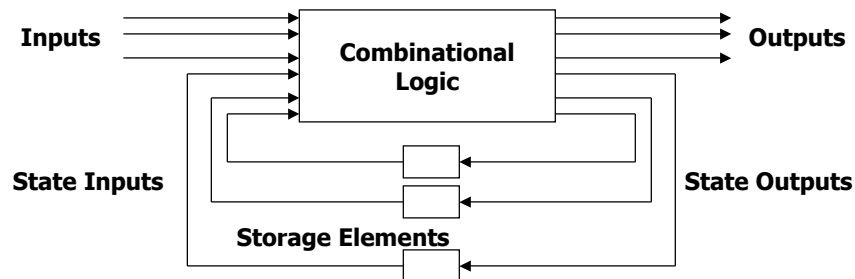


Finite State Machines

- ⌘ Sequential circuits
 - ☒ primitive sequential elements
 - ☒ combinational logic
- ⌘ Models for representing sequential circuits
 - ☒ finite-state machines (Moore and Mealy)
- ⌘ Basic sequential circuits revisited
 - ☒ shift registers
 - ☒ counters
- ⌘ Design procedure
 - ☒ state diagrams
 - ☒ state transition table
 - ☒ next state functions
- ⌘ Hardware description languages

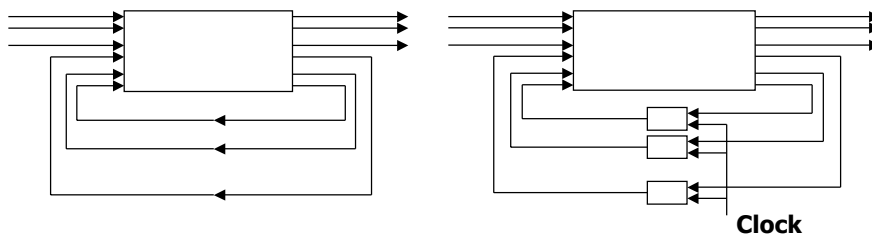
Abstraction of state elements

- ⌘ 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



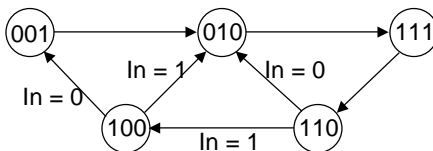
Forms of sequential logic

- ⌘ Asynchronous sequential logic – state changes occur whenever state inputs change (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 - the clock)



Finite state machine representations

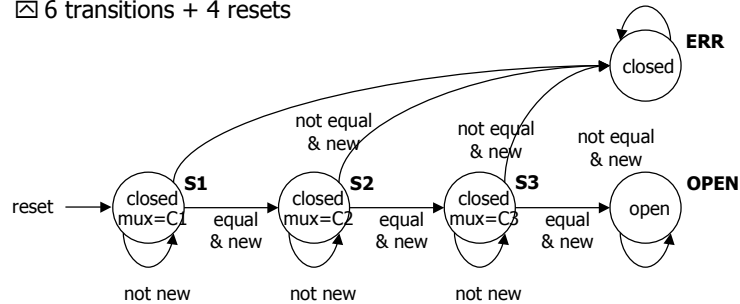
- ⌘ States: determined by possible values in sequential storage elements
- ⌘ Transitions: change of state
- ⌘ Clock: controls when state can change by controlling storage elements
- ⌘ Sequential logic
 - ☑ sequences through a series of states
 - ☑ based on sequence of values on input signals
 - ☑ clock period defines elements of sequence



Example finite state machine diagram

⌘ Combination lock from introduction to course

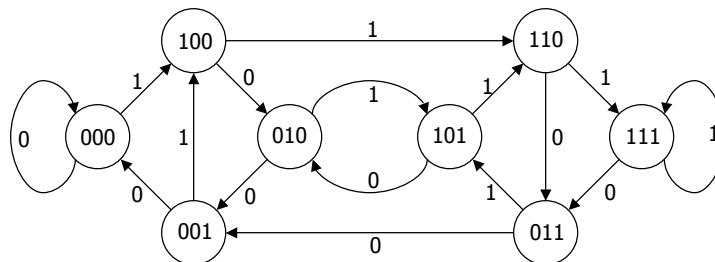
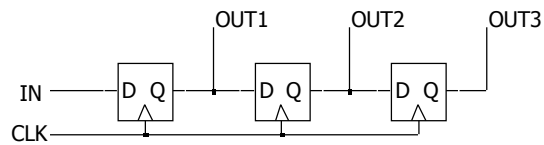
- ☒ 5 states
- ☒ 5 self-transitions + 1 reset to state S1
- ☒ 6 transitions + 4 resets



Can any sequential system be represented with a state diagram?

