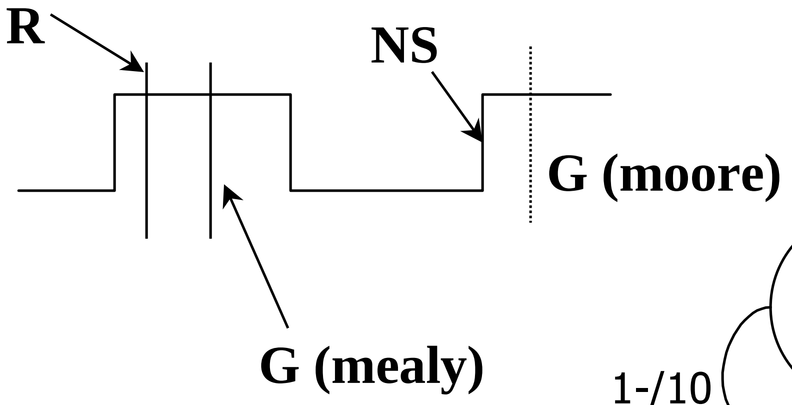
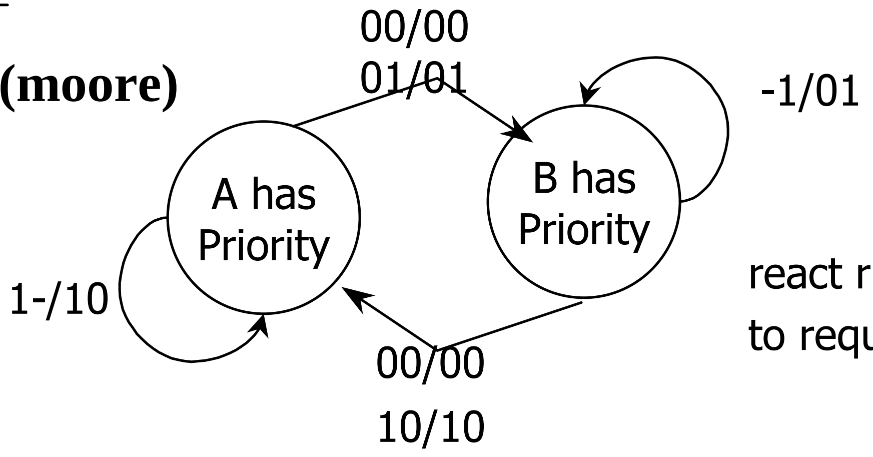
Remaining Topics

- ❑ Basic State Machine Design Study Problems
  - ➢ Study Problems: 8.3, 8.5, 8.6, 8.7, 8.14, 9.2, 9.5,
- ❑ Moore v. Mealy and Timing Issues
- ❑ Communicating State Machines
- ❑ State Minimization
- ❑ State Assignment
- ❑ State Machines in Verilog
- ❑ Datapath and Control Architecture
- ❑ Counter Based Design
- ❑ Computer Organization

Mealy vs. Moore machines

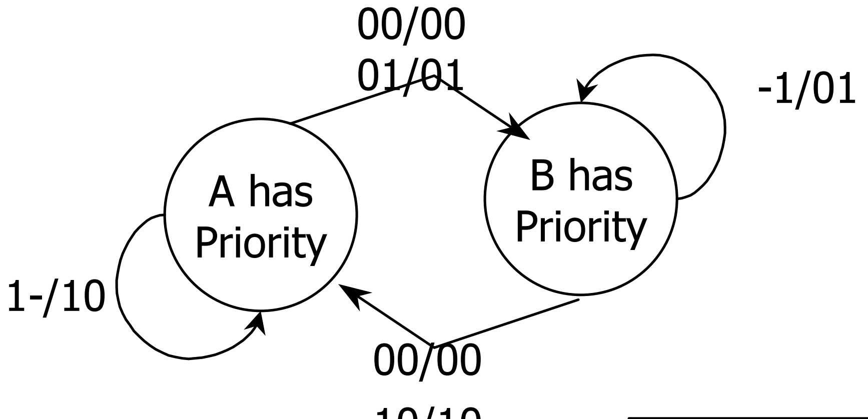
- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our ant brain is a Moore machine
  - output does not react immediately to input change
- We could have specified a Mealy FSM
  - outputs have immediate reaction to inputs
  - as inputs change, so does next state, doesn't commit until clocking event

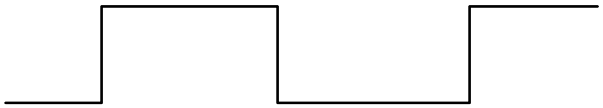


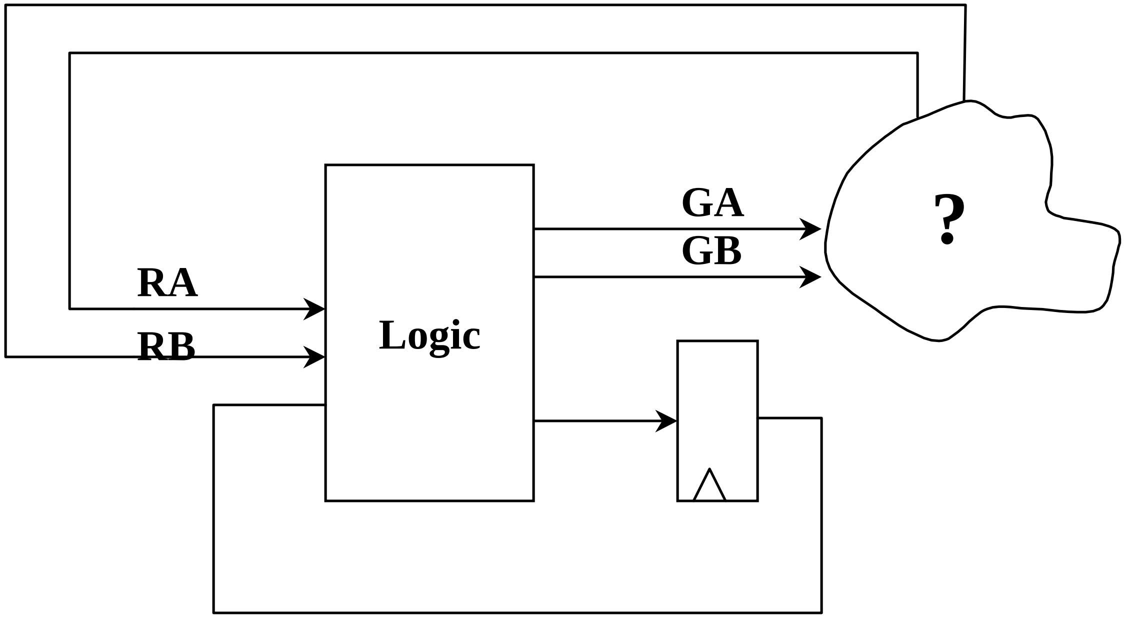


react right away
to requests

Timing Issues

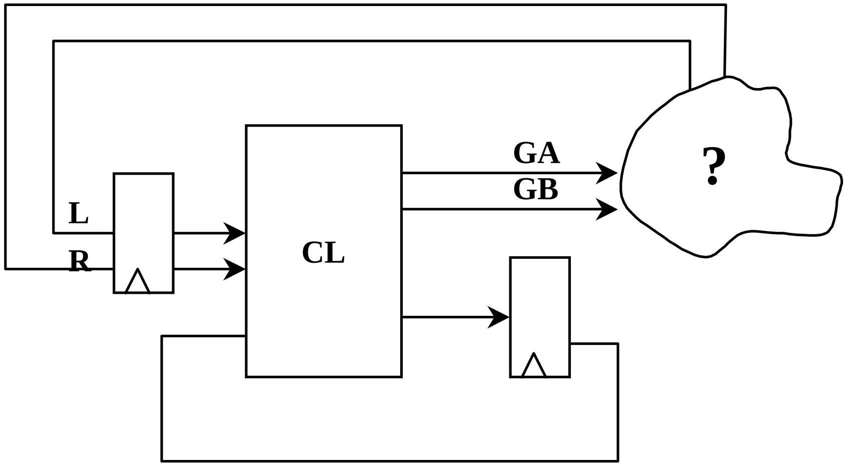






Is this safe?

