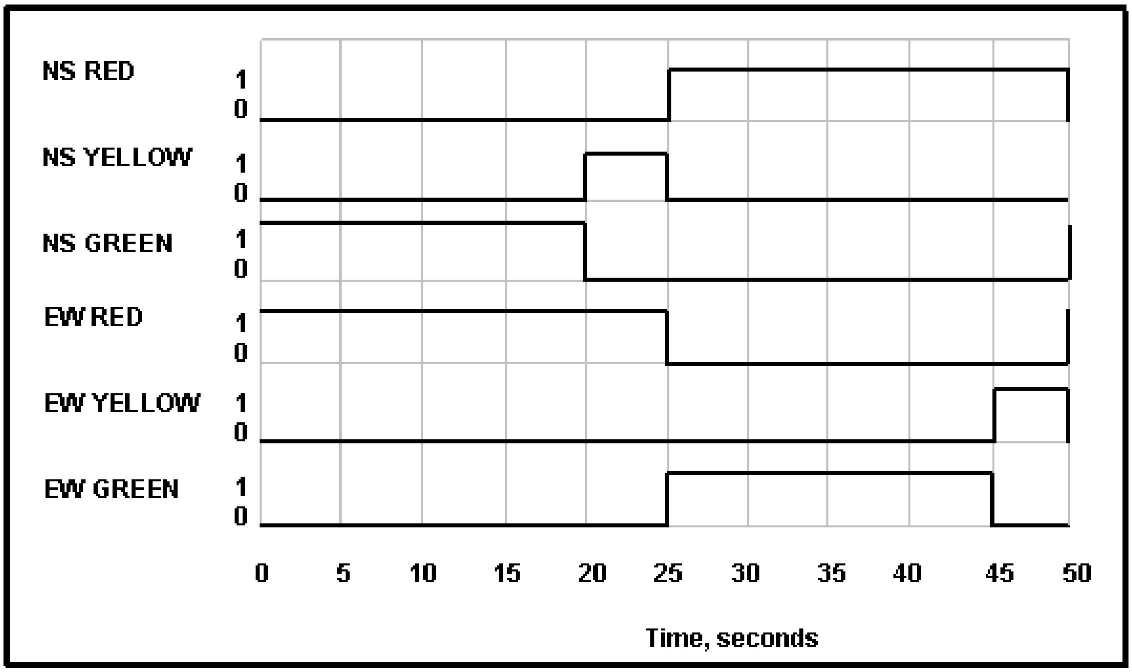


Figure 5.12: Traffic light sequence represented as a timing diagram.

The timing diagram of figure 5.12 represents an alternative view of the system. In fact, this is what the controller outputs to the traffic light power circuits would look like on the screen of a logic analyzer. This kind of view is significant because it introduces us to another important concept; the idea of the states of the system represented as a set of ***state vectors***.

The state vectors are just the combination of all of the possible states of the inputs and outputs that our system might be in. Thus, if you create a table of all of the vectors, then you have an alternative way of representing all of the possible states of the system. Figure 5.13 is a representation of the timing diagram of figure 5.12, but now represented as a state vector set.

Time	EW GREEN	EW YELLOW	EW RED	NS GREEN	NS YELLOW	NS RED	Hexadecimal State
0	1	0	0	0	0	1	21
5	1	0	0	0	0	1	21
10	1	0	0	0	0	1	21
15	1	0	0	0	0	1	21
20	0	1	0	0	0	1	11
25	0	0	1	1	0	0	0C
30	0	0	1	1	0	0	0C
35	0	0	1	1	0	0	0C
40	0	0	1	1	0	0	0C
45	0	0	1	0	1	0	0A
50	1	0	0	0	0	1	21

Figure 5.13 Timing diagram represented as a set of state vectors. The hexadecimal state representation on the right is a shorthand way of collecting the individual bit values of the six light signals and aggregating them into a binary, or hexadecimal number that represents the state. Thus, these binary or hexadecimal state "numbers" have no numerical significance but are simply a shorthand way of collecting together and representing six Boolean variables.

At each clock transition the state of the system can be described by the binary vector set or by the hexadecimal vector set. We have to be a bit careful here because each of the hexadecimal states shown in Figure 5.13 is not a number. Rather, each state is a collection of individual bits that are represented as an ensemble of bits that together form a state vector.

The hexadecimal representation is convenient, as you'll see, for creating the actual memory image for the traffic controller, but the more accurate and informative representation would be to leave the description in terms of binary vectors. Unfortunately, it's just too convenient not to use hex.

Let's now begin the actual process of designing the controller. As a first pass through, we'll ignore the E/W traffic sensors and just consider a simpler model, one with each direction getting 20 seconds of time with a 5-second yellow. Better to focus on the issues. Figure 5.14 shows the simplified algorithm converted to state variables. Also, we've redrawn it so that every state lasts 5 seconds in duration. By increasing the number of states, even though several of the states "seem to" represent the same condition, we can convert the problem to one of equal time intervals. Actually, the four states that represent the 20-second time intervals are not the same. They just have the same output function.

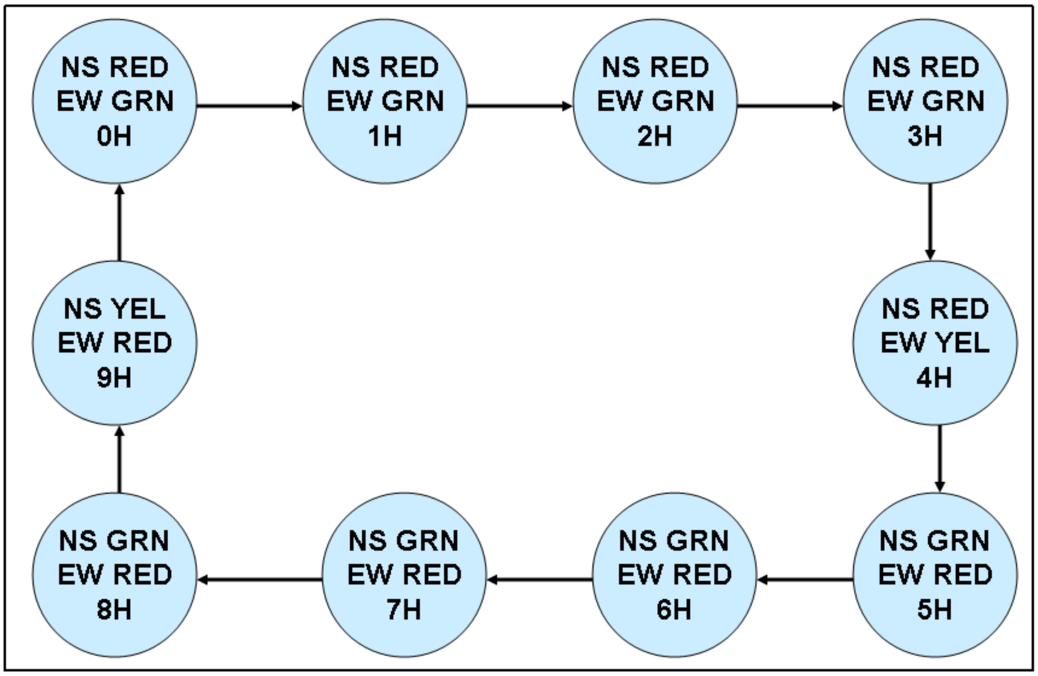


Figure 5.14 Algorithm for traffic intersection represented as a set of state variables. Each state is numbered in sequence and the period is 5 seconds long.

Now we've given each state a unique numerical identifier, numbering them from 0000 to 1001, or 0H to 9H. This may look suspiciously like a memory address to you, but my lips are sealed. I won't reveal the good part until the end. Let's specify the outputs and the binary weight of each bit. This is only for the sake of convenience, but we'll press on.

Consider the 6 outputs for the state 0000 (0 Hex).

- `NS red is on:        NSR or Data bit 0 (D0) = 1`
- `NS yellow is off:    NSY or Data bit 1 (D1) = 0`
- `NS green is off:     NSG or Data bit 2 (D2) = 0`
- `EW red is off:       EWR or Data bit 3 (D3) = 0`
- `EW yellow is off:    EWY or Data bit 4 (D4) = 0`
- `EW green is on:      EWG or Data bit 5 (D5) = 1`

Thus, the state variable for state 0000 is 100001, or state 0 hex = 21 hex. We'll also use the collective term ***data bits*** to represent the ensemble of state variables and it will be convenient to give the individual bits that control the lights a collective identity so that we may deal with them as a group, rather than as individual entities. However, we should always keep in mind that each output variable is independent of the others. Thus, for example, adding a number such as A3 to the hexadecimal state value, 21, would not have any logical significance. That said, the next step is to convert our specification to a truth table.

Remember the 32-bit memory array that we looked at earlier? Let's see it again, but this time we'll expand the number of outputs to 6 in order to account for all 6 outputs for the traffic signals. Thus, we need a memory with a total of 96 memory cells, arranged as 16 by 6. Figure 5.15 shows the memory (truth table) layout for this project.

