Chapter #8: Finite State Machine Design

Contemporary Logic Design

Randy H. Katz
University of California, Berkeley

June 1993

Motivation

- *Counters:* Sequential Circuits where State = Output
- *Generalizes to Finite State Machines:*
  Outputs are Function of State (and Inputs)
  Next States are Functions of State and Inputs
  Used to implement circuits that control other circuits
  "Decision Making" logic
- *Application of Sequential Logic Design Techniques*
  Word Problems
  Mapping into formal representations of FSM behavior
  Case Studies

Chapter Overview

Concept of the State Machine

- Partitioning into Datapath and Control
- When Inputs are Sampled and Outputs Asserted

Basic Design Approach

- Six Step Design Process

Alternative State Machine Representations

- State Diagram, ASM Notation, VHDL, ABEL Description Language

Moore and Mealy Machines

- Definitions, Implementation Examples

Word Problems

- Case Studies

Concept of the State Machine

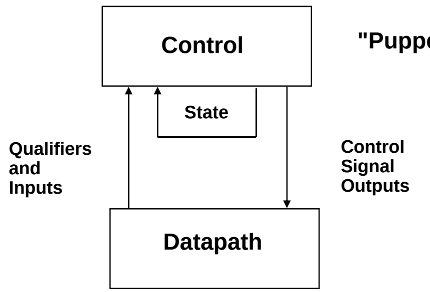
Computer Hardware = Datapath + Control

Registers
Combinational Functional
Units (e.g., ALU)
Busses

Qualifiers →

← *Control*

FSM generating sequences
of control signals
Instructs datapath what to
do next



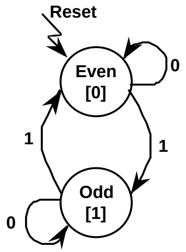
"Puppeteer who pulls the strings"

"Puppet"

Concept of the State Machine

Example: Odd Parity Checker

Assert output whenever input bit stream has odd # of 1's



State Diagram

Present State	Input	Next State	Output
Even	0	Even	0
Even	1	Odd	0
Odd	0	Odd	1
Odd	1	Even	1

Symbolic State Transition Table

Present State	Input	Next State	Output
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

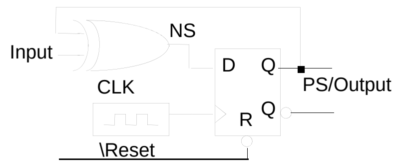
Encoded State Transition Table

Concept of the State Machine

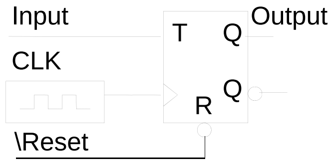
Example: Odd Parity Checker

Next State/Output Functions

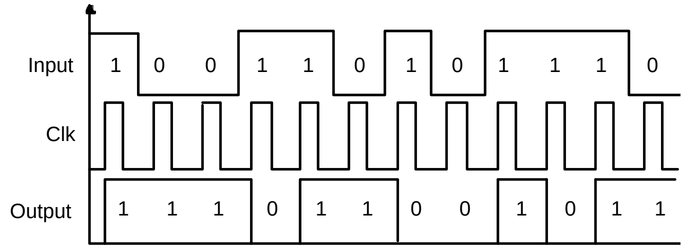
NS = PS xor PI; OUT = PS



D FF Implementation



T FF Implementation



Timing Behavior: Input 1 0 0 1 1 0 1 0 1 1 1 0

Concept of State Machine

Timing:
When are inputs sampled, next state computed, outputs asserted?

State Time: Time between clocking events

- Clocking event causes state/outputs to transition, based on inputs
- For set-up/hold time considerations:

  Inputs should be stable before clocking event

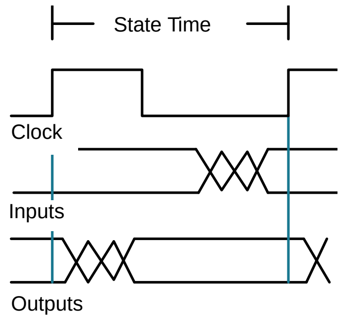
- After propagation delay, Next State entered, Outputs are stable

NOTE: Asynchronous signals take effect immediately
Synchronous signals take effect at the next clocking event

E.g., tri-state enable: effective immediately
sync. counter clear: effective at next clock event

Concept of State Machine

Example: Positive Edge Triggered Synchronous System



On rising edge, inputs sampled
outputs, next state computed

After propagation delay, outputs and
next state are stable

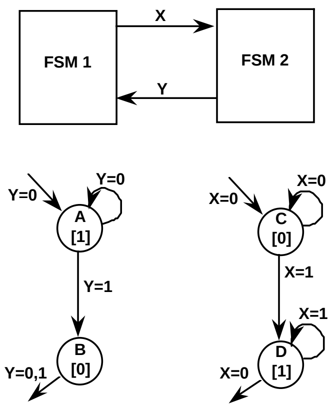
Immediate Outputs:
affect datapath immediately
could cause inputs from datapath to change

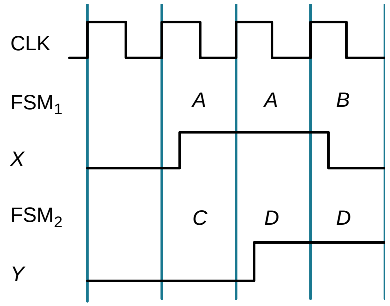
Delayed Outputs:
take effect on next clock edge
propagation delays must exceed hold times

Concept of the State Machine

Communicating State Machines

One machine's output is another machine's input





Machines advance in lock step

Initial inputs/outputs: X = 0, Y = 0

Basic Design Approach

Six Step Process

1. **Understand the statement of the Specification**
2. **Obtain an abstract specification of the FSM**
3. **Perform a state mininimization**
4. **Perform state assignment**
5. **Choose FF types to implement FSM state register**
6. **Implement the FSM**

1, 2 covered now; 3, 4, 5 covered later;
4, 5 generalized from the counter design procedure

Basic Design Approach

Example: **Vending Machine FSM**

General Machine Concept:

deliver package of gum after 15 cents deposited

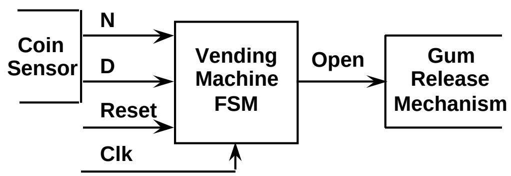
single coin slot for dimes, nickels

no change

Step 1. *Understand the problem:*

Draw a picture!