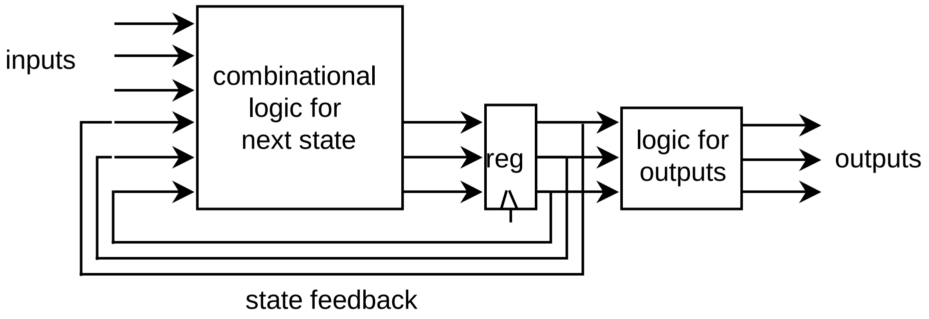
Mealy Machine with Synchronizers

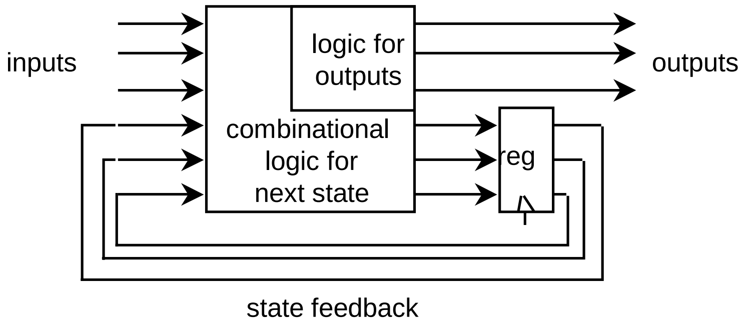


- ❑Synchronous (or registered) Mealy machine
  - ➢registered state AND outputs
  - ➢avoids ‘glitchy’ outputs
  - ➢easy to implement in PLDs
- ❑Moore machine with no output decoding
  - ➢outputs computed on transition to next state rather than after entering
  - ➢view outputs as expanded state vector

Comparison of Mealy and Moore machines

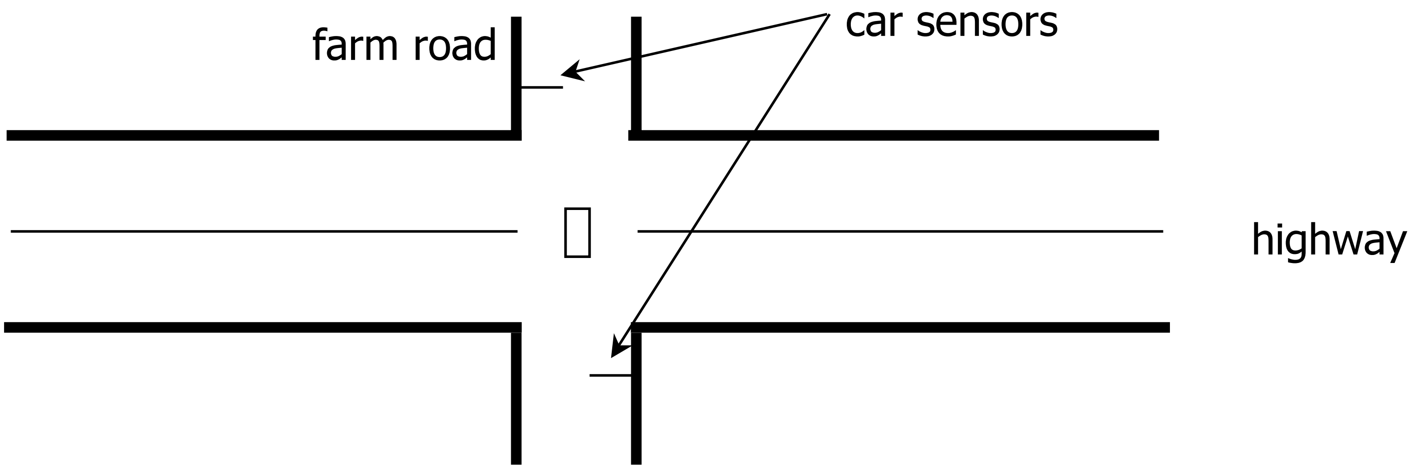
- ❑ Mealy machines tend to have less states
  - ➢ different outputs on arcs (n^2) rather than states (n)
- ❑ Moore machines are safer to use
  - ➢ outputs change at clock edge (always one cycle later)
  - ➢ in Mealy machines, input change can cause output change as soon as logic is done – a big problem when two machines are interconnected – asynchronous feedback
- ❑ Mealy machines react faster to inputs
  - ➢ react in same cycle – don't need to wait for clock
  - ➢ in Moore machines, more logic may be necessary to decode state into outputs – more gate delays after





Communicating State Machines (Decomposition)

- Example: 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



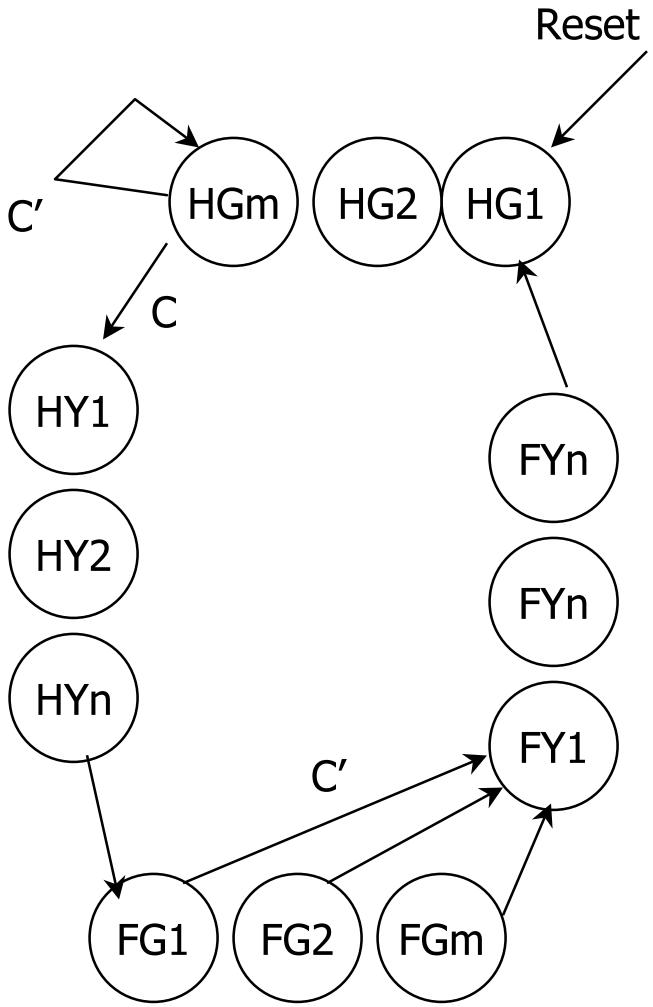
Example: traffic light controller (cont’)

- Tabulation of inputs and outputs

inputs	description
reset	place FSM in initial state
C	detect vehicle on the farm road

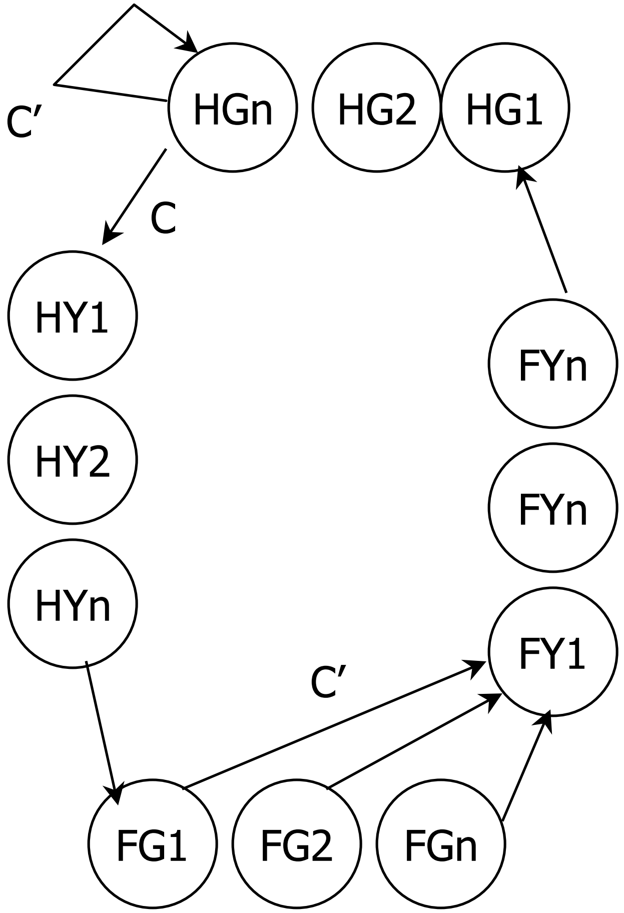
outputs	description
HG, HY, HR	highway lights
FG, FY, FR	Farm road lights