⌘ Shift register

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



Counters are simple finite state machines

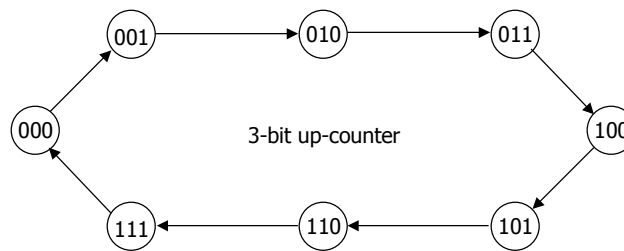
⌘ Counters

☒ proceed through well-defined sequence of states in response to enable

⌘ Many types of counters: binary, BCD, Gray-code

☒ 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, ...



How do we turn a state diagram into logic?

⌘ Counter

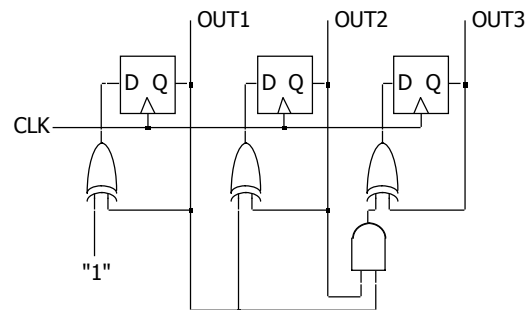
☒ 3 flip-flops to hold state

☒ logic to compute next state

☒ clock signal controls when flip-flop memory can change

☒ wait long enough for combinational logic to compute new value

☒ don't wait too long as that is low performance



FSM design procedure

- ⌘ Start with counters
 - ☒ simple because output is just state
 - ☒ simple because no choice of next state based on input

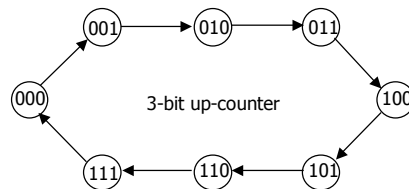
- ⌘ State diagram to state transition table
 - ☒ tabular form of state diagram
 - ☒ like a truth-table

- ⌘ 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

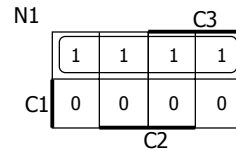
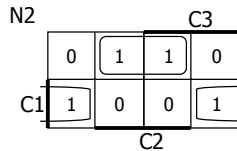
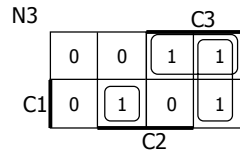
Implementation

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

$N1 \leq C1'$
 $N2 \leq C1C2' + C1'C2$
 $\leq C1 \text{ xor } C2$
 $N3 \leq C1C2C3' + C1'C3 + C2'C3$
 $\leq (C1C2)C3' + (C1' + C2')C3$
 $\leq (C1C2)C3' + (C1C2)'C3$
 $\leq (C1C2) \text{ xor } C3$



Autumn 2000

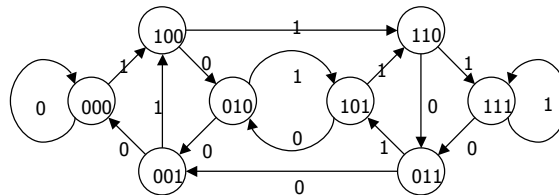
CSE370 - VII - Finite State Machines

11

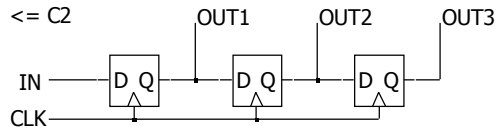
Back to the shift register

- ⌘ 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



$N1 \leq In$
 $N2 \leq C1$
 $N3 \leq C2$



Autumn 2000

CSE370 - VII - Finite State Machines

12

More complex counter example

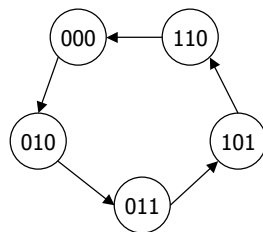
⌘ Complex counter

- ☒ 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
A	X	1	X	1	
	B				

		C			
		1	1	0	X
A	X	0	X	1	
	B				

		C			
		0	1	0	X
A	X	1	X	0	
	B				

$$C+ \leq A$$

$$B+ \leq B' + A'C'$$

$$A+ \leq BC'$$

Self-starting counters (cont'd)

