§4 FINITE STATE MACHINE DESIGN & OPTIMIZATION

4.1 Combinational Vs Sequential Circuits

- Digital circuits can generally be divided into ***combinational*** circuits and ***sequential*** circuits. Combinational (*combinatorial*) circuits are those whose outputs are functions of current inputs only. Sequential circuits are those whose outputs depends not only on current inputs, but also previous inputs.

  Combinational circuit: *Outputs = F (current inputs)*

  Sequential circuit: *Outputs = F (current inputs, past inputs)*

- Sequential circuits therefore requires ***memory elements*** such as latches or bistables.

- ***Finite State Machines*** (FSMs) are mathematical abstractions of sequential circuits. A FSM is a system comprising *states*, *inputs* and *outputs*. It models time as discrete instants at which input or output can change.

- If the states and output transitions are constrained to occur at pre-defined times such as clock edges, the FSM is known as ***synchronous***.

- If the states and outputs change in response to input changes, which can occur at *any time*, the FSM is known as ***asynchronous***.

4.2 General Models for Finite State Machines

Figure 4.1 shows the ***Mealy model*** of a FSM:-

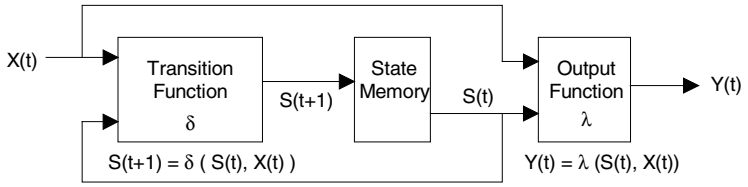


Figure 4.1 Mealy model of FSM

- A Mealy model of FSM contains three components: a ***state memory*** which stores the current state S(t); a ***state transistion function*** δ which determines the next state depending on the current state S(t) and the input X(t), $S(t+1) = \delta(S(t),X(t))$; and finally an ***output function*** λ which generates the output Y(t), which is determined both by S(t) and X(t).

- An alternative way of representing a FSM is to use the ***Moore model***. It is similar to the Mealy model except that the output is dependent only on the current state S(t), and not on the input X(t).

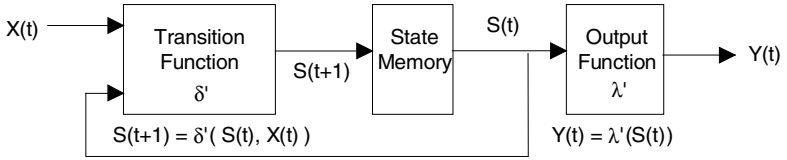


Figure 4.2 Moore model of FSM

- A Moore machine can always be mapped into a Mealy machine (and vice versa).

- Mealy machines usually have fewer state variables, hence they are more widely used in engineering applications. Fewer state variables implies fewer memory elements. However, Moore machines are simpler to analyse mathematically, and therefore are more widely used in *algebraic FSM theory* (which we don't study on this course). In the following discussion, we will (mainly) restrict ourselves to the Mealy models.

- One problem with the Mealy model as shown in figure 4.1 is that the output may have *glitches* (why?) A slightly modified form as shown below is more commonly used. Here the transition and output functions are combined, and the state memory latches are extended to latch the primary outputs. Beware, the output Y(t) now changes on the next clock cycle!

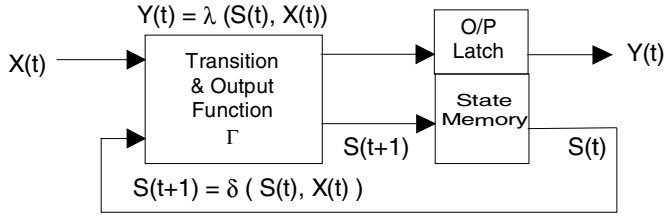


Figure 4.3 Mealy machine with registered output

- If we view combinational circuit as state machine with no memory, register/memory as state machine with no transition logic, all digital systems can be viewed as networks of FSMs. However the FSM models are only useful for circuits with inherent sequencing characteristics such as counters and control

circuits. Circuits such as registers and adders are better described by the functions they perform on input data.

- ***Autonomous Finite State Machines*** are special FSMs with no inputs. They can easily be analysed algebraically. A very useful example of an autonomous FSM is the maximal length ***linear feed-back shift register*** (LFSR) which will be studied later on in this course.

- ***Communicating Finite State Machines*** consists of two or more FSMs interacting with each other. An example of this is:

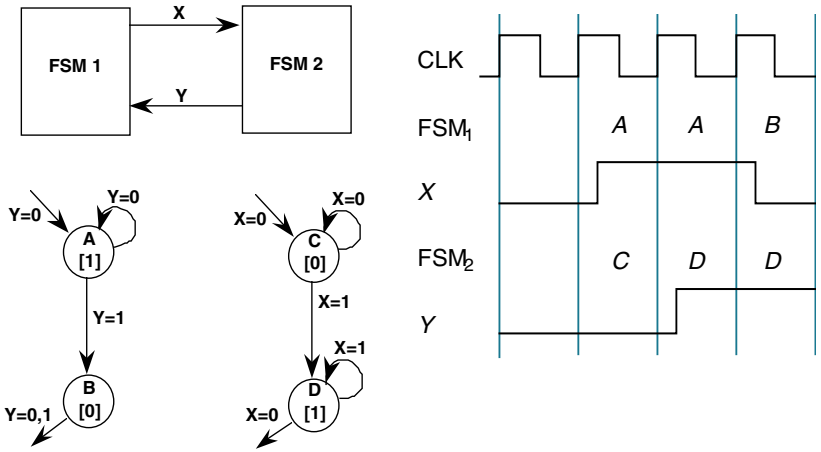


Figure 4.4 Communciation FSM

These two machines advance in locked steps. Assuming initially X=0 and Y=0, then the behaviour of the machine is as shown in the timing diagram.

4.3 FSM Design Steps

- The steps for designing a FSM are:-
  1. Understand the specification;
  2. Define the problem using a state diagram and/or a state table;
  3. Simplify the state table by eliminating redundant internal states (state minimization problem);
  4. Assign (binary) codes to the states (state assignment problem);
  5. Determine the logic equation for the transition function and output function;
  6. Minimise the logic equations;
  7. Map the design to a given technology or device.

