Finite State Machines (FSMs)

Dr. Cyril Prasanna Raj P.

Director – Angstromers Engg. Services

Professor – Dept. of ECE

CIT, Bangalore

Introduction

- RISC-V is often used in embedded systems
  - Most RISC-V platforms (e.g., SiFive, GD32VF103, ESP32-C3) are microcontrollers or small SoCs, where:
    - Applications run without an OS (bare-metal), or with a small RTOS.
    - Code needs to be predictable, modular, and efficient.

- FSM-based coding is **highly appropriate and effective for both hardware and software development** in the RISC-V ecosystem. It's especially beneficial for clarity, control flow, and reliability — all critical in RISC-V's modular, open-source philosophy.

- FSM-based C programming is very suitable and recommended for developing structured, responsive, and maintainable applications on RISC-V platforms. It's especially powerful in embedded, real-time, and low-level applications.

FSM Examples

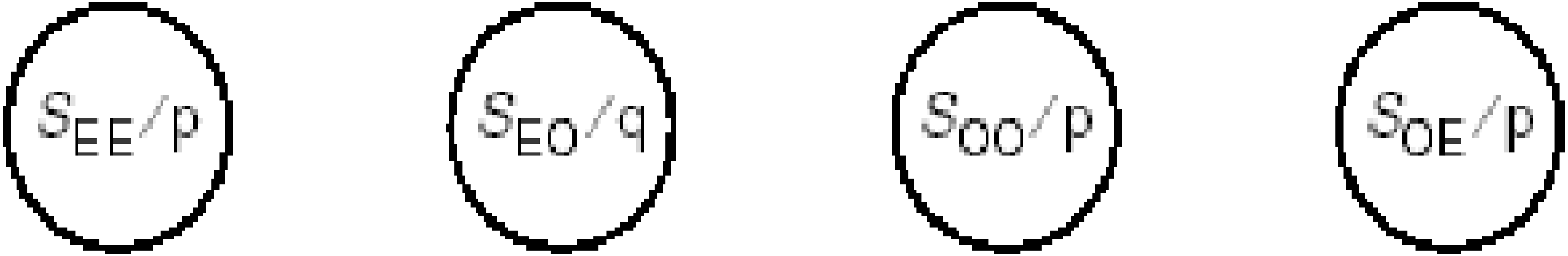
- Consider the following system description
  - A sequential system has
    - One input = { **a,b,c** }
    - One output = { **p,q** }
- Output is
  - **q** when input sequence has even # of **a**’s and odd # of **b**’s
  - **p** otherwise

FSM Example

- We begin forming a state machine for the system description by reviewing the possible states. In addition, assign each state name.
- S_{EE}: even # of a’s and even # of b’s/output is p
- S_{EO}: even # of a’s and odd # of b’s/output is q
- S_{OO}: odd # of a’s and odd # of b’s/output is p
- S_{OE}: odd # of a’s and even # of b’s/output is p
- Note that this machine can be a moore machine. So, we can associate the output with each state.

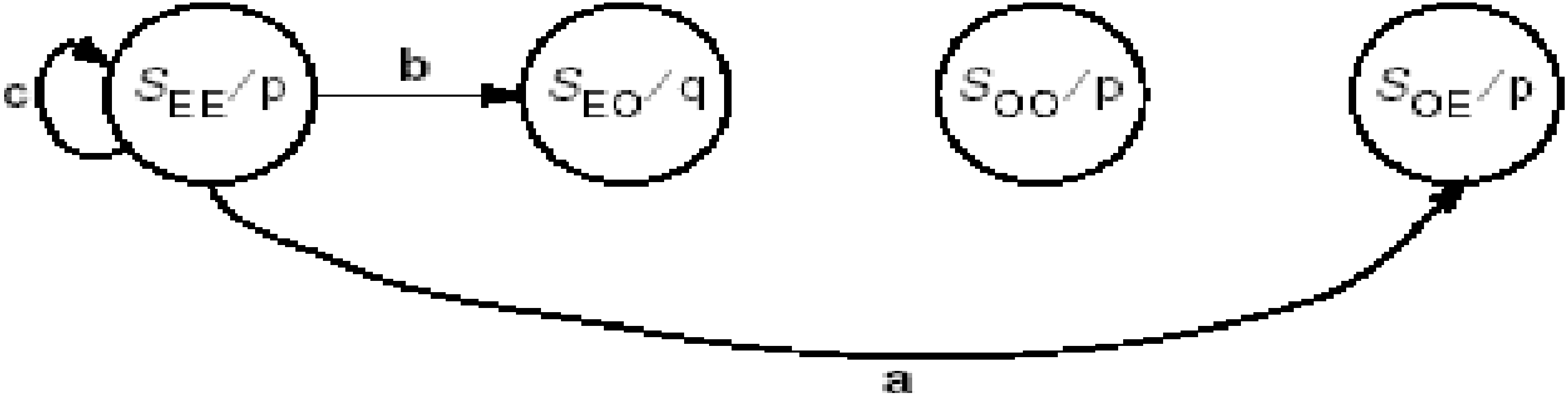
FSM Example

- Now draw a circle with each state



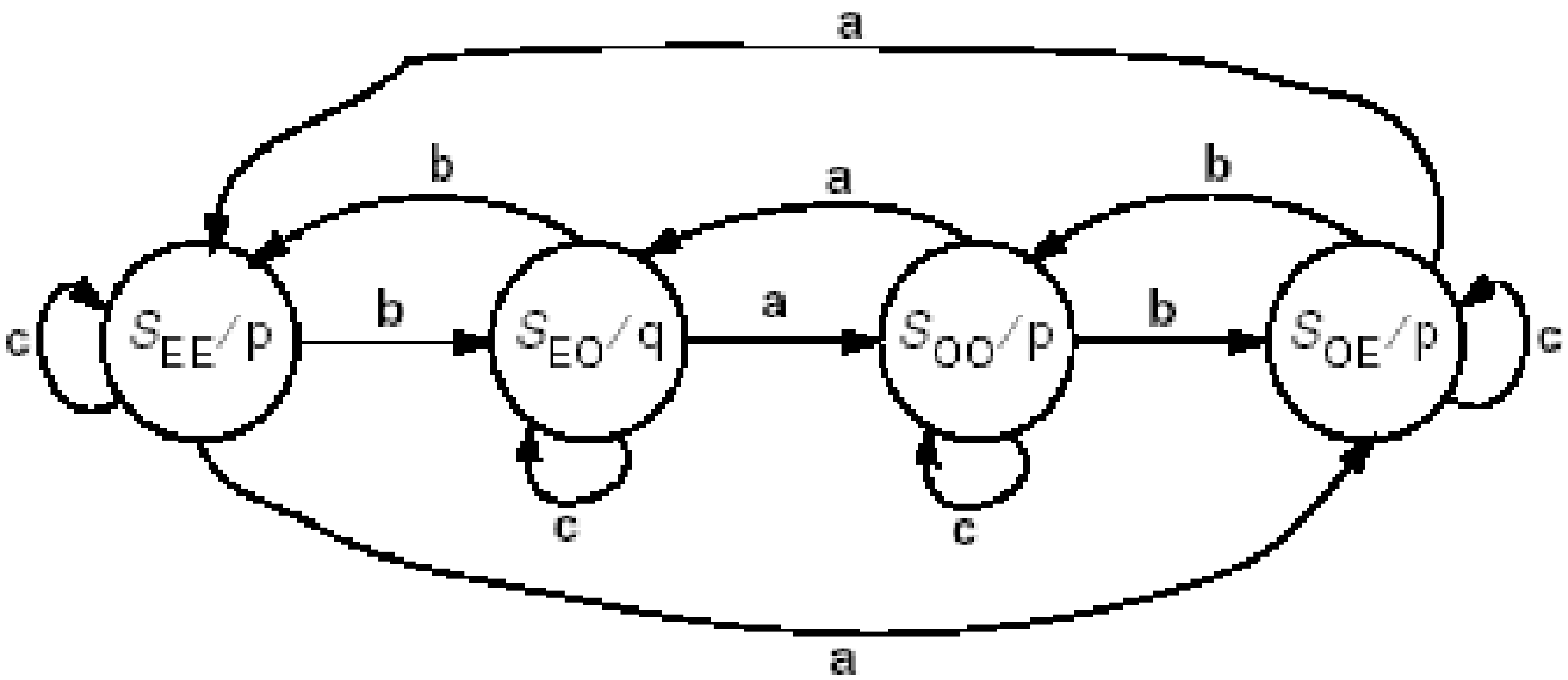
FSM Example

- Finally, for each state, consider the effect for each possible input.
  - For instance, starting with state SEE, the next state for the three input **a, b**, and **c** are determined as follows.



FSM Example

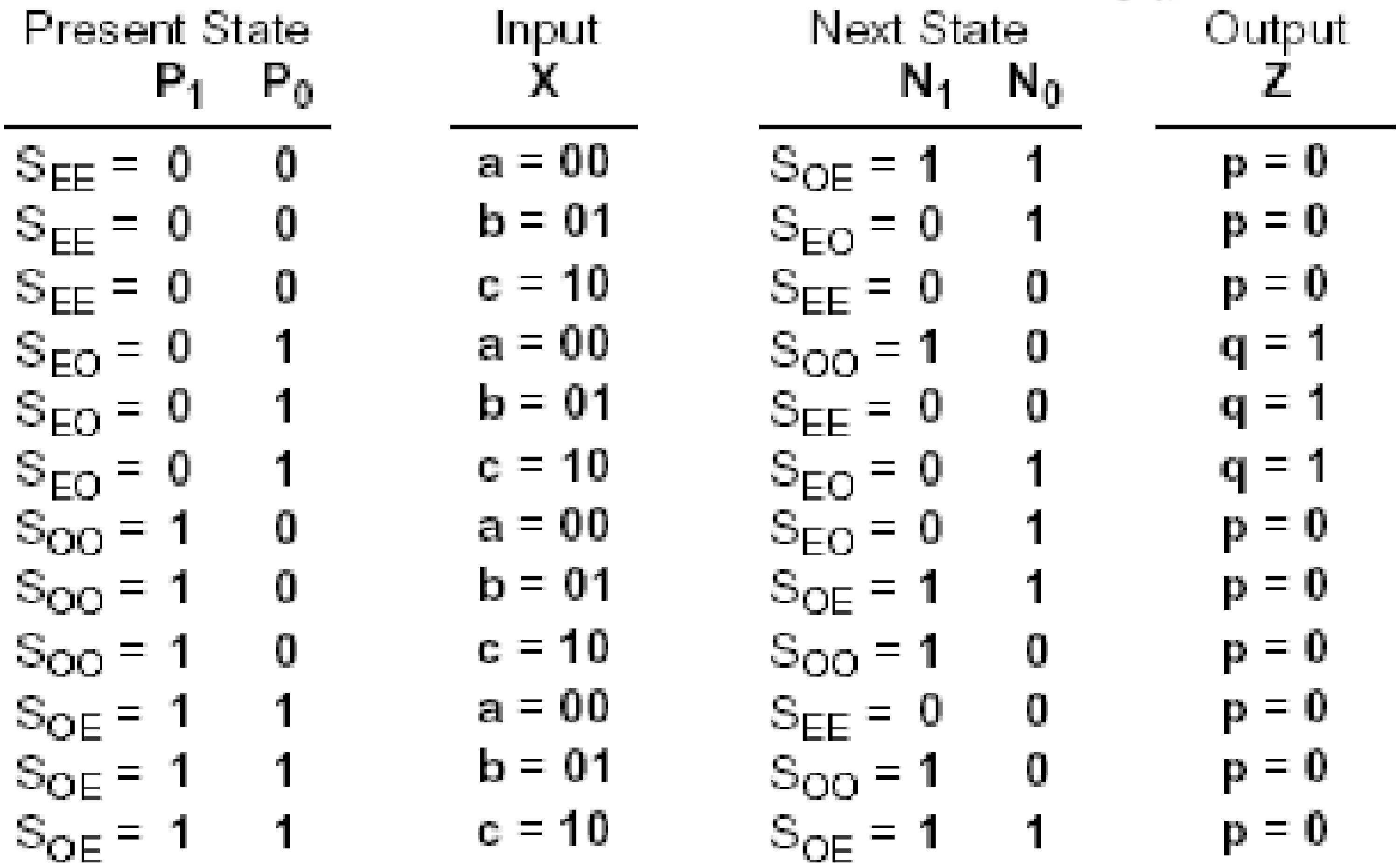
- Finishing the state diagram, the following is obtained.



FSM Example

- A state table can also be formed for this state diagram as follows.
  - First, assign a binary number to each state
    - SEE = 00, SEO = 01, SOO = 10, SOE = 11
  - Assign a binary number to each input
    - a = 00, b = 01, c = 10
  - Assign a binary number to each output
    - p = 0, q = 1
  - Then for each state, find the next state for each input. In this case there are three possible input values, so, three possible state transitions from each state.
- The state table on the following slide shows the results for this example.