Block Diagram



Vending Machine Example

Step 2. *Map into more suitable abstract representation*

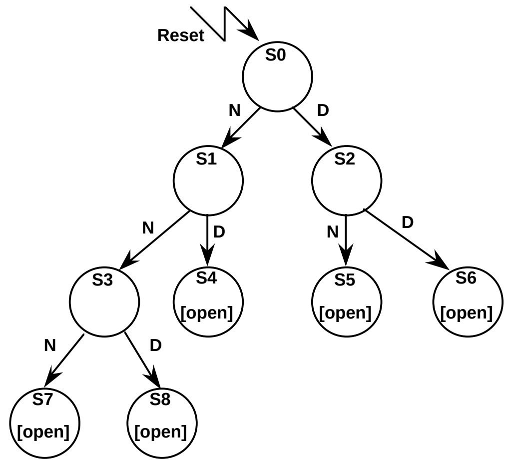
Tabulate typical input sequences:

three nickels
nickel, dime
dime, nickel
two dimes
two nickels, dime

Draw state diagram:

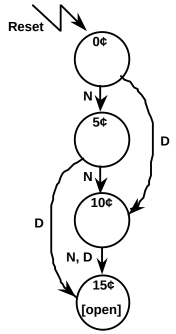
Inputs: N, D, reset

Output: open



Vending Machine Example

Step 3: State Minimization



reuse states whenever possible

Present State	Inputs D	Inputs N	Next State	Output Open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Symbolic State Table

Vending Machine Example

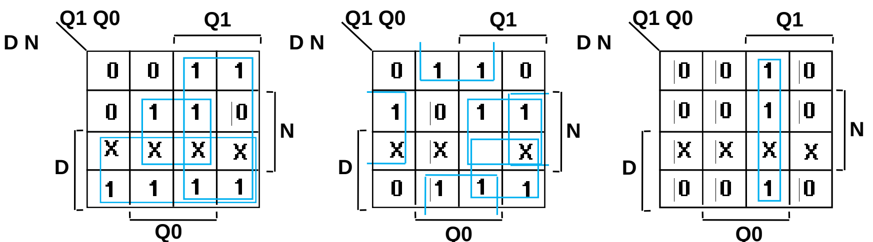
Step 4: State Encoding

Present State Q_1	Present State Q_0	Inputs D	Inputs N	Next State D_1	Next State D_0	Output Open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	X	X	X
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	X	X	X
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	X	X	X
1	1	0	0	1	1	1
		0	1	1	1	1
		1	0	1	1	1
		1	1	X	X	X

Parity Checker Example

Step 5. *Choose FFs for implementation*

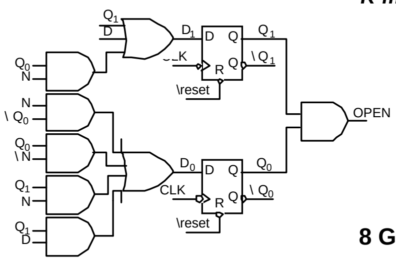
D FF easiest to use



K-map for D1

K-map for D0

K-map for Open



D1 = Q1 + D + Q0 N

D0 = N $\overline{\text{Q0}}$ + Q0 $\overline{\text{N}}$ + Q1 N + Q1 D

OPEN = Q1 Q0

8 Gates

Parity Checker Example

Step 5. *Choosing FF for Implementation*

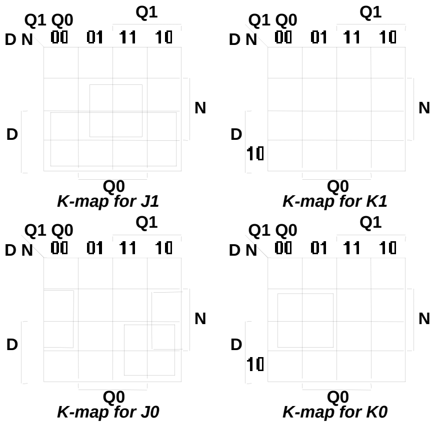
J-K FF

Present State Q_1	Present State Q_0	Inputs D	Inputs N	Next State D_1	Next State D_0	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	X	0	X
		0	1	0	1	0	X	1	X
		1	0	1	0	1	X	0	X
		1	1	X	X	X	X	X	X
0	1	0	0	0	1	0	X	X	0
		0	1	1	0	1	X	X	1
		1	0	1	1	1	X	X	0
		1	1	X	X	X	X	X	X
1	0	0	0	1	0	X	0	0	X
		0	1	1	1	X	0	1	X
		1	0	1	1	X	0	1	X
		1	1	X	X	X	X	X	X
1	1	0	0	1	1	X	0	X	0
		0	1	1	1	X	0	X	0
		1	0	1	1	X	0	X	0
		1	1	X	X	X	X	X	X

Remapped encoded state transition table

Vending Machine Example

Implementation:

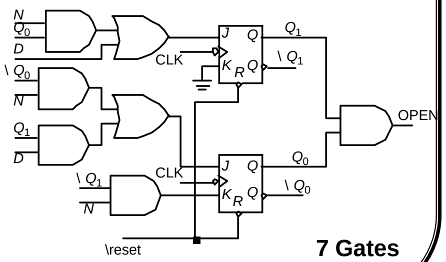


$J1 = D + Q0\ N$

$K1 = 0$

$J0 = \overline{Q0}\ N + Q1\ D$

$K0 = \overline{Q1}\ N$



7 Gates

Alternative State Machine Representations

Why State Diagrams Are Not Enough

Not flexible enough for describing very complex finite state machines

Not suitable for gradual refinement of finite state machine

Do not obviously describe an *algorithm:* that is, well specified sequence of actions based on input data

algorithm = sequencing + data manipulation

separation of control and data

Gradual shift towards program-like representations:

- Algorithmic State Machine (ASM) Notation
- Hardware Description Languages (e.g., VHDL)

Alternative State Machine Representations

Algorithmic State Machine (ASM) Notation

Three Primitive Elements:

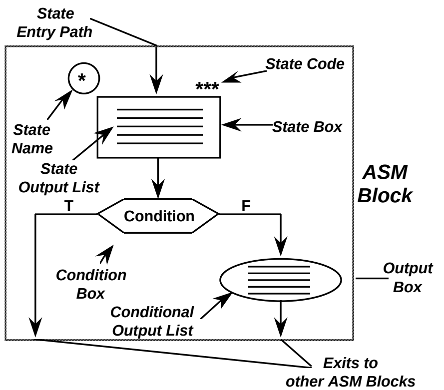
- State Box
- Decision Box
- Output Box

State Machine in one state block per state time

Single Entry Point

Unambiguous Exit Path for each combination of inputs

Outputs asserted high (.H) or low (.L); Immediate (I) or delayed til next clock



Alternative State Machine Representations

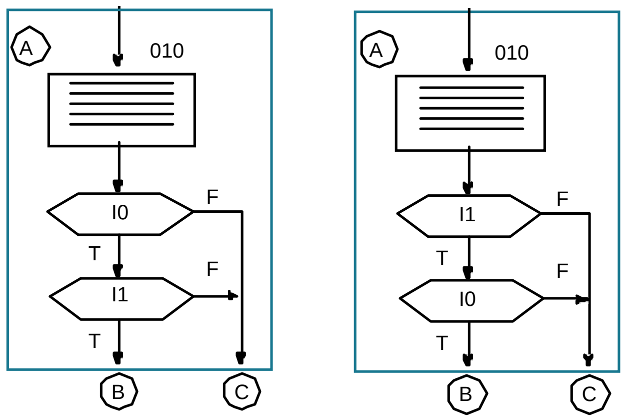
ASM Notation

Condition Boxes:

Ordering has no effect on final outcome

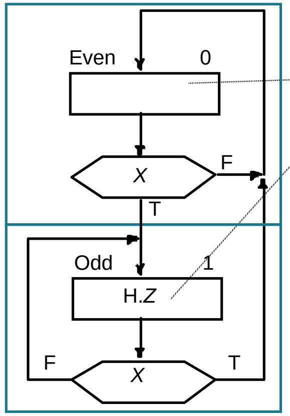
Equivalent ASM charts:

A exits to B on (I0 • I1) else exit to C



Alternative State Machine Representations

Example: Parity Checker



Trace paths to derive state transition tables

Input X, Output Z

Nothing in output list implies Z not asserted

Z asserted in State Odd

Symbolic State Table:

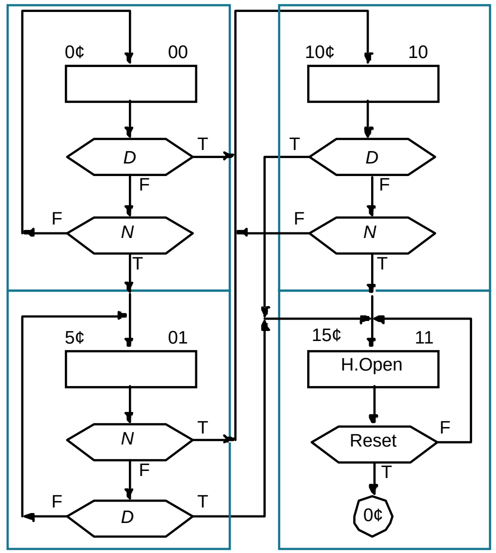
Input	*Present State*	*Next State*	*Output*
F	Even	Even	—
T	Even	Odd	—
F	Odd	Odd	A
T	Odd	Even	A

Encoded State Table:

Input	*Present State*	*Next State*	*Output*
0	0	0	0
1	0	1	0
0	1	1	1
1	1	0	1

Alternative State Machine Representations

ASM Chart for Vending Machine



Alternative State Machine Representations

Hardware Description Languages: VHDL

```
ENTITY parity_checker IS
 PORT (
  x, clk: IN BIT;
  z: OUT BIT);
END parity_checker;

ARCHITECTURE behavioral OF parity_checker IS
 BEGIN
  main: BLOCK (clk = ‘1’ and not clk’STABLE)

  TYPE state IS (Even, Odd);
  SIGNAL state_register: state := Even;

  BEGIN state_even:
   BLOCK ((state_register = Even) AND GUARD)
    BEGIN
     state_register <= Odd WHEN x = ‘1’
      ELSE Even
   END BLOCK state_even;

  BEGIN state_odd:
   BLOCK ((state_register = Odd) AND GUARD)
    BEGIN
     state_register <= Even WHEN x = ‘1’
      ELSE Odd;
   END BLOCK state_odd;

  z <= ‘0’ WHEN state_register = Even ELSE
     ‘1’ WHEN state_register = Odd;
 END BLOCK main;
END behavioral;
```

Interface Description

Architectural Body

Guard Expression

Determine New State

Determine Outputs

Alternative State Machine Representations

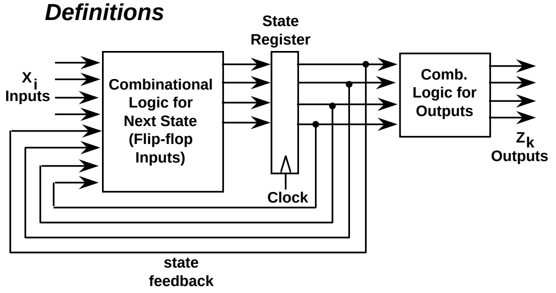
ABEL Hardware Description Language

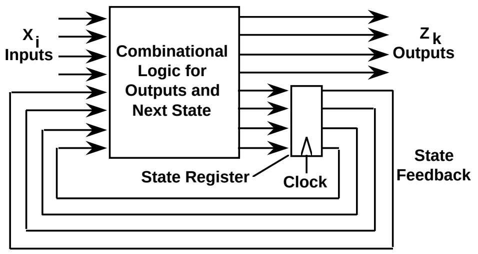
```
module parity
title 'odd parity checker state machine'
u1 device 'p22v10';

"Input Pins
 clk, X, RESET pin 1, 2, 3;

"Output Pins
 Q, Z    pin 21, 22;

 Q, Z    istype 'pos,reg';

"State registers
SREG  =  [Q, Z];
S0   =  [0, 0]; " even number of 0's
S1   =  [1, 1];  " odd number of 0's

equations
 [Q.ar, Z.ar] = RESET; "Reset to state S0

state_diagram SREG
state S0:
 if X then S1
 else S0;
state S1:
 if X then S0
 else S1;
```

```
test_vectors ([clk, RESET, X] -> [SREG])
 [0,1,.X.] -> [S0];
 [.C.,0,1] -> [S1];
 [.C.,0,1] -> [S0];
 [.C.,0,1] -> [S1];
 [.C.,0,0] -> [S1];
 [.C.,0,1] -> [S0];
 [.C.,0,1] -> [S1];
 [.C.,0,0] -> [S1];
 [.C.,0,0] -> [S1];
 [.C.,0,0] -> [S1];
end parity;
```

Moore and Mealy Machine Design Procedure

Definitions





Moore Machine

Outputs are function solely of the current state

Outputs change synchronously with state changes

Mealy Machine

Outputs depend on state AND inputs

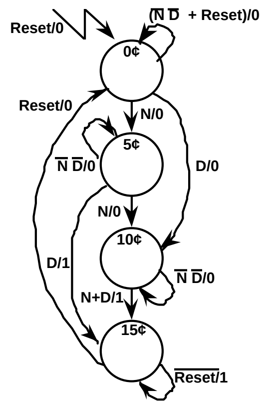
Input change causes an immediate output change

Asynchronous signals

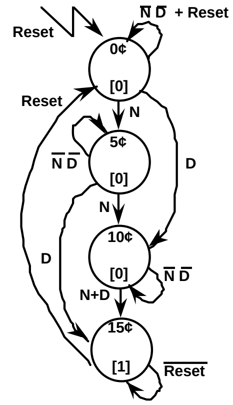
Moore and Mealy Machines

State Diagram Equivalents

Moore Machine



Mealy Machine



Outputs are associated with State

Outputs are associated with Transitions

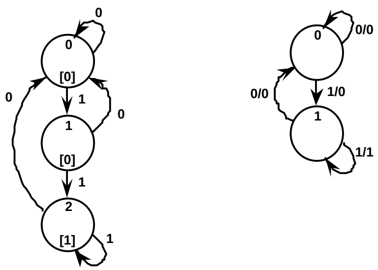
Moore and Mealy Machines

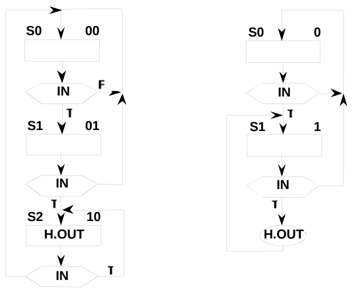
States vs. Transitions

Mealy Machine typically has fewer states than Moore Machine for same output sequence