- Steps 3, 4 and 6 optimise the design. They are valuable, but not necessary, in order for a design to work properly.

Step 1: Let us consider the design of FSM using a simple Vending Machine example. The specification is: a) Accepts 10p and 20p coins; b) Delivers a can of drink costing 30p; c) Provide change where appropriate. This is modelled as below:

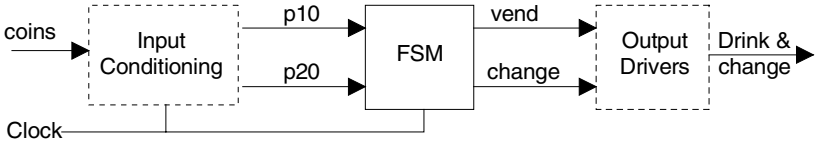


Figure 4.5 Vending Machine

- We further assume that only one coin can be inserted at any one time. For each coin inserted, a pulse is generated on *p10* or *p20* lasting for one clock period. If *vend*=1, a can of drink is delivered; if *change*=1, a 10p is returned.

4.4 Step 2: State Diagram Representation

- A common way of representing a FSM is to use a state diagram. A state is depicted as a circle with output arrows. Each arrow defines a transition. Next to the arrow is the input and output conditions as shown below:-

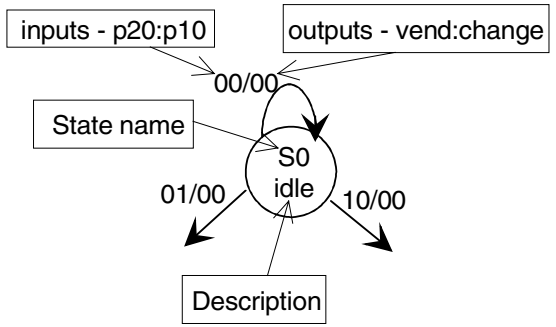


Figure 4.6 Notations used in a state diagram

FSM must remain in S0 until there is a *p20* or *p10* input. If *p10*=1, the circuit should not activate vend or change, but must remember that the credit is 10p, and must

therefore move to another state (say S1). Therefore we can draw a state transition diagram as:-

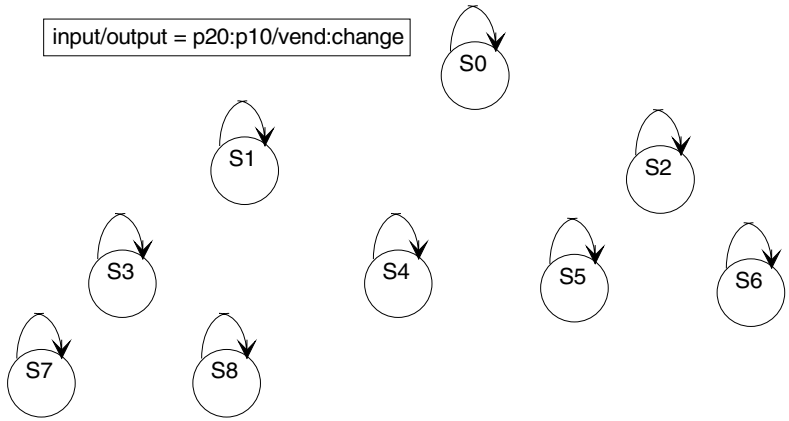


Figure 4.7 State Diagram for the drink vending machine

Step 3: State Minimization

The credit remaining at states S4, S5, S6, S7 and S8 is zero because a can of drink has been delivered and any change have been given. Therefore in theory, the complete tree must be repeated beginning from these states. The tree can therefore grow indefinitely.

However, states S4-8 are equivalent to state S0. We can therefore merge all these states and create a cyclic state diagram:-

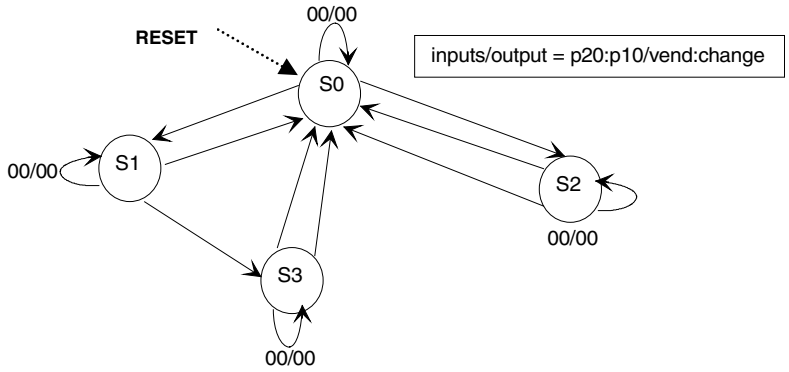


Figure 4.8 Cyclic state diagram for the vending machine

*Two states are said to be **equivalent** if they have identical next states and outputs.* Therefore, from the above diagram, we can see that at both S2 and S3, the machine has received 20p and will respond to future inputs in the exactly the same way. Hence S3≡S2. Two equivalent states can be merged. Therefore we can reduce this diagram further as:-

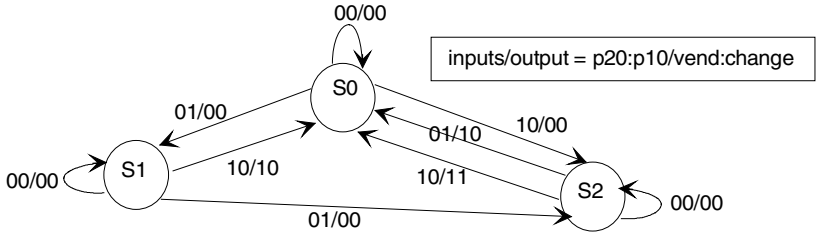


Figure 4.9 Reduced state diagram for the vending machine

One last modification: although we have reduced the FSM from 4 to 3 states, we have to eventually encode the states with an N-bit binary number. Therefore there will still be a fourth state which is illegal. Let us called this S3 and force a transition (always) from S3 to S0. This ensures that even if S3 is entered accidentally, the FSM will not be hung in an invalid state forever! The final state diagram is thus:

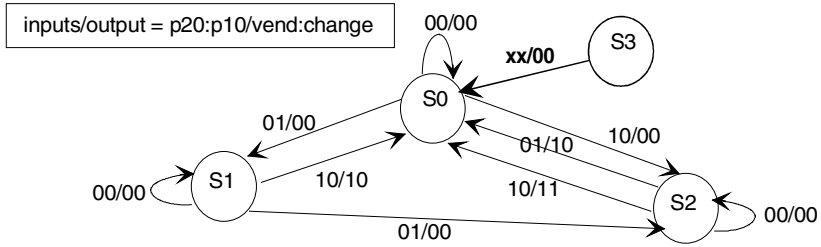


Figure 4.10 Final state diagram for the vending machine

4.5 State Transition Table

We are now in a position to draw up the state table:-

Current State	Inputs *p20:p10*	Next State	Outputs *vend:change*
S0	0 0	S0	0 0
S0	0 1	S1	0 0
S0	1 0	S2	0 0
S1	0 0	S1	0 0

S1	0 1	S2	0 0
S1	1 0	S0	1 0
S2	0 0	S2	0 0
S2	0 1	S0	1 0
S2	1 0	S0	1 1
S3	x x	S0	0 0

Alternatively we can use a more compact representation:

Inputs - p20:p10

current state	00	01	11	10
S0	S0,00	S1,00	S0,00	S2,00
S1	S1,00	S2,00	S0,00	S0,10
S2	S2,00	S0,10	S0,00	S0,11
S3	S0,00	S0,00	S0,00	S0,00

Each entry in the table represents the tuple (next state, outputs - *vend*:*change*).

4.6 Transition and Output Logic Equations

To design a circuit for this state table, the states must be assigned binary bit-patterns. Let S0=00, S1=01 and S2=10. (S3=11 is the illegal state). Note that this state assignment is entirely arbitrary. We could have assigned S0=10, S1=01 and S2=11. As will be seen later, the state assignment has an impact on the optimality of the circuit.

The state table now looks like:

Inputs - p20:p10

current state *s*1:*s*0	00	01	11	10
00	00,00	01,00	00,00	10,00
01	01,00	10,00	00,00	00,10
10	10,00	00,10	00,00	00,11
11	00,00	00,00	00,00	00,00

The logic equations for the four outputs s1(next), s0(next), vend and change can easily be derived:

s1(next) = /s1/s0*p20*/p10 + /s1*s0*/p20*p10 + s1*/s0*/p20*/p10*
s0(next) = /s1/s0*/p20*p10 + /s1*s0*/p20*/p10*
vend = s1/s0*p20 + s1*/s0*p10 + /s1*s0*p20*/p10*
change = s1/s0*p20*/p10*

At this stage, logic minimisation procedure such as ***Karnaugh Map*** or the ***Quinne-McCluskey*** algorithm could be employed to attempt to reduce the logic further.

Once these equations are defined, the design of the FSM is complete. It is now ready for implementation. The exact implementation depends on the technology used. For example it can be Altera MAX/FLEX or Xilinx XC4000/Virtex FPGAs. The circuit might look like:

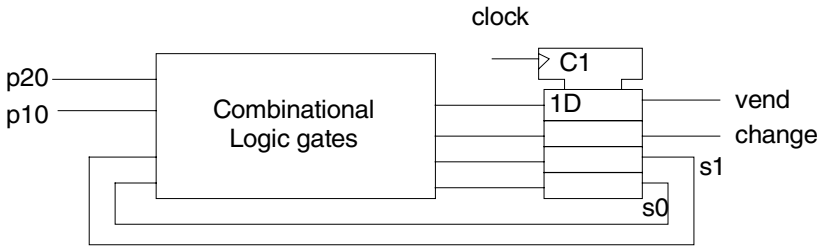


Figure 4.11 Implementation of the vending machine FSM

4.8 FSM Example: Huffman Code Decoder

Huffman codes are used in many applications such as JPEG and MPEG compression in order to reduce the number of bits required to send messages (symbols). It relies on known probability of occurrence of a set of fixed symbols.

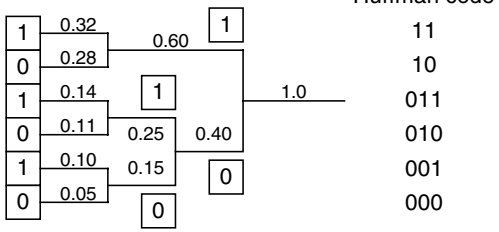
Consider the following symbol set (A to F) with the given probabilities:-

Symbol	Binary Code	Frequency of Occurrence
A	000	0.32
B	001	0.28
C	010	0.14
D	011	0.11
E	100	0.10
F	101	0.05

The binary code is a fixed length code. It is obvious that if one were to use fewer bits for the symbol 'A', possibly more for symbol F, then on average the number of bits needed to send messages will be less than 3-bits per symbol.

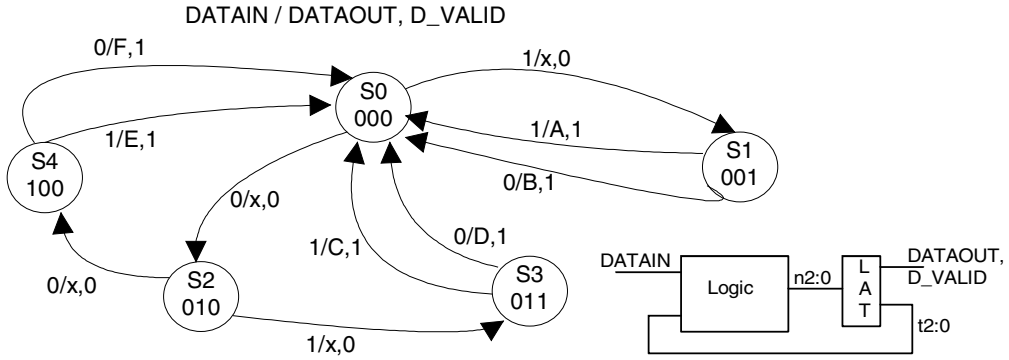
Here is how Huffman code for this symbol set can be constructed as :

Symbol	Binary Code	Freq.	Huffman code
A	000	0.32	11
B	001	0.28	10
C	010	0.14	011
D	011	0.11	010
E	100	0.1	001
F	101	0.05	000



1. First, group together two symbols with the lowest probabilities (in this case 'E' and 'F') and treat is as a combined symbol 'EF' with probability = P(E)+P(F);
2. Keep repeat 1) until a tree is constructed;
3. From the root of the tree, backtrack and allocate a '0' to one branch of the tree and a '1' to the other branch of the tree;
4. Huffman code is formed by tracing the sequence of '0' and '1' from the root of the tree to the leave as shown.