FSM Example



Present State			Input	Next State			Output
	P_1	P_0	X		N_1	N_0	Z
S_{EE} =	0	0	a = 00	S_{OE} =	1	1	p = 0
S_{EE} =	0	0	b = 01	S_{EO} =	0	1	p = 0
S_{EE} =	0	0	c = 10	S_{EE} =	0	0	p = 0
S_{EO} =	0	1	a = 00	S_{OO} =	1	0	q = 1
S_{EO} =	0	1	b = 01	S_{EE} =	0	0	q = 1
S_{EO} =	0	1	c = 10	S_{EO} =	0	1	q = 1
S_{OO} =	1	0	a = 00	S_{EO} =	0	1	p = 0
S_{OO} =	1	0	b = 01	S_{OE} =	1	1	p = 0
S_{OO} =	1	0	c = 10	S_{OO} =	1	0	p = 0
S_{OE} =	1	1	a = 00	S_{EE} =	0	0	p = 0
S_{OE} =	1	1	b = 01	S_{OO} =	1	0	p = 0
S_{OE} =	1	1	c = 10	S_{OE} =	1	1	p = 0

FSM Example

- The Boolean function for the output can be determined from a Karnaugh map as follows.
  - Note that an input 11 is not possible since we only have three inputs that we have assigned to 00, o1 and 10. We can therefore use don’t cares for this possible input.

X_1X_0 \ P_1P_0	00	01	11	10
00	0	1	0	0
01	0	1	0	0
11	X	X	X	X
10	0	1	0	0

$$Z = \overline{P_1}P_0$$

FSM Example

- The Boolean function for the next state bit can also be determined from Karnaugh maps as follows

X_1X_0 \ P_1P_0	00	01	11	10
00	1	1	0	0
01	0	0	1	1
11	X	X	X	X
10	0	0	1	1

X_1X_0 \ P_1P_0	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	X	X	X	X
10	0	1	1	0

$$N_1 = \overline{P_1 \oplus X_1 \oplus X_0}$$

$$N_0 = P_0 X_1 + \overline{P_0}\,\overline{X_1} = \overline{P_0 \oplus X_1}$$

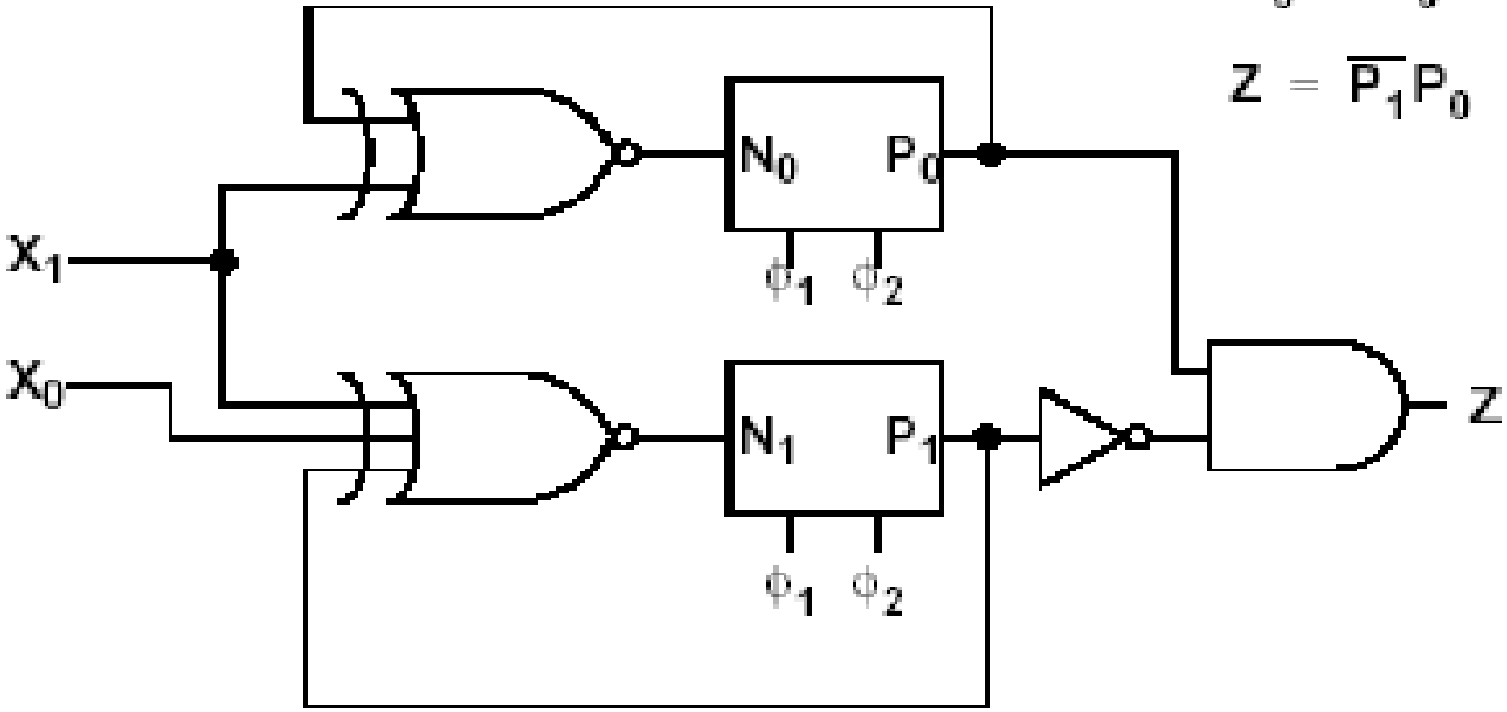
FSM Example

- The following logic circuit can be made with these Boolean functions

$$N_1 = \overline{P_1 \oplus X_1 \oplus X_0}$$

$$N_0 = \overline{P_0 \oplus X_1}$$

$$Z = \overline{P_1} P_0$$



FSM Example

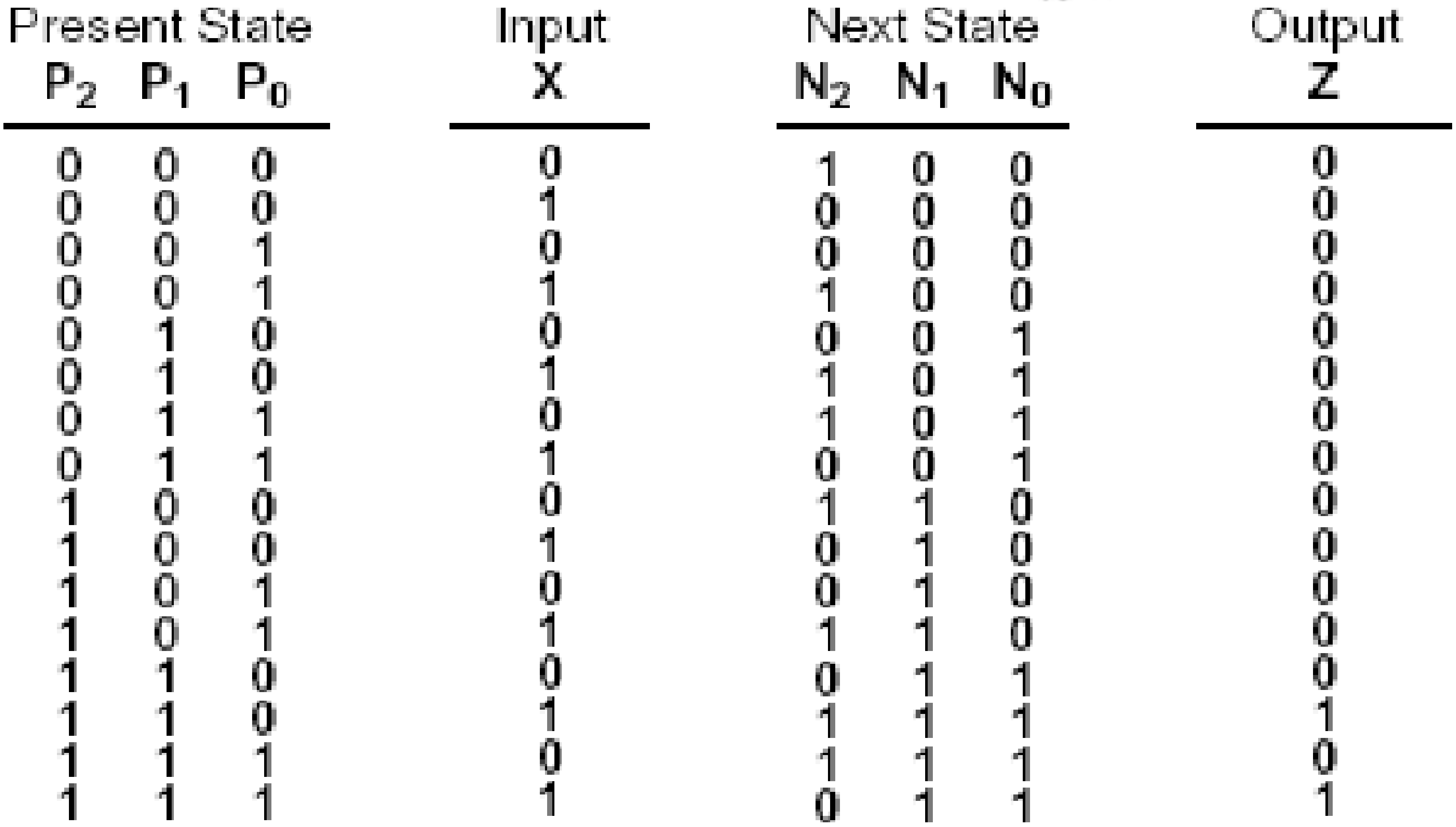
- A sequential circuit is defined by the following Boolean functions with input X, present states P0, P1 and P2 and next states N0, N1 and N2

- $N_2 = X(P_1 \oplus P_0) + \bar{X}\overline{(P_1 \oplus P_0)}$
- $N_1 = P_2$
- $N_0 = P_1$
- $Z = XP_1P_2$

- Derive the state table
- Derive the state Diagram

FSM Example

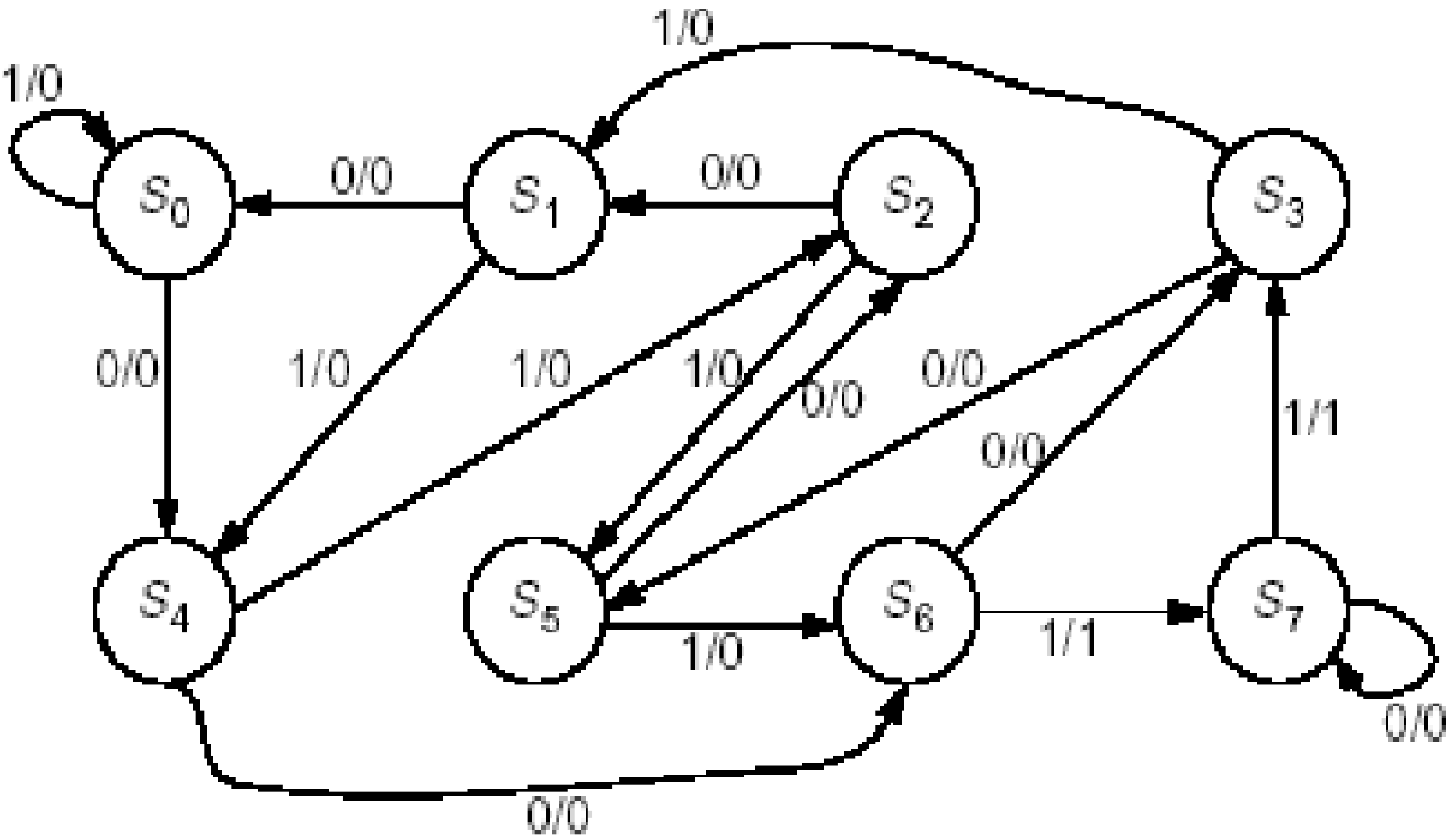
- The state table is formed as follows.



