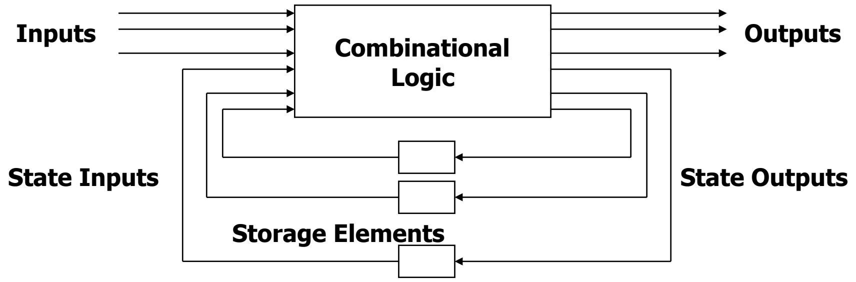
Finite State Machines

- Finite State Machines (FSMs)
  - general models for representing sequential circuits
  - two principal types based on output behavior (Moore and Mealy)
- Basic sequential circuits revisited and cast as FSMs
  - shift registers
  - counters
- Design procedure for FSMs
  - state diagrams
  - state transition table
  - next state functions
  - potential optimizations
- Hardware description languages

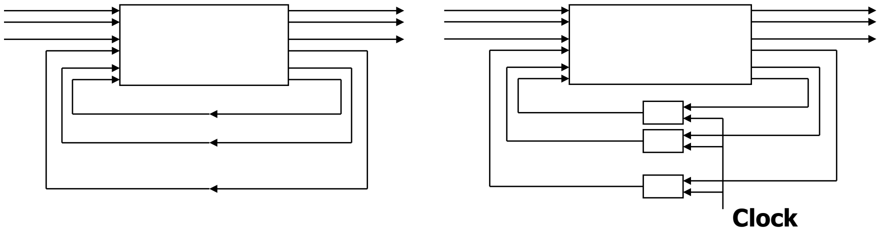
Abstraction of state elements

- Divide circuit into combinational logic and state
- Localize the feedback loops and make it easy to break cycles
- Implementation of storage elements leads to various forms of sequential logic



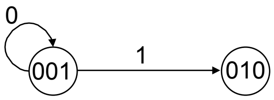
Forms of sequential logic

- Asynchronous sequential logic – state changes occur whenever state inputs change (seq. elements may be simple wires or delay elements)
- Synchronous sequential logic – state changes occur in lock step across all storage elements (using a periodic waveform to trigger FFs)



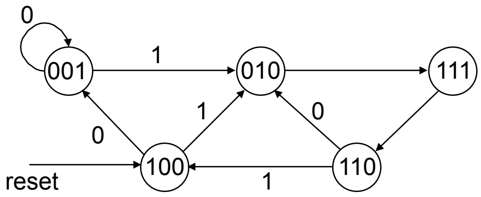
Finite state machine representations

- States: determined by possible values in sequential storage elements
- Transitions: change of state
- Clock: controls when state can change by controlling storage elements
- Sequential logic
  - transitions through a series of states
  - which transitions are taken depends on values of input signals
  - clock period defines elements of input sequence



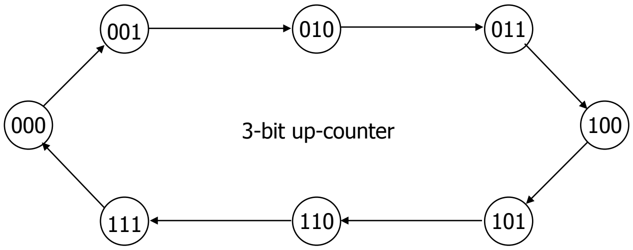
Example finite state machine diagram

- 5 states
- 8 other transitions between states
  - 6 conditioned by input
  - 1 self-transition (on 0 from 001 to 001)
  - 2 independent of input (to/from 111)
- 1 reset transition (from all states) to state 100
  - represents 5 transitions (from each state to 100), one a self-arc
  - simplifies condition on other transitions –all would include **AND reset'**)
  - short-hand – rather than drawing a transition arc from each state



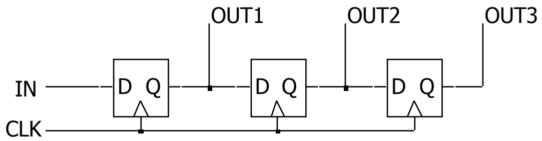
Counters are simple finite state machines

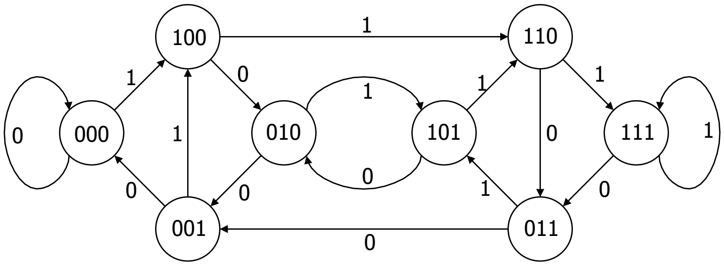
- Counters
  - proceed through well-defined sequence of states (if enabled)
- Many types of counters: binary, BCD, Gray-code, etc.…
  - 3-bit up-counter: 000, 001, 010, 011, 100, 101, 110, 111, 000, ...
  - 3-bit down-counter: 111, 110, 101, 100, 011, 010, 001, 000, 111, ...



Can any sequential system be represented with a state diagram?