Same I/O behavior

Different # of states



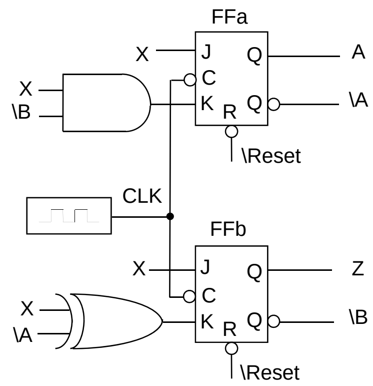


Equivalent ASM Charts

Moore and Mealy Machines

Timing Behavior of Moore Machines

Reverse engineer the following:



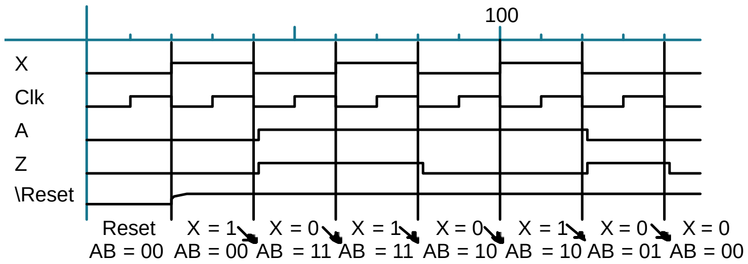
Input X
Output Z
State A, B = Z

Two Techniques for Reverse Engineering:

- **Ad Hoc: Try input combinations to derive transition table**
- **Formal: Derive transition by analyzing the circuit**

Ad Hoc Reverse Engineering

Behavior in response to input sequence 1 0 1 0 1 0:



Partially Derived State Transition Table

A	B	X	A+	B+	Z
0	0	0	?	?	0
		1	1	1	0
0	1	0	0	0	1
		1	?	?	1
1	0	0	1	0	0
		1	0	1	0
1	1	0	1	1	1
		1	1	0	1

Formal Reverse Engineering

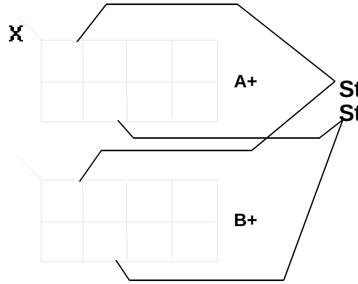
Derive transition table from next state and output combinational functions presented to the flipflops!

Ja = X   **Ka = X • $\overline{B}$**   **Z = B**
Jb = X   **Kb = X xor $\overline{A}$**

FF excitation equations for J-K flipflop:

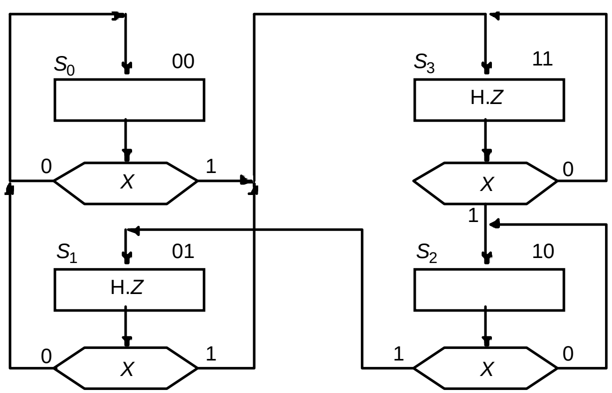
$$A+ = Ja \bullet \overline{A} + \overline{Ka} \bullet A = X \bullet \overline{A} + (\overline{X} + B) \bullet A$$
$$B+ = Jb \bullet B + Kb \bullet B = X \bullet B + (X \bullet A + X \bullet A) \bullet B$$

Next State K-Maps:



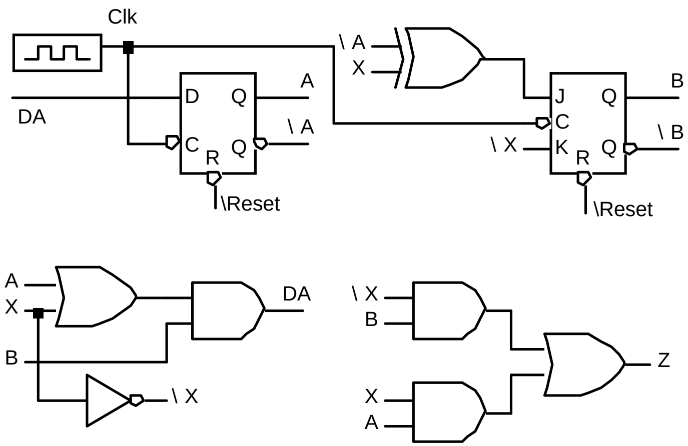
State 00, Input 0 -> State 00
State 01, Input 1 -> State 01

Complete ASM Chart for the Mystery Moore Machine



Note: All Outputs Associated With State Boxes
No Separate Output Boxes — Intrinsic in Moore Machines

Reverse Engineering a Mealy Machine



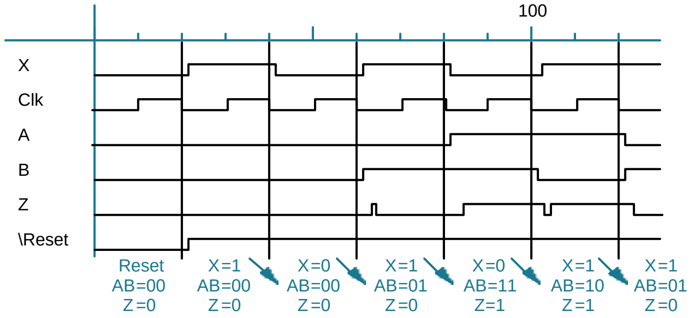
Input X, Output Z, State A, B

State register consists of D FF and J-K FF

Moore and Mealy Machine

Ad Hoc Method

Signal Trace of Input Sequence 101011:



Note glitches in Z!

Outputs valid at following falling clock edge

Partially completed state transition table based on the signal trace

A	B	X	A+	B+	Z
0	0	0	0	1	0
		1	0	0	0
0	1	0	?	?	?
		1	1	1	0
1	0	0	?	?	?
		1	0	1	1
1	1	0	1	0	1
		1	?	?	?