Present State			Input	Next State			Output
P_2	P_1	P_0	X	N_2	N_1	N_0	Z
0	0	0	0	1	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	1	0	0	0
0	1	0	0	0	0	1	0
0	1	0	1	1	0	1	0
0	1	1	0	1	0	1	0
0	1	1	1	0	0	1	0
1	0	0	0	1	1	0	0
1	0	0	1	0	1	0	0
1	0	1	0	0	1	0	0
1	0	1	1	1	1	0	0
1	1	0	0	0	1	1	0
1	1	0	1	1	1	1	1
1	1	1	0	1	1	1	0
1	1	1	1	0	1	1	1

FSM Example

- The state Diagram can be drawn as follows



Mapping English Language Description to Formal Specifications

Case Studies:

- String Pattern Recognizer
- Traffic Light Controller

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:

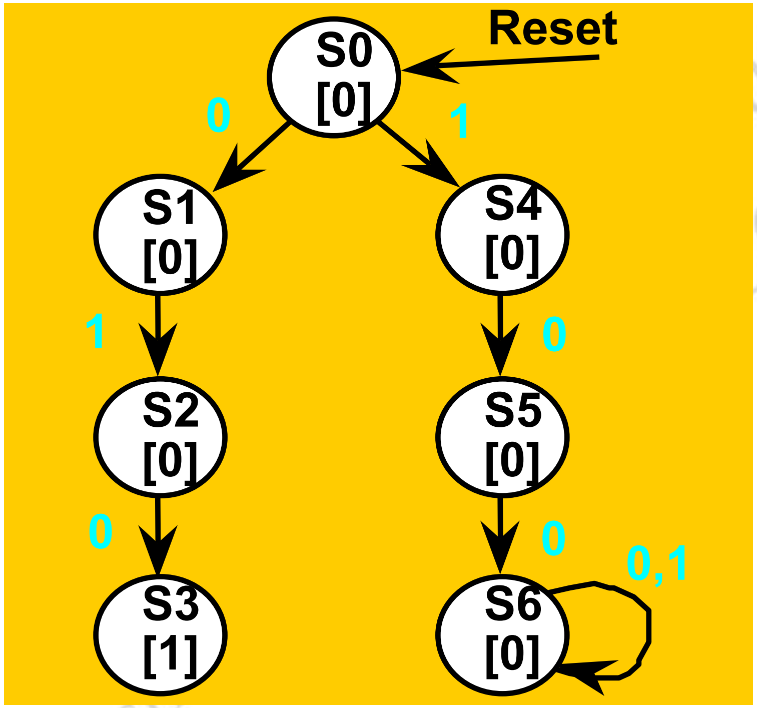
```
X:  00101010010…
Z:  00010101000…

X:  11011010010…
Z:  00000001000…
```

Finite State Machine Word Problems

Finite String Recognizer

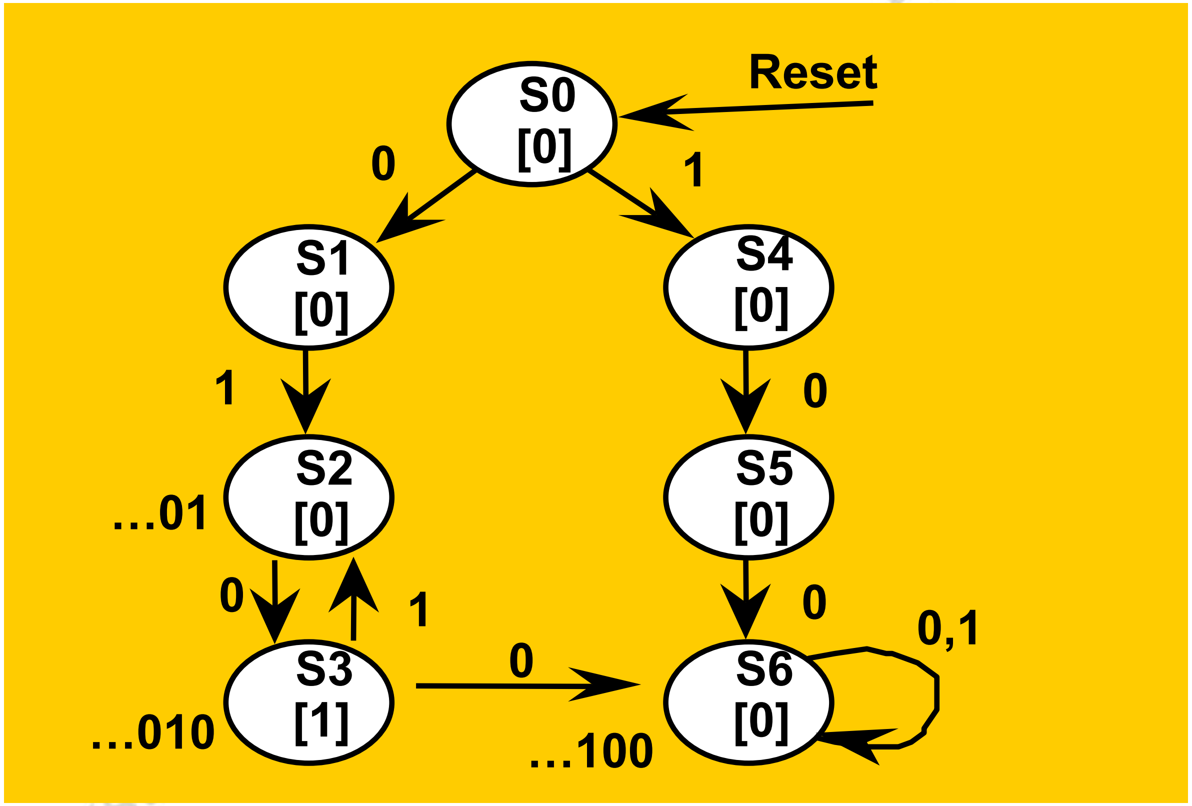
Step 2. Draw State Diagrams/ASM Charts for the strings that must be recognized. i.e., 010 and 100.



