Sequential System Design using ASM Charts

Aims

- to discuss datapath design
- to review FSM design
- to introduce ASM charts
- to discuss the rules for ASM chart design
- to discuss the implementation of sequential systems from the ASM chart representation
- to introduce the use of map-entered variables
Typical Processor Structure

It is sensible to separate a processing system into distinct functional blocks (hierarchy, locality, modularity etc):

memory – which is a specialist system to design – we will ignore,
data path – repeated blocks of identical functionality – we will look at briefly
I/O – which has little architectural interests
control – more irregular and requires special purpose logic to form each control signal

Processor: amount of datapath/control depends upon the approach, microcode is 75% control

-datapath often has a regular structure
-control irregular

Different design approach for each!!
General form of the datapath, n-bit wide block of combinatorial logic. This could simply be synthesised from an RTL description, however, this is usually very inefficient since simplifications made by the designer may not be translated very efficiently by the compiler, this is then replicated and results in the use of a large area of Silicon and Silicon costs money.

The datapath needs to be efficient and fast!
Datapath design requires a structured approach encompassing design concepts such as hierarchy, regularity, modularity and locality.

Generally, you have n bits of data which are to be processed in n identical circuits. In addition, data operators may generally be sequenced in time or space, which leads to the notion of physically placing linked data operators to each other.

Generally, data flows in one direction, i.e. horizontally, and control signals are introduced in the orthogonal direction.

Example shown is a magnitude comparator. Operators are aligned horizontally with data bits are arranged vertically. Data is routed horizontally from operator to operator by wires (metal 2) and control information is routed vertically (metal 1).

Generally, datapaths allow optimization of the area of the layout by incorporating the regular routing strategy into the operator cell design.
Datapath Example: Bit Slice Design

Design one bit path and then replicate …

Schematic …

Layout …

Layout of data path is a repetition of a one-bit wide slice of the whole data path, the width of each ‘slice’ is the data pitch, the data flows horizontally with control lines in the orthogonal direction. Additional busses running along with data path. Translates well into the way the chip will be laid out on Silicon.

Advantages
• only need to layout single bit slice – reduced design time
• gives a compact rectangular layout
• control block can be easily placed alongside
• short interconnects between successive operations and from control block to data path – high speed

Disadvantages
• full-custom layout

CMOS design – remember we can use one metal layer for data, another for control. Such regular routing strategies help optimise the area of layout of the design.
Datapath Example: Full Adder Layout

Standard cell design using automatic place and route – connections made at bottom, no regularity

Data path design, data flows horizontally, transistors rotated – optimal Silicon area
The control logic initiates properly sequenced commands to the data path – and uses status conditions issued by the data path to determine the correct sequence of commands. Typically, the control network is irregular and requires careful design. The control block requires memory, hence, it is a sequential system not a combinatorial system.

There are a number of ways we can implement the control block, in particular, using finite state machines, random logic or structured logic. We will look at the use of finite state machines, their design and their implementation using a number of approaches.

Recall …

A combinatorial network – output depends only on the present value of the inputs
A sequential network – output depends upon both present and past inputs, i.e. it has memory!
There are two types of synchronous state machines, the Moore and Mealy machines.

The behaviour of synchronous sequential logic circuits (FSMs) can be described in a number of ways. At any given clock pulse the state of the circuit is the present state. Signals present at the input at that time are the inputs. This combination of present state and input results in two things: a transition to the next state and an output. At the next clock pulse, the process is repeated with the present state being the previous next state. There will be a limited number of states and as such the machine will cycle through the states depending upon the input conditions.

The memory element is often implement using flip-flops. If we have $n$ flip-flops to store the current state of the machine, then it has $2^n$ distinct states. Hence, the name finite state machine!
Design of Sequential Systems

In order to implement a FSM follow these steps:

1. determine the inputs and outputs and the states required – construct a *state diagram*
2. label each state with a unique number – *state assignment*
3. complete a *state transition table*
4. extract logic equations for the input and output forming logic
5. produce a complete circuit diagram
The State Diagram (1)

Used to graphically describe the behaviour of the FSM.

Each state is represented by a node in the graph (a circle) at the arrival of a clock pulse there will be a state transition any output will depend upon the state (Moore model)

Each state is assigned a unique state value (binary number).

A -> 00
B -> 01
C -> 10
D -> 11

-> 4 state FSM, need two flip-flops …

State assignments can be optimised … later lectures!
The State Diagram (2)

A typical FSM will allow external signals to control its operation

- State A
  - Transition on X=0
  - Transition on X=1 to State B
The Moore State Diagram

For a Moore machine, an output changes within a state.

Not outputs are sometimes not shown … the output is only high when indicated.
The Mealy State Diagram

For a Mealy machine, an output changes upon a state transition

There may be multiple inputs to control state transitions, and also multiple outputs derived from each state.

One reason why state diagrams are not very good, they can become very cluttered!
Example: Simple Counter (1)

Modulo-2 counter

- counts 0, 1, 2, 3, 4, 5, 6, 0 ...
- the output is given by the current state value (binary)
- 7 states – 3 flip flops needed, outputs ABC
- assign each state its equivalent binary number, i.e.
  