Moore and Mealy Machines

Formal Method

$$A+ = B \bullet (A + X) = A \bullet B + B \bullet X$$

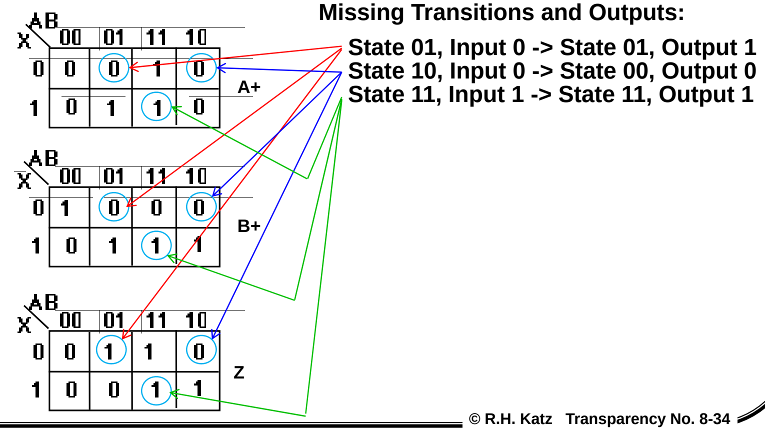
$$B+ = Jb \bullet \overline{B} + \overline{Kb} \bullet B = (\overline{A} \text{ xor } X) \bullet \overline{B} + X \bullet B$$

$$= A \bullet \overline{B} \bullet X + \overline{A} \bullet \overline{B} \bullet \overline{X} + B \bullet X$$

$$Z = A \bullet X + B \bullet \overline{X}$$

Missing Transitions and Outputs:

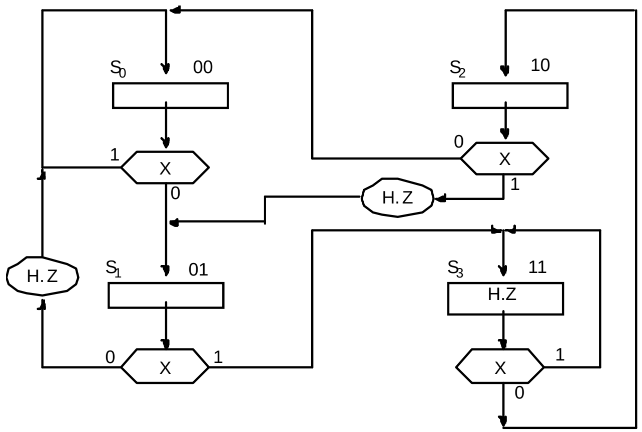
- **State 01, Input 0 -> State 01, Output 1**
- **State 10, Input 0 -> State 00, Output 0**
- **State 11, Input 1 -> State 11, Output 1**



Moore and Mealy Machines

ASM Chart for Mystery Mealy Machine

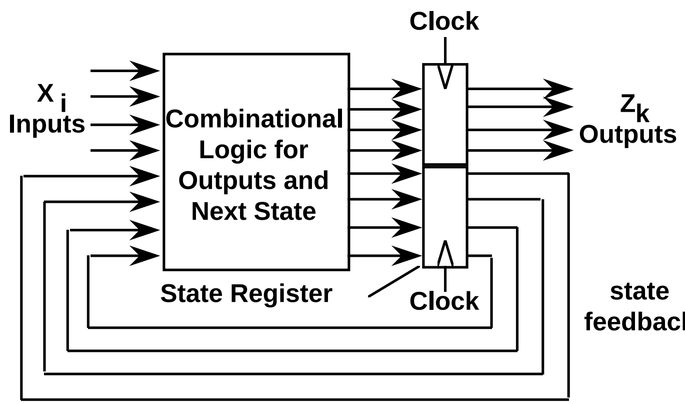
S0 = 00, S1 = 01, S2 = 10, S3 = 11



NOTE: Some Outputs in Output Boxes as well as State Boxes
This is intrinsic in Mealy Machine implementation

Moore and Mealy Machines

Synchronous Mealy Machine



latched state AND outputs

avoids glitchy outputs!

Finite State Machine Word Problems

Mapping English Language Description to Formal Specifications

Four Case Studies:

- **Finite String Pattern Recognizer**
- **Complex Counter with Decision Making**
- **Traffic Light Controller**
- **Digital Combination Lock**

We will use state diagrams and ASM Charts

Finite State Machine Word Problems

Finite String Pattern Recognizer

A finite string recognizer has one input (X) and one output (Z). The output is asserted whenever the input sequence …010… has been observed, as long as the sequence 100 has never been seen.

Step 1. Understanding the problem statement

Sample input/output behavior:

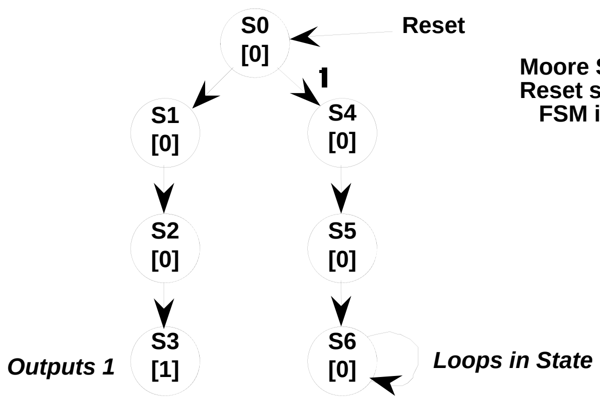
X: 00101010010…
Z: 00010101000…

X: 11011010010…
Z: 00000001000…

Finite State Machine Word Problems

Finite String Recognizer

Step 2. Draw State Diagrams/ASM Charts for the strings that must be recognized. I.e., 010 and 100.

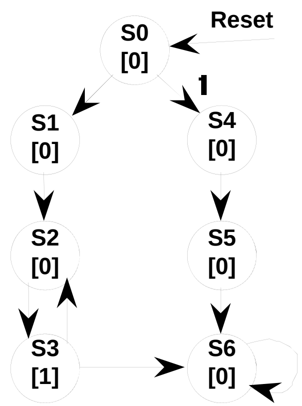


Moore State Diagram
Reset signal places FSM in S0

Finite State Machine Word Problems

Finite String Recognizer

Exit conditions from state S3: have recognized …010
if next input is 0 then have …0100!
if next input is 1 then have …0101 = …01 (state S2)

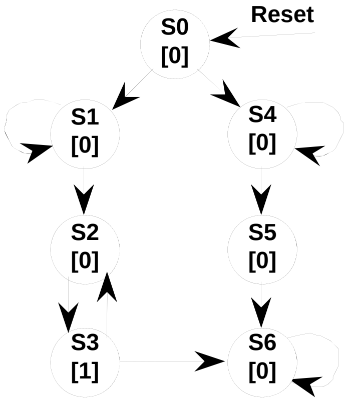


Finite State Machine Word Problems

Finite String Recognizer

Exit conditions from S1: recognizes strings of form ...0 (no 1 seen)
loop back to S1 if input is 0

Exit conditions from S4: recognizes strings of form ...1 (no 0 seen)
loop back to S4 if input is 1



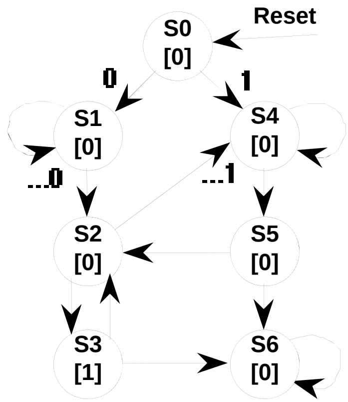
Finite State Machine Word Problems

Finite String Recognizer

S2, S5 with incomplete transitions

S2 = ...01; If next input is 1, then string could be prefix of (01)1(00)
S4 handles just this case!