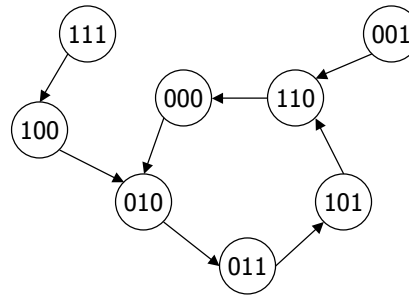
⌘ Re-deriving state transition table from don't care assignment

	C		
0	0	0	0
A	1	1	1
	B		

	C		
1	1	0	1
A	1	0	0
	B		

	C		
0	1	0	0
A	0	1	0
	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



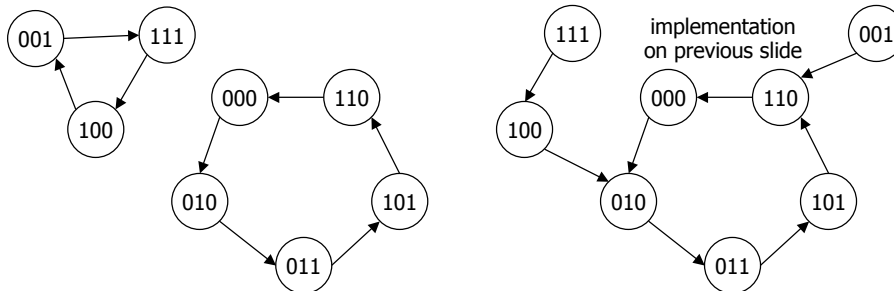
Self-starting counters

⌘ 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
- ☒ may limit exploitation of don't cares



Activity

⌘ 2-bit up-down counter (2 inputs)

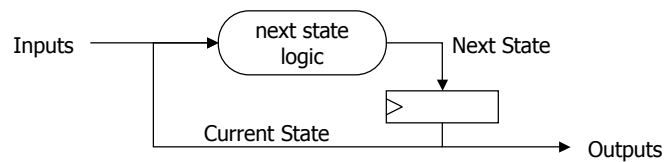
direction: $D = 0$ for up, $D = 1$ for down

count: $C = 0$ for hold, $C = 1$ for count

Activity (cont'd)

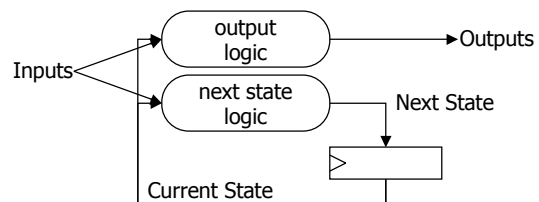
Counter/shift-register model

- ⌘ 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



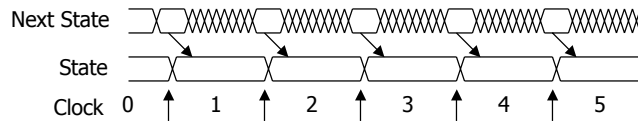
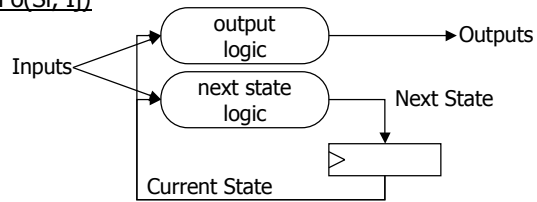
General state machine model

- ⌘ 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)