  0 – 000, 1 – 001, 2 – 010, 3 – 011, …, 6 – 110

Number of flip-flops

To represent the total number of required states you need at least $\log_2(\text{states})$ flip-flops, 1 state – 0, 2 states – 1, 3-4 states – 2, 5-8 states – 3, 9-16 states – 4

Undefined states are not used.
Example: Simple Counter (2)

Draw a state transition table and assign state values (state codes)

<table>
<thead>
<tr>
<th>State No.</th>
<th>Current State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example: Simple Counter (3)

Generate Logic

\[ A^+ = \bar{A}.B + \bar{A}.\bar{C} \]

\[ \begin{array}{c|cc|c|c|}
   C & 00 & 01 & 11 & 10 \\
   \hline
   0 & 1 & 0 & 0 & 1 \\
   1 & 1 & 0 & x & 0 \\
\end{array} \]

… using D-type flip-flops \( D = A^+ \)

do something similar for \( B^+ \) and \( C^+ \) – an exercise
Example: Mealy Machine (1)

Sequence detector

The circuit asserts Z when any binary input sequence ends with 101, note: the circuit does not reset, i.e.

\[
\begin{align*}
X &: 0 0 1 1 0 1 1 0 0 1 0 0 1 0 0 0 1 0 1 0 1 0 0 \ldots \\
Z &: 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 \ldots
\end{align*}
\]


We have a 3 digit sequence to detect, therefore each ‘digit’ will correspond to a state.
Example: Mealy Machine (2)

a)
If a 1 is received, we have identified the start of a possible sequence, hence, move to the next state (state S0 detects the first ‘1’)

b)
If a 0 is then received, then we have identified the second digit in the sequence, move to the next state (state S1 detects the second digit ‘0’)

c)
If a 1 is received then the sequence has been detected and we need to assert the output (state S2 detects the final digit ‘1’). As this could also be the start of another sequence, we move back to state S1 to detect the possible next ‘0’.

d)
If we are in S0 and a zero is received, then we haven’t detected a start of sequence, so stay in state S0. If we are in S1 and a ‘1’ is received then this could be the start of a new sequence, so stay in S1.

e)
If we are in S2 and a 0 is received, then we haven’t detected a correct sequence, so return to state S0 and start again.

Make sure all paths are covered, for all input combinations.
Example: Mealy Machine (3)

State Transition Table:

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Present Output</th>
<th>$A^<em>B^</em>$</th>
<th>$Z$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$X=0$</td>
<td>$X=1$</td>
<td>$X=0$</td>
<td>$X=1$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_0$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$S_0$</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

3 states, therefore we need two flip-flops $A$ and $B$, let $S_0$: $A=0, B=0$ (00), $S_1$: $A=0, B=1$ (01), $S_2$: $A=1, B=0$ (10)
Example: Mealy Machine (4)

Define logic:

<table>
<thead>
<tr>
<th>A⁺</th>
<th>X</th>
<th>B⁺</th>
<th>X</th>
<th>Z⁺</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
<td>11</td>
<td>X</td>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

A⁺ = \overline{X}B  
B⁺ = X  
Z =XA

use D-type flip-flops D_A = A⁺, D_B = B⁺
Example: Mealy Machine (5)

This is a Mealy machine
In the previous example we designed a sequence detector that asserted an output when an input pattern ending with 101 was detected. In this case the output was asserted during the transition between states. What if we wanted to assert the output whilst in a state – this is an example of a Moore machine, since the output will purely depend upon the state, not the input.
The Hardware Algorithm

If -- then --- else constructs:

e.g.

A := 0;
if C=1 then
B := A
else
B := \text{not} A
end if;

Flow Chart:

Algorithms imply a sequence of events as shown above. These are often described using flow diagrams. We shall see that these are equivalent to finite state machines in logic hardware.

A flow chart is a convenient way to specify the sequence of procedural steps and decision paths for an algorithm. A flow chart for a hardware algorithm translates the word statement to an information diagram that enumerates the sequence of operations together with the conditions necessary for their execution. A special flow chart that has been developed specifically to define digital hardware algorithms is called an algorithmic state machine (ASM) chart.

We will see that a natural extension of the flow diagram is the Algorithmic State Machine chart or ASM chart. This will allow us to describe state machines which can be represented by algorithms. Thus, if we have a high level description of a system in terms of algorithms we can synthesise the system as a set of finite state machines.
The Algorithmic State Machine Chart

- **Formal** diagrammatic representation
- Shows state-transition functions and output functions
- Easier to understand the operation of the digital system using an ASM chart
- Aids the design of a state machine for the implementation of an algorithm
- Less ambiguous than a state diagram representation

Sometimes called SM charts.

The algorithmic state machine (ASM) chart is a diagrammatic method of describing a state machine in an algorithmic form (flow chart equivalent of nested if-then-else statements) and provides a formal design methodology. This part of the course is concerned with developing this methodology. Developed at Hewlett Packard.

Flow chart representation of the operation of a digital system. State diagrams can become cluttered, ambiguous and difficult to design/optimise for large number of inputs and states.
The state box represents one of the finite set of states that the machine can exist in.
It has an entry path and an exit path.
Each state has a unique state code which represents the binary vector of the memory element outputs for that state.
A unique (meaningful) name should also be given to each state.
Within the box a list of state outputs, i.e., Moore outputs, (from the output forming logic) can be provided.
A shorthand way of providing the output values is to only list those outputs which are true.
Sometimes the state name can be placed within the state box, in this case the state name and output list are separated by a /.
State outputs are active only when the machine resides in a state.
Basic Components: The Decision Box

- Alternative symbol can be a diamond
- The decision box involves the inputs to the state machine and gives the conditions that qualify the state transitions and conditional outputs.
- The box contains a Boolean expression involving the inputs and has a true and false exit path.
- The labels T and F or Y and N may be used to identify the exits, the true and false exit paths.
- For positive logic $T = 1$ and $F = 0$ and the binary values are often used in place of the labels.
- The exit paths may lead to other decision boxes, state boxes or conditional output boxes.