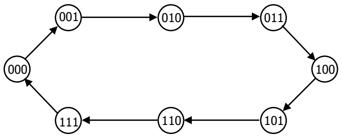
- Shift register
  - input value shown on transition arcs
  - output values shown within state node

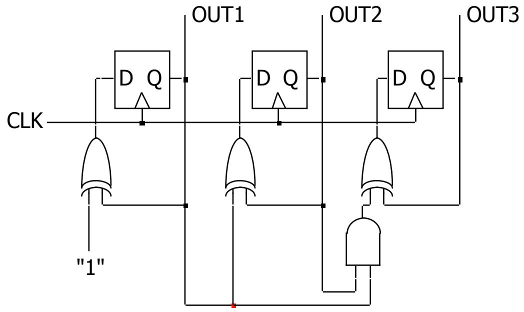




How do we turn a state diagram into logic?

- Counter
  - 3 flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait long enough for combinational logic to compute new value
    - though waiting too long is a waste of time



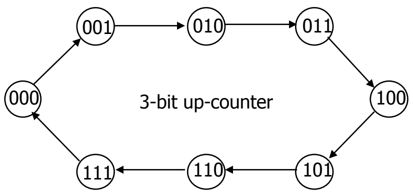


FSM design procedure

- We started with counters
  - simple because the output is just its state
  - simple because there is no input used to choose next state
- State diagram to state transition table
  - tabular form of state diagram
  - like a truth-table
- State encoding
  - decide on representation of states
  - for counters it is simple: just its value
- Implementation
  - flip-flop for each state bit
  - combinational logic based on encoding

FSM design procedure: state diagram to encoded state transition table

- Tabular form of state diagram
- Like a truth-table (specify output for all input combinations)
- Encoding of states: easy for counters – just use value



current state		next state	
0	000	001	1
1	001	010	2
2	010	011	3
3	011	100	4
4	100	101	5
5	101	110	6
6	110	111	7
7	111	000	0

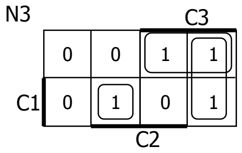
Implementation

- D flip-flop for each state bit
- Combinational logic based on state encoding

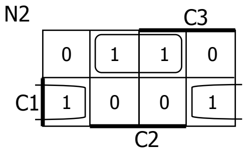
C3	C2	C1	N3	N2	N1
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Verilog notation to show function represents an input to D-FF

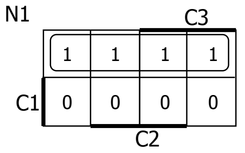
N1 <= C1′
<= C1 xor 1
N2 <= C1C2′ + C1′C2
<= C1 xor C2
N3 <= C1C2C3′ + C1′C3 + C2′C3
<= (C1C2)C3′ + (C1′ + C2′)C3
<= (C1C2)C3′ + (C1C2)′C3
<= (C1C2) xor C3


N3

				C3
	0	0	1	1
C1	0	1	0	1
		C2		


N2

				C3
	0	1	1	0
C1	1	0	0	1
		C2		

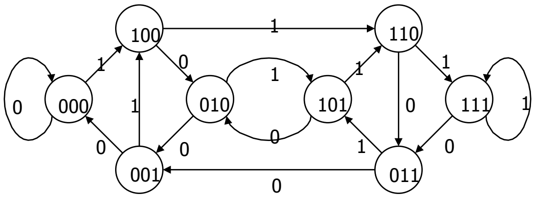

N1

				C3
	1	1	1	1
C1	0	0	0	0
		C2		

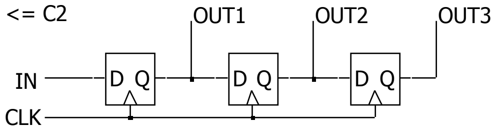
Back to the shift register

- Input determines next state

In	C1	C2	C3	N1	N2	N3
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	1	1
0	1	1	1	0	1	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	1	0	1
1	0	1	1	1	0	1
1	1	0	0	1	1	0
1	1	0	1	1	1	0
1	1	1	0	1	1	1
1	1	1	1	1	1	1

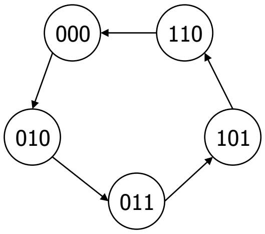


N1 <= In
N2 <= C1
N3 <= C2



More complex counter example

- Complex counter
  - repeats 5 states in sequence
  - not a binary number representation
- Step 1: derive the state transition diagram
  - count sequence: 000, 010, 011, 101, 110
- Step 2: derive the state transition table from the state transition diagram

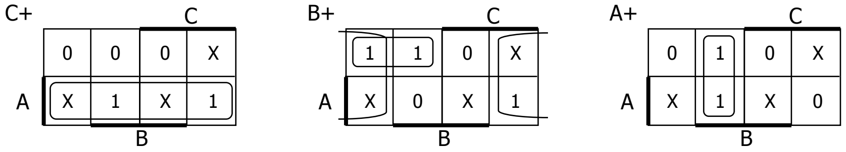


Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	–	–	–
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	–	–	–
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	–	–	–

note the don't care conditions that arise from the unused state codes

More complex counter example (cont'd)

- Step 3: K-maps for next state functions



C+

0	0	0	X
X	1	X	1

B+

1	1	0	X
X	0	X	1

A+

0	1	0	X
X	1	X	0

C+ <= A

B+ <= B′ + A′C′

A+ <= BC′

Self-starting counters (cont'd)