Our clock generator is somewhere else in this system. Compared to the clock frequencies that we've been discussing so far, 0.2 Hz (one cycle every 5 seconds) is positively glacial, but that's all the speed that we'll need for this example. As an aside, let's consider the possibility of increasing the clock frequency to 1 Hz. This would have the apparent effect of increasing the number of states in the system, but there are other ways we might be able to redesign the system to handle that. However, one advantage of increasing the number of states is that we lower the response time of the system.

Suppose that we wanted to add an emergency vehicle detector to our intersection. That's a photosensitive device that is designed to detect the strobe lights on the tops of emergency vehicles. If a police car was rushing towards this intersection at 100 kilometers per hour (~28 meters per second) then the police car could travel as much as 139 meters (with a 5-second clock period) before all the lights in the intersection turned red. Clearly there is room for improvements in our traffic intersection algorithm.

The output value from the D register is the address of the 6 memory cells that determine the data for the next state of the system after the rising edge of the clock pulse arrives. We can summarize the data by actually building the ***state table***. This is just a convenient way of bringing all of the pieces together. Keep in mind that we still need to add a mechanism for sequencing through the various states in the system.

Figure 5.16 is the state table for our preliminary design. We call it preliminary because we still need to add the sequencing mechanism. That will come shortly. Referring to figure 5.16, you can see that we've let the cat out of the bag. The numerical state identifiers, 0000 through 1001 are, indeed, the address in the READ ONLY MEMORY ***ROM*** that will become the truth table for our design. You're probably already familiar with the term "ROM" from your PC's BIOS ROM. The ROM is just a memory device that can be pre-programmed so that its data values are retained even if power is turned off. This is just what we need for our traffic controller design.

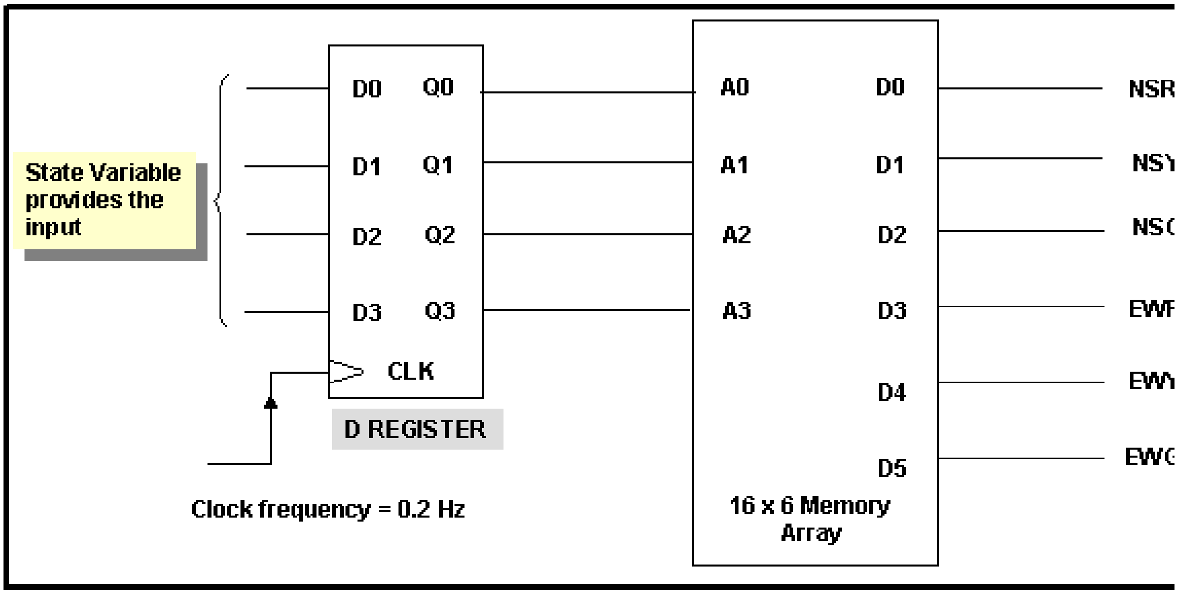


Figure 5.15 The state machine with the storage register added to provide the sequencing mechanism. This circuit still lacks the mechanism for sequencing through the various states in the system and for changing the algorithm if an E/W car is waiting,

State					Outputs						
Q3	Q2	Q1	Q0	ROM Address	D5	D4	D3	D2	D1	D0	ROM Contents
0	0	0	0	0H	1	0	0	0	0	1	21H
0	0	0	1	1H	1	0	0	0	0	1	21H
0	0	1	0	2H	1	0	0	0	0	1	21H
0	0	1	1	3H	1	0	0	0	0	1	21H
0	1	0	0	4H	0	1	0	0	0	1	11H
0	1	0	1	5H	0	0	1	1	0	0	0CH
0	1	1	0	6H	0	0	1	1	0	0	0CH
0	1	1	1	7H	0	0	1	1	0	0	0CH
1	0	0	0	8H	0	0	1	1	0	0	0CH
1	0	0	1	9H	0	0	1	0	1	0	0AH
1	0	1	0	AH	X	X	X	X	X	X	Don't Care
1	0	1	1	BH	X	X	X	X	X	X	Don't Care
1	1	0	0	CH	X	X	X	X	X	X	Don't Care
1	1	0	1	DH	X	X	X	X	X	X	Don't Care
1	1	1	0	EH	X	X	X	X	X	X	Don't Care
1	1	1	1	FH	X	X	X	X	X	X	Don't Care

Figure 5.16 State table for the traffic controller showing the state variables (ROM addresses) and the data contained in each ROM address. The values identified as "X" are referred to as "Don't Cares" because the state machine will never go to these states.

Notice that our ROM actually has memory cells without any data in them. These are the ***Don't Care*** conditions at address 0A hex through 0F hex. The reason for this is that our design only requires ten valid states (00 hex through 09 hex). Even though our four-variable truth table allows for sixteen states, we won't need them as long as our state machine sequence never takes us to them.

Adding the state sequencing mechanism is quite straight-forward. All we need to do is increase the number of data outputs to our state table that we will feed back to the D storage register. Thus, we've generalized the concept of the state machine to show that it isn't necessary to feed all of the outputs back to the inputs. We only need to bring

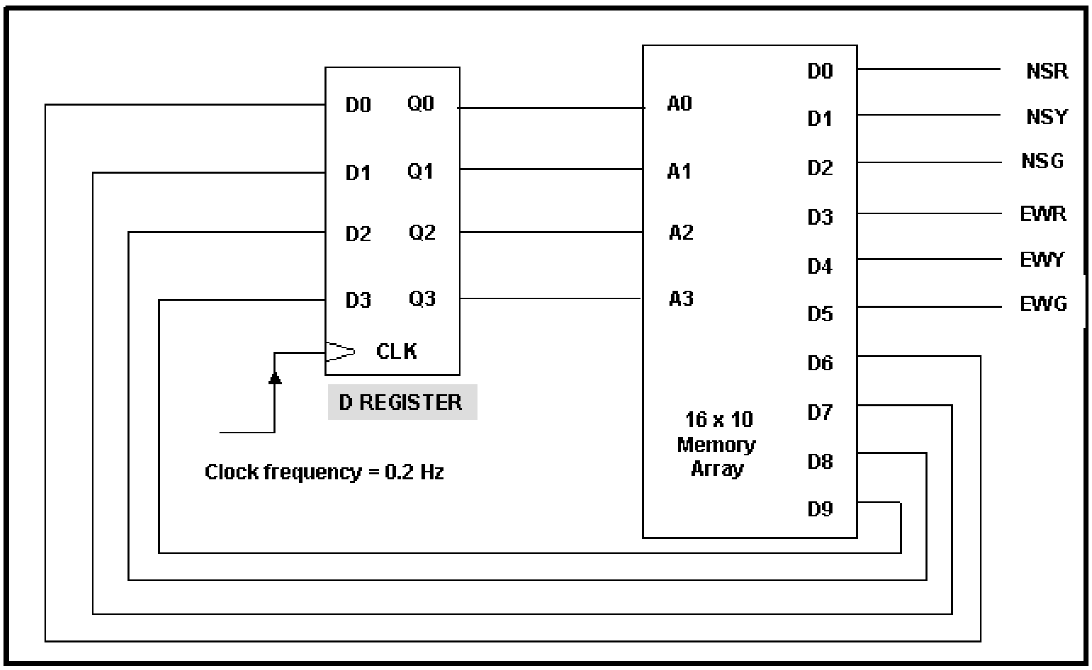


Figure 5.17 State machine with the addition of extra data bits in the ROM in order to provide a mechanism for sequencing through the states.

enough outputs back to the inputs to provide a mechanism to sequence the state machine through its different states. We call these additional outputs the ***excitation outputs*** because their function is to provide the sequencing (excitation) mechanism for the state machine. Figure 5.17 shows the new design. Notice that outputs D0 through D5 are the variables that directly control the state of the traffic lights and four outputs, D6 through D9, are the excitation outputs.

Now, it will be necessary to expand the number of output bits in the ROM to provide the feedback to the D register for the sequencing system. Our ROM still has the same number of memory addresses but we've added 4 more output bits. Thus, the data in each ROM address is now 10 binary digits. Figure 5.18 shows the expanded state table.