Condition expression can be a single variable or a Boolean expression that is evaluated according to input conditions.
• The conditional output box describes certain outputs which only become active if certain conditions described in terms of the system inputs become true.

• The outputs are given in the conditional output list and the box is always associated with a decision box which defines the condition.

• This box is **always associated with a Mealy model state machine.**
The three basic components can be combined together to form the basic unit of an ASM chart which is the ASM block.

- Each ASM block contains exactly one state box together with the decision boxes and conditional output boxes associated with that state.
- An ASM block has only one entry path, but any number of exit paths.
- Each exit path MUST lead to another state and so is an entry path to another ASM block.
- Each ASM block describes the machine operation during the time that the machine resides in that state.
- When the ASM block is entered, the outputs specified in the output list in the state box become true.
- The conditions in the decision boxes are evaluated to determine which path (or paths) is followed through the ASM block.
- When a conditional output box is encountered along such a path, the corresponding conditional outputs become true.
- A path through an ASM block from entry to exit is called a link path.
- Each link path corresponds to a single Boolean expression which contributes to the complete expression for a conditional output or a next state function.
- Within an ASM block, the state box is the only element which indicates a time factor and all other boxes are assumed to be activated concurrently. Hence, when the block is entered, the ordering of the decision boxes and conditional output boxes is immaterial from a time point of view as all conditional functions and outputs are evaluated simultaneously, irrespective of their positions. Different from a program or signal flowchart.

Note: State outputs and conditional outputs are asserted concurrently on entering a state!
Example of Link Paths

When state S1 is entered, Z1 and Z2 become 1.

• If X1=0, Z3 and Z4 also become 1. If X2=0, then the exit to the next state will occur at exit path 1 (blue line); if X2=1, then the exit to the next state will occur at exit path 2 (red line)
• If X1=1 and X3=0, then Z5 also becomes 1 and the exit to the next state will occur at exit path 3 (green line)
• etc etc etc

We will discuss the importance of link paths later.

There are certain rules which must be obeyed when designing ASM charts. The first is that for every valid combination of input variables, there must be exactly one exit path defined. We will come to the others later.
ASM charts are not necessarily restricted to sequential systems.

A purely combinatorial circuit can be described using an ASM chart. There is no change of state – combinational behaviour. In this case the state box has no physical significance, but one MUST be present!
Equivalent ASM Charts

![ASM Charts Diagram]
State assignment is such that state outputs are redundant, i.e. bit3=C, bit2=B and bit1=A (outputs are derived from the outputs from the registers no output forming logic required) – same as the modulo-2 counter in the previous lecture.

What type of machine: Mealy or Moore? … answer is Moore!

Implementation is then forming the next state variables and the outputs from the current state variables and any inputs (of which there aren’t any in this example). State assignment is such that state outputs are redundant, i.e. bit3=C, bit2=B and bit1=A (outputs are derived from the outputs from the registers no output forming logic required) – same as the modulo-2 counter in the previous lecture.

What type of machine: Mealy or Moore? … answer is Moore!

Implementation is then forming the next state variables and the outputs from the current state variables and any inputs (of which there aren’t any in this example).
What effect does state assignment have?

Gray code – only 1 bit changes as the number increments

000
001
011
010
110
111
101
100

It is not essential for state assignment to be the same as the outputs – optimization … we will look at this later.

What output forming logic would be required if the state assignment was same as previous slide? … we would have to form the output code using logic. The input forming logic forming the next state would be simple.
Which is Mealy, which is Moore? How do you tell them apart?

Mealy FSM implementations can save a state but rely on careful logic design to avoid output glitches when inputs change.
Link Path Extraction (1)

How do we determine next-state functions from the ASM chart?
… the ASM chart not only depicts sequential behaviour but also provides the
functions required for synthesis – different to the state diagram.

Each link path identifies a next-state Boolean function of the input variables
– each state can have a number of link paths.

The Boolean expressions represent the next-state functions which will result
when all link paths through all ASM blocks have been identified.
Link Path Extraction (2)

Link path extraction checks the validity of each link path out of a state box …

Key objectives:

• identify boolean expressions for each link path out of a state

• check for identical paths

• check that all exit conditions are covered to prevent hung states

• check for identical states in the chart

The Boolean function for each link path is determined to ensure there are no common paths out of a state and that every condition is satisfied – i.e. there is always an exit path out of a state whatever the input conditions.

Link path extraction is also useful in determining whether there are equivalent states in an ASM chart – these can be identified because they will have the identical input and exit paths in both states, thus, one state may be removed to reduce the complexity.
Rule: All exits must be covered, i.e. ALL boolean conditions must be satisfied and have a valid link path.

Remember:

State outputs in a state box will stay active during the whole state time irrespective of the link path followed.
Conditional outputs will only become active if the link path containing the conditional output box is followed – the conditional outputs will remain active for the whole state time.

\[
\begin{align*}
X \& Y \\
0 0 & L_1 \text{ followed} \\
0 1 & L_2 \text{ followed} \\
1 0 & L_2 \text{ followed} \\
1 1 & L_2 \text{ followed}
\end{align*}
\]

- all exit conditions covered
Link Path Extraction: Example 2

C

1

W

0

B

1

A

\( C \oplus B = W \)

\( C \oplus A = \overline{W} \)

All link paths covered.