- Too many states

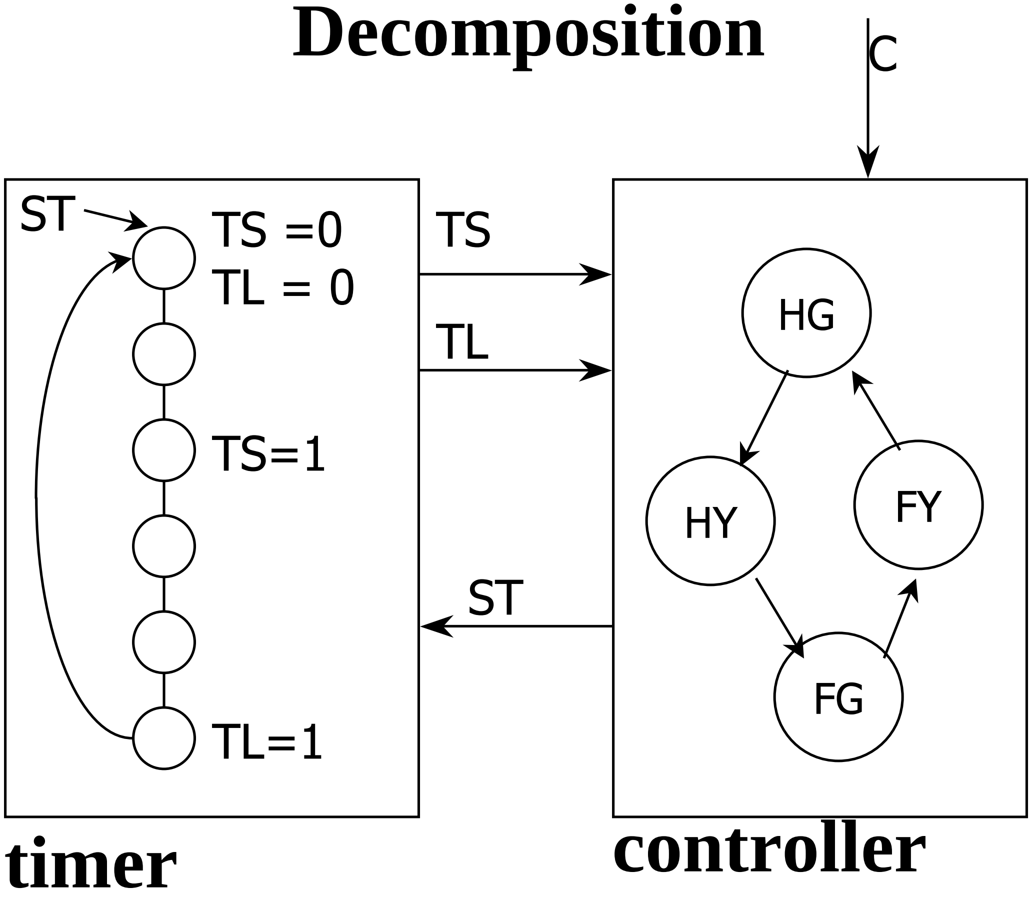


m>>n

Decomposition (Parallel Processes)

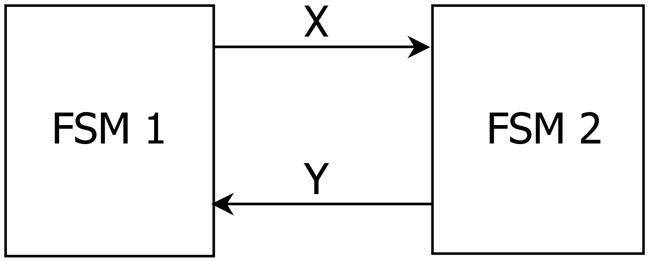


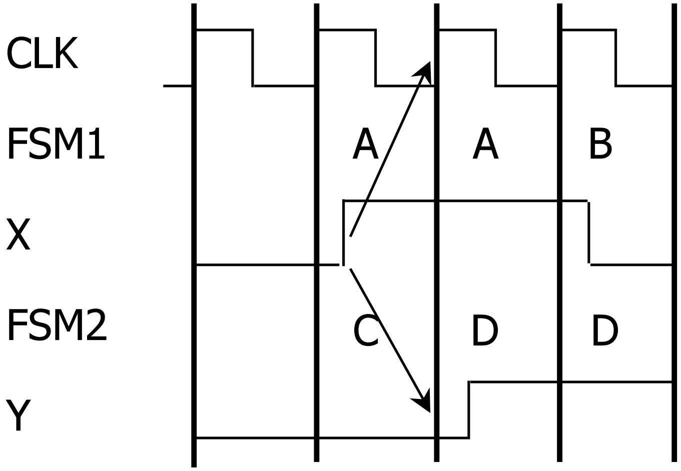
Decomposition



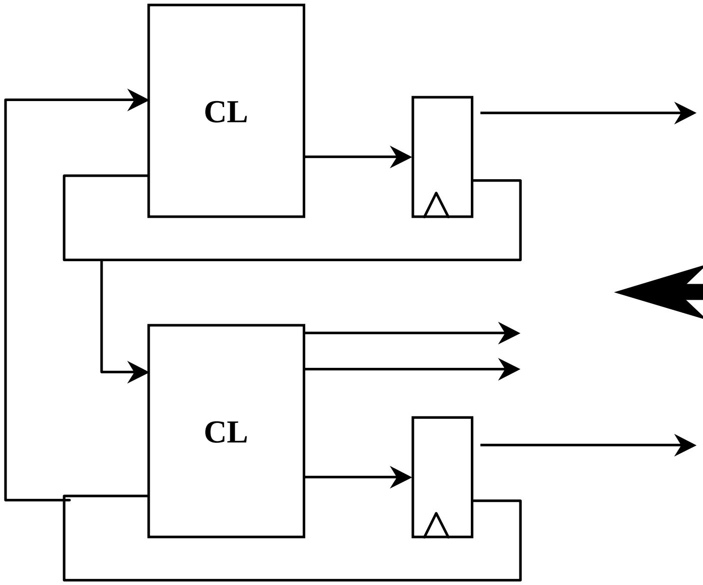
Communicating finite state machines

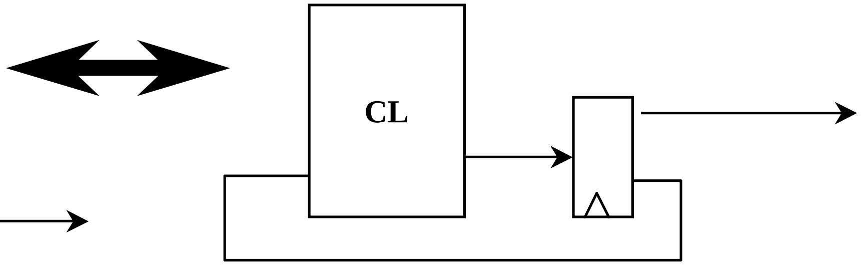
- ❑ One machine's output is another machine's input





- **Its really one machine, designed as two**
- **HW version of parallel processes**
- **Be careful w/ Mealy**