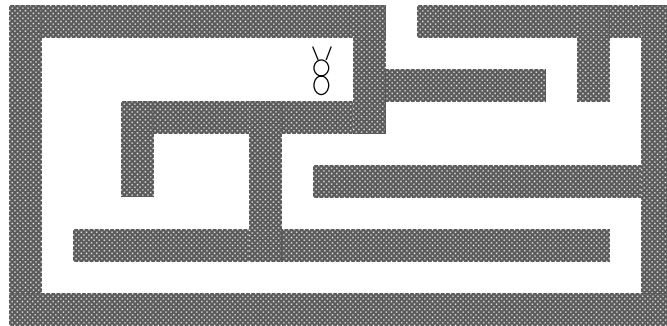
State machine model (cont'd)

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

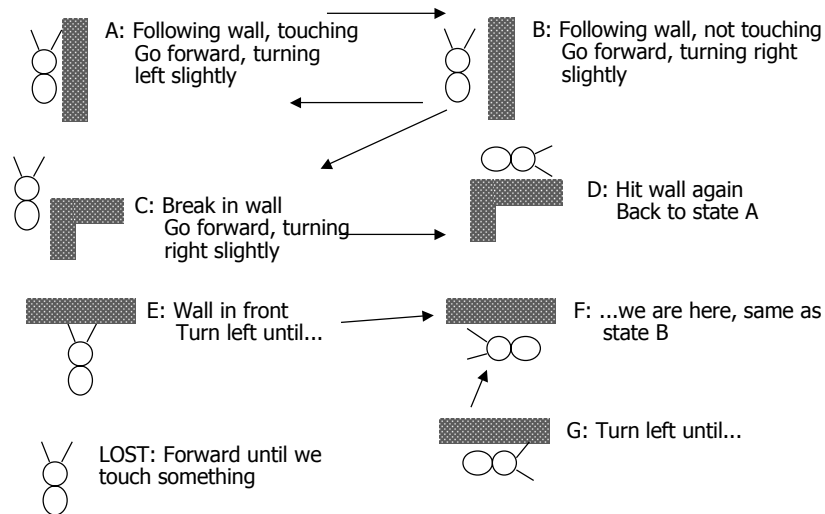


Example: ant brain (Ward, MIT)

- ⌘ Sensors: L and R antennae, 1 if in touching wall
- ⌘ Actuators: F - forward step, TL/TR - turn left/right slightly
- ⌘ Goal: find way out of maze
- ⌘ Strategy: keep the wall on the right

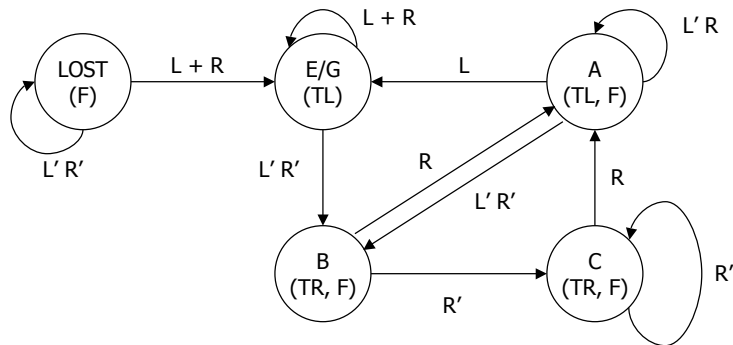


Ant behavior



Designing an ant brain

⌘ State diagram

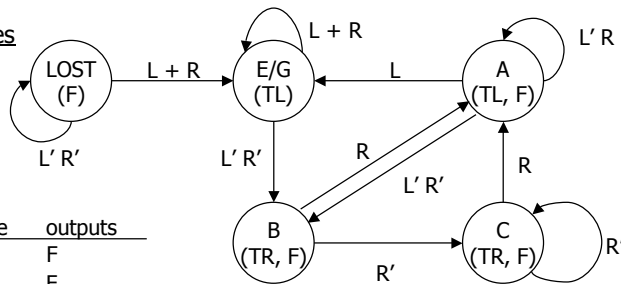


Synthesizing the ant brain circuit

- ⌘ Encode states using a set of state variables
 - ☒ arbitrary choice - may affect cost, speed
- ⌘ Use transition truth table
 - ☒ define next state function for each state variable
 - ☒ define output function for each output
- ⌘ Implement next state and output functions using combinational logic
 - ☒ 2-level logic (ROM/PLA/PAL)
 - ☒ multi-level logic
 - ☒ next state and output functions can be optimized together

Transition truth table

- ⌘ Using symbolic states and outputs



state	L	R	next state	outputs
LOST	0	0	LOST	F
LOST	-	1	E/G	F
LOST	1	-	E/G	F
A	0	0	B	TL, F
A	0	1	A	TL, F
A	1	-	E/G	TL, F
B	-	0	C	TR, F
B	-	1	A	TR, F
...