<table>
<thead>
<tr>
<th>W</th>
<th>0</th>
<th>C \rightarrow A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>C \rightarrow B</td>
</tr>
</tbody>
</table>
Link Path Extraction: Example 3

\[ A \cdot B = XY \]

\[ A \cdot \overline{C} = \overline{X} \overline{Y} \quad \text{Path 1} \]

\[ A \cdot \overline{C} = X \quad \text{Path 2} \]

\[ A \cdot C = X \overline{Y} + X = X + Y \]

All link paths covered.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>A -&gt; B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A -&gt; C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>A -&gt; C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>A -&gt; C</td>
</tr>
</tbody>
</table>
Link Path Extraction: Example 4

\[ B \rightarrow B = YZ \quad \text{Path 1} \]
\[ B \rightarrow B = YZW \quad \text{Path 2} \]
\[ B \rightarrow A = YZW \]
\[ B \rightarrow C = \overline{Y} \]

All link paths covered.

<table>
<thead>
<tr>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B \rightarrow C</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>B \rightarrow C</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>B \rightarrow A</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>B \rightarrow B</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>B \rightarrow C</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>B \rightarrow C</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B \rightarrow B</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>B \rightarrow B</td>
</tr>
</tbody>
</table>
We have two next states which follow from the same initial state – which is taken?

Rule: For every valid combination of input variables there must be only one exit path (link path) from the state to a single next state.
Ambiguity is less obvious. If both X and Y are true then there is a unique state transition A->C. However, if either X or Y are false then two next states are indicated.
Invalid ASM Structures (3)

Path to B is invalid.

To enter state B requires X to be both true and false, XX'
No path to B.

Again, to enter state B X would have to be both true and false, i.e. $XYX'$
More difficult this one …

Extract the link path from A to B, i.e. $XY'(X'+Y)=XY'X'+XY'Y=0$, which can never be true.
Feedback Loops in ASM Charts

Can you see any problems here?

What if Y is 0? … no internal feedback within an ASM block

What is the next state when Y=0? … it is undefined.
Parallel structure … many link paths but only one next state

Serial structure … one path only to the next state

An ASM block can have several parallel paths that lead to the same exit path, and more than one of these paths can be active during the state time – this purely determines the conditional outputs, not the state transition. Such a structure would be invalid for a software flow chart – but this is not the case for the ASM chart. An equivalent serial ASM chart can be constructed, here there is only ever one path to the next state, although again multiple conditional outputs can be active. Due to the concurrent nature of the ASM block, both structures are identical, as the conditional outputs will be initiated simultaneously when the state is entered – there is no delay.
It may be possible to simply your ASM chart design by allowing ASM blocks to share decision boxes or conditional output boxes, thereby avoiding duplication. The fact that ASM blocks will overlap is not a problem since they can still be identified as link paths are extracted.

The decision box 'W' belongs to both states B and C.
ASM Chart Design: Mealy Example (1)

How do we produce an ASM chart design …

… previously we have used state diagram (bubble graph) representations.

Recall the 101 sequence detector Mealy machine example …

![ASM Chart Design: Mealy Example (1)](image)

The circuit asserts Z when any binary input sequence ends with 101, note: the circuit does not reset.

Recall: We need to remember receiving the three symbols 1->0->1, which implies three states. The output is asserted when at the same time the final ‘1’ in the sequence is detected.

S₀ – waiting for a ‘1’ state
S₁ – waiting for a ‘0’ state
S₂ – waiting for the second ‘1’ state

• Three states, therefore, three ASM blocks are required
• No Moore outputs, if they were they would be placed in the state boxes
• Mealy outputs appear as conditional output boxes
• In this example, each ASM block only has one decision box associated with it since only one input variable is tested

Remember … this is a Mealy machine example … what do we know about Mealy machines and ASM charts????
ASM Chart Design: Mealy Example (2)

a) If a 1 is received, we have identified the start of a possible sequence, hence, move to the next state (state S₀ detects the first ‘1’).
b) If a 0 is then received, then we have identified the second digit in the sequence, move to the next state (state S1 detects the second digit ‘0’).
c) If a 1 is received then the sequence has been detected and we need to assert the output (state S2 detects the final digit ‘1’). As this could also be the start of another sequence, we move back to state S1 to detect the possible next ‘0’.

S₂ detects the second ‘1’ in the sequence – assert the output
d) If we are in S0 and a zero is received, then we haven't detected a start of sequence, so stay in state S0. If we are in S1 and a '1' is received then this could be the start of a new sequence, so stay in S1.
e) If a ‘0’ is detected in S₂ then return to S₀ to start all over again.

If we are in S₂ and a 0 is received, then we haven’t detected a correct sequence, so return to state S₀ and start again.

Make sure all paths are covered, for all input combinations.
Recall the 101 sequence detector Moore machine example …

The circuit asserts Z (state output) when a binary input sequence ends with 101, note: the circuit does not reset.

Recall: We need to remember receiving the three symbols 1->0->1, which implies at least three states. We need an extra state to assert the output since we are dealing with a Moore machine.