S5 = ...10; If next input is 1, then string could be prefix of (10)1(0)
S2 handles just this case!



Final State Diagram

Finite State Machine Word Problems

Finite String Recognizer

```
module string
title '010/100 string recognizer state machine
  Josephine Engineer, Itty Bity Machines, Inc.'
u1 device 'p22v10';

"Input Pins
 clk, X, RESET    pin 1, 2, 3;

"Output Pins
 Q0, Q1, Q2, Z    pin 19, 20, 21, 22;

 Q0, Q1, Q2, Z istype 'pos,reg';

"State registers
SREG  =  [Q0, Q1, Q2, Z];
S0  =  [0,0,0,0]; " Reset state
S1  =  [0,0,1,0]; " strings of the form ...0
S2 = [0,1,0,0]; " strings of the form ...01
S3 = [0,1,1,1]; " strings of the form ...010
S4 = [1,0,0,0]; " strings of the form ...1
S5 = [1,0,1,0]; " strings of the form ...10
S6 = [1,1,0,0]; " strings of the form ...100

equations
 [Q0.ar, Q1.ar, Q2.ar, Z.ar] = RESET; "Reset to S0

state_diagram SREG
state S0: if X then S4 else S1;
state S1: if X then S2 else S1;
state S2: if X then S4 else S3;
state S3: if X then S2 else S6;
state S4: if X then S4 else S5;
state S5: if X then S2 else S6;
state S6: goto S6;

test_vectors ([clk, RESET, X] -> [Z])
 [0,1,.X.] -> [0];
 [.C.,0,0] -> [0];
 [.C.,0,0] -> [0];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [1];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [1];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [1];
 [.C.,0,0] -> [0];
 [.C.,0,1] -> [0];
 [.C.,0,0] -> [0];
end string;
```

ABEL Description

Finite State Machine Word Problems

Finite String Recognizer

Review of Process:

- **Write down sample inputs and outputs to understand specification**
- **Write down sequences of states and transitions for the sequences to be recognized**
- **Add missing transitions; reuse states as much as possible**
- **Verify I/O behavior of your state diagram to insure it functions like the specification**

Finite State Machine Word Problems

Complex Counter

A sync. 3 bit counter has a mode control M. When M = 0, the counter counts up in the binary sequence. When M = 1, the counter advances through the Gray code sequence.

Binary: 000, 001, 010, 011, 100, 101, 110, 111
Gray: 000, 001, 011, 010, 110, 111, 101, 100

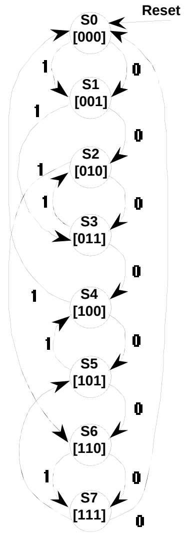
Valid I/O behavior:

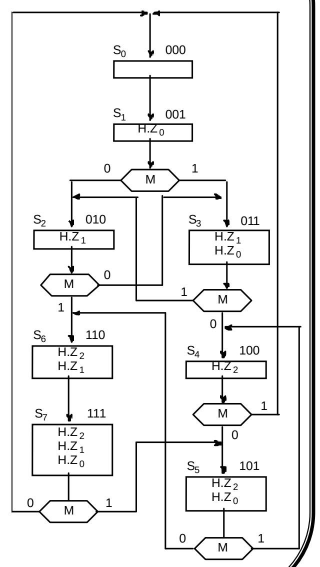
Mode Input M	Current State	Next State (Z2 Z1 Z0)
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111

Finite State Machine Word Problems

Complex Counter

One state for each output combination
Add appropriate arcs for the mode control





Finite State Machine Word Problems

Complex Counter

```
module counter
title 'combination binary/gray code upcounter
  Josephine Engineer, Itty Bity Machines, Inc.'
u1 device 'p22v10';

"Input Pins
 clk, M, RESET    pin 1, 2, 3;

"Output Pins
 Z0, Z1, Z2    pin 19, 20, 21;

 Z0, Z1, Z2   istype 'pos,reg';

"State registers
SREG  =  [Z0, Z1, Z2];
S0  =  [0,0,0];
S1  =  [0,0,1];
S2 = [0,1,0];
S3 = [0,1,1];
S4 = [1,0,0];
S5 = [1,0,1];
S6 = [1,1,0];
S7 = [1,1,1];

equations
 [Z0.ar, Z1.ar, Z2.ar] = RESET; "Reset to state S0

state_diagram SREG
state S0: goto S1;
state S1: if M then S3 else S2;
state S2: if M then S6 else S3;
state S3: if M then S2 else S4;
state S4: if M then S0 else S5;
state S5: if M then S4 else S6;
state S6: goto S7;
state S7: if M then S5 else S0;

test_vectors ([clk, RESET, M] -> [Z0, Z1, Z2])
 [0,1,.X.] -> [0,0,0];
 [.C.,0,0] -> [0,0,1];
 [.C.,0,0] -> [0,1,0];
 [.C.,0,1] -> [1,1,0];
 [.C.,0,1] -> [1,1,1];
 [.C.,0,1] -> [1,0,1];
 [.C.,0,0] -> [1,1,0];
 [.C.,0,0] -> [1,1,1];
end counter;
```

ABEL Description

Finite State Machine Word Problems

Traffic Light Controller

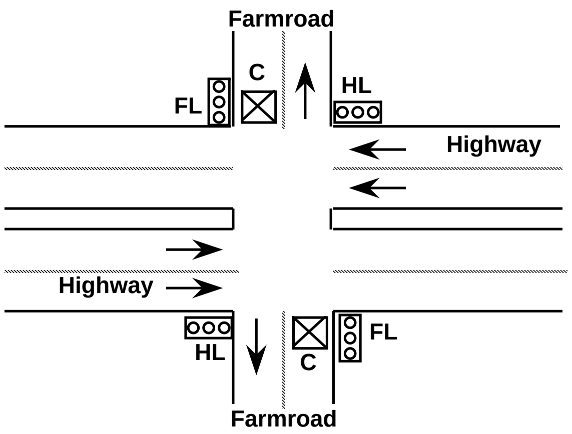
A busy highway is intersected by a little used farmroad. Detectors C sense the presence of cars waiting on the farmroad. With no car on farmroad, light remain green in highway direction. If vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green. These stay green only as long as a farmroad car is detected but never longer than a set interval. When these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green. Even if farmroad vehicles are waiting, highway gets at least a set interval as green.

Assume you have an interval timer that generates a short time pulse (TS) and a long time pulse (TL) in response to a set (ST) signal. TS is to be used for timing yellow lights and TL for green lights.