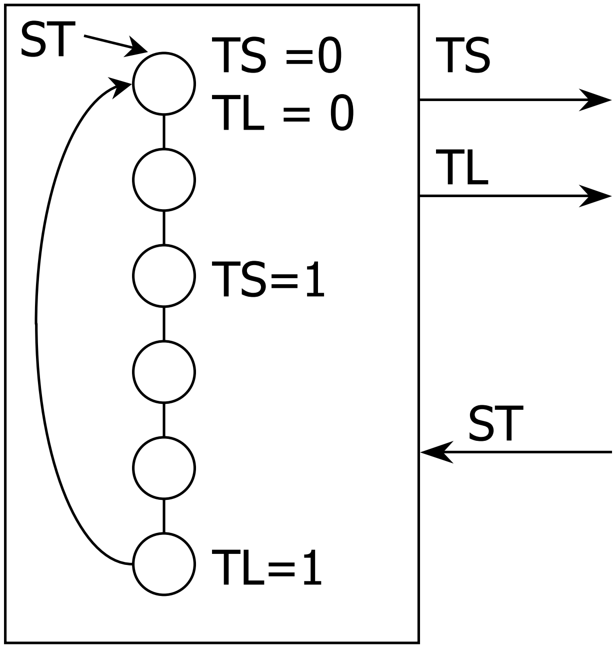




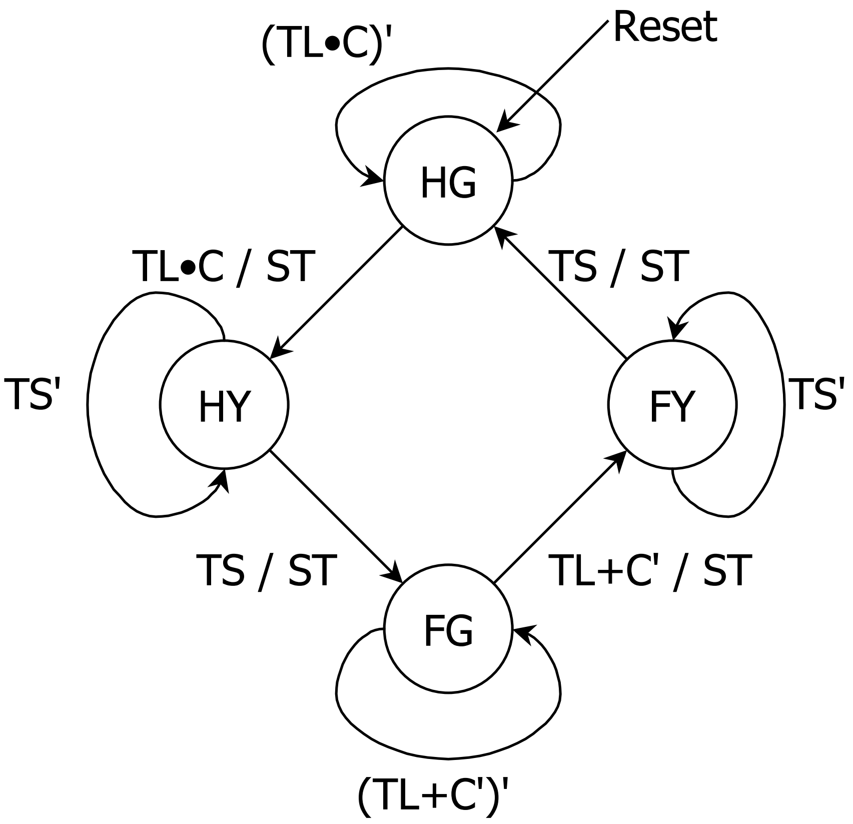
Decomposition: traffic light controller

- Controller State diagram

States: HG, HY, FG, FY



timer



Finite state machine optimization

- State minimization
  - fewer states require fewer state bits
  - fewer bits require fewer logic equations
- Encodings: state, inputs, outputs
  - state encoding with fewer bits has fewer equations to implement
    - however, each may be more complex
  - state encoding with more bits (e.g., one-hot) has simpler equations
    - complexity directly related to complexity of state diagram
  - input/output encoding may or may not be under designer control

FSM Optimization: Traffic Light Controller

- ❑ State Minimization
- ❑ State Encoding
- ❑ Output Encoding

output encoding – similar problem
to state assignment
(Green = 00, Yellow = 01, Red = 10)

Inputs			Present State	Next State	Outputs			
C	TL	TS				ST	H	F
0	–	–	HG	HG	0	Green	Red	
–	0	–	HG	HG	0	Green	Red	
1	1	–	HG	HY	1	Green	Red	
–	–	0	HY	HY	0	Yellow	Red	
–	–	1	HY	FG	1	Yellow	Red	
1	0	–	FG	FG	0	Red	Green	
0	–	–	FG	FY	1	Red	Green	
–	1	–	FG	FY	1	Red	Green	
–	–	0	FY	FY	0	Red	Yellow	
–	–	1	FY	HG	1	Red	Yellow	

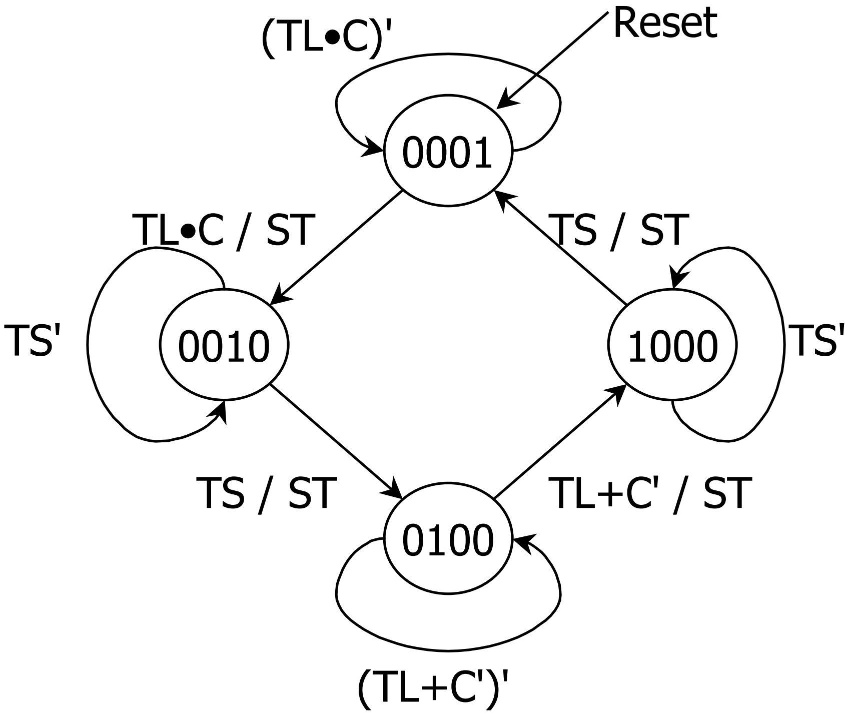
SA1:	HG = 00	HY = 01	FG = 11	FY = 10	(gray code)
SA2:	HG = 00	HY = 10	FG = 01	FY = 11	(sequential)
SA3:	HG = 0001	HY = 0010	FG = 0100	FY = 1000	(one-hot)

One-hot state assignment

- Simple
  - easy to encode
  - easy to debug
- Small logic functions
  - each state function requires only predecessor state bits as input
- Good for programmable devices
  - lots of flip-flops readily available
  - simple functions with small support (signals its dependent upon)
- Impractical for large machines
  - too many states require too many flip-flops
  - decompose FSMs into smaller pieces that can be one-hot encoded
- Many slight variations to one-hot
  - one-hot + all-0

State Assignment: One Hot

SA3: HG = 0001 HY = 0010 FG = 0100 FY = 1000



SA3 (One Hot) : Read Next State logic from the state diagram!

NS3 = C'•PS2 + TL•PS2 + TS'•PS3 NS2 = TS•PS1 + C•TL'•PS2

NS1 = C•TL•PS0 + TS'•PS1 NS0 = C'•PS0 + TL'•PS0 + TS•PS3

ST = C•TL•PS0 + TS•PS1 + C'•PS2 + TL•PS2 + TS•PS3

H1 = PS3 + PS2 H0 = PS1

F1 = PS1 + PS0 F0 = PS3

State Assignment: Comparison of Results

- SA1 (Gray Code)

  NS1 = C•TL'•PS1•PS0 + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
  NS0 = C•TL•PS1'•PS0' + C•TL'•PS1•PS0 + PS1'•PS0

  ST = C•TL•PS1'•PS0' + TS•PS1'•PS0 + TS•PS1•PS0' + C'•PS1•PS0 + TL•PS1•PS0
  H1 = PS1 H0 = PS1'•PS0
  F1 = PS1' F0 = PS1•PS0'

- SA2 (Sequential)

  NS1 = C•TL•PS1' + TS'•PS1 + C'•PS1'•PS0
  NS0 = TS•PS1•PS0' + PS1'•PS0 + TS'•PS1•PS0

  ST = C•TL•PS1' + C'•PS1'•PS0 + TS•PS1
  H1 = PS0 H0 = PS1•PS0'
  F1 = PS0' F0 = PS1•PS0

- SA3 (One Hot)

  NS3 = C'•PS2 + TL•PS2 + TS'•PS3 NS2 = TS•PS1 + C•TL'•PS2
  NS1 = C•TL•PS0 + TS'•PS1 NS0 = C'•PS0 + TL'•PS0 + TS•PS3

  ST = C•TL•PS0 + TS•PS1 + C'•PS2 + TL•PS2 + TS•PS3
  H1 = PS3 + PS2 H0 = PS1
  F1 = PS1 + PS0 F0 = PS3

But, if we already had a counter...