Finally, we are now in a position to go back and include the extra inputs for the EW traffic that we've neglected so far. Let's refresh our memory (it’s been a long session) by reviewing the algorithm that we’ve previously developed. Figure 5.19 illustrates the state diagram that we developed, modified so that we may utilize the same 5 second clock period for each state.

Current State ROM Address	Next State				Outputs						ROM Contents
	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	
0H	0	0	0	1	1	0	0	0	0	1	061H
1H	0	0	1	0	1	0	0	0	0	1	0A1H
2H	0	0	1	1	1	0	0	0	0	1	0E1H
3H	0	1	0	0	1	0	0	0	0	1	121H
4H	0	1	0	1	0	1	0	0	0	1	151H
5H	0	1	1	0	0	0	1	1	0	0	18CH
6H	0	1	1	1	0	0	1	1	0	0	1CCH
7H	1	0	0	0	0	0	1	1	0	0	20CH
8H	1	0	0	1	0	0	1	1	0	0	24CH
9H	0	0	0	0	0	0	1	0	1	0	00AH
AH	X	X	X	X	X	X	X	X	X	X	Don't Care
BH	X	X	X	X	X	X	X	X	X	X	Don't Care
CH	X	X	X	X	X	X	X	X	X	X	Don't Care
DH	X	X	X	X	X	X	X	X	X	X	Don't Care
EH	X	X	X	X	X	X	X	X	X	X	Don't Care
FH	X	X	X	X	X	X	X	X	X	X	Don't Care

Figure 5.18 Modified state table now includes extra data bits for sequencing

This new addition leads us to the state machine design in Figure 5.20. The addition of these two independent inputs, ET and WT, which represent the in-the-road sensors for waiting eastbound and waiting westbound traffic, respectively, is extremely significant. The conditions of the independent inputs can affect the sequencing of the state machine. Thus, we've added a completely new dimension to our Algorithmic State Machine: the ability to change its state flow in response to a change in its input data. Do you recognize this new feature? You should. It's called a computer.

With the addition of independent inputs that are able to define new states for the state machine, our state machine is beginning to look like a decision making device. Of course, it isn't really making a decision because we've pre-programmed into it exactly what it can do in response to the new data. The state machine in your computer is several thousand times more complex than this simple one, but you're beginning to get the picture. We call the state transition table (ROM) for the microprocessor the ***microcode.*** It is the microcode that gives the computer its personality. The computer's ***instruction set architecture***, or ISA is defined by the microcode.

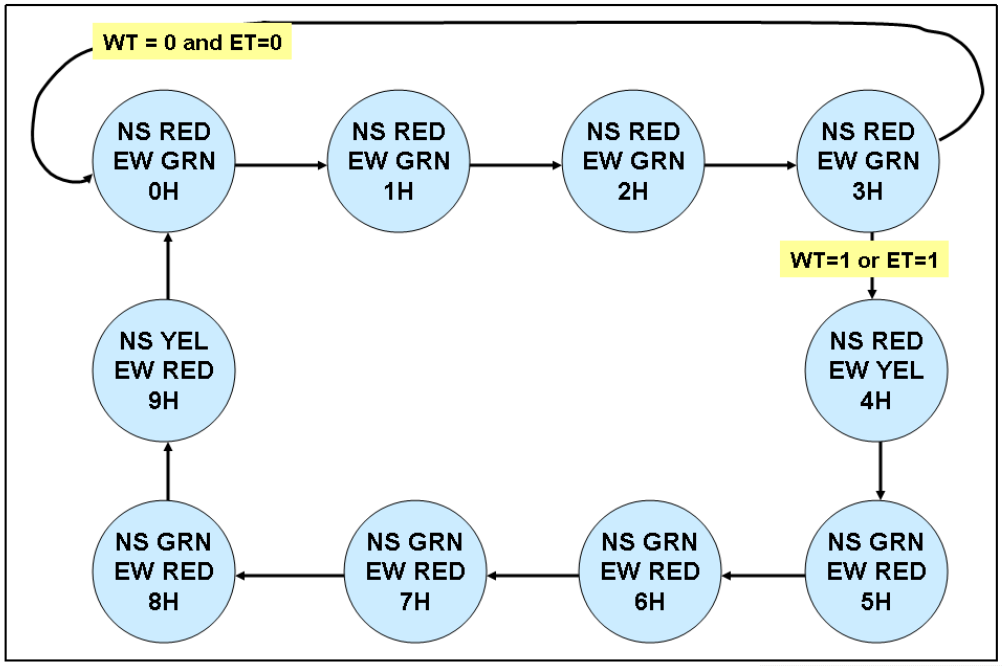


Figure 5.19 State transition diagram for state machine including inputs for waiting EW traffic

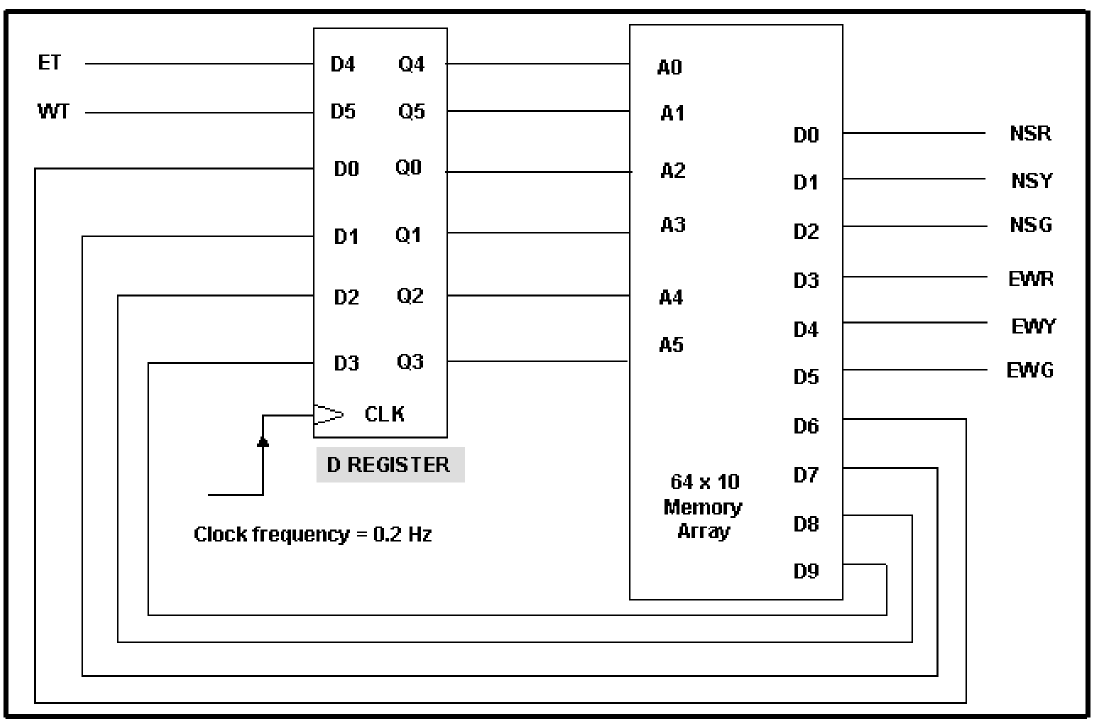


Figure 5.20 State machine design with independent inputs

When a computer fetches an instruction from memory, the bit pattern that represents the instruction is the external combination of inputs to the microcode ROM. This pattern establishes the sequence of steps required to interpret and execute that instruction. Ever wonder why you get the infamous ***Blue Screen of Death***? Sometimes (not always) it may be caused by an errant pointer that sends the program off to fetch an instruction from an illegal area of memory. The bit pattern there may be data values, or garbage, but not the bit pattern of a legal instruction. Since the microcode cannot interpret this pattern, the processor notifies the operating system and the program halts itself. Anyway, back to the problem at hand.

Let's re-examine our state ROM. We've added two more inputs, so the number of possible address combinations goes up to 64. Our state ROM is now 64 locations by 10 bits wide. Our simple traffic system controller has grown to 640 memory cells. It isn't hard to see why the microcode engine in a modern computer has millions of memory cells. Figure 5.21 is a table of the state ROM in all of its gory detail.

Of course the next logical step in the process would be to put back into our design the provisions to the emergency vehicle procedures. That would add one more input to the design, giving a total of 7 address bits to the ROM. Why stop there? We could add a left turn signal for the NS traffic. That would add at least one more output (NS turn light) and one more input (NS turn traffic waiting, NST).

If we proceed to develop the algorithm for this more complex situation we'll probably see that the number of states in our system has increased beyond 16 so we'll need to add an additional output to feed back to the input for sequencing. That gives us a total of 11 outputs and 8 inputs to the ROM. If you have nothing to do some rainy weekend, why not create the state table for this real traffic intersection? I'll leave it as an optional, but highly instructive exercise for the motivated student.