- Re-deriving state transition table from don't care assignment

C+

			C	C
	0	0	0	0
A	1	1	1	1
		B	B	

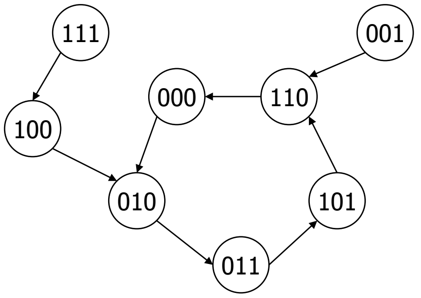
B+

			C	C
	1	1	0	1
A	1	0	0	1
		B	B	

A+

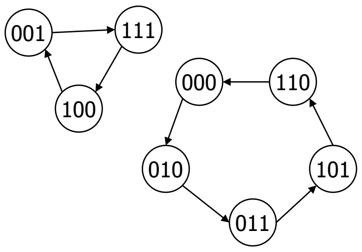
			C	C
	0	1	0	0
A	0	1	0	0
		B	B	

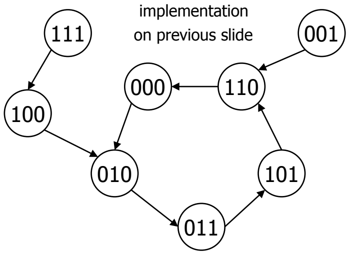
Present State			Next State		
C	B	A	C+	B+	A+
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	0	1	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	1	1	0	0



Self-starting counters

- Start-up states
  - at power-up, counter may be in an unused or invalid state
  - designer must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - design counter so that invalid states eventually transition to a valid state
    - this may or may not be acceptable
  - may limit exploitation of don't cares



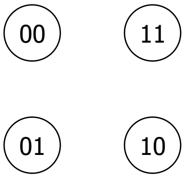


implementation on previous slide

Activity

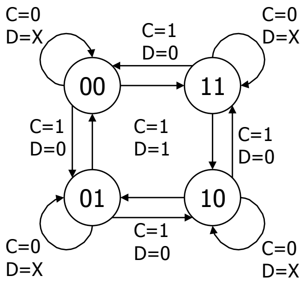
- 2-bit up-down counter (2 inputs)
  - direction: D = 0 for up, D = 1 for down
  - count: C = 0 for hold, C = 1 for count

S1	S0	C	D	N1	N0



Activity

- 2-bit up-down counter (2 inputs)
  - direction: D = 0 for up, D = 1 for down
  - count: C = 0 for hold, C = 1 for count



S1	S0	C	D	N1	N0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

Activity (cont'd)

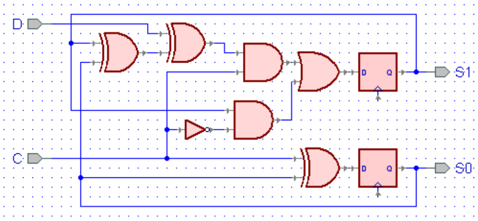
S1	S0	C	D	N1	N0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	0	1	1	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	0

			S1		
	0	0	1	1	
	0	0	1	1	D
C	1	0	1	0	
	0	1	0	1	
		S0			

N1 <= C'S1
+ CDS0'S1' + CDS0S1
+ CD'S0S1' + CD'S0'S1
<= C'S1
+ C (D' (S1 ⊕ S0) + D (S1 == S0))
<= C'S1 + C (D ⊕ (S1 ⊕ S0))

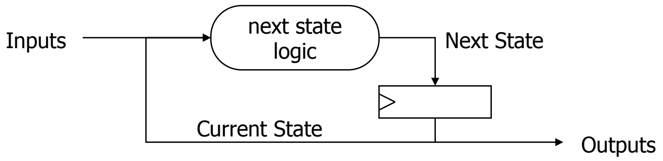
			S1		
	0	1	1	0	
	0	1	1	0	D
C	1	0	0	1	
	1	0	0	1	
		S0			

N0 <= CS0' + C'S0



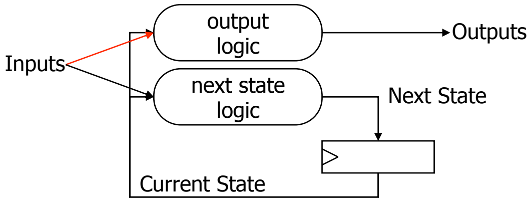
Counter/shift-register model

- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - values of flip-flops



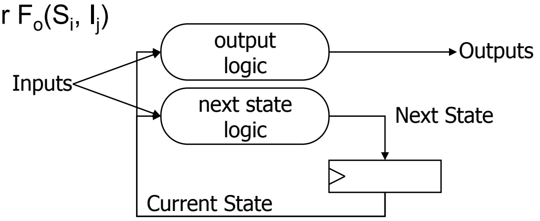
General state machine model

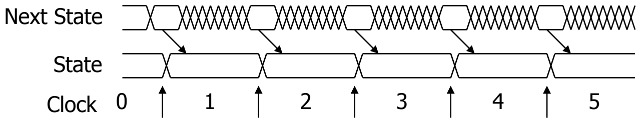
- Values stored in registers represent the state of the circuit
- Combinational logic computes:
  - next state
    - function of current state and inputs
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)



State machine model (cont'd)