Moore State Diagram
Reset signal places FSM in S0

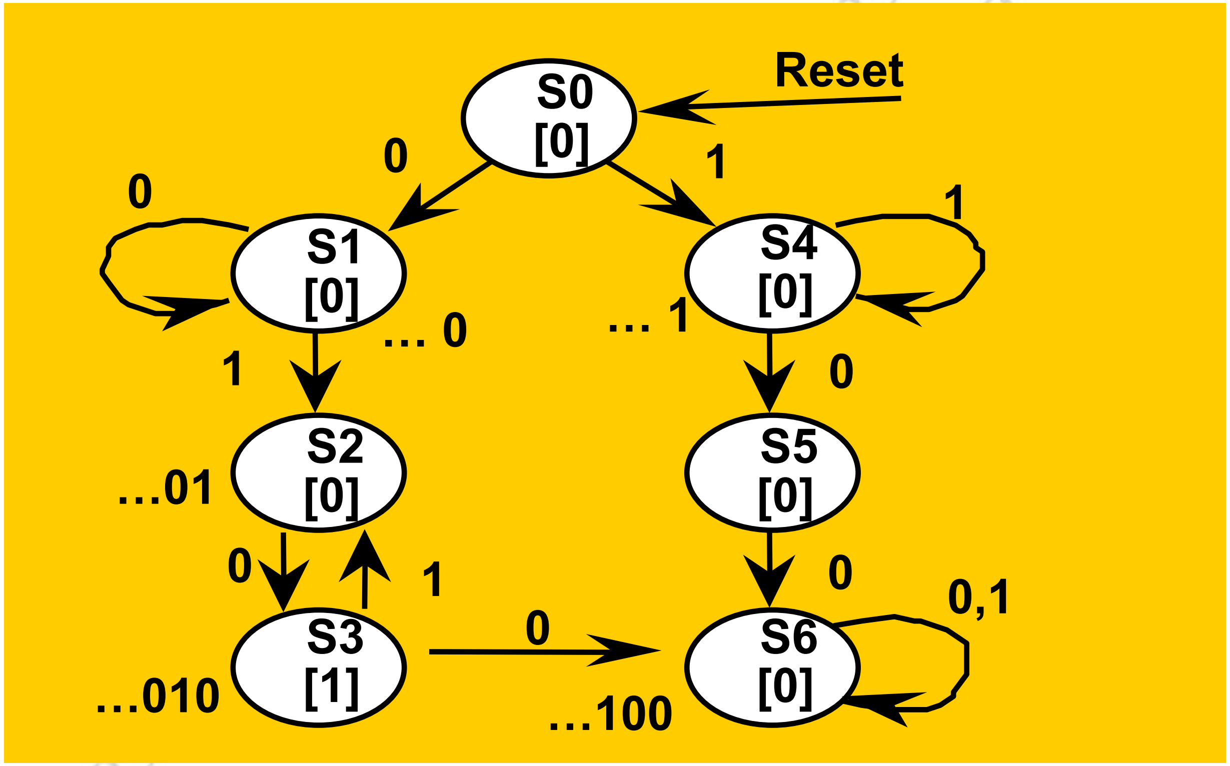
Finite State Machine Word Problems

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

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

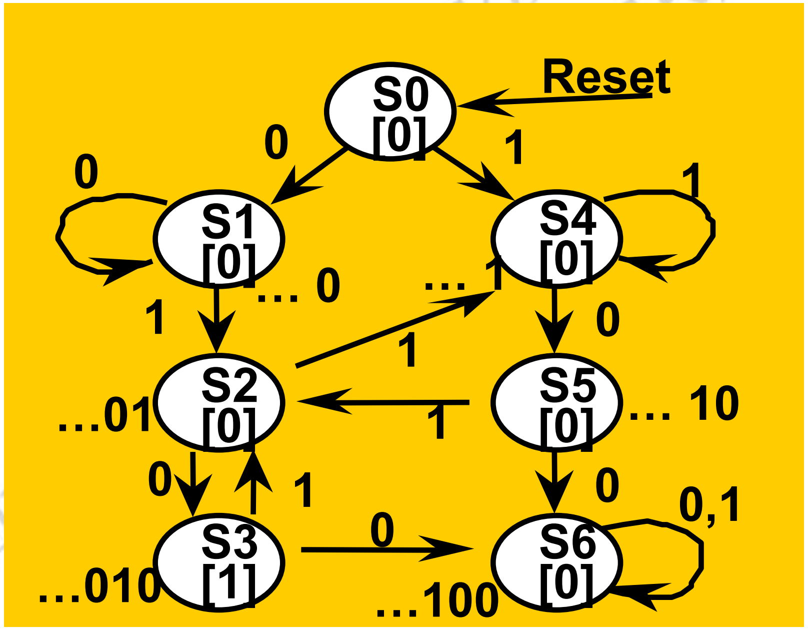


Finite State Machine Word Problems

S2, S5 with incomplete transitions

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

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



Finite State Machine Word Problems

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

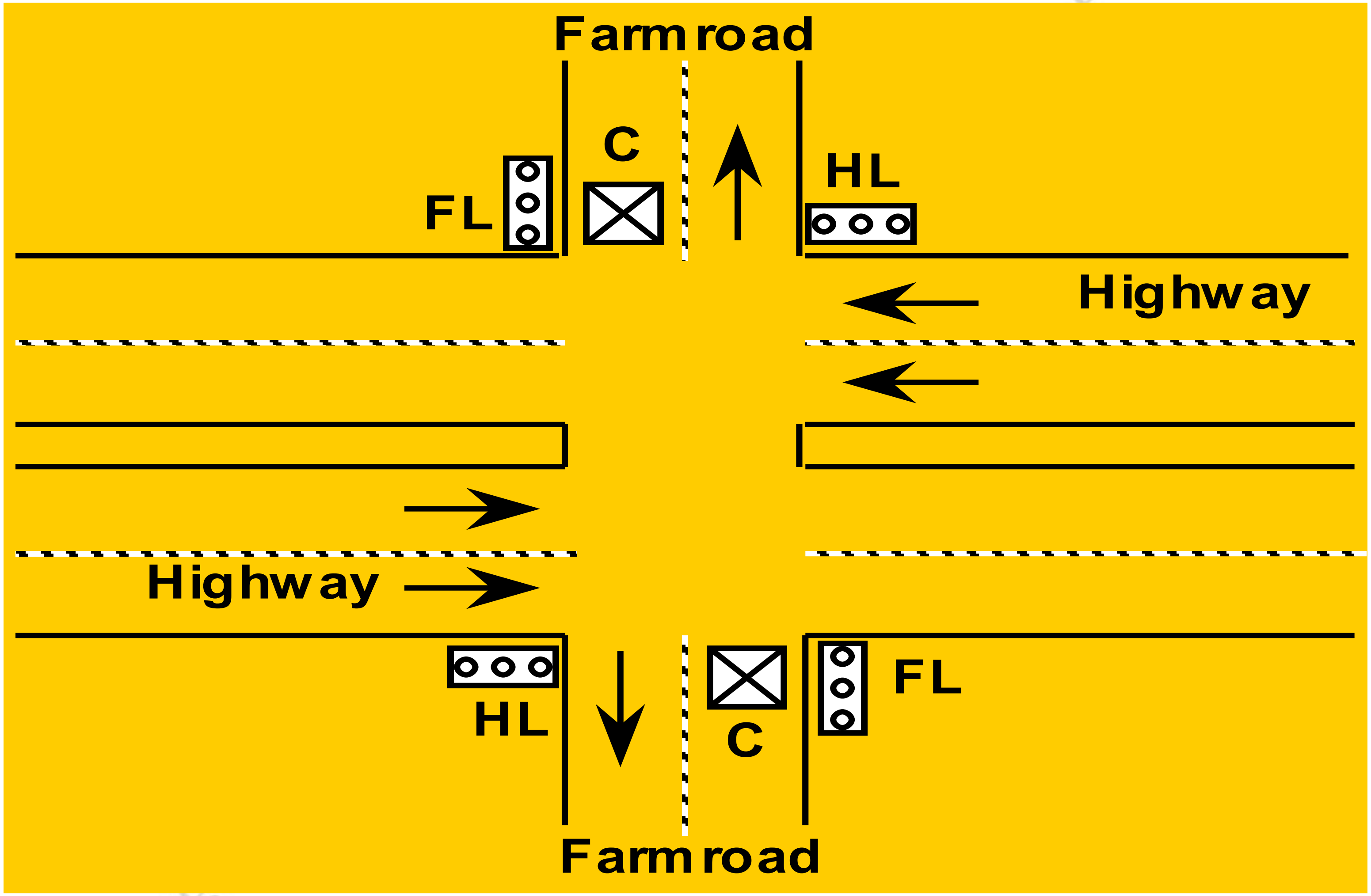
Finite State Machine Word Problems

Traffic Light Controller

- A busy highway is intersected by a little used farm road. Detectors C sense the presence of cars waiting on the farm road.
- With no car on farm road, light remain green in highway direction.
- If vehicle on farm road, highway lights go from Green to Yellow to Red, allowing the farm road lights to become green.
- These stay green only as long as a farm road 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 farm road 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 longtime 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

Picture of Highway/Farmroad Intersection:



Finite State Machine Word Problems

- 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

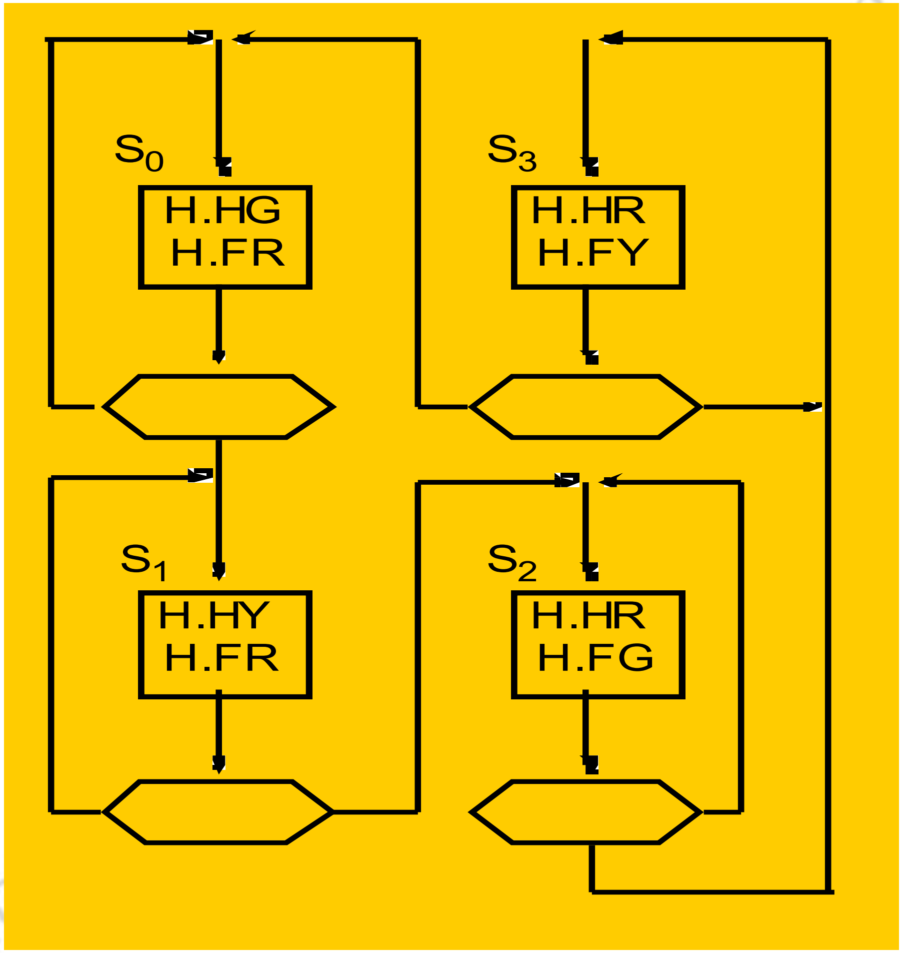
Output Signal	*Description*
HG, HY, HR	assert green/yellow/red highway lights
FG, FY, FR	assert green/yellow/red farmroad lights
ST	start timing a short or long interval

- 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

Start with basic sequencing and outputs:

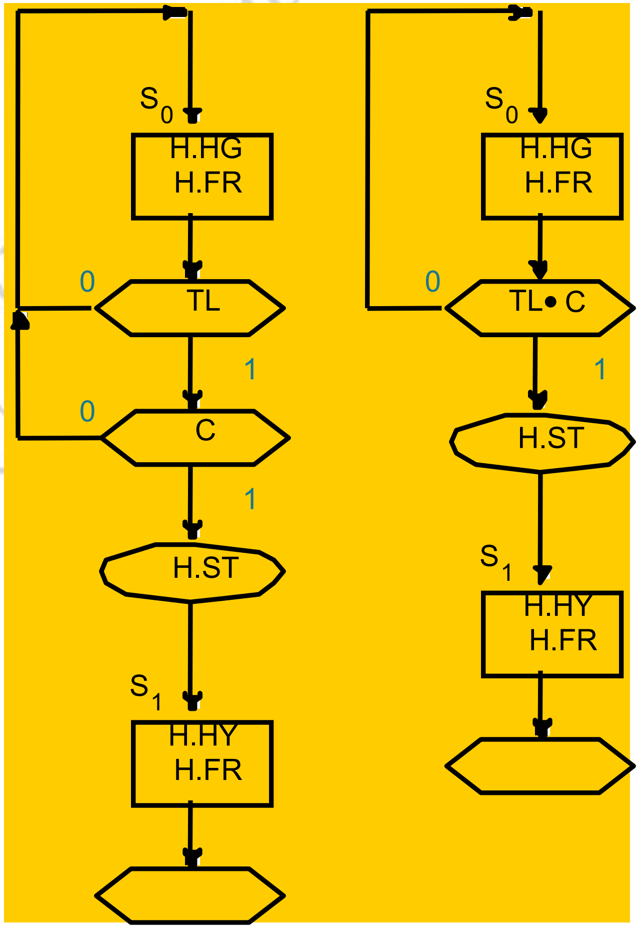


Finite State Machine Word Problems

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

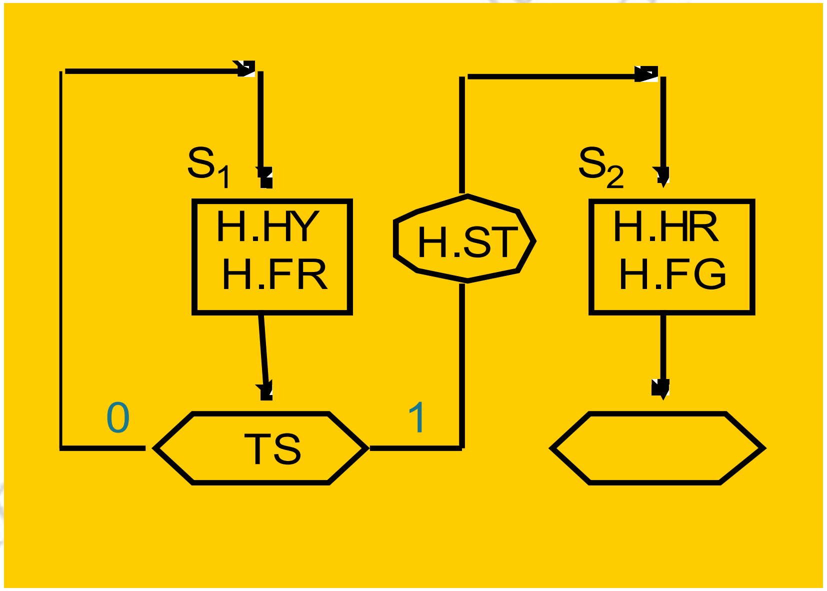
Traffic Light Controller

S1 to S2 Transition:

Set ST on exit from S0

Stay in S1 until TS asserted

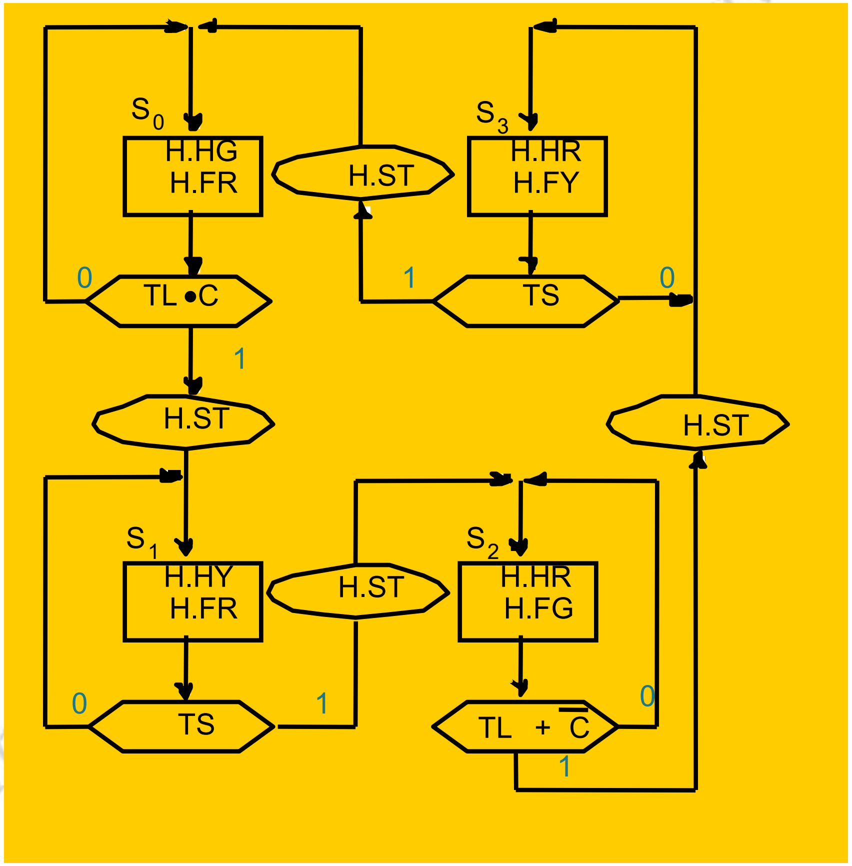
Similar situation for S3 to S4 transition



Finite State Machine Word Problems

Traffic Light Controller

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

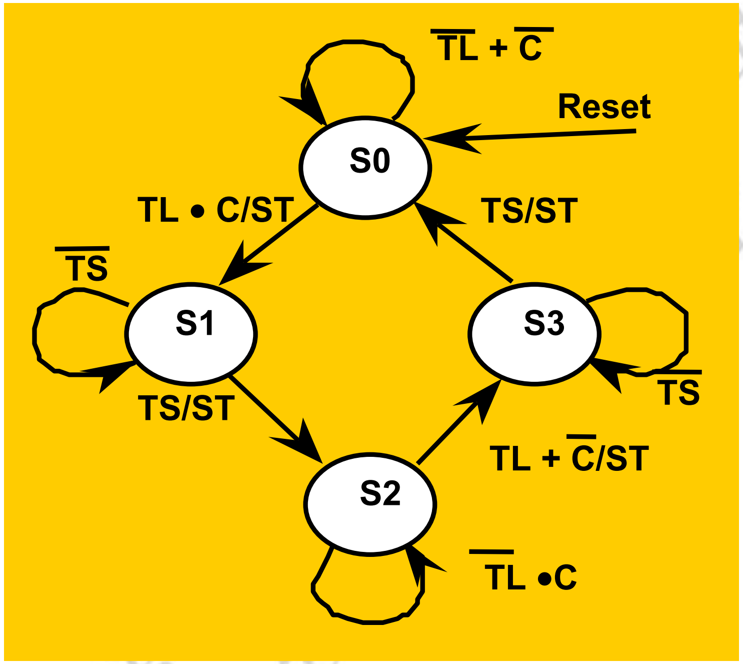


Complete ASM Chart for Traffic Light Controller

Finite State Machine Word Problems

Traffic Light Controller

Compare with state diagram:



S0: HG

S1: HY

S2: FG

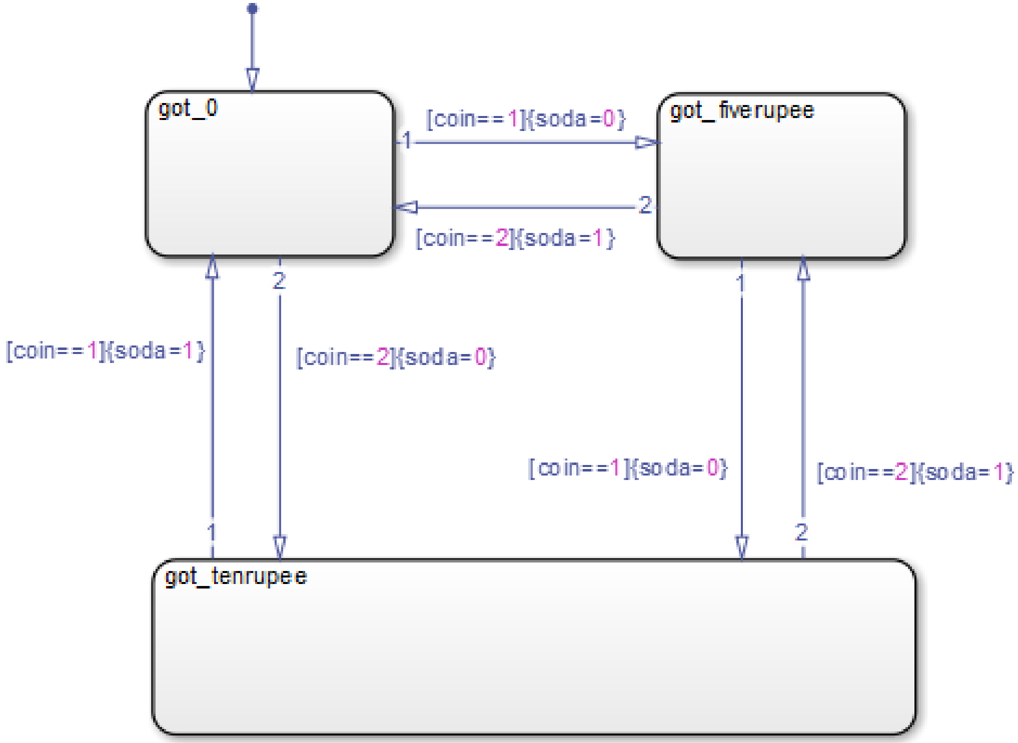
S3: FY

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

Model a Vending Machine by Using Mealy Semantics

- The vending machine requires 15 rupees to release a can of soda. The purchaser can insert a five rupees coin or a ten rupees coin, one at a time, to purchase the soda. Develop a state machine (Mealy) that dispenses soda after receiving inputs as coins.

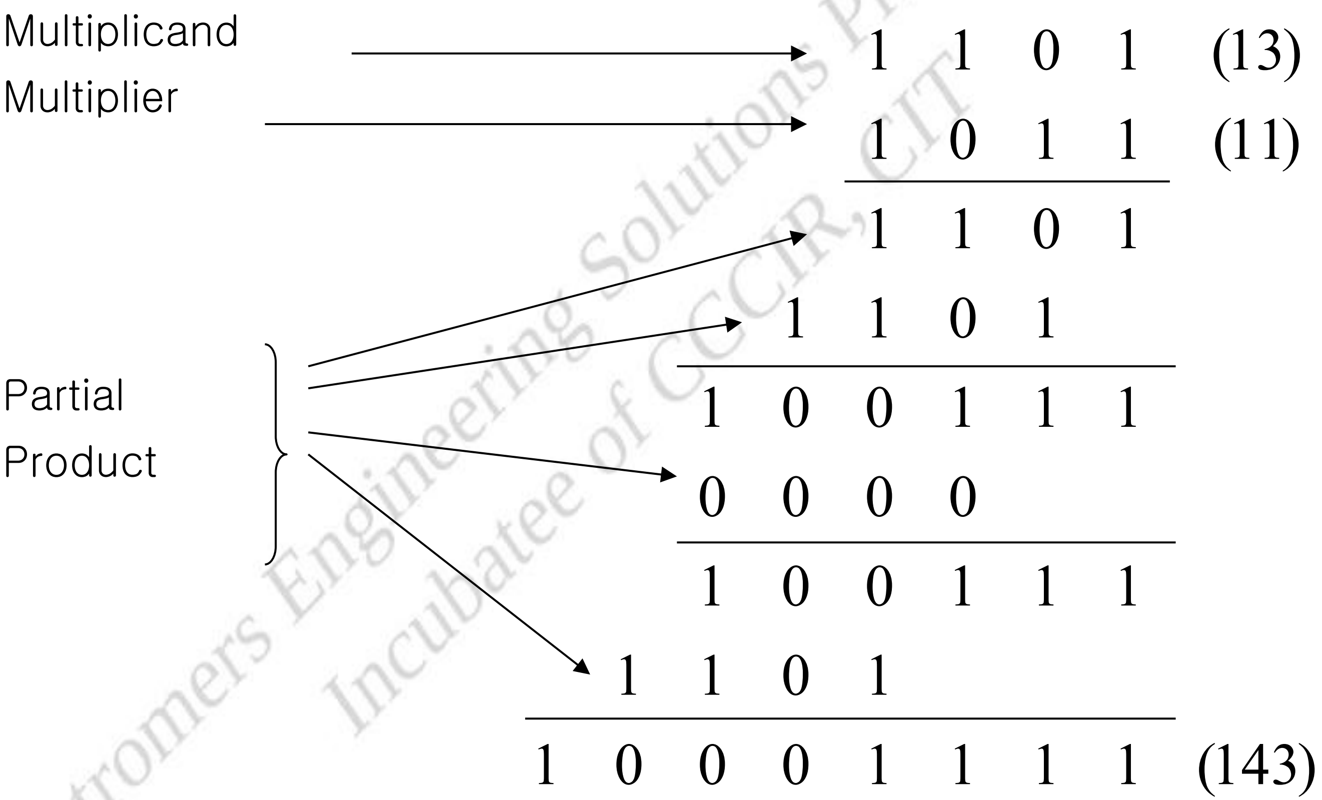


Vending machine states

- got_0 — No coin has been received or no coins are left.
  - If a fiverupee is received (coin == 1), output soda remains 0, but state got_ fiverupee becomes active.
  - If a tenrupee is received (coin == 2), output soda remains 0, but state got_tenrupee becomes active.
  - If input coin is not a tenrupee or a fiverupee, state got_0 stays active and no soda is released (output soda = 0).
- got_rupeesfive — A fiverupee was received.
  - If another fiverupee is received (coin == 1), state got_tenrupee becomes active, but no can is released (soda remains at 0).
  - If a tenrupee is received (coin == 2), a can is released (soda = 1), the coins are banked, and the active state becomes got_0 because no coins are left.
  - If input coin is not a tenrupee or a fiverupee, state got_fiverupee stays active and no can is released (output soda = 0).
- got_rupeesten — A tenrupee was received.
  - If a fiverupee is received (coin == 1), a can is released (soda = 1), the coins are banked, and the active state becomes got_0 because no coins are left.
  - If a tenrupee is received (coin == 2), a can is released (soda = 1), 15 cents are banked, and the active state becomes got_ fiverupee because a fiverupee (change) is left.
  - If input coin is not a tenrupee or a fiverupee, state got_tenrupee stays active and no can is released (output soda = 0).

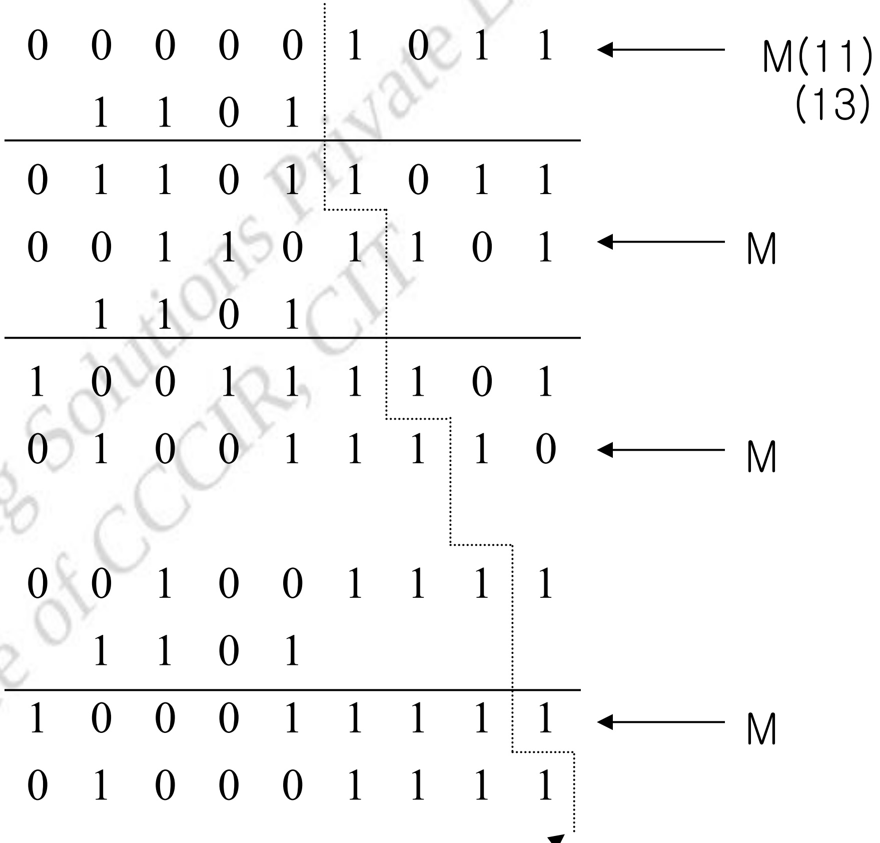
Design of a Binary Multiplier

Multiplication of 13_{10} by 11_{10} in binary



```
              1  1  0  1   (13)
              1  0  1  1   (11)
             ------------
              1  1  0  1
           1  1  0  1
       ------------------
        1  0  0  1  1  1
        0  0  0  0
       ------------------
        1  0  0  1  1  1
     1  1  0  1
  -----------------------
  1  0  0  0  1  1  1  1   (143)
```

Design of a Binary Multiplier

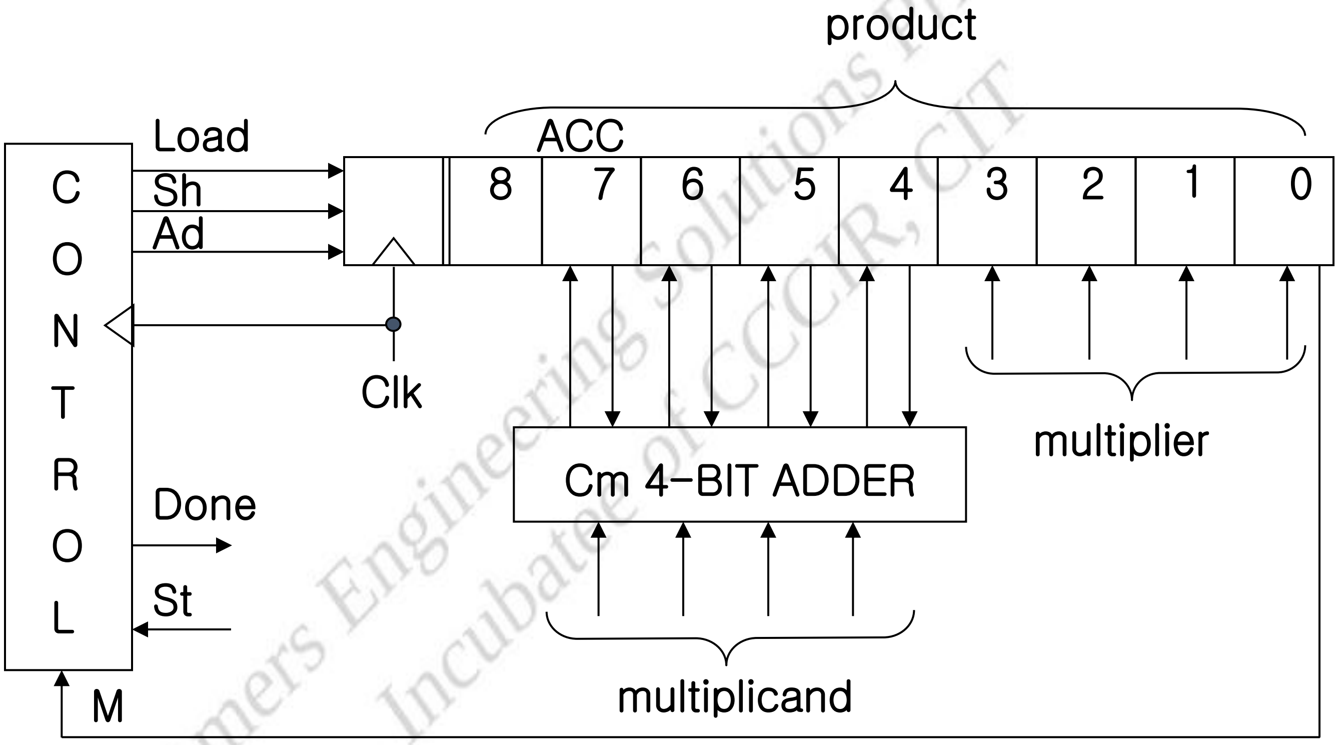


initial contents of product register	0	0	0	0	0	1	0	1	1	M(11) (13)
(add multiplicand since M=1		1	1	0	1					
after addition	0	1	1	0	1	1	0	1	1	
after shift	0	0	1	1	0	1	1	0	1	M
(add multiplicand since M=1)		1	1	0	1					
after addition	1	0	0	1	1	1	1	0	1	
after shift	0	1	0	0	1	1	1	1	0	M
(skip addition since M=0)										
after shift	0	0	1	0	0	1	1	1	1	
(add multiplicand since M=1)		1	1	0	1					
after addition	1	0	0	0	1	1	1	1	1	M
after shift (final answer)	0	1	0	0	0	1	1	1	1	