$S_0$ – waiting for the first ‘1’ in sequence state
$S_1$ – waiting for the ‘0’ in sequence state
$S_2$ – waiting for the final ‘1’ in sequence state
$S_3$ – sequence received state - assert output

• Four states, therefore, four ASM blocks are required
• Moore machine – no conditional outputs, outputs only asserted within state boxes
• In this example, each ASM block only has one decision box associated with it since only one input variable is tested
If a 1 is received, we have identified the start of a possible sequence, hence, move to the next state (state S0 detects the first ‘1’).
ASM Chart Design: Moore Example (3)

b) 

S1 detects the ‘0’ in the middle of the sequence, if it is received then move to S2

b) 

If a 0 is then received, then we have identified the second digit in the sequence, move to the next state (state S1 detects the second digit ‘0’)
ASM Chart Design: Moore Example (4)

c)

If a ‘1’ is detected in $S_1$ then stay in $S_1$
… it could be a new sequence!

c)
Stay in S1 for ‘1’s received … could be the start of a new sequence.
If a ‘0’ is detected in $S_2$ then a valid sequence has not been received, start again!

If when in $S_2$ a ‘0’ is received, then the sequence was invalid so we start again and return to $S_0$. 
If a ‘1’ is detected in S₂ then a valid sequence has been received, enter S₃ and assert the output in S₃.

If in S₂ a ‘1’ is received, then we have identified the sequence … Moore example, so we can only assert the output in the next state S₃.
If a ‘0’ is detected in S₃ then it could be the ‘0’ in the middle of a new (overlapping) sequence, return to S₂.

If whilst in S₃ a ‘0’ is received, it could be the middle of another sequence, so we return to S₂.
If a ‘1’ is detected in S₃, then it could be the start of a new sequence (overlapping) sequence, return to S₁.

If in S₃ a ‘1’ is received, then it could be the start of a new sequence so we move to S₁.
Mealy v. Moore Designs

Which do we use in our design?

*Mealy* – reduced number of states, simplified ASM chart …

… consider the effects of asynchronous inputs!

*Moore* – increased number of states – is this a problem? …

… truly synchronous though!

Mealy, what effect would fluctuating asynchronous inputs have on the next state and output forming logic?
Realisation of FSM Designs

1. determine the inputs and outputs and the states required
   – construct an **ASM chart**

2. label each state with a unique number – **state assignment**
   (optimise – minimum state locus)

3. complete a **state table**

4. extract logic equations for the input and output forming logic
   (map entered variables)

5. produce a complete circuit diagram
Example: Sequence Detector (1)

Design a system to detect the occurrence of the sequence:

00 00 11 10

on two inputs $X_1$ and $X_2$ and output a binary 1 when the sequence is detected.

We will use this example to illustrate the ASM synthesis process.

A system is required to detect the occurrence of sequence of pairs of inputs 00, 00, 11, 10 on two inputs and give an output 1 during the final combination of the detected sequence. Overlapping sequences are detected.

Mealy system since the output is to be initiated at the point when the final combination in the sequence is detected.

We will implement the system as a single finite state machine.
Example: Sequence Detector (2)

$S_1$ detects the first '00' on inputs $X_1X_2$

... if not return to $S_1$
Example: Sequence Detector (3)

$S_2$ detects the second ‘00’ on inputs $X_1X_2$

... if not return to $S_1$
Example: Sequence Detector (4)

S₃ detects ‘11’ on inputs X₁X₂

... if ’00’ is detected then it may be the second in the sequence, so return to S₃

... otherwise return to S₁
Example: Sequence Detector (5)

$S_4$ detects ’10’ on inputs $X_1, X_2$, assert the output and return to $S_1$

... if ’00’ is detected then it may be the first in the sequence, so return to $S_2$

... otherwise return to $S_1$

Note we have labelled the states but have not yet allocated the state vectors.
Example: Sequence Detector (6)

Alternative representation
...
put Boolean expressions
in the decision boxes
Example: Sequence Detector (7)

<table>
<thead>
<tr>
<th>Present State</th>
<th>X1</th>
<th>X2</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>S2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td>0</td>
<td>S3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>S3</td>
<td>0</td>
<td>0</td>
<td>S3</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>S4</td>
</tr>
<tr>
<td>S4</td>
<td>0</td>
<td>0</td>
<td>S2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>S1</td>
</tr>
</tbody>
</table>

Check that all paths are valid to ensure the SM chart is correct!
State Assignment

We next have to assign a unique code (state code/vector) to each state …

- Most difficult step to solve
- Different assignments give different solution complexities
- Any assignment providing a unique code for each state leads to a valid synthesis
- State code represents the condition of the state register in hardware.

Recall, the states of the machine will be represented by the condition of a state register which, in hardware synthesis, will be made up from a collection of memory elements such as flip-flops. The number of flip-flops used, \( k \), is chosen such that \( 2^k \) is greater than the number of states and the number of states is greater than \( 2^{k-1} \).

The state assignment process involves allotting one of the \( 2^k \) possible k-bit codes to each of the states.

Recall from the gray code converter example, the complexity of the implementation depends upon the state code assignment. We can optimise the logic by optimising the state code assignment process!
Example: Sequence Detector (8)

Arbitrary state assignments …

4 states, therefore we need 2 flip-flops

<table>
<thead>
<tr>
<th>State Name</th>
<th>Case A State Code</th>
<th>Case B State Code</th>
<th>Case C State Code</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AB</td>
<td>AB</td>
<td>AB</td>
</tr>
<tr>
<td>S_1</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>S_2</td>
<td>01</td>
<td>11</td>
<td>01</td>
</tr>
<tr>
<td>S_3</td>
<td>11</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>S_4</td>
<td>10</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

What is the optimum assignment?