Finite State Machine Word Problems

Traffic Light Controller

Picture of Highway/Farmroad Intersection:



Finite State Machine Word Problems

Traffic Light Controller

- **Tabulation of Inputs and Outputs:**

Input Signal	*Description*
reset	place FSM in initial state
C	detect vehicle on farmroad
TS	short time interval expired
TL	long time interval expired

Output Signal	*Description*
HG, HY, HR	assert green/yellow/red highway lights
FG, FY, FR	assert green/yellow/red farmroad lights
ST	start timing a short or long interval

- **Tabulation of Unique States: Some light configuration imply others**

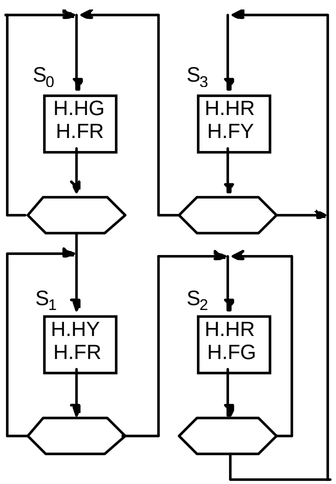
State	*Description*
S0	Highway green (farmroad red)
S1	Highway yellow (farmroad red)
S2	Farmroad green (highway red)
S3	Farmroad yellow (highway red)

Finite State Machine Word Problems

Traffic Light Controller

Refinement of ASM Chart:

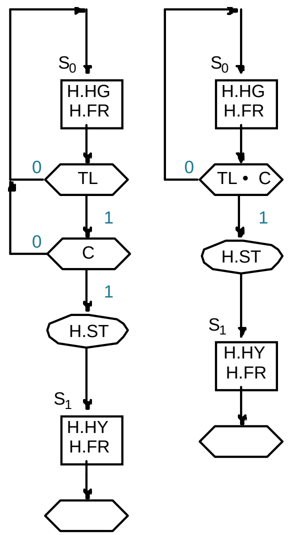
Start with basic sequencing and outputs:



Finite State Machine Word Problems

Traffic Light Controller

Determine Exit Conditions for S0:
Car waiting and Long Time Interval Expired- C • TL

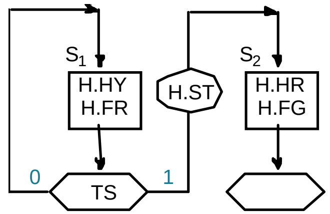


Equivalent ASM Chart Fragments

Finite State Machine Word Problems

Traffic Light Controller

S1 to S2 Transition:
- **Set ST on exit from S0**
- **Stay in S1 until TS asserted**
- **Similar situation for S3 to S4 transition**

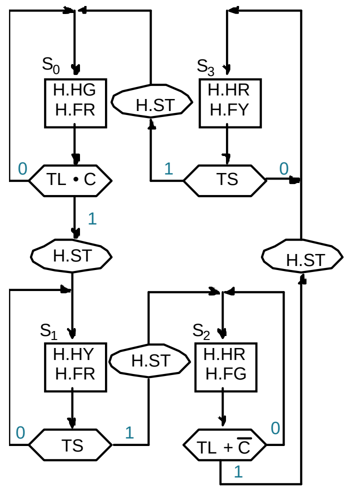


Contemporary Logic Design
Finite State Machine Design

Finite State Machine Word Problems

Traffic Light Controller

S2 Exit Condition: no car waiting OR long time interval expired



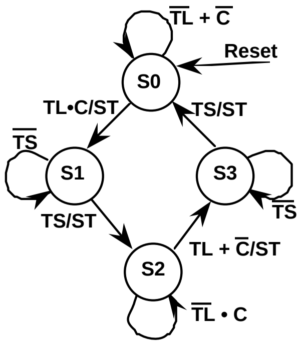
Complete ASM Chart for Traffic Light Controller

Contemporary Logic Design
Finite State Machine Design

Finite State Machine Word Problems

Traffic Light Controller

Compare with state diagram:



S0: HG

S1: HY

S2: FG

S3: FY

Advantages of State Charts:

- **Concentrates on paths and conditions for exiting a state**
- **Exit conditions built up incrementally, later combined into single Boolean condition for exit**
- **Easier to understand the design as an algorithm**

Contemporary Logic Design
Finite State Machine Design

Finite State Machine Word Problems

Traffic Light Controller

```
module traffic
title 'traffic light FSM'
u1 device 'p22v10';

"Input Pins
 clk, C, RESET, TS, TL
pin   1, 2,     3,  4,  5;

"Output Pins
 Q0, Q1, HG, HY, HR,
 FG, FY, FR, ST
pin 14, 15, 16, 17, 18,
    19, 20, 21, 22;

 Q0, Q1 istype 'pos,reg';
 ST, HG, HY, HR,
 FG, FY, FR istype 'pos,com';

"State registers
SREG  =  [Q0, Q1];
S0  =  [ 0,  0];
S1  =  [ 0,  1];
S2 = [ 1,  0];
S3 = [ 1,  1];

equations
 [Q0.ar, Q1.ar] = RESET;
 HG = !Q0 & !Q1;
 HY = !Q0 & Q1;
 HR = (Q0 & !Q1) # (Q0 & Q1);
 FG = Q0 & !Q1;
 FY = Q0 & Q1;
 FR = (!Q0 & !Q1) # (!Q0 & Q1);

state_diagram SREG
state S0:  if (TL & C) then S1 with ST = 1
   else S0 with ST = 0
state S1: if TS then S2 with ST = 1
   else S1 with ST = 0
state S2:  if (TL # !C) then S3 with ST = 1
   else S2 with ST = 0
state S3:  if TS then S0 with ST = 1
   else S3 with ST = 0

test_vectors
([clk,RESET, C, TS, TL]->[SREG,HG,HY,HR,FG,FY,FR,ST])
 [.X.,   1,.X.,.X.,.X.]->[ S0, 1, 0, 0, 0, 0, 1, 0];
 [.C.,   0,  0,  0,  0]->[ S0, 1, 0, 0, 0, 0, 1, 0];
 [.C.,   0,  1,  0,  1]->[ S1, 0, 1, 0, 0, 0, 1, 0];
 [.C.,   0,  1,  0,  0]->[ S1, 0, 1, 0, 0, 0, 1, 0];
 [.C.,   0,  1,  1,  0]->[ S2, 0, 0, 1, 1, 0, 0, 0];
 [.C.,   0,  1,  0,  0]->[ S2, 0, 0, 1, 1, 0, 0, 0];
 [.C.,   0,  1,  0,  1]->[ S3, 0, 0, 1, 0, 1, 0, 0];
 [.C.,   0,  1,  1,  0]->[ S0, 1, 0, 0, 0, 0, 1, 0];
end traffic;
```

ABEL Description

Finite State Machine Word Problems

Digital Combination Lock

"3 bit serial lock controls entry to locked room. Inputs are RESET, ENTER, 2 position switch for bit of key data. Locks generates an UNLOCK signal when key matches internal combination. ERROR light illuminated if key does not match combination. Sequence is: (1) Press RESET, (2) enter key bit, (3) Press ENTER, (4) repeat (2) & (3) two more times."

Problem specification is incomplete:

- how do you set the internal combination?
- exactly when is the ERROR light asserted?