The decoder circuit is implemented with a FSM:-



4.8 FSM implementations - hardware considerations

4.8.1 Generic Block Diagram

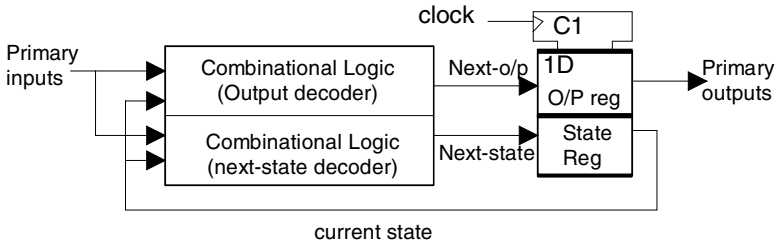


Figure 4.12 Generic block diagram of a FSM

4.8.2 Implementation alternatives

- The combinational part of the FSM can be implemented using:
  1. standard logic gates - suitable only for simple designs
  2. Programmable ROM - suitable for designs with many outputs/states
  3. Programmable Logic Devices (CPLDs) - suitable for most FSMs
  4. Field Programmable Gate Arrays (FPGA) - suitable for more than FSMs

- Using PROM has the advantage that no logic minimization is needed. It implements the logic 'exhaustively': all possible combination of inputs are implemented. For example take a 4 i/p function:

Figure 4.13 Implementation of FSM with PROM

- Unfortunately PROM size grows exponentially with number of inputs! However, for logic with small number of inputs and large number of outputs, PROM can offer efficient implementation.

- Programmable Logic Devices and FPGAs are more suitable for most FSM implementation.

4.8.3 Asynchronous inputs

So far we have only considered the cases where the input signals are assumed to be synchronous to the FSM itself. In practise, this may not be true. Some or all of the inputs may be derived from external circuits (such as a transducer, comparator etc.) which may change at any time. This could give rise to a timing problem known as ***input race hazard***. (Race hazard in general will be discussed in a later lecture with the topic of metastability in digital circuits.)

Let us consider the problem in some details. Assume that we have a state transition as shown in figure 4.14. S1 to S2 transition depends on input A going from 0 to 1. Assuming that the encoding of S1 and S2 are 00 and 11 respectively, and that A is an asynchronous input, figure 4.16 shows the timing relationship.

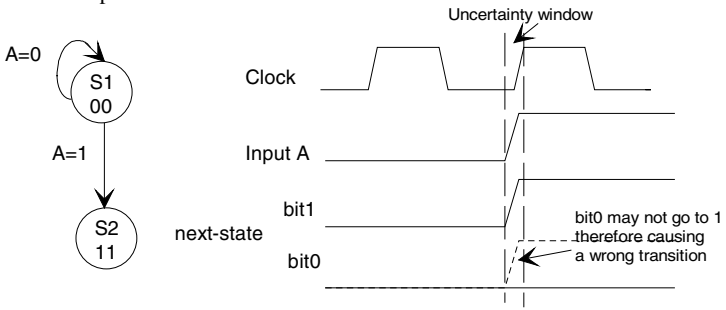


Figure 4.14 Asychronous Input to FSM

Let us consider what happens if there is a 0 -> 1 transition on A occurring around the rising edge of the clock (which is also the active edge of the FSM). The delay in the combinational logic and the setup time of the flip-flop may cause one of the state flip-flop to change to the next-state value, but not the other. In this case, bit 1 changes from 0 to 1, but not bit 0. This will obviously cause errors in the state transition. The FSM will therefore fail to work properly.

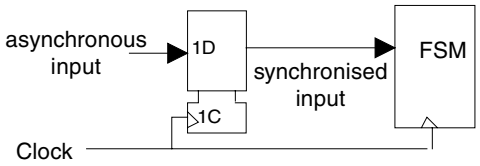


Figure 4.15 Synchronising asynchronous inputs

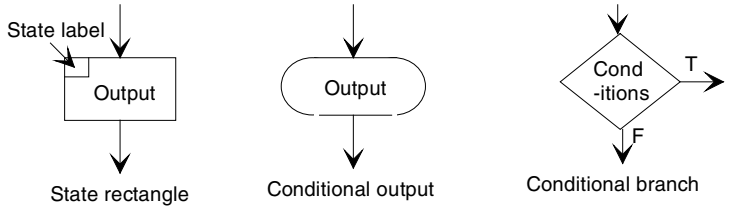
A simple but effective cure for this input race problem is to synchronise all asynchronous inputs with a latch which is clocked by the same clock signal as the FSM as shown in figure 4.15.

4.8.4 Types of flip-flops

Many modern PLDs provide flexible output structure. The output from a programmable macrocell can be configured to be inverting/non-inverting, register or combinational output, with/without feedback. The type of flip-flop is also user selectable. These could be a D flip-flop, S-R flip-flop, J-K flip-flop or T flip-flop. So far we have assumed that we only use D-type flip-flops. In some cases, using T or J-K flip-flops may yield a more efficient implementation (i.e. fewer product terms in the boolean equations). Which flip-flop type is optimum is totally problem dependent. Most CAD software (good ones) actually try different types and choose the best option.