This arbitrary assignment is perfectly valid but may not be the optimum assignment in terms of minimising the logic required to implement the machine. We will return to this example later.
The state map is similar to a Karnaugh map and has one cell for each possible code. The map is labelled with the state variables and the state name corresponding to each assigned code is written in the appropriate cell.
Minimum State Locus (1)

Relatively simple approach to produce a good solution by optimising state assignment.

• a state transition will cause a 1, 2, …, \( k \) bit change to occur, i.e. a state transition from code 010 to code 101 will result in 3 bits changing

• listing all state possible state transitions in the ASM and summing the number of bit changes gives the state locus

• we assign state codes to minimise the state locus, as a result the input forming logic ‘should’ be simplified

• the resulting fewer bit changes also makes the system more reliable

Minimum state locus does not necessarily produce an optimum design, but generally it produces a good design, with simplified input forming (next state) logic and possibly output forming logic. It is generally better than an ad-hoc assignment of the state codes.
Consider the state map example with the following state transitions …

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>a → b</td>
<td>000 → 010</td>
<td>1 bit change</td>
</tr>
<tr>
<td>a → c</td>
<td>000 → 011</td>
<td>2 bit changes</td>
</tr>
<tr>
<td>b → d</td>
<td>010 → 111</td>
<td>2 bit changes</td>
</tr>
<tr>
<td>c → d</td>
<td>011 → 111</td>
<td>1 bit change</td>
</tr>
<tr>
<td>d → e</td>
<td>111 → 101</td>
<td>1 bit change</td>
</tr>
<tr>
<td>e → b</td>
<td>101 → 000</td>
<td>3 bit changes</td>
</tr>
</tbody>
</table>

**Minimum State Locus (2)**

State locus: 10 bit changes

… we can reduce the state locus by using an alternative mapping.
**Minimum State Locus (3)**

Try a new state assignment …

<table>
<thead>
<tr>
<th>State Name</th>
<th>State Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>000</td>
</tr>
<tr>
<td>b</td>
<td>010</td>
</tr>
<tr>
<td>c</td>
<td>100</td>
</tr>
<tr>
<td>d</td>
<td>110</td>
</tr>
<tr>
<td>e</td>
<td>111</td>
</tr>
</tbody>
</table>

State map:

```
A
0 00 01 11 10
0 (a) – – (b)
1 (c) – (e) (d)
```

– unassigned codes
Minimum State Locus (4)

New state locus …

\[
\begin{array}{c|c|c|c}
\text{State} & \rightarrow & \text{Next State} & \text{New Value} & \text{Change} \\
\hline
a & \rightarrow & b & 000 & 010 & 1 \text{ bit change} \\
a & \rightarrow & c & 000 & 100 & 1 \text{ bit change} \\
b & \rightarrow & d & 010 & 110 & 1 \text{ bit change} \\
c & \rightarrow & d & 100 & 110 & 1 \text{ bit change} \\
d & \rightarrow & e & 110 & 111 & 1 \text{ bit change} \\
e & \rightarrow & b & 111 & 010 & 2 \text{ bit changes} \\
\end{array}
\]

State locus 7 bit changes

… this is the minimum state locus
Minimum State Locus (5)

What do you notice about the state map?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>-</td>
<td>-</td>
<td>b</td>
</tr>
<tr>
<td>1</td>
<td>c</td>
<td>-</td>
<td>e</td>
<td>d</td>
</tr>
</tbody>
</table>

… states which have transitions are adjacent on the state map.

i.e.

a → b, b → d, d → e

a → c, c → d

- unit distance

the only transition not adjacent is e → b, however, we cannot change this …

If we assign 011 to e, this makes e to b unit distance, however, it will increase the transition d to e to distance 2, and the state locus still remains at 7.
Minimum state locus only takes into account the changes in the state variables and exit state functions. However, the next state functions are also dependent on the inputs to the ASM. There are other methods of state assignment, in particular reduced dependency where an attempt can be made to assign states in such a way to simplify the next-state functions from an input contributions point of view.
Example: Sequence Detector (10)

- **S_1** – 3 link paths
- **S_2** – 3 link paths
- **S_3** – 4 link paths
- **S_4** – 3 link paths
The state table describes all state transitions along with the input conditions to cause these state transitions and the status of any outputs.

From the state table, the logic implementation can be derived!
**Example: Sequence Detector (12)**

Produce next state Karnaugh maps:

<table>
<thead>
<tr>
<th>$A^+$</th>
<th>AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1X_2$</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$B^+$</th>
<th>AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1X_2$</td>
<td>00</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$$A^+ = B\overline{X}_1\overline{X}_2 + ABX_1X_2$$

$$B^+ = \overline{X}_1\overline{X}_2$$
**Example: Sequence Detector (13)**

Produce Karnaugh maps for any outputs:

<table>
<thead>
<tr>
<th>$X_1, X_2$</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

SEQ = $X_1 \overline{X_2} \overline{AB}$
Example: Sequence Detector (14)

Use D-type flip-flops and general logic gates …
Example: Sequence Detector (15)

How about implementing the sequence detector using MUXs and D-type flip-flops?

Using AB as control for a 4:1 MUX
Map Entered Variables (1)

It is possible to display and manipulate sparse functions (numerous don’t cares in the state table) of many input variables (greater than 4) on Karnaugh maps of smaller dimension by entering variables on the Karnaugh map along with the 0’s and 1’s where appropriate – these are map-entered variables.

Map entered variables are generally used to simplify a Karnaugh map to a reduced number of functions. For example, try manipulating a Karnaugh map of 6 variables … designs can easily have more!