State assignment strategies

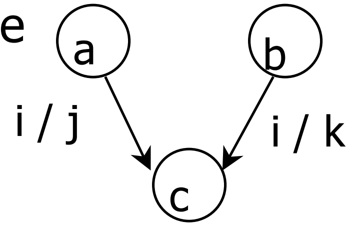
- Possible strategies
  - sequential – just number states as they appear in the state table (Timer)
  - random – pick random codes
  - one-hot – use as many state bits as there are states (Small)
  - output – use outputs to help encode states (Intersection)
  - heuristic – rules of thumb that seem to work in most cases (Opcode)
- No guarantee of optimality – another intractable problem

Heuristics for state assignment

- 1. Adjacent codes to states that share a common next state
  - group 1's in next state map

I	Q	Q^+	O
i	a	c	j
i	b	c	k

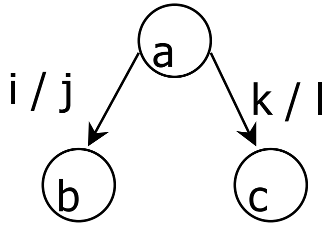
$c = i * a + i * b$



- 2. Adjacent codes to states that share a common ancestor state
  - group 1's in next state map

I	Q	Q^+	O
i	a	b	j
k	a	c	l

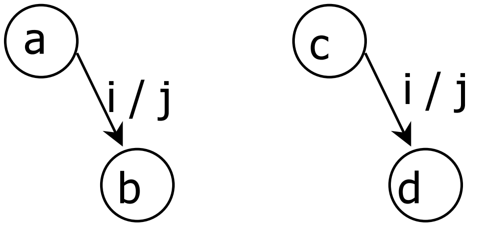
$b = i * a$
$c = k * a$



- 3. Adjacent codes to states that have a common output behavior
  - group 1's in output map

I	Q	Q^+	O
i	a	b	j
i	c	d	j

$j = i * a + i * c$
$b = i * a$
$d = i * c$



General approach to heuristic state assignment

- All current methods are variants of this
  - 1) determine which states “attract” each other (weighted pairs)
  - 2) generate constraints on codes (which should be in same cube)
  - 3) place codes on Boolean cube so as to maximize constraints satisfied
    sum (distance*weighted)
- Different weights make sense depending on whether we are optimizing for two-level or multi-level forms
- Can't consider all possible embeddings of state clusters in Boolean cube
  - heuristics for ordering embedding
  - to prune search for best embedding
  - expand cube (more state bits) to satisfy more constraints -- eventually becomes one-hot

Example

Common Next State

S3: S1, S4, S5

S2: S1, S2, S5

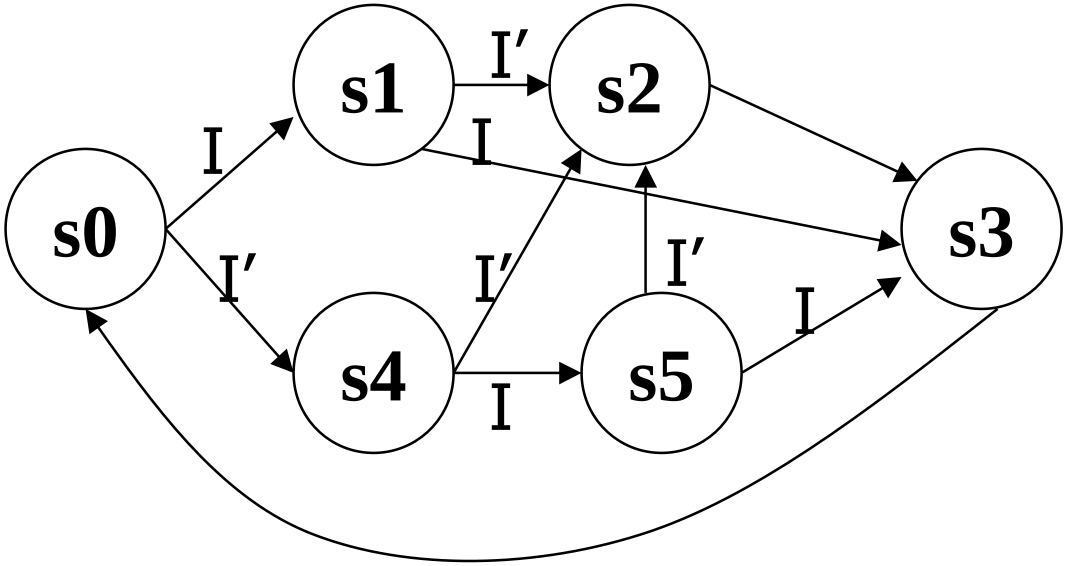
Common Ancestor

S0: S1,S4

S5: S2,S3

S4: S2,S5

S1: S2,S3



b2b1b0	00	01	11	10
0	?	S4	S5	S0
1	?	S1	S2	S3

Can we satisfy all Constraints?
How do I use empty cells?

Symbolically: S2 = I'(S1+S2+S4). If codes for S1,S2,S4 then logic is simpler

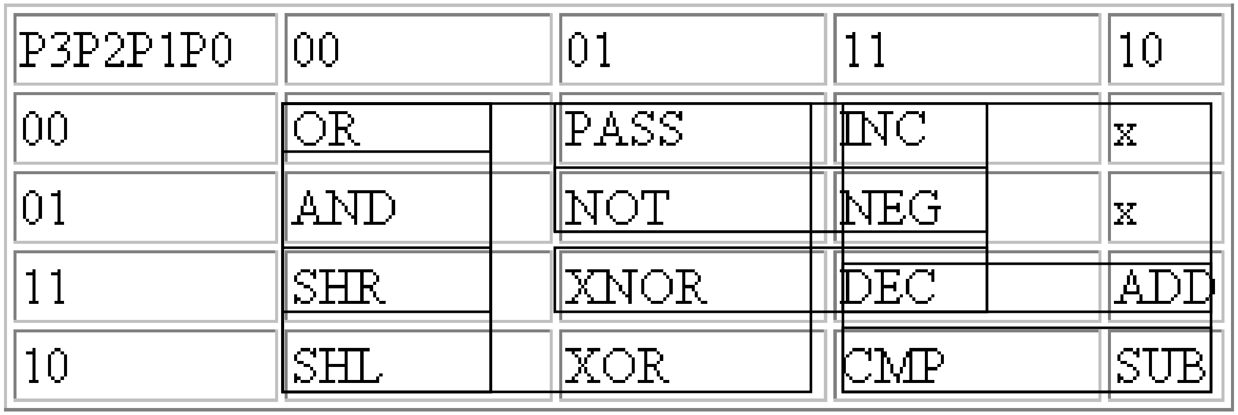
S1 = I(S0)
S4 = I'(S0)

If codes for S1, S4 are close then logic is simpler

Example: Output Encoding

If state machine's Ouputs are opcodes

s0 = [NEG] + [NOT]
s1 = [INC] + [DEC] + [PASS] + [NEG] + [NOT]
s2 = [DEC] + [SUB] + [CMP] + [XNOR]
s3 = [OR]
s4 = [ARITHMETIC]'
s5 = [SH(L/R)]
s6 = [AND] + [OR] + [SHR]
s7 = [ARITHMETIC]
s8 = [SHL]
s9 = ([ADD] + [DEC])'



P3P2P1P0	00	01	11	10
00	OR	PASS	INC	x
01	AND	NOT	NEG	x
11	SHR	XNOR	DEC	ADD
10	SHL	XOR	CMP	SUB

State Encoding: Output-based encoding

- Reuse outputs as state bits - use outputs to help distinguish states
  - why create new functions for state bits when output can serve as well
  - fits in nicely with synchronous Mealy implementations