State				WT	ET	Next State				Outputs					
A5	A4	A3	A2	A1	A0	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	1
0	0	0	0	1	1	0	0	0	1	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0	1	0	0	0	0	1
0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	1
0	0	0	1	1	0	0	0	1	0	1	0	0	0	0	1
0	0	0	1	1	1	0	0	1	0	1	0	0	0	0	1
0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	1
0	0	1	0	0	1	0	0	1	1	1	0	0	0	0	1
0	0	1	0	1	0	0	0	1	1	1	0	0	0	0	1
0	0	1	0	1	1	0	0	1	1	1	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0	0	0	0	1
0	0	1	1	0	1	0	1	0	0	1	0	0	0	0	1
0	0	1	1	1	0	0	1	0	0	1	0	0	0	0	1

0	0	1	1	1	1	0	1	0	0	1	0	0	0	0	1
0	1	0	0	0	0	0	1	0	1	0	1	0	0	0	1
0	1	0	0	0	1	0	1	0	1	0	1	0	0	0	1
0	1	0	0	1	0	0	1	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	1	0	1	0	1	0	0	0	1
0	1	0	1	0	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	0	1	0	1	1	0	0	0	1	1	0	0
0	1	0	1	1	0	0	1	1	0	0	0	1	1	0	0
0	1	0	1	1	1	0	1	1	0	0	0	1	1	0	0
0	1	1	0	0	0	0	1	1	1	0	0	1	1	0	0
0	1	1	0	0	1	0	1	1	1	0	0	1	1	0	0
0	1	1	0	1	0	0	1	1	1	0	0	1	1	0	0
0	1	1	0	1	1	0	1	1	1	0	0	1	1	0	0
0	1	1	1	0	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	0	1	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	0	1	0	0	0	0	0	1	1	0	0
0	1	1	1	1	1	1	0	0	0	0	0	1	1	0	0
1	0	0	0	0	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	0	1	1	0	0	1	0	0	1	1	0	0
1	0	0	0	1	0	1	0	0	1	0	0	1	1	0	0
1	0	0	0	1	1	1	0	0	1	0	0	1	1	0	0
1	0	0	1	0	0	0	0	0	0	0	0	1	0	1	0
1	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0
1	0	0	1	1	0	0	0	0	0	0	0	1	0	1	0
1	0	0	1	1	1	0	0	0	0	0	0	1	0	1	0
1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	1
1	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1
1	0	1	0	1	0	0	0	0	0	0	0	1	0	0	1
1	0	1	0	1	1	0	0	0	0	0	0	1	0	0	1
1	0	1	1	0	0	0	0	0	0	0	0	1	0	0	1
1	0	1	1	0	1	0	0	0	0	0	0	1	0	0	1
1	0	1	1	1	0	0	0	0	0	0	0	1	0	0	1
1	0	1	1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	0	0	0	0	0	0	0	0	0	0	1	0	0	1
1	1	0	0	0	1	0	0	0	0	0	0	1	0	0	1
1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	1
1	1	0	0	1	1	0	0	0	0	0	0	1	0	0	1
1	1	0	1	0	0	0	0	0	0	0	0	1	0	0	1
1	1	0	1	0	1	0	0	0	0	0	0	1	0	0	1

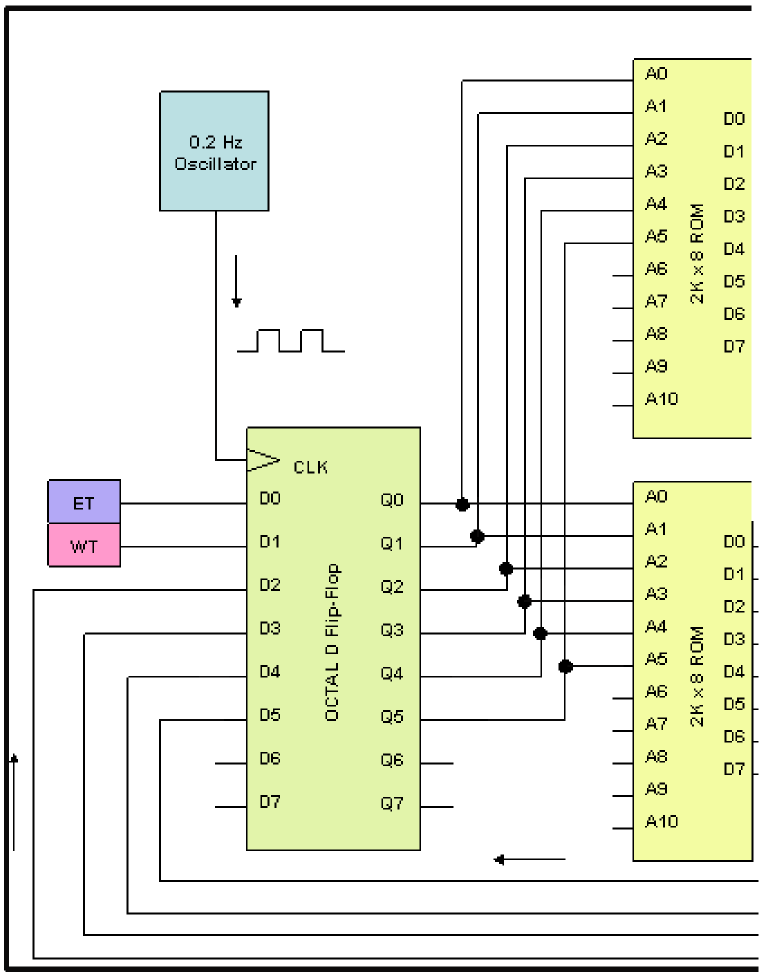
1	1	0	1	1	0	0	0	0	0	0	0	1	0	0	1
1	1	0	1	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	0	0	0	0	0	0	0	0	1	0	0	1
1	1	1	0	0	1	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	0	0	0	0	0	0	0	1	0	0	1
1	1	1	0	1	1	0	0	0	0	0	0	1	0	0	1
1	1	1	1	0	0	0	0	0	0	0	0	1	0	0	1
1	1	1	1	0	1	0	0	0	0	0	0	1	0	0	1
1	1	1	1	1	0	0	0	0	0	0	0	1	0	0	1
1	1	1	1	1	1	0	0	0	0	0	0	1	0	0	1

Figure 5.21 State table for the traffic controller ROM. Note how the unused states, beginning with state 1010 have a data output equal to 09H, corresponding to the NS red and EW red lights being turned on. This is a safety feature in the unlikely event that the system enters an illegal state.

Figure 5.22 shows the important components and data paths within the circuit without getting bogged down in the minute details of actually making such a circuit work in the real world. The oscillator block produces the 0.2 Hz clock stream that we need to sequence the state machine. The Octal D Flip-Flop is an integrated circuit building block that contains 8 D-FF's in a single package with a common clock input. Thus, all 8 FF's are always clocked on the same rising clock edge.

The next two devices are the heart of our traffic controller. They contain the ROM code that we developed to implement the Algorithmic State Machine. Two separate ROMs were used because we need 10 outputs and most commercially available ROMs only have 8 outputs, so we need to partition the design among the two ROMs. For convenience, the 6 outputs of the upper ROM are used to control the lights and the 4 outputs of the lower ROM to sequence through the states. You can see that the 4 outputs from the lower ROM go back and become the inputs to the D-FF's. This clearly illustrates the separate functions of the sequencing function, ***d(s,i)***, and the output function, ***f(s,i).***

The ROMs used in this example each hold 16K bits, organized as 2K by 8. Since we're only using a total of 64 addresses in each ROM, you could argue that this is not the greatest engineering solution in the history of Computer Architecture. However, memory is relatively inexpensive, and, in this particular instance, a ROM with 16K memory cells is less expensive than a much smaller one because the smaller ROM is an obsolete part and difficult, if not impossible to buy.



A	B	Cin		Sum	Cout
0	0	0		0	0
1	0	0		1	0
0	1	0		1	0
1	1	0		0	1
0	0	1		1	0
1	0	1		0	1
0	1	1		0	1
1	1	1		1	1

Table 1: Truth table for a 2-bit adder circuit.

Figure 5.22 Simplified schematic diagram for the traffic intersection controller

Before we move on let's take what we've learned about state machines and try to focus these principles on how a computer works. Figure 5.23 shows the basic sequencing mechanism of a digital computer. You can clearly see the elements of our Mealy machine. Conveniently we’ve omitted pretty much everything else inside the computer, but they’re just connected to the outputs of the microsequencer ROM. The external inputs to the microsequencer would include the machine language instructions as well as all of the other inputs which can affect the program execution flow of the machine.

Suppose that we want to add two numbers together inside of our computer. With respect to our new-found knowledge of state machine design, how might we do this? Let's assume that we want to add together two, 4-bit numbers. The circuit that adds two numbers together inside a digital computer is part of the ***arithmetic and logic unit,*** or ***ALU***. You actually know enough about digital design to create the basic building blocks of the ALU. The basic adder circuit has 3 inputs (A,B and Carry In) and 2 outputs (SUM and Carry Out). We can summarize the behavior of a basic 2-bit adder circuit in the truth table shown as Table 1.