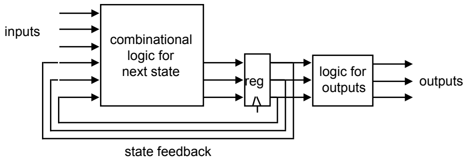
- States: S_1, S_2, ..., S_k
- Inputs: I_1, I_2, ..., I_m
- Outputs: O_1, O_2, ..., O_n
- Transition function: $F_s(S_i, I_j)$
- Output function: $F_o(S_i)$ or $F_o(S_i, I_j)$



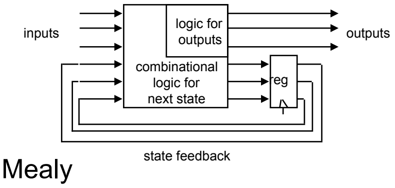


Comparison of Mealy and Moore machines (cont'd)

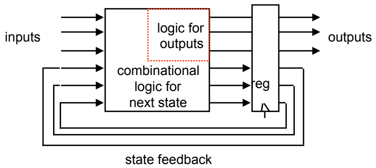
- Moore



- Mealy



- Synchronous Mealy

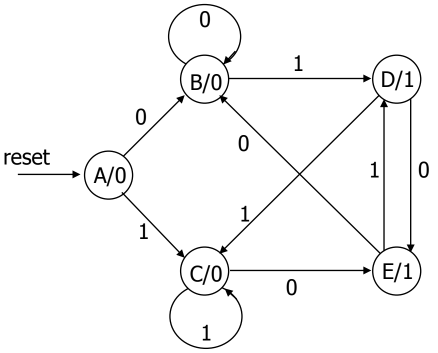


Comparison of Mealy and Moore machines

- Mealy machines tend to have less states
  - outputs depend on arc taken from a state to another state (n^2) rather than just the state of the FSM (n)
- Moore machines are safer to use
  - outputs change at next clock edge
  - in Mealy machines, input change can cause async output change (after prop delay of logic) – a BIG problem when two machines are interconnected – asynchronous feedback may occur if one isn't careful (input to fsm1, changes output of fsm1, which is an input to fsm2, whose output changes, and turns out to be input to fsm1)
- Mealy machines advantage? – they 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 that are needed – more gate delays after clock edge

Specifying outputs for a Moore machine

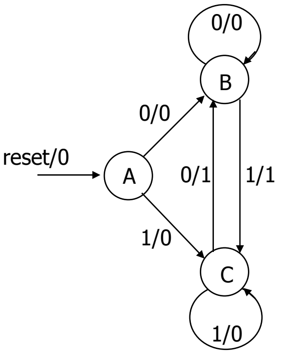
- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10



		current	next	
reset	input	state	state	output
1	–	–	A	
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	D	0
0	0	C	E	0
0	1	C	C	0
0	0	D	E	1
0	1	D	C	1
0	0	E	B	1
0	1	E	D	1

Specifying outputs for a Mealy machine

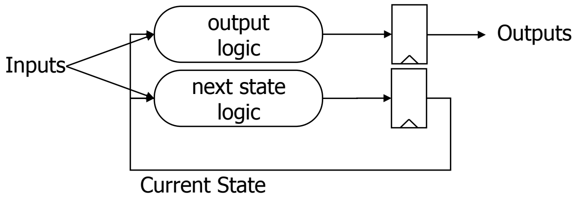
- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10



		current	next	
reset	input	state	state	output
1	–	–	A	0
0	0	A	B	0
0	1	A	C	0
0	0	B	B	0
0	1	B	C	1
0	0	C	B	1
0	1	C	C	0

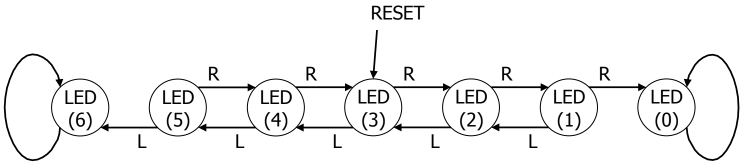
Synchronous Mealy machine (really Moore)

- Synchronous (or registered) Mealy machine
  - state AND output FFs
  - avoids ‘glitchy’ outputs (no hazards)
  - easy to implement in programmable logic (function blocks + FF)
- Same as a Moore machine with no output decoding
  - outputs computed on transition to next state rather than after entering state
  - view outputs as expanded state vector – “output-encoded state”

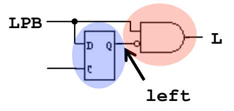


Tug-of-War Game FSM

- Tug of War game
  - 7 LEDS, 2 push buttons (LPB, RPB)



Light Game FSM Verilog



positive edge detector

```
module Light_Game (LEDS, LPB, RPB, CLK, RESET);

   input LPB ;
   input RPB ;
   input CLK ;
   input RESET;
   output [6:0] LEDS ;

   reg [6:0] position;
   reg left;
   reg right;
```

combinational logic and wires

```
   wire L, R;
   assign L = ~left && LPB;
   assign R = ~right && RPB;
   assign LEDS = position;
```

sequential logic

```
   always @(posedge CLK)
        begin
             left <= LPB;
             right <= RPB;
             if (RESET) position <= 7'b0001000;
             else if ((position == 7'b0000001) || (position == 7'b1000000)) ;
             else if (L) position <= position << 1;
             else if (R) position <= position >> 1;
        end

endmodule
```

Do you see a problem with this game?