4.9 Alternative FSM Representation - Algorithmic State Machine (ASM) Chart

Based on flowchart notations, the Algorithmic State Machine chart was popularized by Christopher Clare in the book "Designing Logic Systems using State Machines". An ASM chart has only three elements:



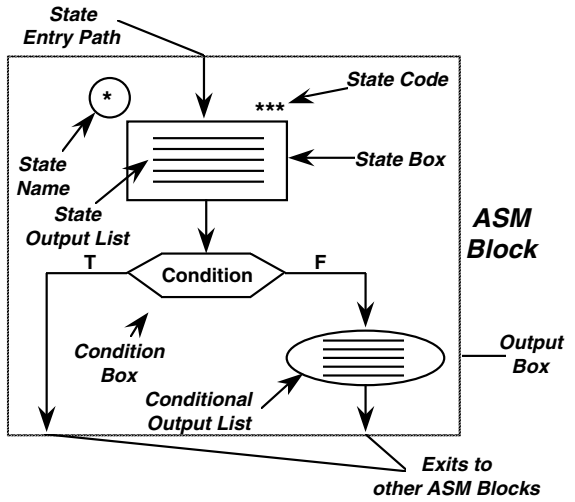


Figure 4.16 ASM chart notations

The key features of ASM representations are:

1. State machine is in one state block per state time (clock cycle).
2. Single entry point only.
3. Unambiguous exit path for each combination of inputs.
4. Outputs asserted high, low, or high impedance

until the following clock cycle.

To construct an ASM chart you should follow these rules:

1. Each state has one and only one state box.
2. Outputs that depend only on the state number (as in the moore-model) are shown in the square box.
3. Outputs that depend on inputs (as in the mealy-model) are shown in rounded boxes.
4. Decision boxes contains the conditions of input variables under which transition should take place. A path through a decision box deos not have to include all input variables, thus accomodating 'don't cares'.
5. To find out what is done n any state, follow the flow chart until you reach the next state box, then wait for the next CLOCK pulse.

Sometimes multiway decision boxes can be simplified as:

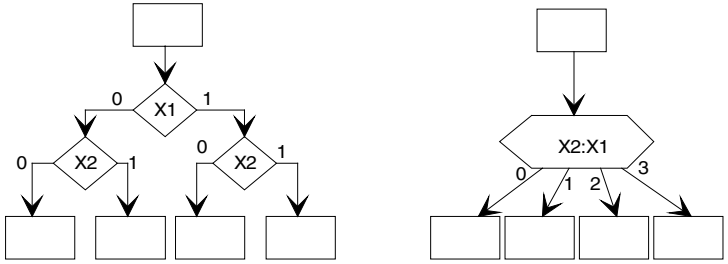


Figure 4.17 Multi-way decision box

ASM charts has the following advantages over (bubble) state diagrams:

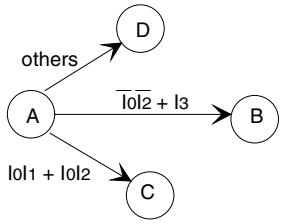


Figure 4.18 Possible conflicts

- They usually reflect the algorithms better and thus is easier for designer to understand.
- They avoid transition conflicts that could occur in state diagrams. For example the following state diagram looks all right, but in fact is illegal. The inputs $I_3I_2I_1I_0$ = 1101,1011,and 1111 will make both transition to be true. This will never occurs with ASM chart.

Here is an ASM representation of a mealy FSM for the vending machine:-

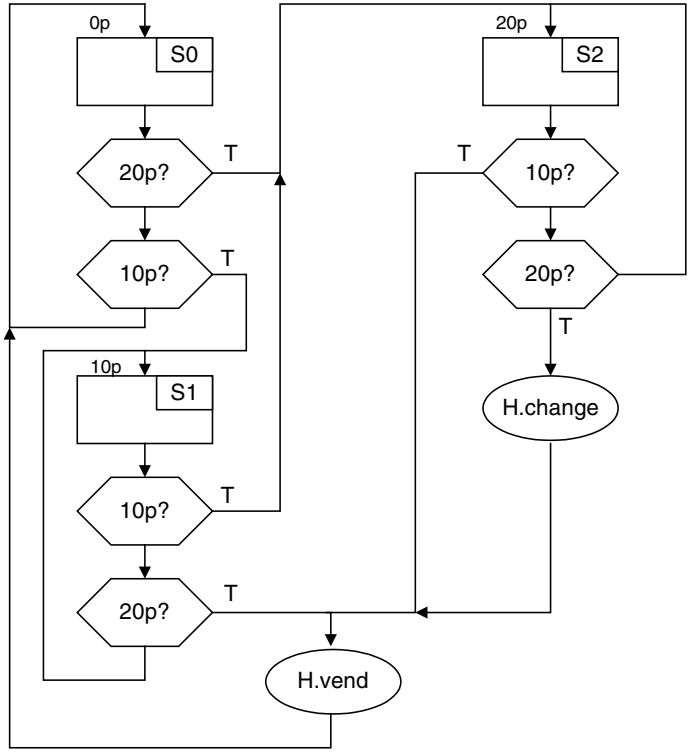


Figure 4.19 Mealy model of the vending machine as ASM chart

4.10 FSM Representation in a Hardware Description Language

We can represent the FSM in a language form such as AHDL (the HDL that you learned in your second year lab). Here is one possible AHDL description of Figure 4.19 using the in-built state machine and table constructs:

```
SUBDESIGN vend
(
   clk, reset     : INPUT;
   p_10, p_20          : INPUT;
   vend, change        : OUTPUT;
)
VARIABLE

    ss: MACHINE OF BITS (stateFF[1..0])

    WITH STATES  (s0, s1, s2, s3);

BEGIN
  ss.clk   = clk;
  ss.reset = reset;

  TABLE
    ss,  p_10,     p_20 =>   vend,  change, ss;

    s0,       0,        0    =>  0,   0, s0;
    s0,       1,        0    =>   0,  0, s1;
    s0,       0,        1    =>   0,  0, s2;
    s1,       0,        0    =>   0,  0, s1;
    s1,       1,        0    =>   0,  0, s2;
    s1,       0,        1    =>   1,  0, s0;
    s2,       0,        0    =>   0,  0, s2;
    s2,       1,        0    =>   1,  0, s0;
    s2,       0,        1    =>   1,  1, s0;
    s3,       x,        x    =>   0,  0, s0;
  END TABLE;
END;
```


4.11 State minimization

- The hardware complexity of a state machine can be improved in three ways:
  1. Minimise the number of states needed in the state machine
  2. Assign the coding to the states optimally
  3. Minimise the logic equation in the transition function and output function

  We shall now examine how the number of states can be minimized.