Dividing line between product and multiplier

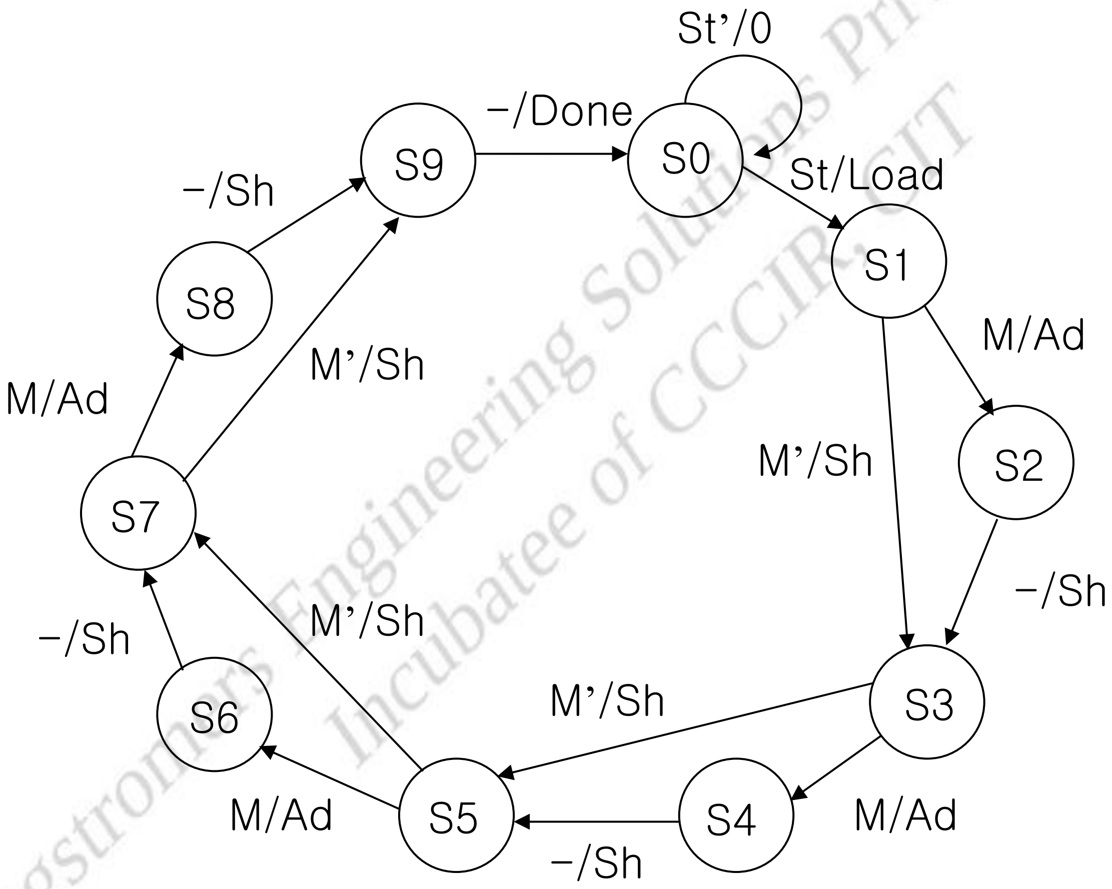
Design of a Binary Multiplier

Block diagram for binary multiplier



Design of a Binary Multiplier

State graph for binary multiplier control

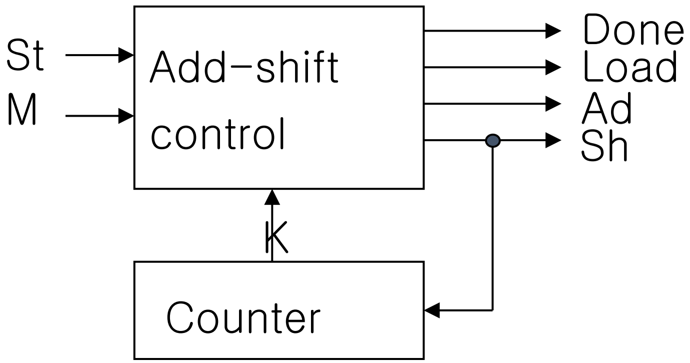


Design of a Binary Multiplier

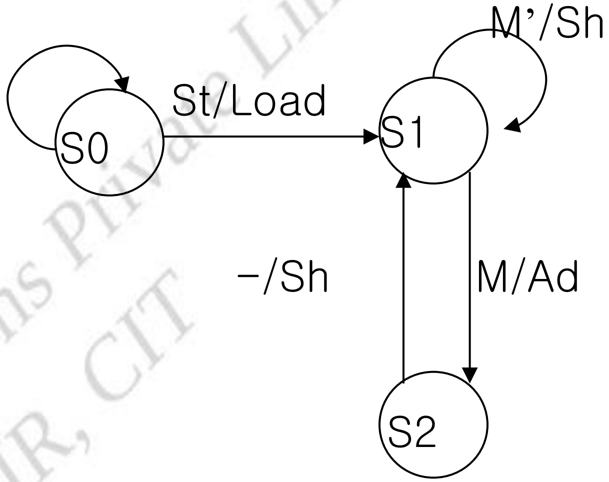
- As the state graph for the multiplier indicates, the control perform two functions – generation add or shift signals as needed and counting the number of shifts
- If the number of bits is large, it is convenient to divide the control network into a counter and an add-shift control.

Design of a Binary Multiplier

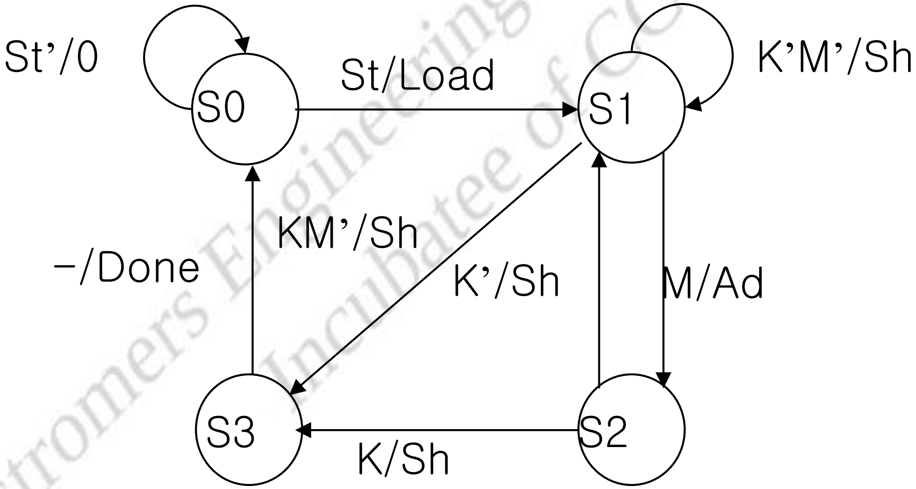
Multiplier control with counter



(a) Multiplier control



(b) State graph for add-shift control



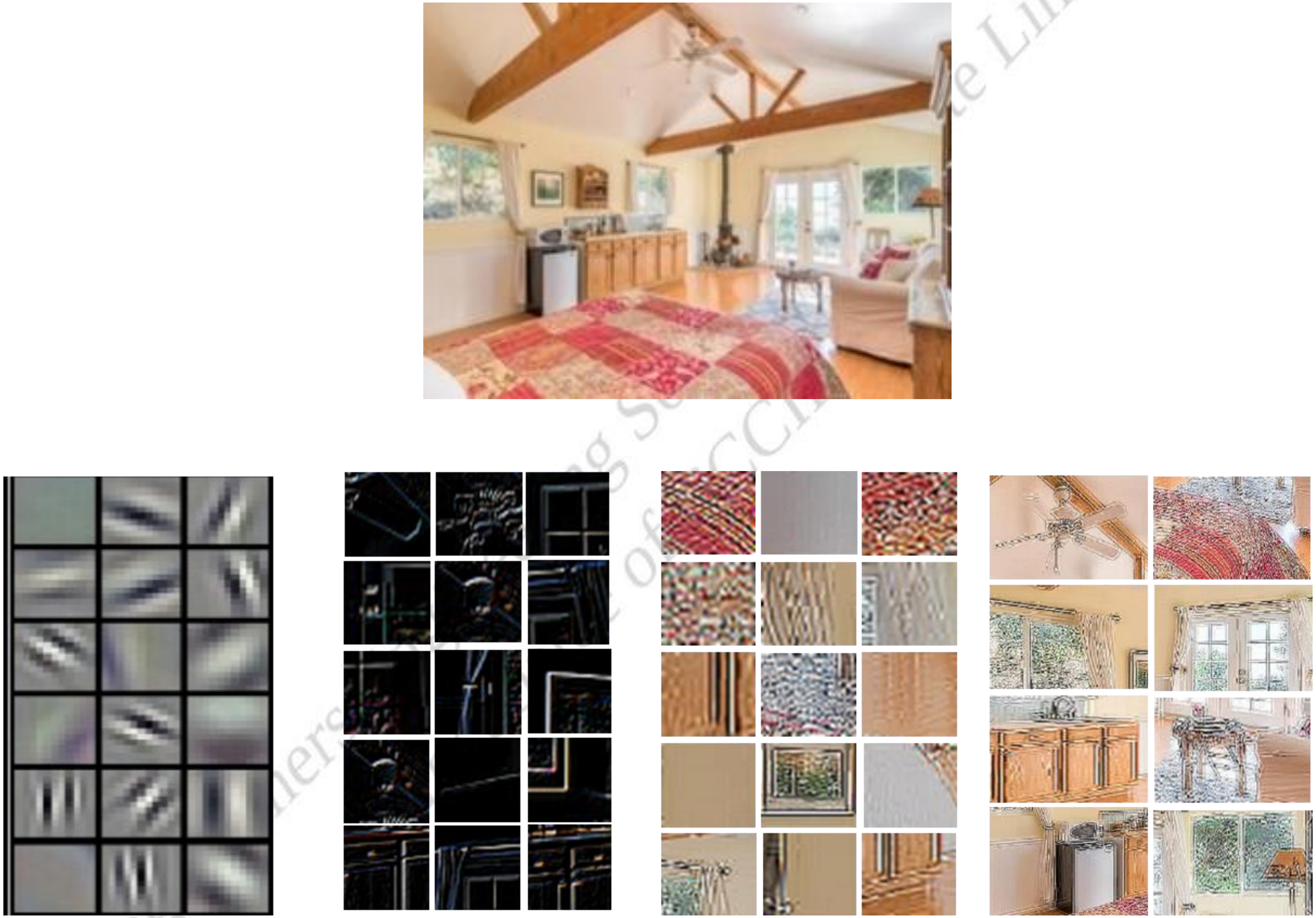
(c) Final state graph for add-shift control