Activity

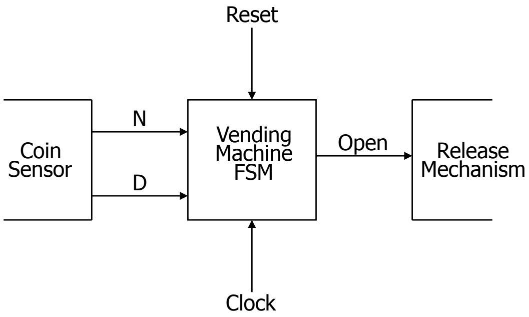
- Where is the problem? What is the fix?

```
always @(posedge CLK) begin
   left <= LPB;
   right <= RPB;
   if (RESET) position <= 7'b0001000;
   else if ( (position == 7'b0000001) || (position == 7'b1000000) )
               position <= position;
   else if (L) position <= position << 1;
   else if (R) position <= position >> 1;
end
```

```
always @(posedge CLK) begin  // no longer biased in favor of L player
   left <= LPB;
   right <= RPB;
   if (RESET) position <= 7'b0001000;
   else if ( (position == 7'b0000001) || (position == 7'b1000000) )
                    position <= position;
   else if (L & ~R) position <= position << 1; // correct error in state diag.
   else if (R & ~L) position <= position >> 1; // favoring L player
   else             position <= position;      // otherwise, just hold
```

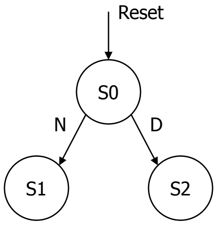
Example: vending machine

- Release item after 15 cents are deposited
- Single coin slot for dimes, nickels
- No change



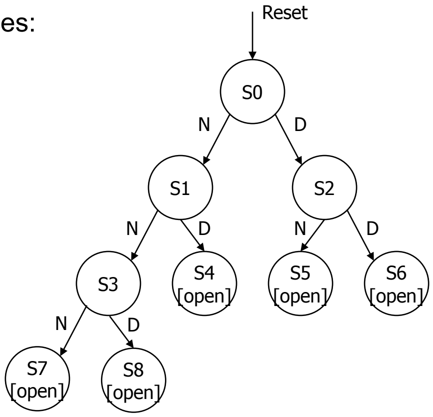
Example: vending machine (cont'd)

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)



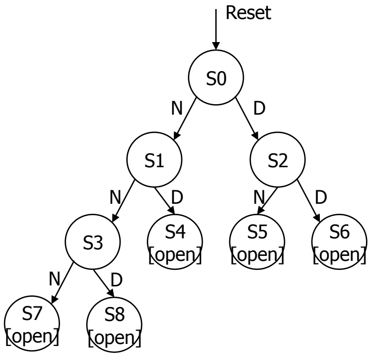
Example: vending machine (cont'd)

- Suitable abstract representation
  - tabulate typical input sequences:
    - 3 nickels
    - nickel, dime
    - dime, nickel
    - two dimes
  - draw state diagram:
    - inputs: N, D, reset
    - output: open chute
  - assumptions:
    - assume N and D asserted for one cycle
    - each state has a self loop for N = D = 0 (no coin)



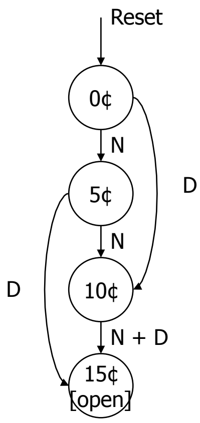
Activity: reuse states

- Redraw the state diagram using as few states as possible



Example: vending machine (cont'd)

- Minimize number of states - reuse states whenever possible



present state	inputs D	N	next state	output open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	–	–
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	–	–
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	–	–
15¢	–	–	15¢	1

symbolic state table

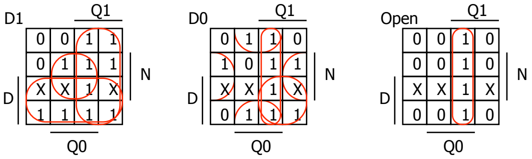
Example: vending machine (cont'd)

- Uniquely encode states

present state Q1	Q0	inputs D	N	next state D1	D0	output open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	–	–	–
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	0
		1	1	–	–	–
1	0	0	0	1	0	0
		0	1	1	1	0
		1	0	1	1	0
		1	1	–	–	–
1	1	–	–	1	1	1