As an example of the use of map entered variables, see the Dice Game Case Study at the end of these notes – or see Roth, ‘Fundamentals of Logic Design’, pg. 575-583.
Consider the following function:

\[ f(X_3, X_2, X_1) = d_0 \overline{X}_3 \overline{X}_2 X_1 + d_1 \overline{X}_3 X_2 X_1 + d_2 X_3 \overline{X}_2 X_1 + d_3 X_3 X_2 X_1 + d_4 \overline{X}_3 \overline{X}_2 \overline{X}_1 + d_5 \overline{X}_3 \overline{X}_2 X_1 + d_6 X_3 \overline{X}_2 X_1 + d_7 X_3 X_2 X_1 \]

where

\[ f_i = d_j \overline{X}_3 + d_k X_3 \]

for the four possible combinations of \( d_j \) and \( d_k \), \( f_i \) will resolve to 0, 1, or \( \overline{X}_3 \) for \( f_0, f_1, f_2, f_3 \) respectively.

Thus, the 3-variable map is reduced to 2 variables by entering a variable, \( X_3 \), onto the map.
Map Entered Variables (3)

Example

\[
\begin{array}{c|cccc}
X_3 & 00 & 01 & 11 & 10 \\
\hline
0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
X_3 & 0 & 1 & f_0 = X_3 \\
0 & 1 & f_1 = 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
X_3 & 0 & 1 & f_2 = \overline{X}_3 \\
1 & 0 & f_3 = 0 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
X_3 & 0 & 1 & f = \overline{X}_1 \overline{X}_2 X_3 + X_1 \overline{X}_2 + \overline{X}_1 X_2 \overline{X}_3 \\
0 & 1 & \overline{X}_3 \\
\end{array}
\]
How do you simplify the Karnaugh map with map-entered variables to produce a logic expression?

General rule …

If a variable $P_i$ is placed in square $m_j$ of a map of function $F$, this means that $F=1$ when $P_i=1$ and the variables are chosen so that $m_j=1$. Given a map with variables $P_1, P_2, \ldots$ entered, the minimum sum-of-products form of $F$ can be found as follows:

Find a sum-of-products expression for $F$ of the form

$$F = MS_0 + P_1MS_1 + P_2MS_2 + \ldots$$

where

- $MS_0$ is the minimum sum obtained by setting $P_i=P_j=\ldots=0$;
- $MS_1$ is the minimum sum obtained by setting $P_1=1, P_j=0 (j\neq 1)$ and replacing ALL 1’s on the map with don’t cares;
- $MS_2$ is the minimum sum obtained by setting $P_2=1, P_j=0 (j\neq 2)$ and replacing ALL 1’s on the map with don’t cares;
- repeat for all.

Eh!
## Map Entered Variables (5)

<table>
<thead>
<tr>
<th>AB</th>
<th>CD</th>
<th>00</th>
<th>01</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>X</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

$s F$

<table>
<thead>
<tr>
<th>AB</th>
<th>CD</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

$s X=Y=0$  
$MS_0 = \overline{A}B + ACD$

$s X=1,Y=0$  
$MS_1 = \overline{A}D$

$s F = MS_0 + XMS_1 + YMS_2$
  
  $= \overline{A}B + ACD + \overline{X}\overline{A}D + YAD$

<table>
<thead>
<tr>
<th>AB</th>
<th>CD</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

$s X=0,Y=1$  
$MS_2 = AD$
One-Hot Assignment (1)

- each state has its own flip-flop
- usually get small excitation equations
- expensive in flip-flops (not a problem for gate array synthesis)
  - ideal if each state requires one output
  - usually can use no-hot initial state (almost one-hot)

i.e. state A – 0001, state B – 0010, state C – 0100, state D – 1000,
One-Hot Assignment (2)

ASM Chart
### One-Hot Assignment (3)

**State Table**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X&lt;sub&gt;1&lt;/sub&gt;</td>
<td>X&lt;sub&gt;2&lt;/sub&gt;</td>
<td>ABCD</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>0001</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1000</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1000</td>
</tr>
</tbody>
</table>
Thus far we have only considered systems that can be described entirely by one ASM chart. It may be possible to design the data and control parts of a system using separate ASM structures which are linked in some way to realise the complete system.

- Thus, large systems can be easily broken down into simpler ASM structures which are linked in some way – hierarchical approach again!
- The simplest systems when the ASM modules share the same clock and hence have identical state times.
- Interaction between linked state machines can be either serial or parallel.
- Linking is performed by making the outputs of one ASM form the control inputs to another and vice versa.

In the serially linked state machine illustrated, machine A will wait in state A₁ until it receives the call instruction CLLA from machine B. Machine A will then proceed through the other states (not illustrated) until it reaches state Aₙ. This will then generate the instruction CLLB which will cause machine B to continue. Meanwhile, machine A returns to its wait state – A₁. It is perfectly feasible to have multiple calls between ASMs.
Two machines may also be connected in a parallel fashion to enable concurrent operation. The simplest parallel mode is when two parallel processes are initiated together, here we can make the initial state of each process make a call for the other as indicated.
Summary

We have discussed

- datapath design – bit-slice approach
- importance of datapath design for efficient layout
- implementation of control circuits
- sequential systems design – the finite state machine
- Moore and Mealy machines
- an overview of state machine design using state diagrams
- the use of ASM charts for sequential system design
- the basic components of the ASM chart, the state box, the decision box, the conditional output box
Summary

• the ASM block
• equivalent representations of ASM charts
• the difference between Moore & Mealy implementations the rules of ASM design, equivalent structures and shared decision boxes
• the relationship between state diagrams and ASM charts
• realisation of ASM designs using state assignment techniques, the state map and state table
• the use of map entered variables for Karnaugh map simplification
• the concept of linked state machines
References


Example: Dice Game (1)
Example: Dice Game (2)

ASM Chart
Example: Dice Game (3)

Lots of don’t cares!