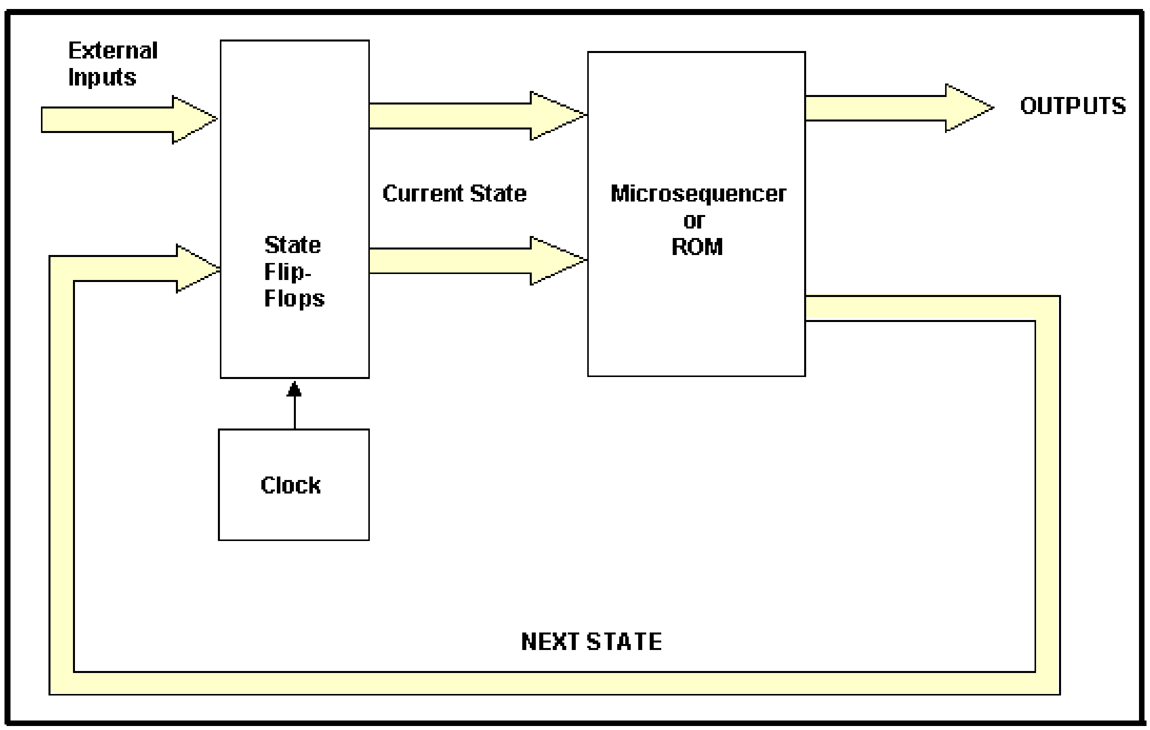


Figure 5.23 A generalized sequential digital machine

Once you design this circuit and place the gates you'll quickly see that you can greatly simplify it if you review the design of the exclusive OR gate. As you can see from the truth table, each stage of the adder simply adds the 2 input numbers plus any carry in from the previous stage and generates a sum and a carry out to the next stage. If we wanted to add together two 32-bit numbers (***ints***), we would need to have 32 of these adders in a row. Anyway, let's see this in figure 5.24.

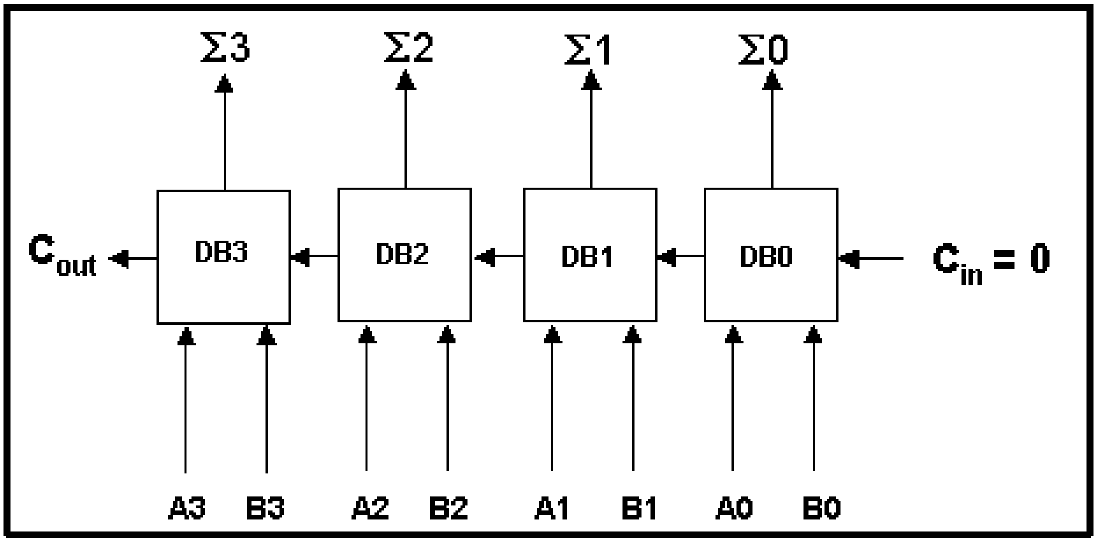


Figure 5.24 Adding two numbers together

Now that we know how the numbers are actually added together, let's look at how a state machine might sequence through the operation to actually add the numbers together, get a result, and save it. Figure 5.25 shows this sequence schematically.

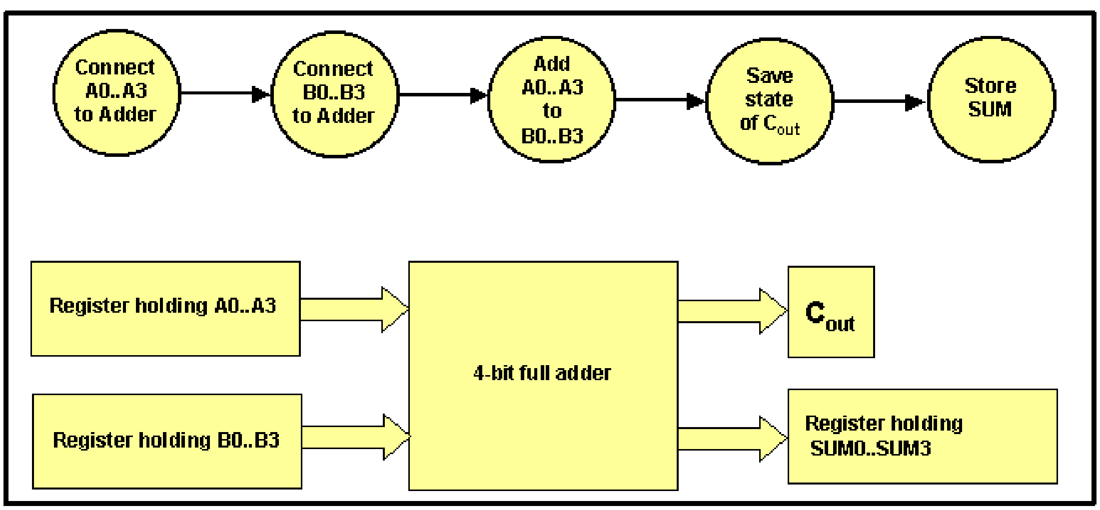


Figure 5.25 State machine sequence for adding two 4-bit numbers

In this particular example it would take us 5 clock cycles to complete the addition. Note that we've neglected some of the preliminary sequences, such as decoding the actual ADD instruction that got us to this point and how the two numbers to be added actually got into the holding registers. Obviously, we're sweeping some additional material under the rug. We'll revisit this problem again in a later chapter in all of its gory detail, so let's just focus on the concepts for now. We'll walk through the steps in the process.

1. The storage register holding the first operand, number A0..A3, is connected to the adder circuit so that the A inputs to the adder now see the first operand.
2. The storage register holding the second operand, number B0..B3, is next connected to the adder.
3. After the appropriate propagation delay (once the B inputs have stabilized), the result appears on the outputs of the adder.
4. The state of the Carry Out bit is saved in the appropriate register. As you'll soon see, we'll also store some other results. For example, if the addition resulted in a sum of zero, we'd also store that information.
5. The sum is stored in an output register for further use.

Let's summarize what we've learned about state machines:

- The next state depends upon the current state, any inputs to the storage register and any outputs that are returned to the input of the storage register.
- A D-type register, comprised of D-type flip-flops, is used to synchronize the state transitions with the edge of a clock. The transition time for the clock must be much faster than any changes that may occur in the state machine. This synchronizes the state machine and prevents the circuit from "running away".
- A flow chart, or state chart for the ASM is the algorithm being implemented.
- The ROM-based state table is one way of implementing the truth table for the design. We could also follow the design steps and use the truth table to create the sum of products (minterm) logical equations for each of the output variables. Once we have the truth table we can then generate the Karnaugh Maps. The K-Maps allow us to create a simplified gate design for the circuit.

The Algorithmic State Machine is the basis for almost all of today's computing engines. The Instruction Set Architecture of a modern computer is determined by its internal microcode ROM which implements the state machine. The processor sequences through a series of states determined by:

- The instruction being executed
- The contents of internal registers
- The results of arithmetic or logical operations
- The type of memory accessing mode being used
- Asynchronous internal or external events (interrupts, RESET, error conditions)

Figure 5.26 shows this schematically.