Example: Moore implementation

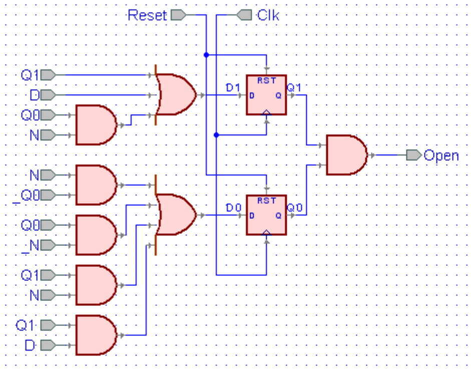
- Mapping to logic



D1				
	0	0	1	1
	0	1	1	1
	X	X	1	X
	1	1	1	1

D0				
	0	1	1	0
	1	0	1	1
	X	X	1	X
	0	1	1	1

Open				
	0	0	1	0
	0	0	1	0
	X	X	1	X
	0	0	1	0



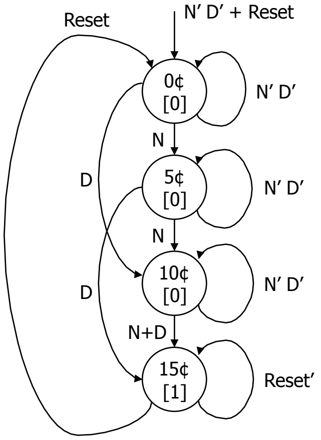
D1 = Q1 + D + Q0 N

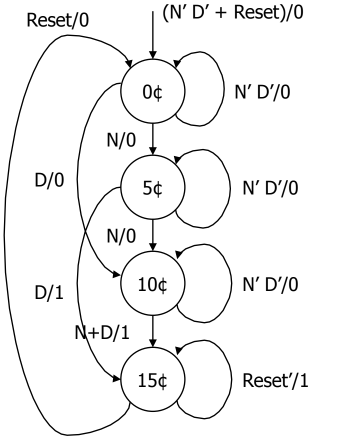
D0 = Q0′ N + Q0 N′ + Q1 N + Q1 D

OPEN = Q1 Q0

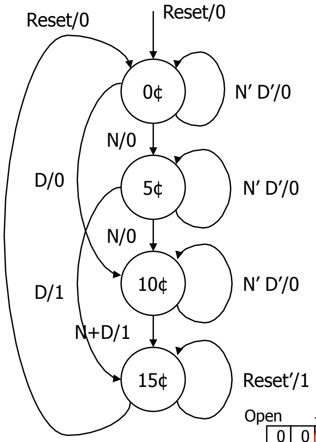
Equivalent Mealy and Moore state diagrams

- Moore machine
  - outputs associated with state
- Mealy machine
  - outputs associated with transitions

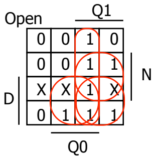




Example: Mealy implementation



present state		inputs		next state		output
Q1	Q0	D	N	D1	D0	open
0	0	0	0	0	0	0
		0	1	0	1	0
		1	0	1	0	0
		1	1	–	–	–
0	1	0	0	0	1	0
		0	1	1	0	0
		1	0	1	1	1
		1	1	–	–	–
1	0	0	0	1	0	0
		0	1	1	1	1
		1	0	1	1	1
		1	1	–	–	–
1	1	–	–	1	1	1



0	0	1	0
0	0	1	1
X	X	1	X
0	1	1	1

D0 = Q0′N + Q0N′ + Q1N + Q1D

D1 = Q1 + D + Q0N

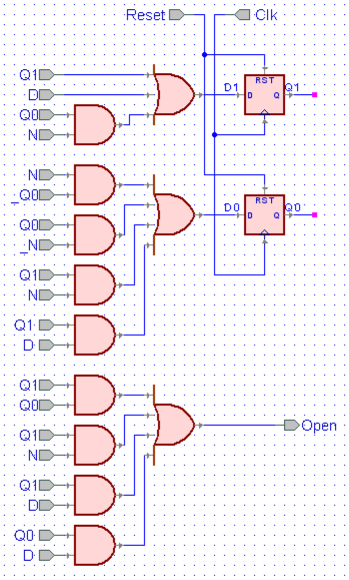
OPEN = Q1Q0 + Q1N + Q1D + Q0D

Example: Mealy implementation

D0 = Q0′N + Q0N′ + Q1N + Q1D

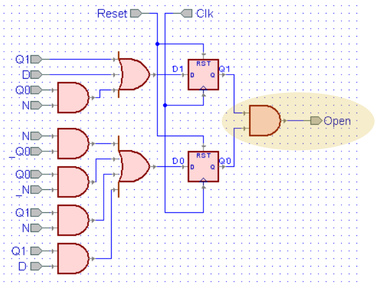
D1 = Q1 + D + Q0N

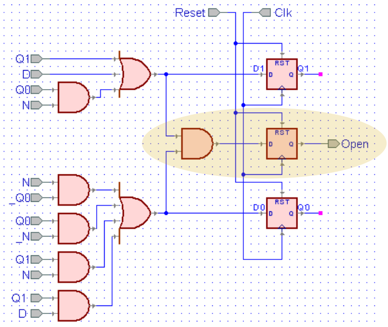
OPEN = Q1Q0 + Q1N + Q1D + Q0D



Vending machine: Moore to synch. Mealy

- OPEN = Q1Q0 creates a combinational delay after Q1 and Q0 change in Moore implementation
- This can be corrected by retiming, i.e., move flip-flops and logic through each other to improve delay – pre-compute OPEN then store it in FF
- OPEN.d = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)
  = Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D
- Implementation now looks like a synchronous Mealy machine
  - another reason programmable devices have FF at end of logic



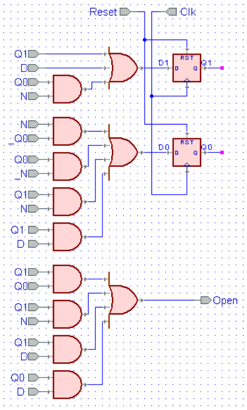


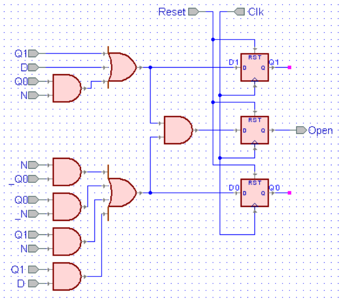
Vending machine: Mealy to synch. Mealy

Open.d			Q1		
	0	0	1	0	
	0	0	1	1	N
D	X	X	1	X	
	0	1	1	1	
		Q0			

- OPEN.d = Q1Q0 + Q1N + Q1D + Q0D
- OPEN.d = (Q1 + D + Q0N)(Q0'N + Q0N' + Q1N + Q1D)
  = Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D