- There are three state minimisation methods commonly used:
  1. Merging states by observation
  2. State Partitioning
  3. Using implication tables

State Merging by observation

We have already used this method in merging two states in the vending machine example. Let us consider another example here. A circuit is required to generate a 1 output (Z) when three successive bits from the serial data input (D) are: 001, 010, 100 or 111. The state diagram is given by:

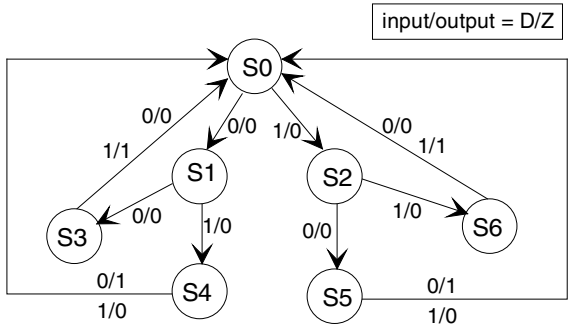


Figure 4.20 State diagram for detecting 001,010,100 and 111

The state stable is given by:

Current State	Input D 0	1
S0	S1, 0	S2, 0
S1	S3, 0	S4, 0
S2	S5, 0	S6, 0
S3	S0, 0	S0, 1

S4	S0, 1	S0, 0
S5	S0, 1	S0, 0
S6	S0, 0	S0, 1

Since S3 and S6 have the same outputs and next state, S3 and S6 are equivalent (i.e. S3 ≡ S6. Similarly, S4 ≡ S5. We can therefore eliminate rows S5 and S6:-

Current State	Input D 0	1
S0	S1, 0	S2, 0
S1	S3, 0	S4, 0
S2	~~S5~~ S4, 0	~~S6~~ S3, 0
S3	S0, 0	S0, 1
S4	S0, 1	S0, 0

Therefore the reduced state diagram becomes:

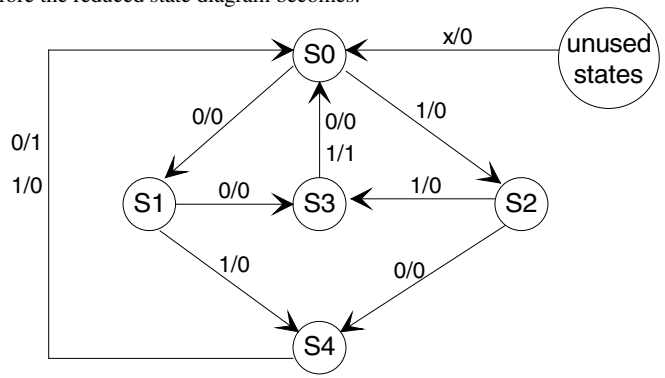


Figure 4.21 Reduced state diagram for sequence detector

Method 2: State reduction by partitioning

Let us consider a FSM described by the following state table:-

Current state	Inputs 0	1
S0	S5,0	S3,0
S1	S9,0	S2,0
S2	S0,1	S5,1

S3	S8,1	S1,1
S4	S8,0	S6,0
S5	S6,1	S0,0
S6	S9,1	S1,1
S7	S4,1	S8,1
S8	S3,1	S4,0
S9	S6,1	S0,0

Only states 5 and 9 have identical next state and outputs, and thus can be merged. However, the best solution for this problem takes only 5 states! (as can be seen later.)

A good strategy for finding the least number of state is to group together the states into sets such that: 1) states in the same set have the same outputs; 2) states in the same set have the same next state for a given input.

Step 1 - partitioning by outputs

Since equivalent states must have the same outputs, therefore we can divide the states into sets with identical next outputs. (If two states have different outputs, they are definitely not equivalent.) In our case, the sets are:

Outputs (input=0:1)	**0:0**	**1:1**	**1:0**
Set	(S0,S1,S4)	(S2,S3,S6,S7)	(S5,S8,S9)
Short form	(0,1,4)	(2,3,6,7)	(5,8,9)
Set Label	A	B	C

Step 2 - Partitioning with next states

Next, take states in each set and find out what their next states are for input=0 and input=1 separately:

Set	**Next State** *input=0*	*input=1*
A = (0,1,4)	{5,9,8} ⊂ C	{3,2,6} ⊂ B
B = (2,3,6,7)	{0,8,9,4} ⊄ any	{5,1,1,8} ⊄ any
C = (5,8,9)	{6,3,6} ⊂ B	{0,4,0} ⊂ A

Notations used here are: $\subset$ means 'belong to a set'; $\not\subset$ means 'not belong to a set'. '(' and ')' are used to enclose a partitioned set of states; '{' and '}' are used to enclose the corresponding next states.

We can see that *set B is not acceptable* because the next states of set B for input=0 is {0,8,9,4}, and it does not belong to a single set. Likewise for input=1. Therefore we must *split set B* in such a way that the next states are enclosed in other sets. In this case divide B = (2,3,6,7) → B1 = (2,7) and B2 = (3,6):

Set	Next State	
	input=0	*input=1*
A = (0,1,4)	{5,9,8} ⊂ C	{3,2,6} ⊄ any
B1 = (2,7)	{0,4} ⊂ A	{5,8} ⊂ C
B2 = (3,6)	{8,9} ⊂ C	{1,1} ⊂ A
C = (5,8,9)	{6,3,6} ⊂ B2	{0,4,0} ⊂ A

Step 3 - Repartition based on next states

Splitting set B affects other next state groups: next state for C (input=0) and A (input=1) once belonged to B. The former now belongs to B2 and therefore set C is a valid partition. Set A however is now *invalid* because its next state set {3,2,6} is now not enclosed in any partition.

We therefore repeat the action in step 2 and split set A into:

A = (0,1,4) → A1 = (0,4) and A2 = (1).