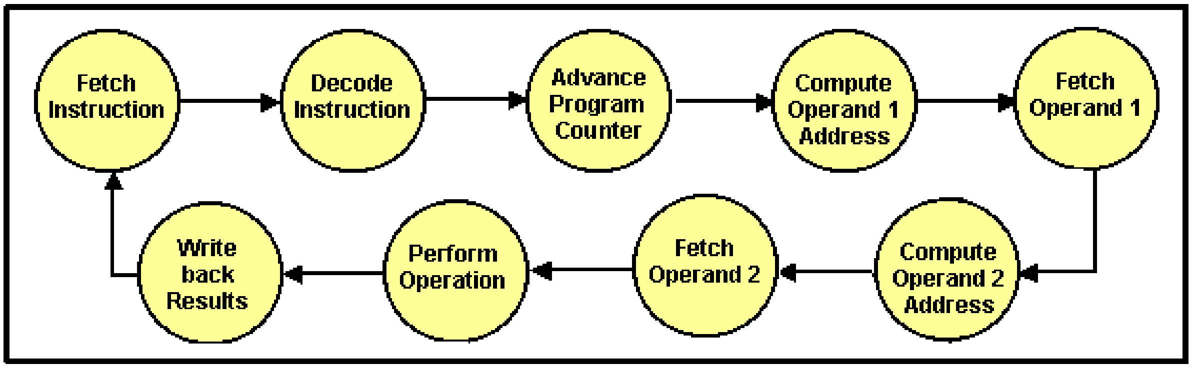


Figure 5.26 State sequencing in a simple microprocessor

Modern hardware design methodologies

Mead and Conway[4] proposed a new methodology for the design of ***Very Large Scale Integration (VLSI)*** circuits. They describe a structured design system based upon a top-down approach to the development of a complex integrated circuit. They say,

> ***The beginnings of a structured design methodology for VLSI systems can be produced by merging together in a hierarchy the concepts presented in this chapter. Designs are then done in a "top down" manner but with a full understanding by the architect of the successive lower levels of the hierarchy.***
>
> ***To begin with, we plan our digital processing systems as combinations of register-to-register data transfer paths, controlled by finite state machines. Then the geometric shapes, relative sizes, and interconnection topologies of all subsystem modules are collectively planned so all modules will merge together snugly, with a minimum of space and time wasted by random interconnect wiring……***
>
> ***A particularly uniform view of such a system of nested modules emerges if we view every module at every level as a finite state machine or data path controlled by a finite state machine. At the lowest level, elements such as the stack and register cells may be viewed as state machines with one feedback term (the output), two external inputs (the control signals) and a 1-bit state register. These rudimentary state machines are grouped in a structured manner to form portions of a state machine, or data path controlled by a state machine, at the next level of the hierarchy.***

Later[5] they go on to describe a method of designing the actual integrated circuits by creating a ***computer-aided design*** tool (CAD) that works like a macro assembler would work for software. If the basic circuit elements could be expressed as a few standard building blocks, or cells, then a symbolic lay-out language of some kind could, using the macro assembler analogy, create an IC layout from these standard cells. They say,

> ***The user defines symbols (macros) that describe the layout of the basic system cells. The locations and orientations of instances of these symbols are described in the language, as functions of the appropriate parameters. These symbolic descriptions may then be mechanically processed in a manner similar to the expansion of a macro assembly language program, to yield the intermediate form description of the system layout, which is analogous to machine code for generating output files.***

What Mead and Conway were describing were two concepts that should be very familiar to you. One is the idea of a structured approach to the design of the hardware. You might call this "software engineering". It starts with developing requirements documentation and then a set of formal specifications. From there, the various functional components (blocks) are defined and by a process of top-down decomposition, the software progresses through the development process.

The second concept will be familiar to you if you've studied how modern compilers convert a high-level language to an intermediate language (assembly language) and then to machine code. Here the low-level machine code is represented by the standard low-level cells that represent the transistor level circuits and interconnects between these cells.

What the authors were describing is a tool that today we call a ***silicon compiler***, and the process by which modern integrated circuits are designed is called ***silicon compilation***. Hardware circuit designers use a high-level development language, either Verilog or VHDL. VHDL is officially defined in IEEE Standard 1076-2001, *IEEE Standard VHDL Reference Manual*[6]. Verilog is considered to be "the other" hardware description language. Verilog started out as proprietary simulation language but was subsequently turned over to the IEEE and published as IEEE Standard 1364-1995, *IEEE Standard Description Language Based on the Verilog Hardware Description Language.*[7]

In 1981, Silicon Compilers, Inc. was founded to *decouple the design description from the implementation*[8]**.** Their value proposition was to take care of the design implementation so that the designers could focus on the algorithm. To do this, they would create libraries and code modules which would be translatable to custom integrated circuit blocks; the same as if expert designers were handcrafting those circuits.

In 1985 the CMOS process became commercially available and this was the breakthrough needed to drive the ASIC industry. As we've seen, the CMOS gate is an almost ideal switch, and with it, the commercial viability of integrated circuits designed using silicon compilation was realized. Prior to the introduction of the CMOS process there was still too much hand-crafting needed to make an integrated circuit work properly. According to *Cheng,*[8] "power consumption went down, noise margin went up, and a 1 was a 1 and a 0 was a 0".

With the commercial viability of ASIC designs the standardization of hardware description language (***HDL***) tools emerged and logic synthesis, the design of digital hardware using silicon compilation, greatly accelerated. Silicon Compilers, Inc. merged with Silicon Design Labs in 1987 to form Silicon Compiler Systems Corporation. In 1990, SCS was acquired by Mentor Graphics, Inc., a leading supplier of Electronic Design Automation Tools.

You can see the impact of silicon compilation in figure 5.27.

Intel microprocessors

	Year of introduction	Transistors
4004	1971	2,250
8008	1972	2,500
8080	1974	5,000
8086	1978	29,000
286	1982	120,000
386™ processor	1985	275,000
486™ DX processor	1989	1,180,000
Pentium® processor	1993	3,100,000
Pentium II processor	1997	7,500,000
Pentium III processor	1999	24,000,000
Pentium 4 processor	2000	42,000,000

Source: Intel Corp.

Figure 5.27 Growth in transistor count for the Intel family of microprocessors. From *Cheng*[9].

If you plot the data from figure 5.27 on semi-logarithmic graph paper, you'll get an amazingly close approximation to a straight line, which is a remarkable validation of Moore's Law. If we consider the number of designers required to design an integrated circuit, such as the 8086, with 29,000 transistors, versus the Pentium 4 processor, with 42,000,000 transistors, we must come to the conclusion that Intel could not possibly have used a Pentium 4 design team roughly 2000 times bigger than the 8086 design team. Thus, the only conclusion that we can draw from this data is that it is the dramatically increased efficiency of each designer to place transistors (or CMOS gates) on a silicon chip is responsible for the increase in circuit density.

So, just as C++ has freed the programmer from assembly language programming issues, so has silicon compilation freed the hardware designer from the low-level issues associated with the process of designing integrated circuits.

As a person familiar with high-level languages, you should quickly become comfortable with the structure of HDL languages. In fact, there are relatively few signs that you are designing hardware, rather than writing software. Several commercial software companies have come to the same conclusion and today, there are commercial tools available which closely integrate the separate hardware and software design processes into a higher level system view.

However, one remaining difference between the hardware and software development processes is the cost of fixing a defect. As software developers, you know that recompiling and rebuilding a software image might take a few days. The processes involved in distributing the new version of the code to the customers are also well established. You might post a new version to an FTP site in order to send out code updates.

The hardware developers do not have this luxury. Even though the design processes are converging, the hardware designer is still faced with the reality that hardware design is a complicated, expensive and time consuming process with little or no margin for error. The cost of a hardware re-spin could easily be $500,000 with a time delay of three months or more. Thus, hardware design tools, even with silicon compilation, are heavily structured towards testing and design simulation. It is not unusual for the time required to develop a new ASIC to be equally divided between the actual time for design and the time required to thoroughly test the hardware in simulation. Of course, the hardware designers always have a Plan B in their back pockets. The universal solution to the presence of a defect in the hardware, is, and will be for the foreseeable future: "***Fix it in software***."

Before we leave this topic it is instructive to actually look at some VHDL code and compare it to the languages we are already familiar with. The following is an example of an adder circuit (*from Ashenden*[10]). We first need to declare an ***entity.*** This is analogous to declaring a variable.

```
entity adder is
        port ( a : in word ;
                    b : in word ;
                    sum : out word ) ;
end entity adder ;
```

Next, we need to describe the internal operation of a module. In other words, we need to write the statements describing the behavior of the variables. This is done in the ***architectural body*** of the code.

```
begin
       add_a_b : process ( a,b ) is
        begin
                sum <= a + b ;
         end process add_a_b ;
 end architecture abstract;
```

The architecture body is named **abstract** and it contains a process **add_a_b**, which describes the operation of the entity. Like template functions in C++ this process assumes that the addition operator '+' has previously been defined for the addition of data type 'word'.