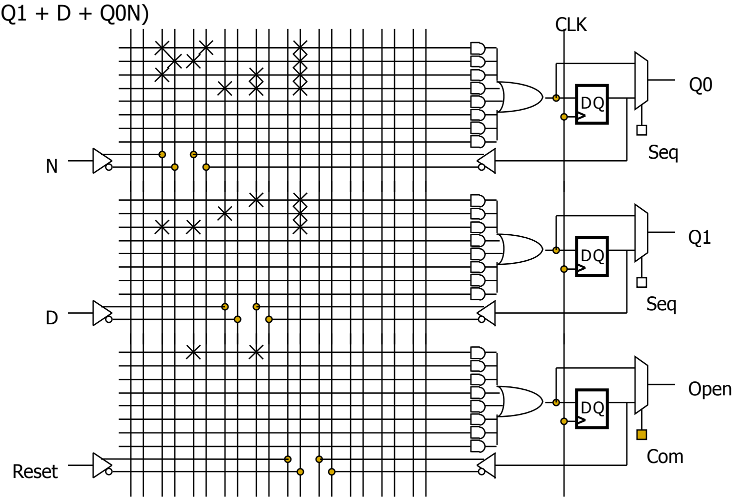
Open.d			Q1		
	0	0	1	0	
	0	0	1	1	N
D	1	0	1	1	
	0	1	1	1	
		Q0			





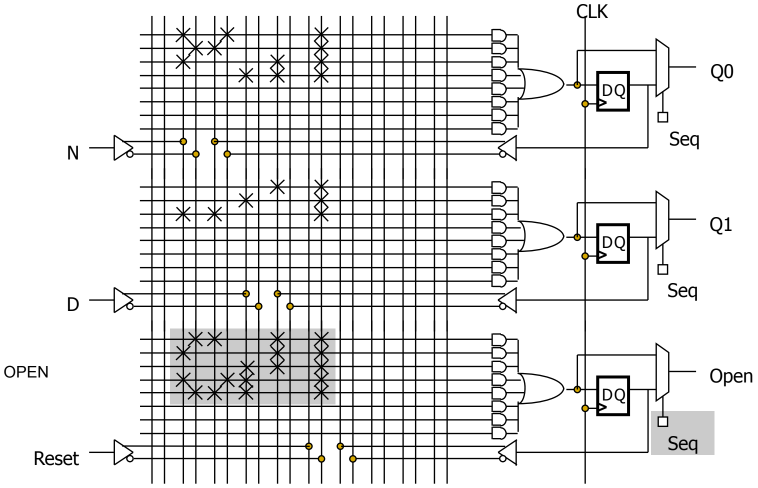
Vending machine example (Moore PLD mapping)

D0 = reset'(Q0'N + Q0N' + Q1N + Q1D)
D1 = reset'(Q1 + D + Q0N)
OPEN = Q1Q0



Vending machine (synch. Mealy PLD mapping)

OPEN = reset'(Q1Q0N' + Q1N + Q1D + Q0'ND + Q0N'D)



One-hot encoded transition table

present state $Q_3Q_2Q_1Q_0$	inputs D N	next state $D_3D_2D_1D_0$	output open
0 0 0 1	0 0	0 0 0 1	0
	0 1	0 0 1 0	0
	1 0	0 1 0 0	0
	1 1	– – – –	–
0 0 1 0	0 0	0 0 1 0	0
	0 1	0 1 0 0	0
	1 0	1 0 0 0	0
	1 1	– – – –	–
0 1 0 0	0 0	0 1 0 0	0
	0 1	1 0 0 0	0
	1 0	1 0 0 0	0
	1 1	– – – –	–
1 0 0 0	– –	1 0 0 0	1

$D_0 = Q_0D'N'$

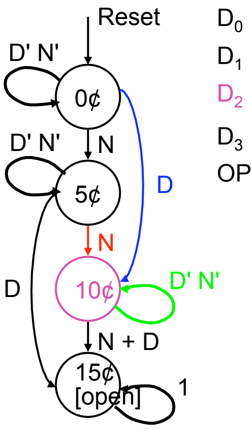
$D_1 = Q_0N + Q_1D'N'$

$D_2 = Q_0D + Q_1N + Q_2D'N'$

$D_3 = Q_1D + Q_2D + Q_2N + Q_3$

$OPEN = Q_3$

Designing from the state diagram



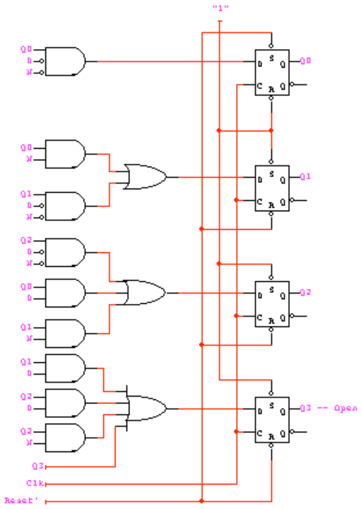
$D_0 = Q_0D'N'$

$D_1 = Q_0N + Q_1D'N'$

$D_2 = Q_0D + Q_1N + Q_2D'N'$

$D_3 = Q_1D + Q_2D + Q_2N + Q_3$

$OPEN = Q_3$

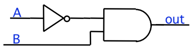


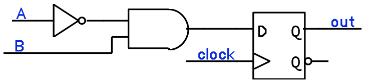
Output encoding

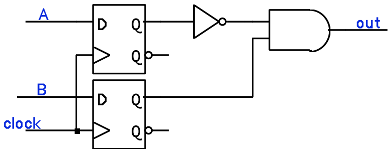
- Reuse outputs as state bits
  - Why create new functions when you can use outputs?
  - Bits from state assignments are the outputs for that state
    - Take outputs directly from the flip-flops
- ad hoc - no tools
  - Yields small circuits for most FSMs
  - Fits nicely with synchronous Mealy machines

Mealy and Moore examples

- Recognize A,B = 0,1
  - Mealy or Moore?