Inputs			Present State	Next State	Outputs (each row is unique		
C	TL	TS			ST	H	F
0	–	–	HG	HG	0	00	10
–	0	–	HG	HG	0	00	10
1	1	–	HG	HY	1	00	10
–	–	0	HY	HY	0	01	10
–	–	1	HY	FG	1	01	10
1	0	–	FG	FG	0	10	00
0	–	–	FG	FY	1	10	00
–	1	–	FG	FY	1	10	00
–	–	0	FY	FY	0	10	01
–	–	1	FY	HG	1	10	01

HG = ST’ H1’ H0’ F1 F0’ + ST H1 H0’ F1’ F0
HY = ST H1’ H0’ F1 F0’ + ST’ H1’ H0 F1 F0’
FG = ST H1’ H0 F1 F0’ + ST’ H1 H0’ F1’ F0’
HY = ST H1 H0’ F1’ F0’ + ST’ H1 H0’ F1’ F0

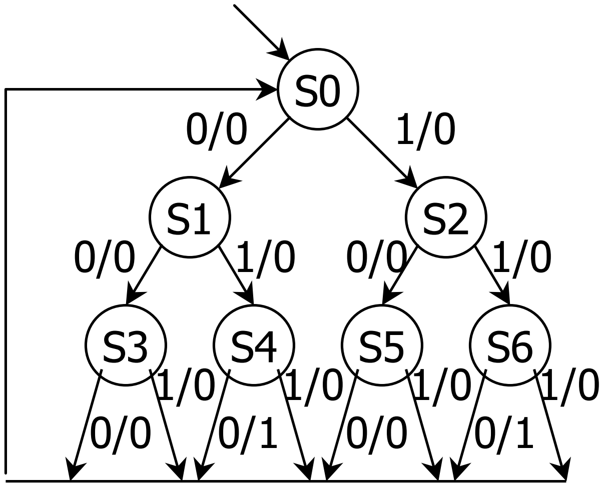
Output patterns are unique to states, we do not need ANY state bits – implement 5 functions (one for each output) instead of 7 (outputs plus 2 state bits)

Current state assignment approaches

- □ For tight encodings using close to the minimum number of state bits
  - ➢ best of 10 random seems to be adequate (averages as well as heuristics)
  - ➢ heuristic approaches are not even close to optimality
  - ➢ used in custom chip design
- □ One-hot encoding
  - ➢ easy for small state machines
  - ➢ generates small equations with easy to estimate complexity
  - ➢ common in FPGAs and other programmable logic
- □ Output-based encoding
  - ➢ ad hoc - no tools
  - ➢ most common approach taken by human designers
  - ➢ yields very small circuits for most FSMs
- □ Tools for
  - ➢ Partitioning (Decomposition)
  - ➢ State and output Encoding

Algorithmic approach to state minimization

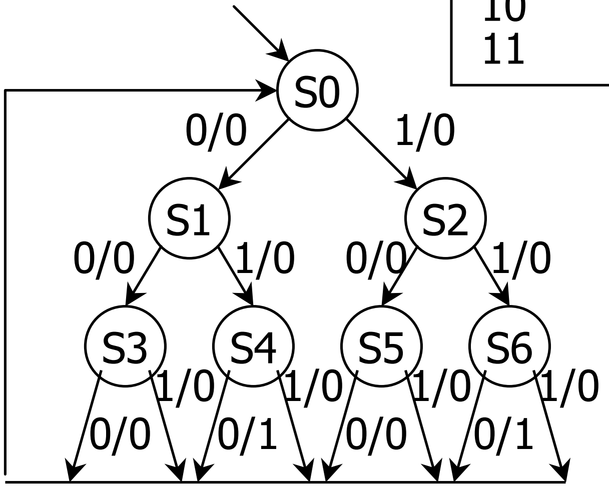
- Goal – identify and combine states that have equivalent behavior
- Equivalent states:
  - same output
  - for all input combinations, states transition to same or equivalent state
- Example: Sequence Detector: 010 or 110



State minimization example: Sequence Detector

- ❑ Goal – identify and combine states that have equivalent behavior
- ❑ Equivalent states:
  - ➢ same output
  - ➢ for all input combinations, states transition to same or equivalent state
- ❑ Sequence detector for 010 or 110

Input		Next State		Output	
Sequence	Present State	X=0	X=1	X=0	X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0



Row Matching Method

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1	S2	0	0
0	S1	S3	S4	0	0
1	S2	S5	S6	0	0
00	S3	S0	S0	0	0
01	S4	S0	S0	1	0
10	S5	S0	S0	0	0
11	S6	S0	S0	1	0

(S0 S1 S2 S3 S4 S5 S6)

(S0 S1 S2 S3 S5) (S4 S6)

(S0 S1 S2) (S3 S5) (S4 S6)

(S0) (S3 S5) (S1 S2) (S4 S6)

S4 is equivalent to S6

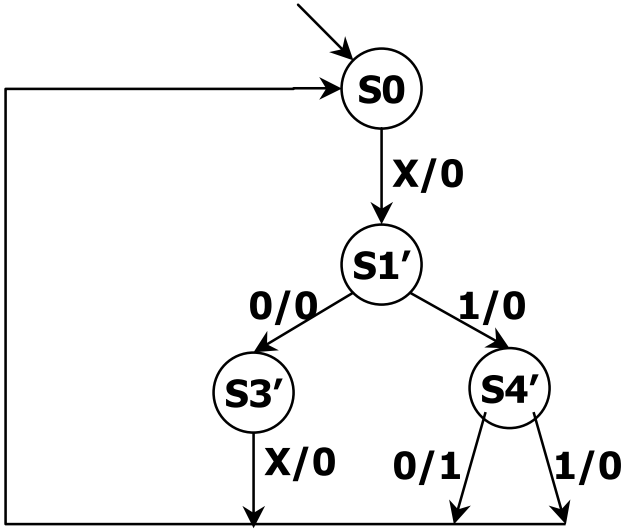
S3 is equivalent to S5

S1 is equivalent to S2

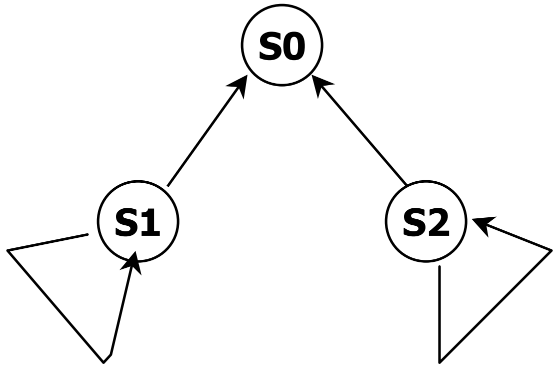
Minimized FSM

- State minimized sequence detector for 010 or 110

Input Sequence	Present State	Next State X=0	Next State X=1	Output X=0	Output X=1
Reset	S0	S1'	S1'	0	0
0 + 1	S1'	S3'	S4'	0	0
X0	S3'	S0	S0	0	0
X1	S4'	S0	S0	1	0

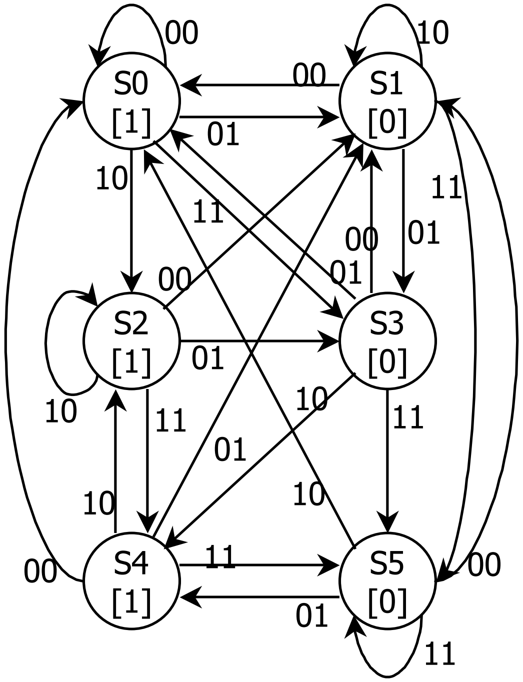


what about this case?