Design of a Binary Multiplier

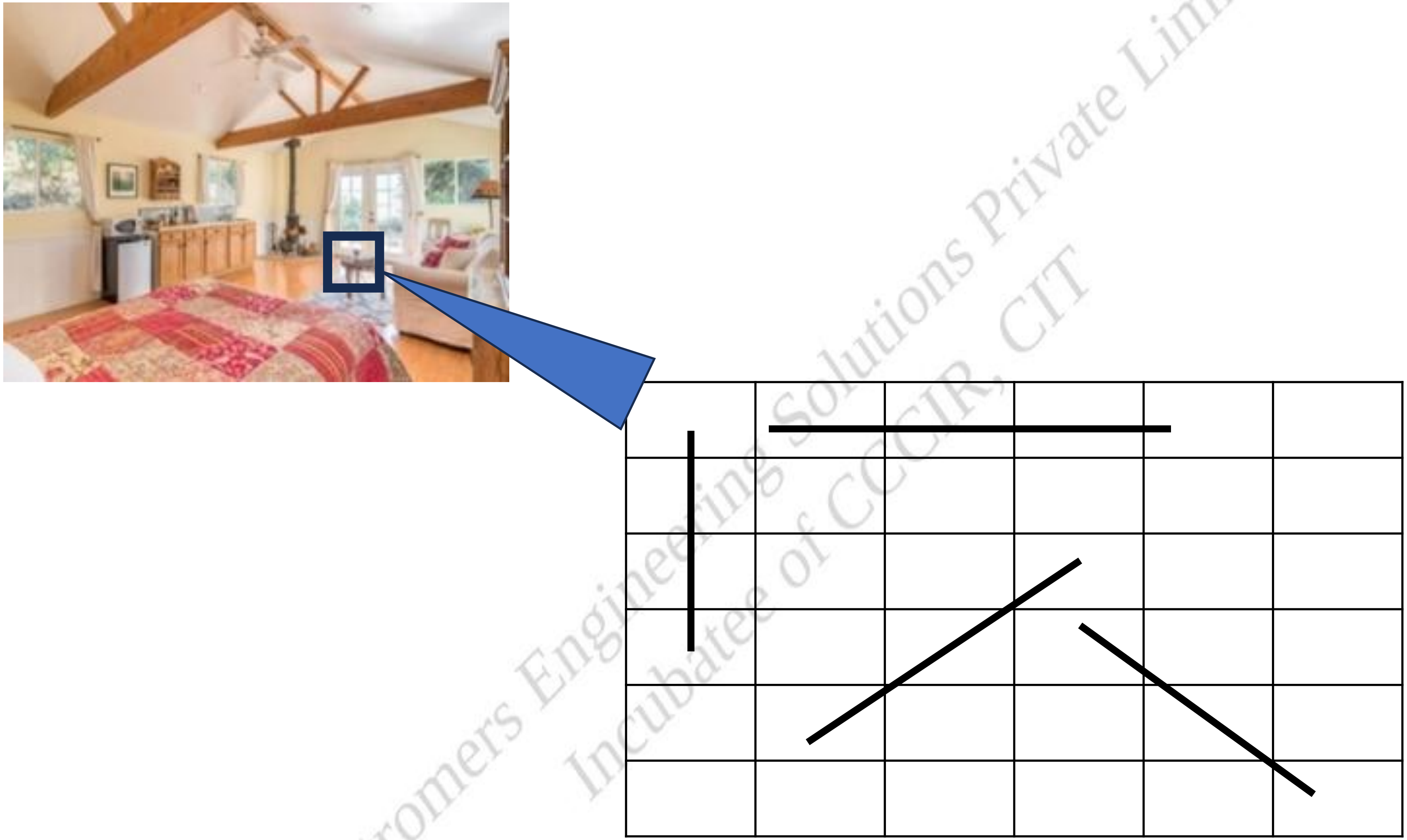
Operation of multiplier using a counter

Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t0	S0	00	000000000	0	0	0	0	0	0	0
t1	S0	00	000000000	1	0	0	1	0	0	0
t2	S1	00	000001011	0	1	0	0	1	0	0
t3	S2	00	011011011	0	1	0	0	0	1	0
t4	S1	01	001101101	0	1	0	0	1	0	0
t5	S2	01	100111101	0	1	0	0	0	1	0
t6	S1	10	010011110	0	0	0	0	0	1	0
t7	S1	11	001001111	0	1	1	0	1	0	0
t8	S2	11	100011111	0	1	1	0	0	1	0
t9	S3	00	010001111	0	1	0	0	0	0	1

Feature Extraction Process



Feature Extraction Process



Feature Extraction Process

These are the network parameters to be learned.

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

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

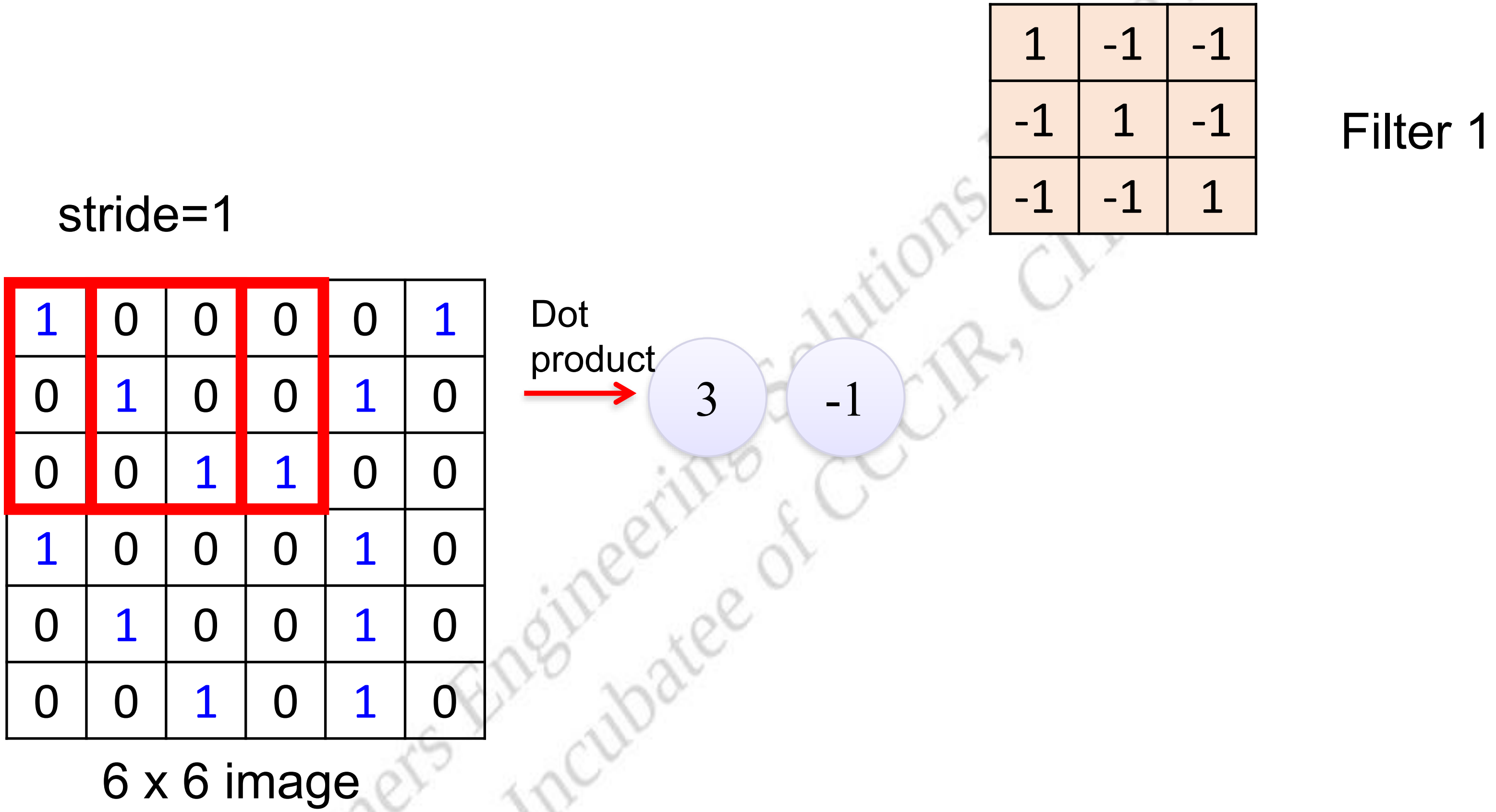
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

⋮ ⋮

Each filter detects a small pattern (3 x 3).

Feature Extraction Process



stride=1

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

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

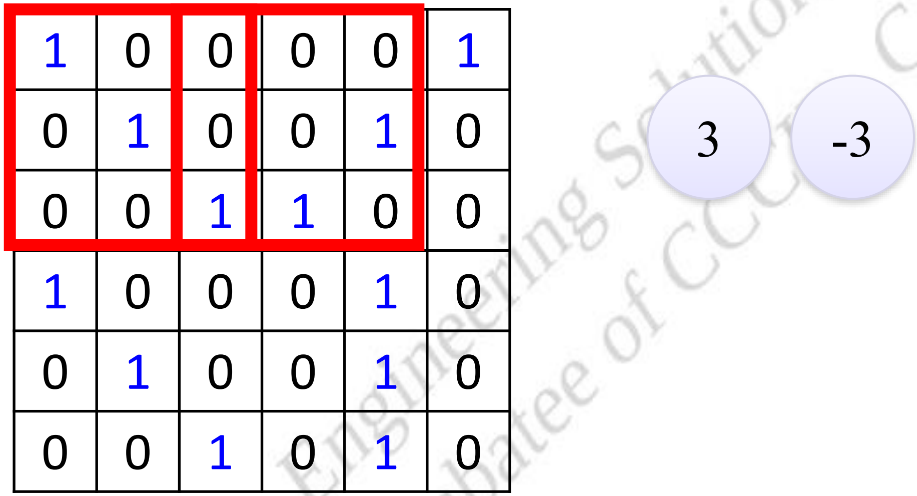
Filter 1

Feature Extraction Process

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

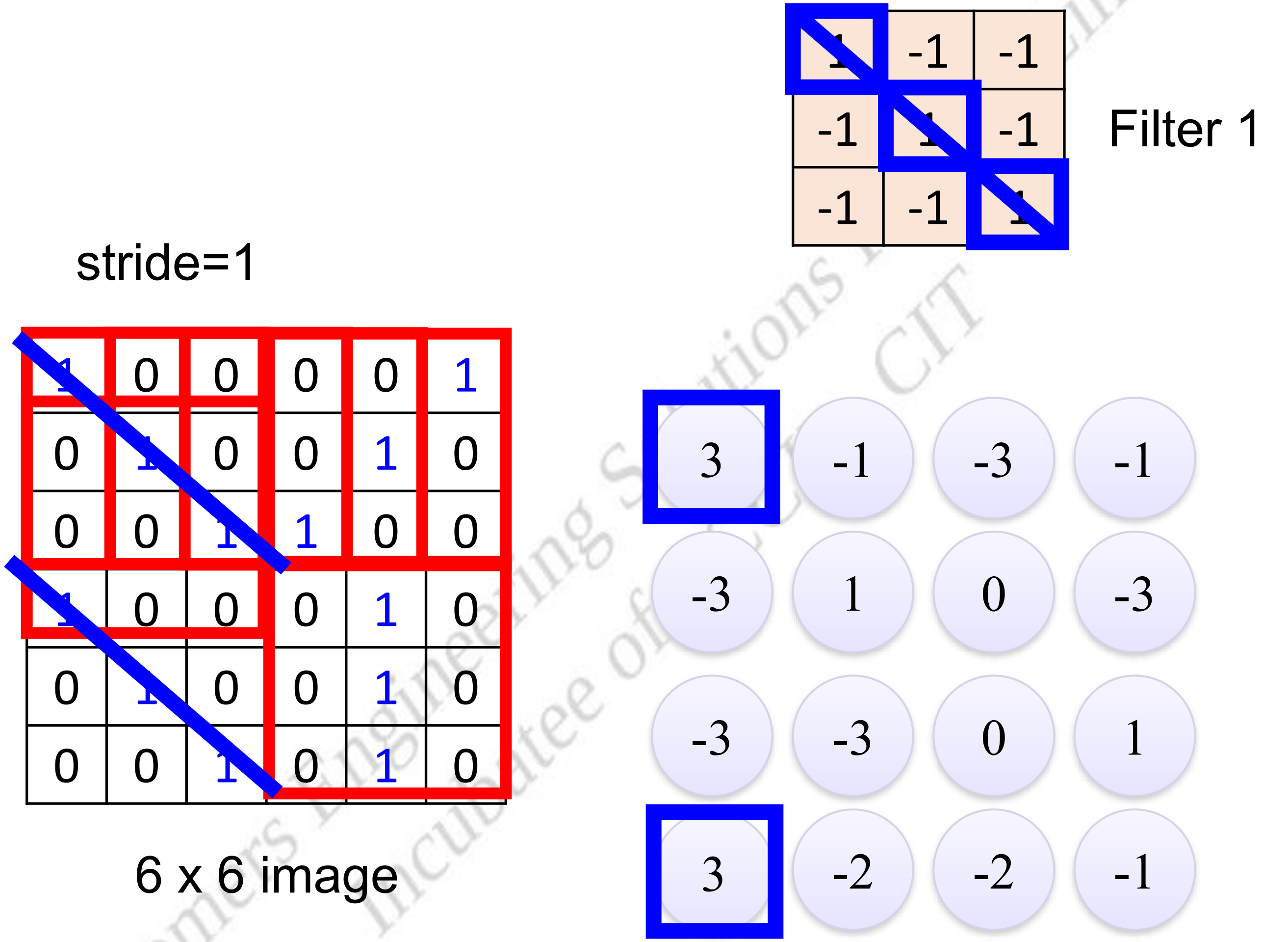
If stride=2



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

6 x 6 image

Feature Extraction Process



1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

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

6 x 6 image

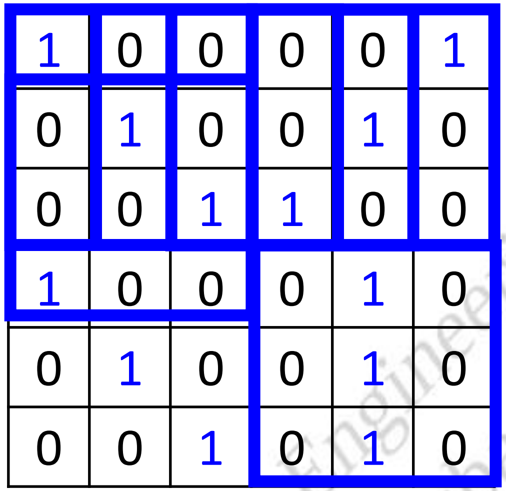
3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