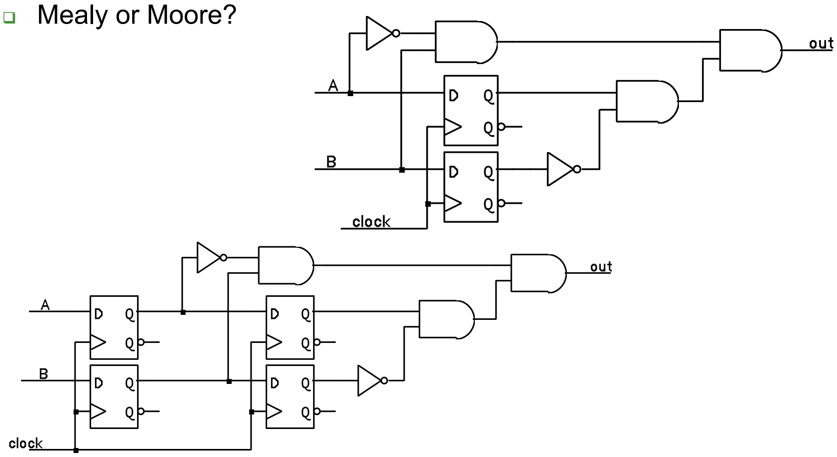






Mealy and Moore examples (cont'd)

- Recognize A,B = 1,0 then 0,1
  - Mealy or Moore?



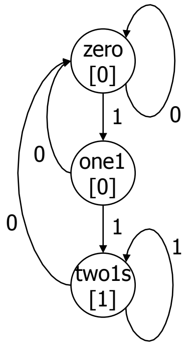
Hardware Description Languages and Sequential Logic

- Flip-flops
  - representation of clocks - timing of state changes
  - asynchronous vs. synchronous
- FSMs
  - structural view (FFs separate from combinational logic)
  - behavioral view (synthesis of sequencers – not in this course)
- Data-paths = data computation (e.g., ALUs, comparators) + registers
  - use of arithmetic/logical operators
  - control of storage elements

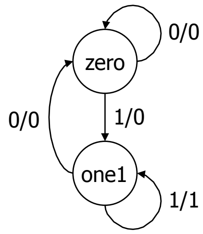
Example: reduce-1-string-by-1

- Remove one 1 from every string of 1s on the input (a *filter*)
  - E.g., 00011100 -> 00001100;   00100110 -> 00000010

Moore



Mealy

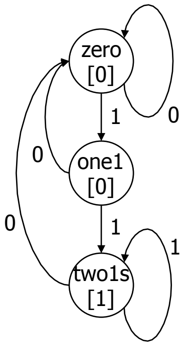


Verilog FSM - Reduce 1s example

- Moore machine

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;

  parameter zero  = 2’b00;
  parameter one1  = 2’b01;
  parameter two1s = 2’b10;

  reg out;
  reg [2:1] state;        // state variables
  reg [2:1] next_state;

  always @(posedge clk)
    if (reset) state = zero;
    else       state = next_state;
```

state assignment
(easy to change,
if in one place)



Moore Verilog FSM (cont'd)

```
always @(in or state)

  case (state)
    zero:
  // last input was a zero
   begin
     if (in) next_state = one1;
     else    next_state = zero;
   end
    one1:
  // we've seen one 1
   begin
     if (in) next_state = two1s;
     else    next_state = zero;
   end
    two1s:
  // we've seen at least 2 ones
   begin
     if (in) next_state = two1s;
     else    next_state = zero;
   end
  endcase
```

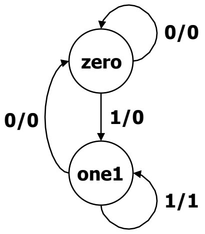
crucial to include
all signals that are
input to state determination

note that output
depends only on state

```
always @(state)
  case (state)
    zero: out = 0;
    one1: out = 0;
   two1s: out = 1;
  endcase

endmodule
```

Mealy Verilog FSM

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables
  reg next_state;

  always @(posedge clk)
    if (reset) state = zero;
    else       state = next_state;

  always @(in or state)
    case (state)
      zero:              // last input was a zero
     begin
       out = 0;
       if (in) next_state = one;
       else    next_state = zero;
     end
      one:               // we've seen one 1
     if (in) begin
        next_state = one; out = 1;
     end else begin
        next_state = zero; out = 0;
     end
    endcase
endmodule
```



Synchronous Mealy Machine

```
module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg state; // state variables

  always @(posedge clk)
    if (reset) state = zero;
    else
     case (state)
      zero:      // last input was a zero
     begin
       out = 0;
       if (in) state = one;
       else    state = zero;
     end
      one:       // we've seen one 1
     if (in) begin
         state = one; out = 1;
     end else begin
         state = zero; out = 0;
     end
    endcase
endmodule
```

Finite state machines 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
- Hardware description languages