More complex state minimization

- Multiple input example



inputs here

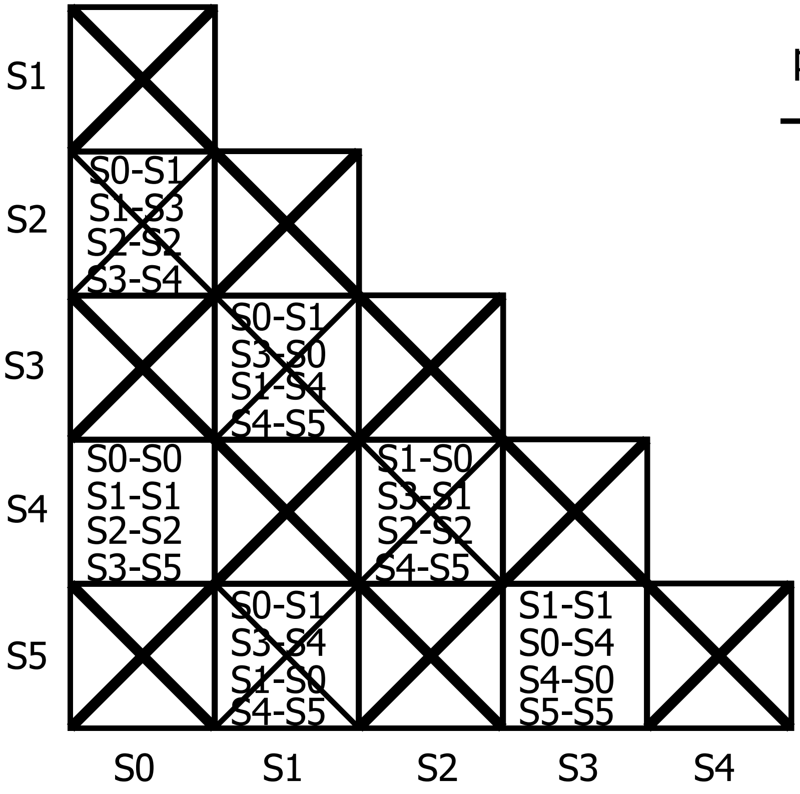
present state	00	01	10	11	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S4	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

next state

symbolic state
transition table

Minimized FSM

- Implication chart method
  - cross out incompatible states based on outputs
  - then cross out more cells if indexed chart entries are already crossed out



	S0	S1	S2	S3	S4
S1					
S2	~~S0-S1~~ ~~S1-S3~~ ~~S2-S2~~ ~~S3-S4~~				
S3		~~S0-S1~~ ~~S3-S0~~ ~~S1-S4~~ ~~S4-S5~~			
S4	S0-S0 S1-S1 S2-S2 S3-S5		~~S1-S0~~ ~~S3-S1~~ ~~S2-S2~~ ~~S4-S5~~		
S5		~~S0-S1~~ ~~S3-S4~~ ~~S1-S0~~ ~~S4-S5~~		S1-S1 S0-S4 S4-S0 S5-S5	

present state	next state 00	01	10	11	output
S0*	S0*	S1	S2	S3*	1
S1	S0*	S3*	S1	S3*	0
S2	S1	S3*	S2	S0*	1
S3*	S1	S0*	S0*	S3*	0

minimized state table
(S0==S4) (S3==S5)

Minimizing incompletely specified FSMs

- Equivalence of states is transitive when machine is fully specified
- But its not transitive when don't cares are present

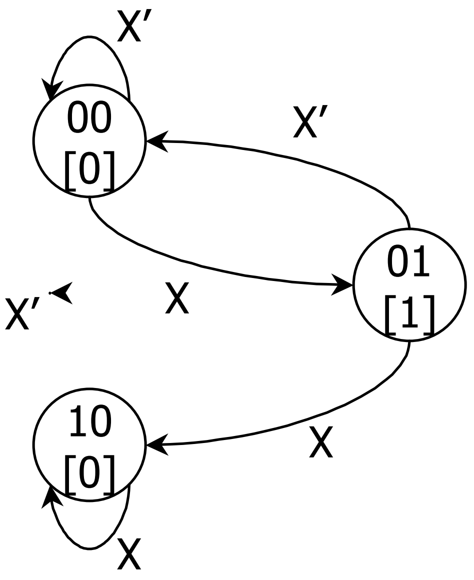
e.g.,

state	output	
S0	– 0	S1 is compatible with both S0 and S2
S1	1 –	but S0 and S2 are incompatible
S2	– 1	

- No polynomial time algorithm exists for determining best grouping of states into equivalent sets that will yield the smallest number of final states

Minimizing states may not yield best circuit

- Example: edge detector - outputs 1 when input changes from 0 to 1



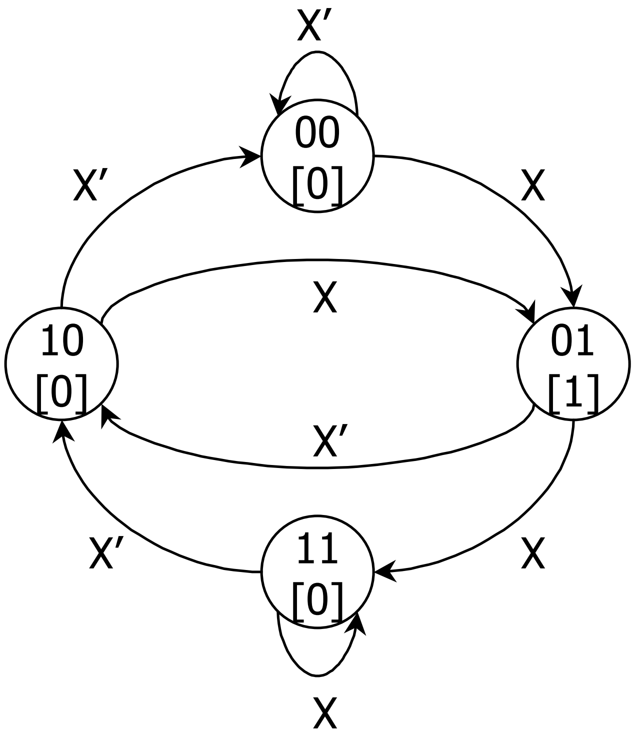
X	Q_1	Q_0	Q_1^+	Q_0^+
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
–	1	1	0	0

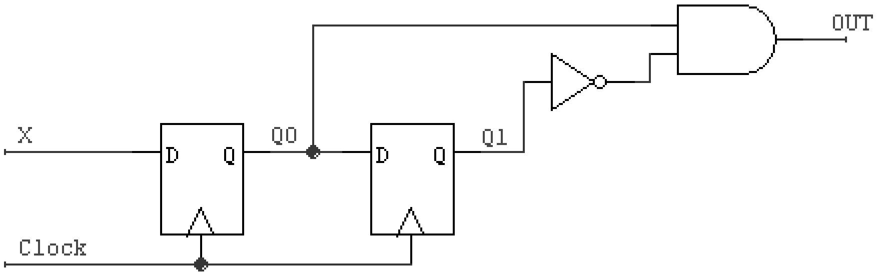
$Q_1^+ = X\ (Q_1 \text{ xor } Q_0)$

$Q_0^+ = X\ Q_1\ Q_0$

Another implementation of edge detector

- "Ad hoc" solution - not minimal but cheap and fast
- State compression from 4 to 3 states not very helpful





Sequential logic implementation summary

- Models for representing sequential circuits
  - abstraction of sequential elements
  - finite state machines and their state diagrams
  - inputs/outputs
  - Mealy, Moore, and synchronous Mealy machines
- Finite state machine design procedure
  - deriving state diagram
  - deriving state transition table
  - determining next state and output functions
  - implementing combinational logic
- Implementation of sequential logic
  - state minimization
  - state assignment
  - support in programmable logic devices

Sequential logic examples

- Finite state machine concept
  - FSMs are the decision making logic of digital designs
  - partitioning designs into datapath and control elements
  - when inputs are sampled and outputs asserted
- Basic design approach: a 4-step design process
- Implementation examples and case studies
  - finite-string pattern recognizer

General FSM design procedure

- ❑ (1) Determine inputs and outputs
- ❑ (2) Determine possible states of machine
  - ➢ – state minimization
- ❑ (3) Encode states and outputs into a binary code
  - ➢ – state assignment or state encoding
  - ➢ – output encoding
  - ➢ – possibly input encoding (if under our control)
- ❑ (4) Realize logic to implement functions for states and outputs
  - ➢ – Verilog model for simulation and synthesis
  - ➢ -- combinational logic implementation and optimization
  - ➢ – choices made in steps 2 and 3 can have large effect on resulting logic

Finite string pattern recognizer (step 1)

- Finite string pattern recognizer
  - one input (X) and one output (Z)
  - output is asserted whenever the input sequence …010… has been observed, as long as the sequence 100 has never been seen