Feature Extraction Process

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

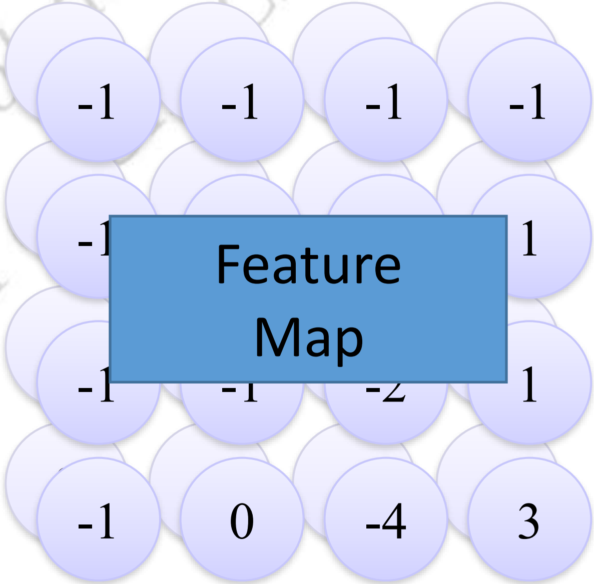
stride=1



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

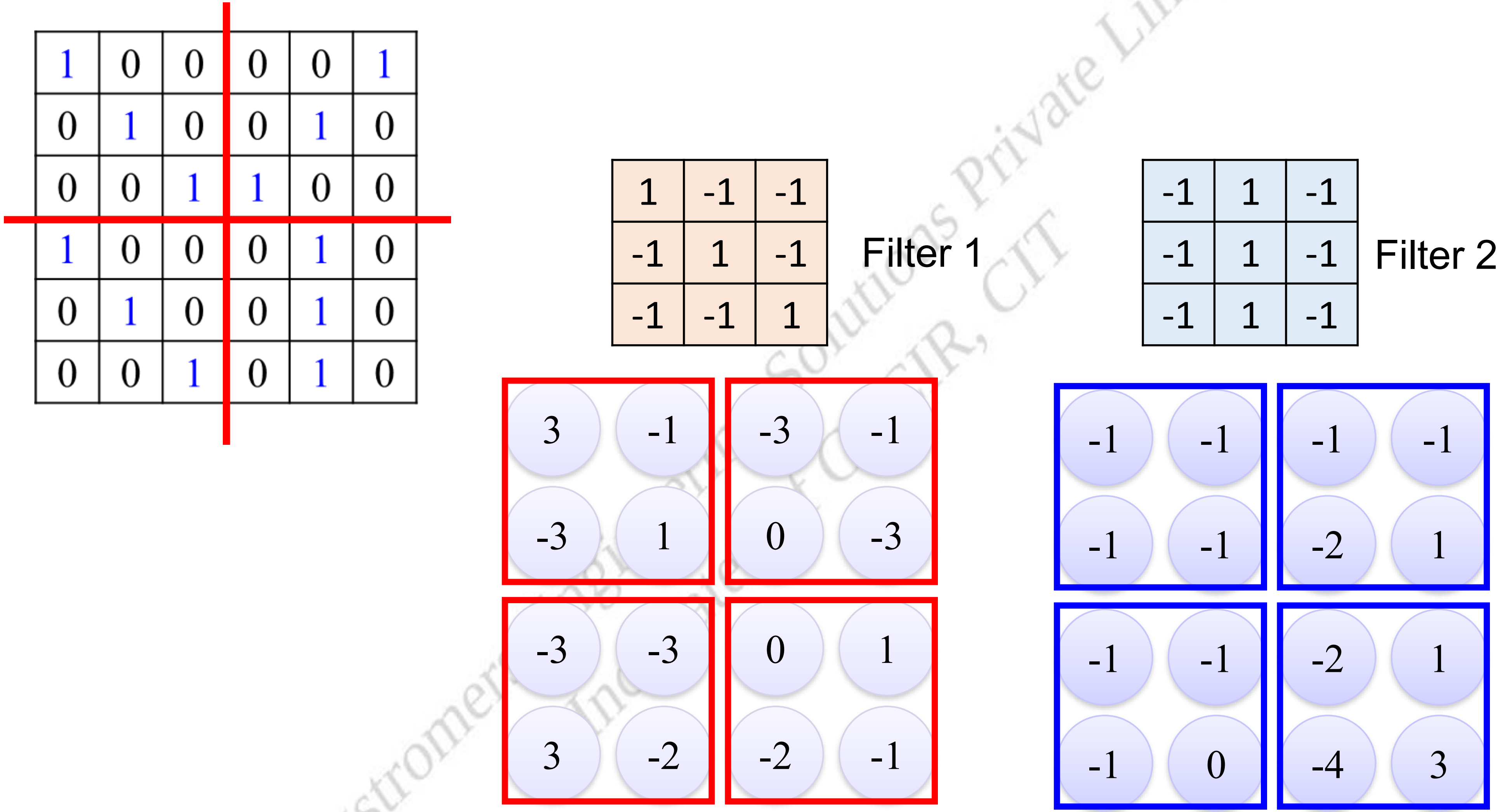
6 x 6 image

Repeat this for each filter



Two 4 x 4 images
Forming 2 x 4 x 4 matrix

Feature Representation



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

Filter 1

1	-1	-1
-1	1	-1
-1	-1	1

Filter 2

-1	1	-1
-1	1	-1
-1	1	-1

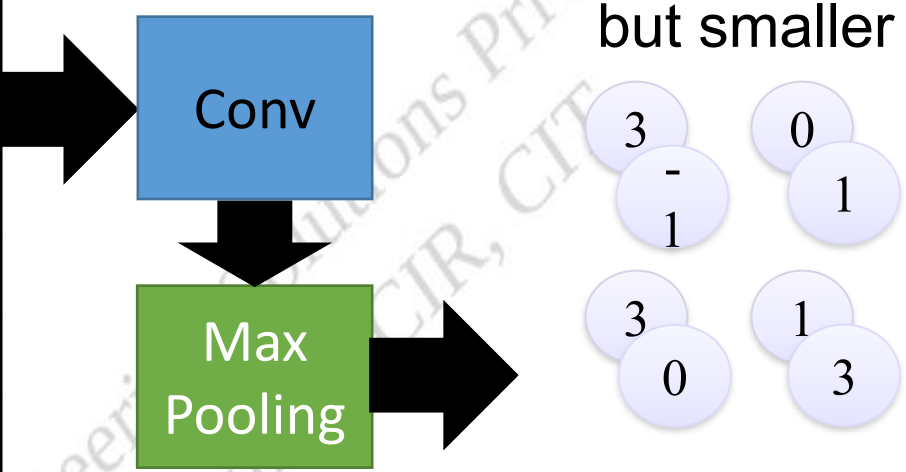
3	-1	-3	-1
-3	1	0	-3
-3	-3	0	1
3	-2	-2	-1

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

Feature Representation

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

6 x 6 image



New image but smaller

2 x 2 image

Each filter is a channel

Programming process

	1	2	3	4	5	6
1	1	0	0	0	0	1
2	0	1	0	0	1	0
3	0	0	1	1	0	0
4	1	0	0	0	1	0
5	0	1	0	0	1	0
6	0	0	1	0	1	0

1,1	1	1,1	1	1,1	1	1,1	1	1,1	1
1,2	0	1,2	0	1,2	0	1,2	0	1,2	0
1,3	0	1,3	0	1,3	0	1,3	0	1,3	0
1,4	0	1,4	0	1,4	0	1,4	0	1,4	0
1,5	0	1,5	0	1,5	0	1,5	0	1,5	0
1,6	1	1,6	1	1,6	1	1,6	1	1,6	1
2,1	0	2,1	0	2,1	0	2,1	0	2,1	0
2,2	1	2,2	1	2,2	1	2,2	1	2,2	1
2,3	0	2,3	0	2,3	0	2,3	0	2,3	0
2,4	0	2,4	0	2,4	0	2,4	0	2,4	0
2,5	1	2,5	1	2,5	1	2,5	1	2,5	1
2,6	0	2,6	0	2,6	0	2,6	0	2,6	0
3,1	0	3,1	0	3,1	0	3,1	0	3,1	0
3,2	0	3,2	0	3,2	0	3,2	0	3,2	0
3,3	1	3,3	1	3,3	1	3,3	1	3,3	1
3,4	1	3,4	1	3,4	1	3,4	1	3,4	1
3,5	0	3,5	0	3,5	0	3,5	0	3,5	0
3,6	0	3,6	0	3,6	0	3,6	0	3,6	0
4,1	1	4,1	1	4,1	1	4,1	1	4,1	1
4,2	0	4,2	0	4,2	0	4,2	0	4,2	0
4,3	0	4,3	0	4,3	0	4,3	0	4,3	0
4,4	0	4,4	0	4,4	0	4,4	0	4,4	0
4,5	1	4,5	1	4,5	1	4,5	1	4,5	1
4,6	0	4,6	0	4,6	0	4,6	0	4,6	0

Assignment

- Use FSM logic for data processing

Code obfuscation

- Code obfuscation is a collective term for techniques that make code more difficult to read and reverse engineer.
- There are many different techniques for a company to protect its intellectual ideas against competitors and their services against unintended use.
- Code obfuscation is a collection of methods that can be used to protect software against reverse engineering.
- Reverse engineering is a process that transforms the output of another process into the input of said process in part or completely.
- An example of this is to reconstruct the source code from a compiled program.
- The application of code obfuscation does not guarantee that a program cannot be reverse engineered, but its purpose is to make the process of reverse engineering prohibitively expensive.
- Combined with other techniques for pro tecting software against third party tampering, code obfuscation can serve as an important tool for software protection in the industry.

Can FSMs Help with Obfuscation

FSMs can add **structural complexity** to code, which might make reverse engineering harder by:

1. **Breaking linear code flow:**
   - Instead of direct if or switch logic, state transitions obscure the actual logic path.
2. **Spreading logic across multiple states:**
   - It forces a reverse engineer to trace through several transitions to understand a full operation.
3. **Encouraging state encoding:**
   - You can encode states non-obviously (e.g., cryptographic hash of a state name) to further confuse analysis.
4. **Introducing "dummy" states:**
   - Adding meaningless or misleading states increases the reverse engineering effort.

Limitations of FSMs for Obfuscation

FSMs are **not a secure obfuscation technique by themselves**. Some limitations:

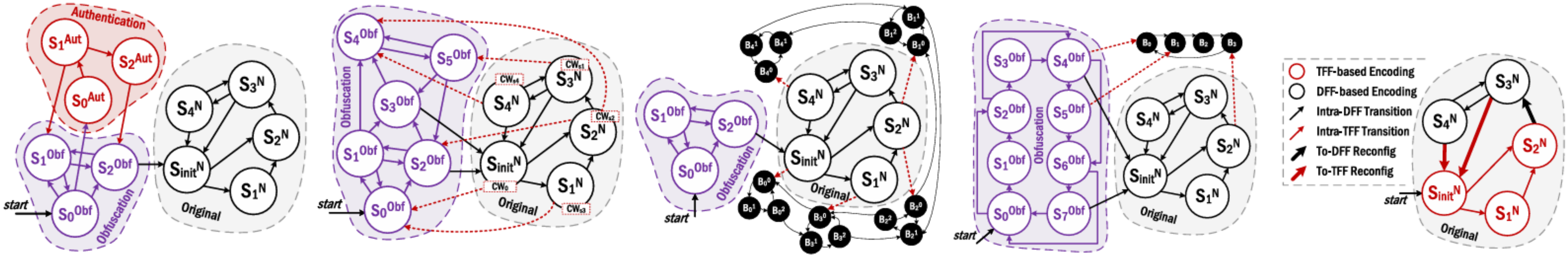
- **Static FSM structures are easy to reverse:** A simple switch-case FSM is trivial to decompile and analyze.
- **Obfuscation ≠ security:** FSMs only raise the **effort** required, they do not prevent reverse engineering.
- **Readable state names and transitions** make the design transparent unless further obfuscation is applied.

To Make FSM-Based Obfuscation Stronger

FSMs for obfuscation, combine them with:

1. **Opaque predicates:** Conditions that are always true/false at runtime but hard to analyze statically.
2. **State encoding:** Use hashed or scrambled identifiers instead of enums.
3. **Mixed logic states:** Split logical operations across multiple states unpredictably.
4. **Control flow flattening:** Restructure logic into a dispatcher loop, hiding original structure.
5. **Anti-debugging hooks:** Add traps or checks to detect reverse engineering tools.

Existing FSM obfuscation solutions



(a) HARPOON [30] (b) Interlocking FSM [32] (c) State Deflection [33] (d) Active Metering [37] (e) JANUS(-HD) [34, 35]

https://www.cs.ucf.edu/courses/cot6410/Spring2023/SampleTopics/FormalVerification/CLAIMED_Rodriguez_ReTrustFSM_Toward_RTL_Hardware_Obfuscation-A_Hybrid_FSM_Approach.pdf

Summary

- Finite State Machine – An Introduction
- Types of FSM
- Moore model
- Mealy Model
- Case study
  - String Recognizer
  - Sequence Detector
  - Traffic Light Controller
- VHDL Implementation of FSMs