The final table becomes:

Set	Next State	
	input=0	*input=1*
A1 = (0,4)	{5,8} ⊂ C	{3,6} ⊂ B2
A2 = (1)	{9} ⊂ C	{2} ⊂ B1
B1 = (2,7)	{0,4} ⊂ A1	{5,8} ⊂ C
B2 = (3,6)	{8,9} ⊂ C	{1,1} ⊂ A2
C = (5,8,9)	{6,3,6} ⊂ B2	{0,4,0} ⊂ A1

This table shows that all next state groupings are enclosed in the partitions. Therefore it is consistent!

The final partition becomes

(0,4)	(1)	(2,7)	(3,6)	(5,8,9)
A1	A2	B1	B2	C

Now the states from each set satisfy both properties that:
1) Next outputs are the same for each state in the same set
2) Next states are the same for each state in the same set.

The state table is now reduced to:

Current state	Input	
	0	*1*
A1	C,0	B2,0
A2	C,0	B1,0
B1	A1,1	C,1
B2	C,1	A2,1
C	B2,1	A1,1

We have successfully reducing a FSM needing 10 states (4 bit state variable) to one requiring only 5 states (and a 3 bit state variable).

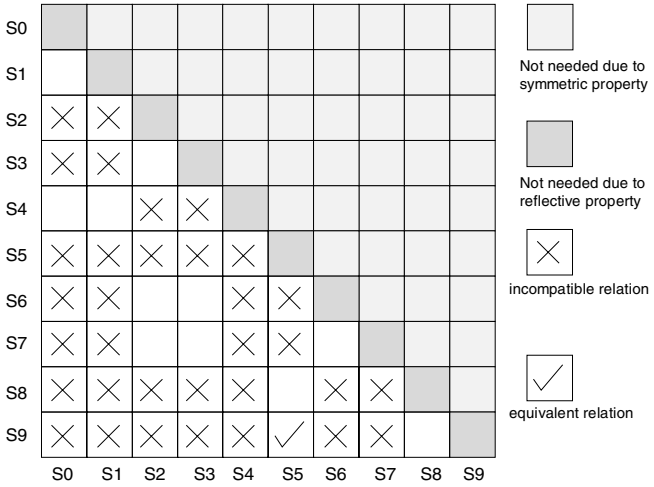
This method works well for small FSM. With large FSM and many input signals, constructing all the partition tables can become tedious. The next method is easier to computerise, and thus better for handling larger problems.

Method 3: Using Implication Tables

In the previous method, we groups states together as large sets and gradually split them up to form smaller sets after testing for *compatibility* at each stage. The approach used in the implication table method is the opposite: each state is compared with every other states for ***incompatibilities*** (i.e. two states are incompatible if their outputs are different or next states cannot be made the same).

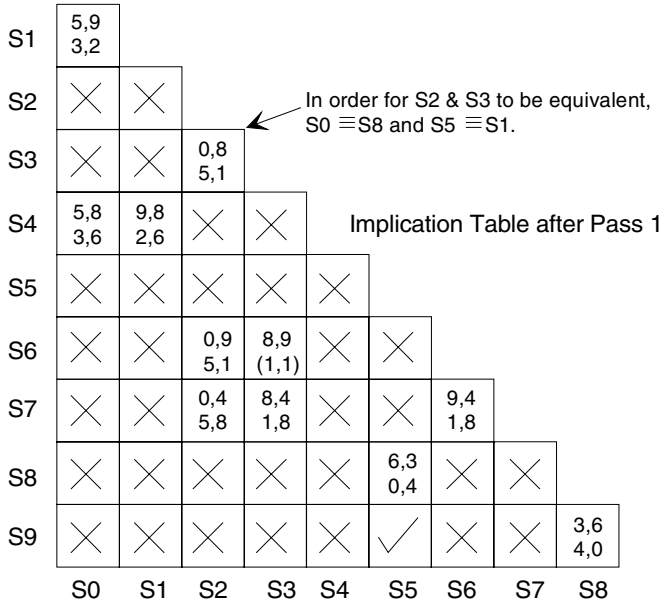
Let us consider the same example used in the previous method. We first construct a rectangular matrix as shown below relating each state to other states. Due to the ***reflexive property*** (i.e. S0 ≡ S0), the *diagonal* entries are equivalent and can thus be

eliminated. Due to the ***symmetrical property*** (i.e. if S1 ≡ S2 then S2 ≡ S1), the *upper triangle* of the matrix is the same as the *lower triangle*, and can therefore be eliminated.



Pass 1:

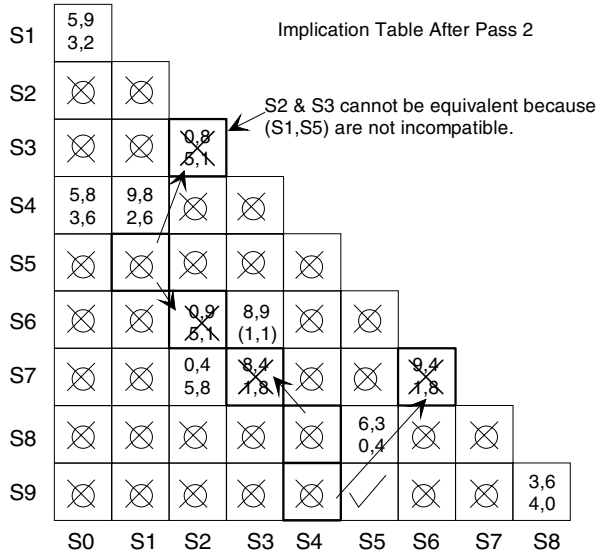
We now examine each of the remaining entries. Any state-pair with different outputs cannot be equivalent. They are called *incompatible*, and we denote that by entering a *cross* as shown above. For example (S7,S9) have outputs of (1:1, 1:0) and therefore is crossed out. We denote the equivalent states with a tick. For example S5≡S9, therefore a tick is inserted at (column S5, row S9). We do the same for the rest of the matrix starting from bottom-right corner and work towards the top-left. What remains could be (but not necessarily be) compatible.



Next we label the remaining cells with the corresponding next states for input=0 and input=1. For example, in order for S2 and S3 to be equivalent, (S0 ≡ S8 and S5 ≡ S1). This entry is thus labled as (0,8) (5,1). This ends pass1.