<table>
<thead>
<tr>
<th>Rb</th>
<th>Reset</th>
<th>D7</th>
<th>D11</th>
<th>D221</th>
<th>Eq</th>
<th>Current State ABC</th>
<th>Next State A<em>B</em>C*</th>
<th>Outputs Win Lose Roll Sp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>000</td>
<td>000</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>000</td>
<td>000</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>001</td>
<td>001</td>
<td>0 0 1 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>001</td>
<td>010</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>001</td>
<td>011</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>001</td>
<td>100</td>
<td>0 0 0 0 1 0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>010</td>
<td>010</td>
<td>1 0 0 0 0 0</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>010</td>
<td>000</td>
<td>1 0 0 0 0 0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>011</td>
<td>011</td>
<td>0 1 0 0 0 0</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>011</td>
<td>000</td>
<td>0 1 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>100</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>101</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>101</td>
<td>101</td>
<td>0 0 0 0 1 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>101</td>
<td>010</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>101</td>
<td>011</td>
<td>0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Example: Dice Game (4)

Next State \( A - A^+ \)

From the state table, \( A^+ = 1 \) when …

\[
\begin{array}{cccccccc}
A & B & C & Rb & Reset & D_7 & D_{11} & D_{231} & Eq \\
0 & 0 & 1 & 0 & - & 0 & 0 & 0 & - \\
1 & 0 & 0 & 0 & - & - & - & - & - \\
1 & 0 & 1 & 1 & - & - & - & - & - \\
1 & 0 & 0 & 0 & - & 0 & - & 0 & 0
\end{array}
\]

… use \( D_7, D_{11}, D_{2312} \) and Eq as the map entered variables.

---

only inputs which affect \( A^+ \)

E₁

E₂

---
Rules for grouping map entered variables:

Map entered variables may be grouped along with other ‘1’s and ‘X’s as previously, with the output be functionally dependent on the map entered variable. However, all ‘1’s included in a map entered variable group must be grouped separately to ensure complete inclusion in the overall next state function.
Example: Dice Game (6)

Next State B – B⁺

From the state table, B⁺ = 1 when …

\[
\begin{array}{cccccccc}
A & B & C & Rb & \text{Reset} & D_7 & D_{11} & D_{231} & \text{Eq} \\
0 & 0 & 1 & 0 & - & 1 & - & - & - \\
0 & 0 & 1 & 0 & - & 0 & 1 & - & - \\
0 & 0 & 1 & 0 & - & 0 & 0 & 1 & - \\
0 & 0 & 1 & 0 & - & 0 & 0 & 0 & - \\
0 & 1 & 0 & - & 0 & - & - & - & - \\
0 & 1 & 1 & - & 0 & - & - & - & - \\
1 & 0 & 1 & 0 & - & - & - & 1 & - \\
1 & 0 & 1 & 0 & - & 1 & - & 0 & - \\
\end{array}
\]

… use Reset, D_7, D_{11}, D_{231} and Eq as the map entered variables.
**Example: Dice Game (7)**

<table>
<thead>
<tr>
<th>CRb</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>R</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>R</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>R</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>E₃</td>
<td>R</td>
<td>X</td>
<td>E₄</td>
</tr>
</tbody>
</table>

\[ E₃ = D₇ + \overline{D₇}D₁₁ + \overline{D₇}\overline{D₁₁}D₂₃₁₂ \]
\[ = D₇ + D₁₁ + \overline{D₁₁}D₂₃₁₂ \]
\[ E₄ = Eq + D₇, Eq = Eq + D₇ \]

R = Reset

\[ B⁺ = ABCRb(D₇ + D₁₁ + D₂₃₁₂) + BReset + ACRb(D₇ + Eq) \]
Example: Dice Game (8)

Next State C – C+

From the state table, \( C^+ = 1 \) when …

\[
\begin{array}{cccccccc}
A & B & C & Rb & Reset & D_7 & D_{11} & D_{231} & Eq \\
0 & 0 & 0 & 1 & - & - & - & - \\
0 & 0 & 1 & 0 & - & - & - & - \\
0 & 0 & 1 & 0 & - & 0 & 0 & 1 \\
0 & 1 & 1 & - & 0 & - & - & - \\
1 & 0 & 0 & 1 & - & - & - & - \\
1 & 0 & 1 & 1 & - & - & - & - \\
1 & 0 & 1 & 0 & - & 1 & - & 0 \\
\end{array}
\]

… use Reset, \( D_7, D_{11}, D_{231} \) and Eq as the map entered variables.
Example: Dice Game (9)

<table>
<thead>
<tr>
<th>C^+</th>
<th>AB</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRb</td>
<td><img src="#" alt="Table" /></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>R</td>
<td>X</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>E₅</td>
<td>R</td>
<td>X</td>
<td>E₆</td>
<td></td>
</tr>
</tbody>
</table>

E₅ = \overline{D₇D₁₁}D₂₃₁₂

E₄ = D₇\overline{Eq}

C^+ = \overline{BRb} + ABC\overline{D₇D₁₁}D₂₃₁₂ + BCR\overline{Reset} + ACD\overline{Eq}