Make reasonable assumptions:

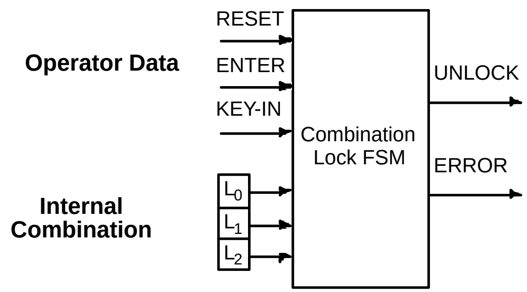
- hardwired into next state logic vs. stored in internal register
- assert as soon as error is detected vs. wait until full combination has been entered

Our design: registered combination plus error after full combination

Finite State Machine Word Problems

Digital Combination Lock

Understanding the problem: draw a block diagram ...



Inputs:
Reset
Enter
Key-In
L0, L1, L2

Outputs:
Unlock
Error

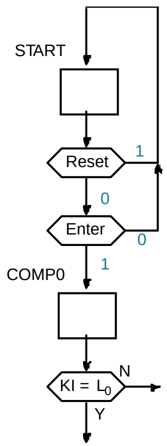
Finite State Machine Word Problems

Digital Combination Lock

Enumeration of states:

what sequences lead to opening the door?
error conditions on a second pass ...

START state plus three key COMParison states



START entered on RESET

Exit START when ENTER is pressed

Continue on if Key-In matches L0

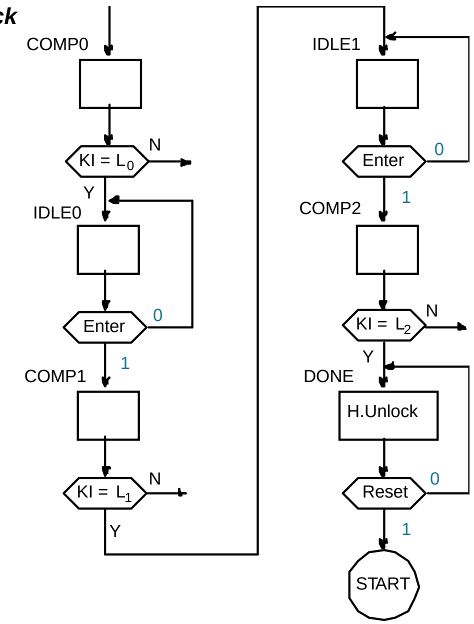
Finite State Machine Word Problems

Digital Combination Lock

Path to unlock:

Wait for Enter Key press

Compare Key-IN

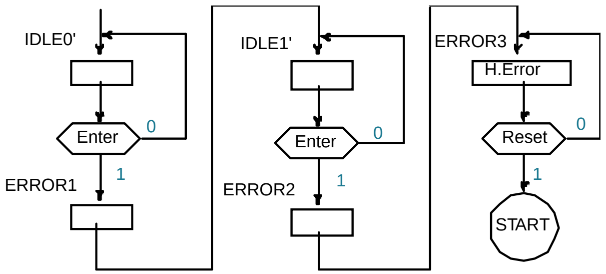


Finite State Machine Word Problems

Digital Combination Lock

Now consider error paths

Should follow a similar sequence as UNLOCK path, except asserting ERROR at the end:



COMP0 error exits to IDLE0'

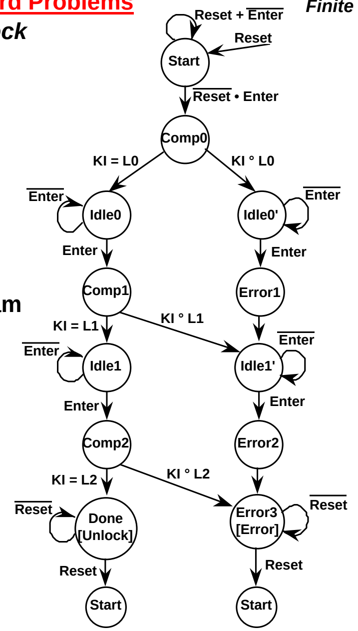
COMP1 error exits to IDLE1'

COMP2 error exits to ERROR3

Finite State Machine Word Problems

Digital Combination Lock

Equivalent State Diagram



Finite State Machine Word Problems

Combination Lock

```
module lock
title 'comb. lock FSM'
u1 device 'p22v10';

"Input Pins
clk, RESET, ENTER, L0, L1, L2, KI
pin   1, 2, 3, 4, 5, 6, 7;

"Output Pins
Q0, Q1, Q2, Q3, UNLOCK, ERROR
pin 16, 17, 18, 19, 14, 15;

Q0, Q1, Q2, Q3 istype 'pos,reg';
UNLOCK, ERROR  istype 'pos,com';

"State registers
SREG   =  [Q0, Q1, Q2, Q3];
START   =  [ 0,  0,  0,  0];
COMP0   =  [ 0,  0,  0,  1];
IDLE0  = [ 0,  0,  1,  0];
COMP1  = [ 0,  0,  1,  1];
IDLE1  = [ 0,  1,  0,  0];
COMP2  = [ 0,  1,  0,  1];
DONE  = [ 0,  1,  1,  0];
IDLE0p  = [ 0,  1,  1,  1];
ERROR1  = [ 1,  0,  0,  0];
IDLE1p  = [ 1,  0,  0,  1];
ERROR2  = [ 1,  0,  1,  0];
ERROR3  = [ 1,  0,  1,  1];

equations
 [Q0.ar, Q1.ar, Q2.ar, Q3.ar] = RESET;
 UNLOCK = !Q0 & Q1 & Q2 & !Q3;"asserted in DONE
 ERROR = Q0 & !Q1 & Q2 & Q3;  "asserted in ERROR3

state_diagram SREG
state START: if (RESET # !ENTER)
             then START else COMP0;
state COMP0: if (KI == L0) then IDLE0 else IDLE0p;
state IDLE0: if (!ENTER) then IDLE0 else COMP1;
state COMP1: if (KI == L1) then IDLE1 else IDLE1p;
state IDLE1: if (!ENTER) then IDLE1 else COMP2;
state COMP2: if (KI == L2) then DONE else ERROR3;
state DONE:  if (!RESET) then DONE else START;
state IDLE0p:if (!ENTER) then IDLE0p else ERROR1;
state ERROR1:goto IDLE1p;
state IDLE1p:if (!ENTER) then IDLE1p else ERROR2;
state ERROR2:goto ERROR3;
state ERROR3:if (!RESET) then ERROR3 else START;

test_vectors

end lock;
```


Chapter Review

Basic Timing Behavior an FSM

- when are inputs sampled, next state/outputs transition and stabilize
- Moore and Mealy (Async and Sync) machine organizations
  outputs = F(state) vs. outputs = F(state, inputs)

First Two Steps of the Six Step Procedure for FSM Design

- understanding the problem
- abstract representation of the FSM

Abstract Representations of an FSM

- ASM Charts, Hardware Description Languages

Word Problems

- understand I/O behavior; draw diagrams
- enumerate states for the "goal"; expand with error conditions
- reuse states whenever possible