We could easily picture how this code snippet could compile to a circuit comprised of 1-bit adder primitives, as in figure 5.24, except that it would be configured to add a 16-bit or larger variable called a 'word'.

Summary of chapter 5

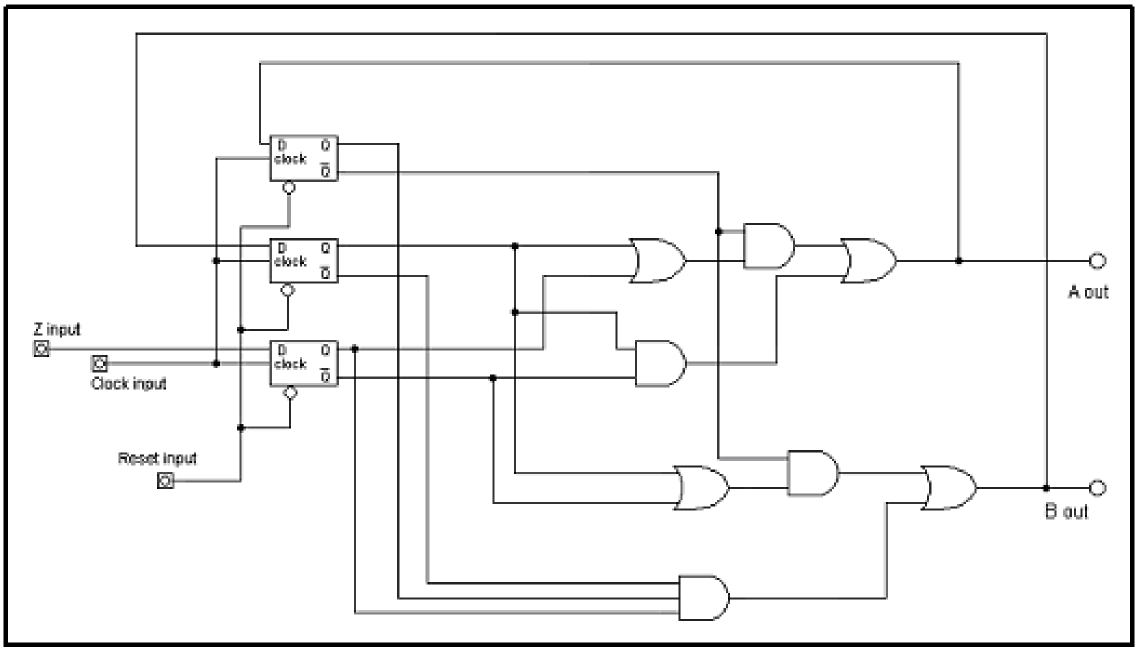
- We started chapter 5 with a discussion of the generalized concept of an algorithm being either a solution based upon a set of software steps or a solution based in hardware.
- We looked at the definition of a finite state machine and saw how we define a state machine as either a Mealy Machine or a Moore Machine.
- We saw how the feedback of the current state, combined with a 'D' type flip-flop, allows us to synchronize and stabilize the state transitions.
- We examined how to use our knowledge of building truth tables to construct the hardware implementation of a state machine.
- We walked through the problem of building an algorithm into a state machine by constructing a memory-based implementation of a traffic intersection controller.
- Finally, we saw how hardware description languages are used to design hardware systems.

Bibliography and references

1- Thomas Richard McCalla, ***Digital Logic and Computer Design,*** ISBN 0-675-21170-0, Merrill, New York, 1992, Pg. 265
2- Carver Mead and Lynn Conway, ***Introduction to VLSI Systems,*** ISBN 0-201-04358-0, Addison-Wesley Publishing Company, Reading, MA, 1980, Pgs. 85-87
3- Claude A. Wiatrowski and Charles H. House, ***Logic Circuits and Microcomputer Systems,*** ISBN 0-07-070090, McGraw-Hill Book Company, New York, 1980, Pgs. 1-11
4- Mead and Conway, ***ibid,*** pg.89
5- Mead and Conway, ***ibid,*** pg.98
6- Peter J. Ashenden, ***The Designer's Guide to VHDL, Second Edition***, ISBN 1-55860-674-2, Morgan-Kaufmann Publishers, San Francisco, 2002, Pg. 671
7-Ashenden, ***ibid***, pg. 677
8-Ed Cheng, http://bwrc.eecs.berkeley.edu/Seminars/Cheng%20-%209.27.02/Silicon%20Compilation,%2021%20years%20young.pdf
9- Cheng, ibid
10- Ashenden, ***ibid***, pgs. 108-111

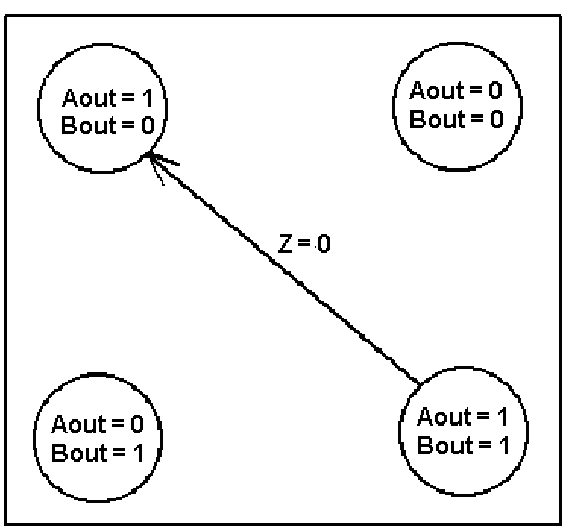
Exercises for Chapter 5

1- The figure below is a state machine comprised of 3 D-type flip-flops and some logic gates.



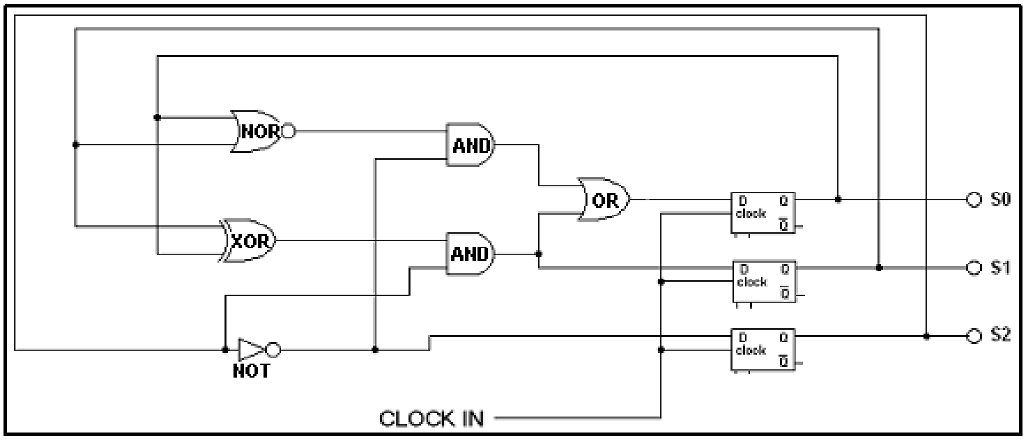
You may assume that the RESET inputs to the 3 flip-flops have been asserted. Complete the truth-table, shown below, for the state machine.

Draw the complete state transition diagram for this state machine. For simplicity, it is not necessary to include the effect of the RESET signal in your drawing. Hint: start with the partial state transition diagram shown below.



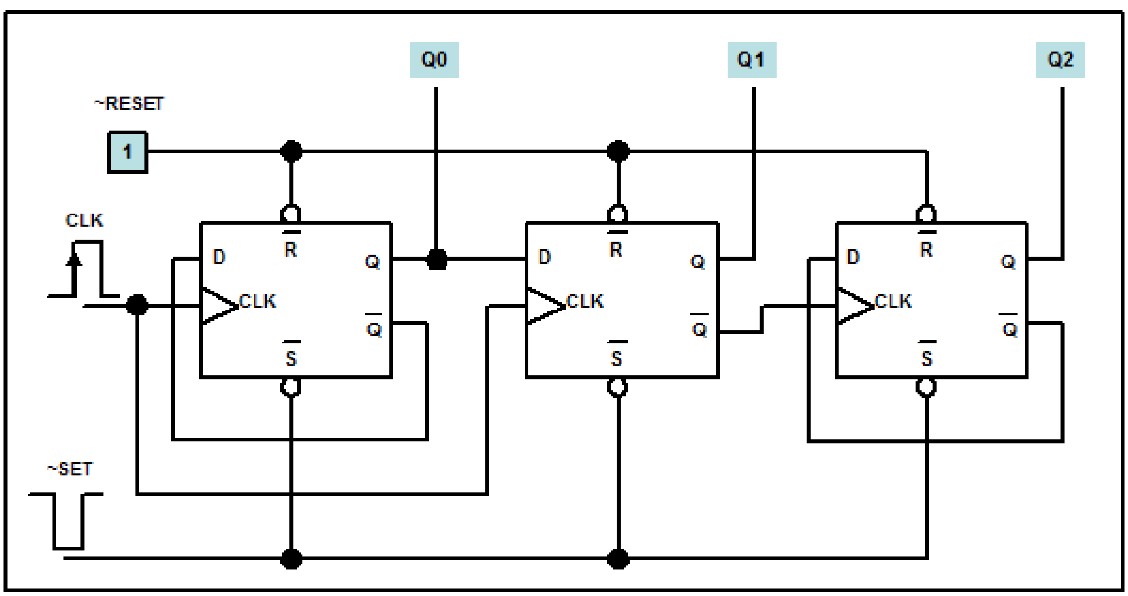
A in	B in	Z	A out	B out
0	0	0		
1	0	0		
0	1	0		
1	1	0		
0	0	1		
1	0	1		
0	1	1		
1	1	1		

2- The diagram shown below is the State Machine for some arbitrary algorithm.