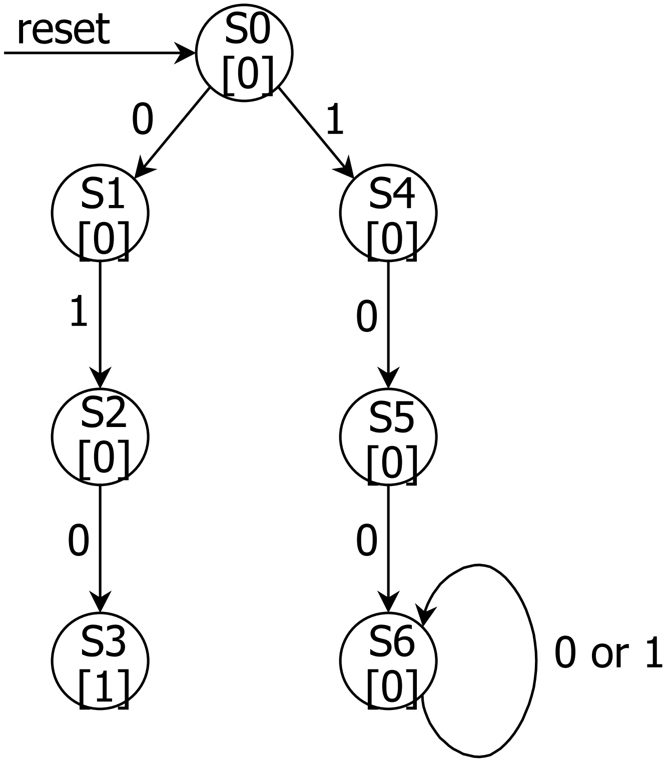
- Step 1: understanding the problem statement
  - sample input/output behavior:

```
X:   0 0 1 0 1 0 1 0 0 1 0 …
Z:   0 0 0 1 0 1 0 1 0 0 0 …

X:   1 1 0 1 1 0 1 0 0 1 0 …
Z:   0 0 0 0 0 0 0 1 0 0 0 …
```

Finite string pattern recognizer (step 2)

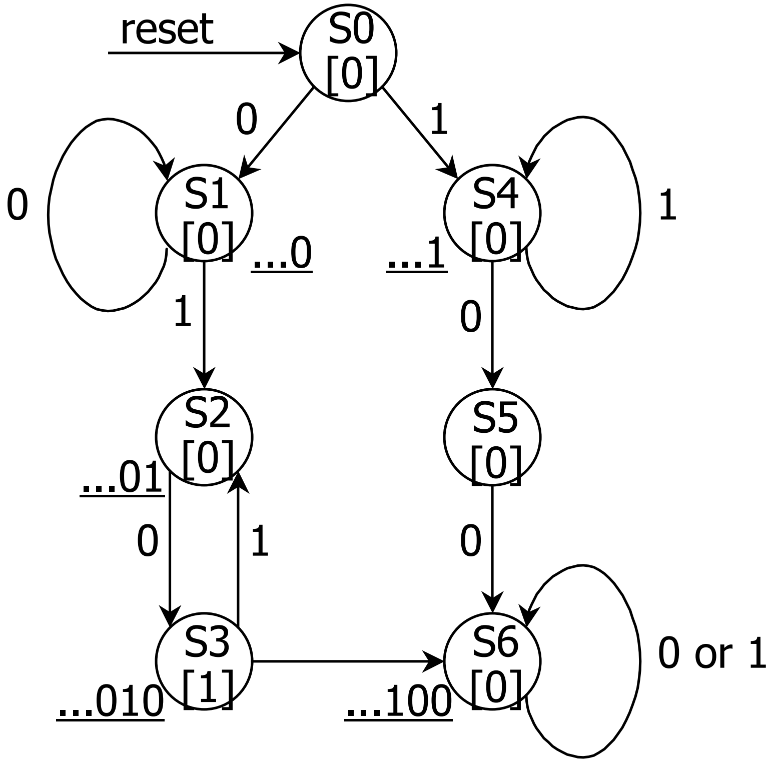
- Step 2: draw state diagram
  - for the strings that must be recognized, i.e., 010 and 100
  - a Moore implementation



Finite string pattern recognizer (step 2, cont'd)

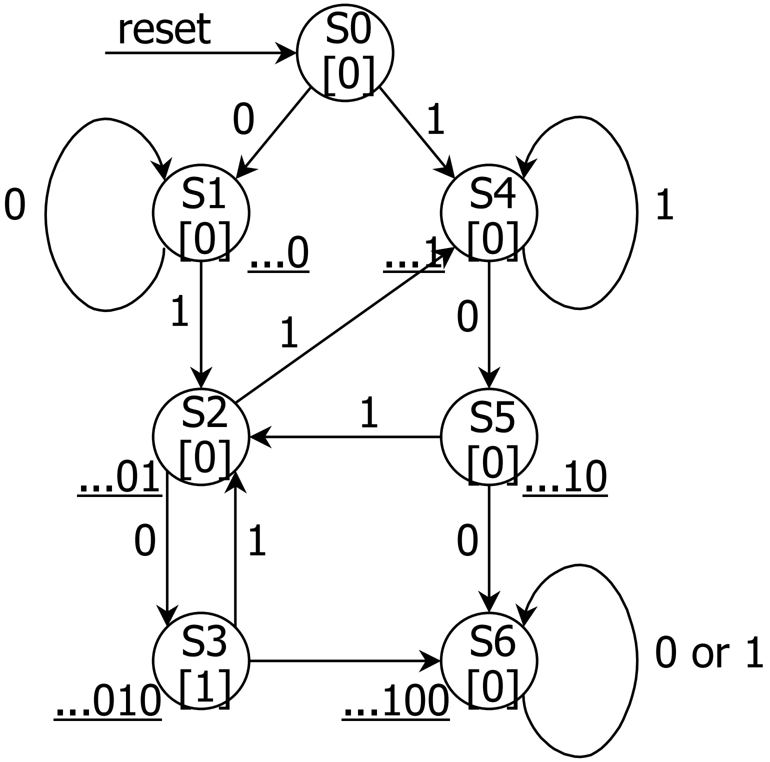
- Exit conditions from state S3: have recognized …010
  - if next input is 0 then have …0100 = ...100 (state S6)
  - if next input is 1 then have …0101 = …01 (state S2)

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 string pattern recognizer (step 2, cont'd)

- S2 and S5 still have 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
- Reuse states as much as possible
  - look for same meaning
  - state minimization leads to smaller number of bits to represent states
- Once all states have a complete set of transitions we have a final state diagram



Finite string pattern recognizer (step 3)

- Verilog description including state assignment (or state encoding)

```
module string (clk, X, rst, Q0, Q1, Q2, Z);
input clk, X, rst;
output Q0, Q1, Q2, Z;

reg state[0:2];
‘define S0 = [0,0,0]; //reset state
‘define S1 = [0,0,1]; //strings ending in   ...0
‘define S2 = [0,1,0]; //strings ending in  ...01
‘define S3 = [0,1,1]; //strings ending in ...010
‘define S4 = [1,0,0]; //strings ending in   ...1
‘define S5 = [1,0,1]; //strings ending in  ...10
‘define S6 = [1,1,0]; //strings ending in ...100

assign Q0 = state[0];
assign Q1 = state[1];
assign Q2 = state[2];
assign Z = (state == ‘S3);
```

```
always @(posedge clk) begin
  if rst state = ‘S0;
  else
    case (state)
      ‘S0: if (X) state = ‘S4 else state = ‘S1;
      ‘S1: if (X) state = ‘S2 else state = ‘S1;
      ‘S2: if (X) state = ‘S4 else state = ‘S3;
      ‘S3: if (X) state = ‘S2 else state = ‘S6;
      ‘S4: if (X) state = ‘S4 else state = ‘S5;
      ‘S5: if (X) state = ‘S2 else state = ‘S6;
      ‘S6: state = ‘S6;
      default: begin
        $display (“invalid state reached”);
        state = 3’bxxx;
    endcase

end

endmodule
```

Finite string pattern recognizer

- ❑ Review of process
  - ➢ understanding problem
    - ▪ write down sample inputs and outputs to understand specification
  - ➢ derive a state diagram
    - ▪ write down sequences of states and transitions for sequences to be recognized
  - ➢ minimize number of states
    - ▪ add missing transitions; reuse states as much as possible
  - ➢ state assignment or encoding
    - ▪ encode states with unique patterns
  - ➢ simulate realization
    - ▪ verify I/O behavior of your state diagram to ensure it matches specification