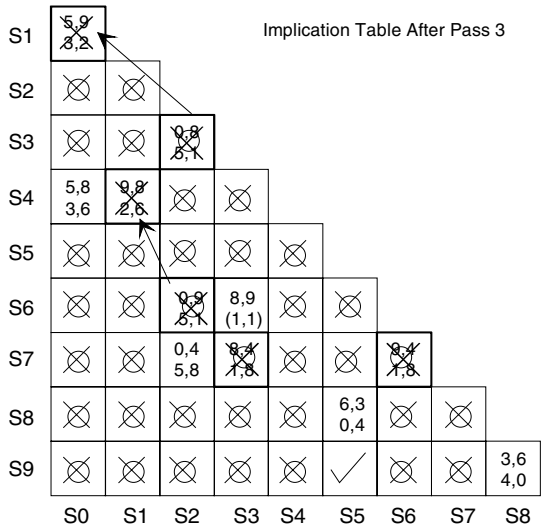
Pass2:

Next we examine each of the *crossed-out* entry and see if any non-crossed entries refer to it. For example S1 ≠ S5, therefore S2 and S3 cannot be equivalent. We can therefore cross it out too. We put a circle round the crossed-out entries which has been processed. At the end of pass2, the table would look like this:

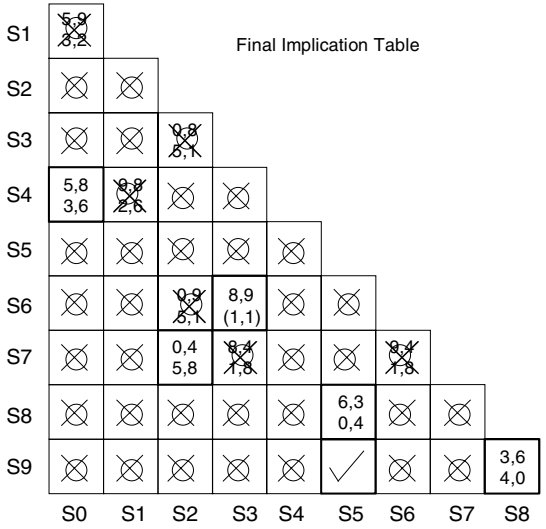


Pass 3:

During pass 2, we crossed out 4 entries. We must now examine and see if these four entries in turn causes any other entries to become *incompatible*. This leads to:



We recursively process the table until all crossed-out entries are dealt with.



The final step in the reduction procedure is to extract the equivalent states. Starting from the lower-right column, we read off the uncrossed entries as:

S8 ≡ S9 S5 ≡ S9 S5 ≡ S8 S3 ≡ S6 S2 ≡ S7 S0 ≡ S4.

These are sometimes called ***pair-wise compatibles***. We can now group them into sets of states, each with the same output states and all equivalent to one another. We can use the ***transitive relationship*** (i.e. if S1 ≡ S2 and S2 ≡ S3, then S1 ≡ S3), and deduce that: S8 ≡ S9, S5 ≡ S9, S5 ≡ S8 ⇒ S5 ≡ S8 ≡ S9.

Therefore, to cover all the initial states of S0 - S9, we must have the sets:

(S0,S4), (S1), (S2,S7), (S3,S6), (S5,S8,S9)

which is identical to the results obtained using the partitioning method.

Warning: The *partitioning* and *implication table methods* only work for ***completely specified*** FSM. For FSM with don't cares, you must be careful with combining the states as shown above because the ***transitive relationship*** may not apply. We will not consider state reduction for *incompletely specified* FSM here (because it will take too long!) We will just assume that CAD software will deal with the general problem of state reduction.

4.12 State assignment

- Once a *reduced* (or minimum) *state diagram/table* is obtained, the next step is to assign binary bit pattern to each of the states.

- State reduction has a *unique solution* independent of the technology used for implementing the circuit. State assignment, on the hand, depends on how the combinational logic function is realised (i.e. using logic gates, PLAs, ROM or other means), and on the type of state storage circuit used (i.e. D-latches or flip-flops).

- In the vending machine example, we have used an arbitrary sequential assignment. There is no guarantee that this assignment gives a good (let alone an optimum) implementation.

- To attempt to try every possible permutation of assignment is impossible. For a state table with r rows requiring a n-bit state variable, the number of permuations is:

$$N = \frac{2^n!}{(2^n - r)!}$$

- McCluskey[1] has shown that many of these assignments are merely rearrangments which make no difference to the implementation. For example, the four-row assignment (*00-01-11-10*) and (*11-10-00-01*) are essentially the same because one is just the inverse of the other. He then showed that the numnber of distinct row assignments for a table with r rows using n-bit state variables is:

$$N_D = \frac{(2^n - 1)!}{(2^n - r)!n!}$$

  Even then, the number of permutations is very large:

rows r	n-bit state variable	possible assignments
4	2	3
8	3	840
9	4	$> 10^7$

- A problem where the complex grows very quickly is difficult, if not impossible, to solve optimally. Such problem is referred to as being ***intractable*** or ***np-complete***. Solving an np-complete problem usually requires the ***use of rules*** (called ***heuristics***).

[1]McCluskey, E.J., Ungar S.H.,(1959) IRE Trans. Electron. Comp., EC-8, 439-40.

4.12.1 One-hot versus Binary coded State Assignment

- In prevous examples, we assign a binary code to each of the state. For an r-state FSM, we need only $\lceil \log_2 r \rceil$ flip-flops to store the state values.

- Sometimes it is more efficient (and convenient) to assign one latch/flip-flop per state. Therefore for an r-state circuit (r rows in state table) would require r flip-flops. This is called ***one-hot state assignment***.

- At any one time, only one bistable circuit would be set, indicating the state of the circuit. The advantages of one-hot state assignment are:
  - No state decoding logic is required
  - State transition logic tends to be simpler than using binary encoding
  - Usually results in a faster implementation if the FSM is complex

- Disadvantages are:
  - Needs many flip-flops, a problem with macrocell (SOP type) logic array, but not a problem with LUT type FPGA which is register-rich.

- One-hot state assignment is particularly efficient for programmable logic that is based on LUT or multiplexer types of structure. This is because each register is associated with a small fan-in logic element (e.g. 4 input or 5 input combinatorial block).