Assume that the state machine has received a RESET pulse and is in state 000 (S0 = 0, S1 = 0, S2 = 0). Create a state transition diagram for this circuit..

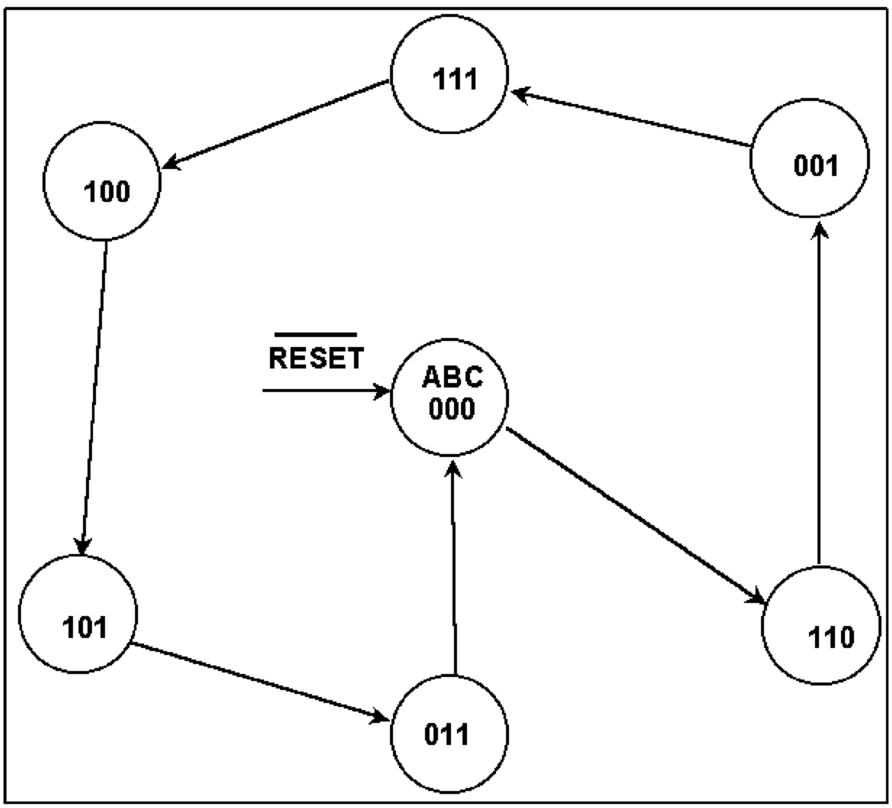
3- The circuit shown below consists of 3 D-type flip flops. The black dots indicate those wires that are physically connected to each other. The ~RESET inputs are permanently tied to logic 1 and are never asserted. Before any clock pulses are received, the ~SET input receives a negative pulse to establish the initial conditions for the circuit. draw the



state transition diagram for this circuit.

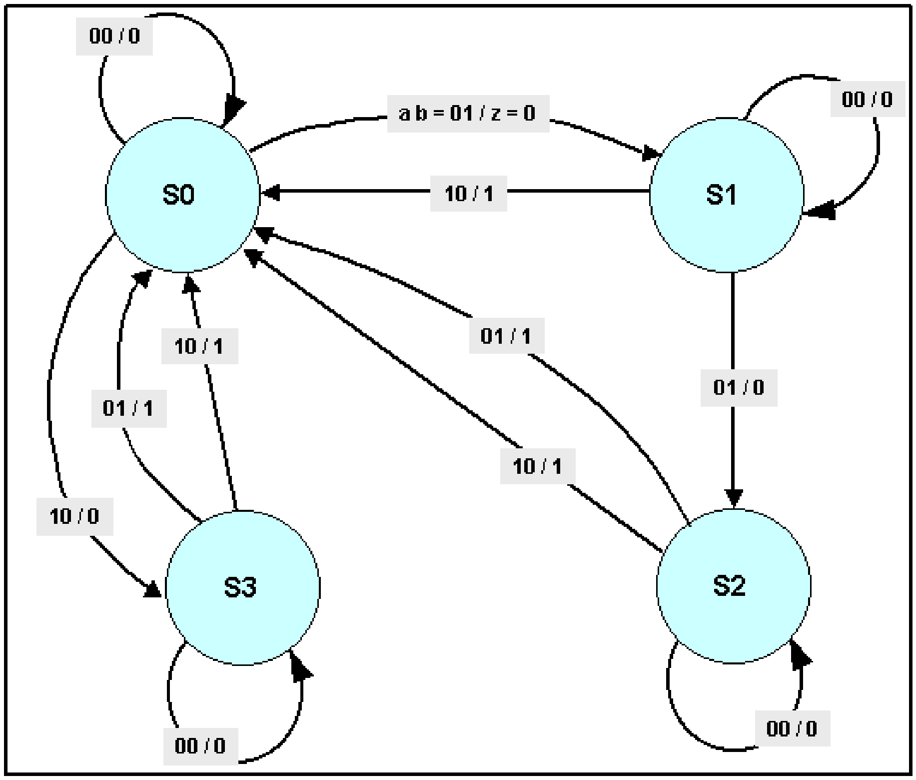
4- The figure, shown below, is a state diagram for a hardware algorithm.

After a reset pulse, the system is initialized to state 000. Shown below is the implementation method for the algorithm. The outputs of the circuit are state variables ABC, as shown. These variables also feed back to the truth table to become the input variables for the next state transition that will occur on the next clock. Draw the truth table that corresponds to this hardware algorithm. Simplify it, and then draw the entire circuit with the gate equivalent circuit replacing the truth table. Hint: remember, that there is one possible state that you can add to the truth table that does not appear in the algorithm. It may help you to include it in your truth table to get some added simplification.



5- You are the chief designer for the Happy Times Storm Door and Vending Machine Company. You've been given the task of redesigning the company's best-selling vending machine, a simple wall model that installs in convenience store rest-rooms. Since electrical power is available you decide to replace the old mechanical model with a spiffy new electronic device.

You decide to use a State Machine design format. The state machine is shown in the figure, below.


00 / 0
a b = 01 / z = 0
00 / 0
S0
10 / 1
S1
10 / 1
01 / 1
01 / 0
01 / 1
10 / 1
10 / 0
S3
S2
00 / 0
00 / 0

The state machine can cycle through 4 possible state, S0-S3. In addition, it has 2 external inputs **a** and **b**, and one output, **z**. Input **a** represents a quarter being deposited in the coin slot. Input **b** represents a dime being deposited in the coin slot. The merchandise cost 30 cents and no change is given if the amount deposited is more than 30 cents. The output, **z = 1**, causes the merchandise to be dispensed.

The four states are defined as follows:

- S0 = Quiescent state, no money deposited.
- S1 = 10 cents deposited
- S2 = 20 cents deposited
- S3 = 25 cents deposited

Assume that a customer deposits 10 cents (ab = 01). This takes it to state S1. This isn't enough money, so there is no merchandise dispensed (z = 0) in this transition. There are two possibilities. If another 10 cents is deposited, it goes to S2. Again, no merchandise is dispensed. However, if 25 cents is deposited, it dispenses the merchandise and returns to state S0.

If 25 cents is the first coin deposited, then it goes to S3, but no merchandise is dispensed. Any other coin being deposited will dispense the merchandise and return it to state S0.

It is not possible to deposit two coins at once, so the input, ab = 11, is not allowed. Also, the state transitions label 00/0 just means that nothing happens on that clock pulse. Complete the Truth Table, simplify the logical equations using the Karnaugh Map and then draw the gate design for this circuit.

6- Design a state machine circuit that will detect the occurrence of the serial bit pattern 1001. Your solution should include the state transition diagram, truth table, K-map and finally, the circuit implementation.

7- The circuit shown below is made up of 4 "D" Type flip-flops and 2 exclusive OR gates. Note that the SET input is not used. Only RESET may be asserted.Make a truth table showing:

a- The state of outputs Q_0, Q_1, Q_2 and Q_3 after a /RESET pulse.

b- The state of the outputs after 16 clock pulses.

c- How many clock pulses are required before the outputs recycle to the same pattern again?

e- Redesign the circuit as a Finite State Machine. For simplicity, design it so that the pattern repeats itself after 8 clock pulses. Create a truth table for each state and for the next state, simplify it and draw the gate circuitry.