Synthesis

- ⌘ 5 states : at least 3 state variables required (X, Y, Z)
- ☒ state assignment (in this case, arbitrarily chosen)

LOST - 000
 E/G - 001
 A - 010
 B - 011
 C - 100

state X,Y,Z	L R	next state X ⁺ , Y ⁺ , Z ⁺	outputs F TR TL
000	00	000	100
000	01	001	100
...
010	00	011	101
010	01	010	101
010	10	001	101
010	11	001	101
011	00	100	110
011	01	010	110
...

it now remains to synthesize these 6 functions

Synthesis of next state and output functions

state X,Y,Z	inputs L R	next state X ⁺ , Y ⁺ , Z ⁺	outputs F TR TL
000	00	000	100
000	- 1	001	100
000	1 -	001	100
001	00	011	001
001	- 1	010	001
001	1 -	010	001
010	00	011	101
010	01	010	101
010	1 -	001	101
011	- 0	100	110
011	- 1	010	110
100	- 0	100	110
100	- 1	010	110

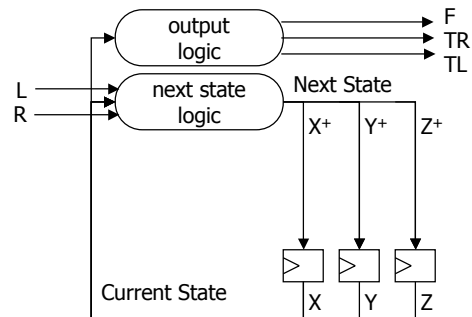
e.g.

$$TR = X + YZ$$

$$X^+ = XR' + YZR' = R'TR$$

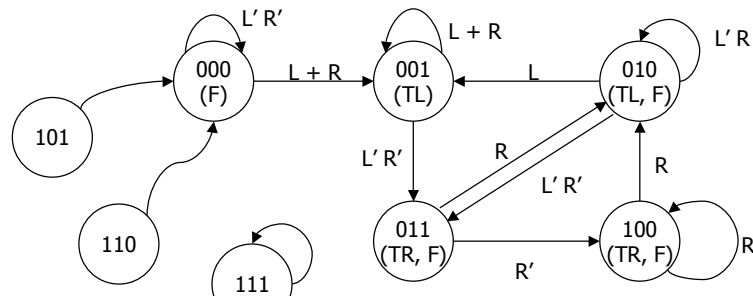
Circuit implementation

- ⌘ Outputs are a function of the current state only - Moore machine



Don't cares in FSM synthesis

- ⌘ What happens to the "unused" states (101, 110, 111)?
- ⌘ They were exploited as don't cares to minimize the logic
 - ☒ if the states can't happen, then we don't care what the functions do
 - ☒ if states do happen, we may be in trouble



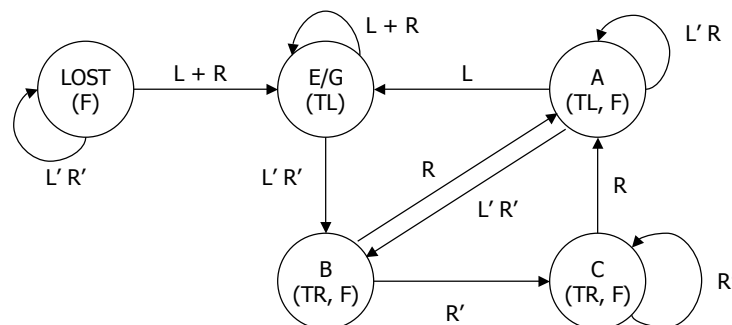
Ant is in deep trouble if it gets in this state

State minimization

- ⌘ Fewer states may mean fewer state variables
- ⌘ High-level synthesis may generate many redundant states
- ⌘ Two states are equivalent if they are impossible to distinguish from the outputs of the FSM, i. e., for any input sequence the outputs are the same
- ⌘ Two conditions for two states to be equivalent:
 - ☒ 1) output must be the same in both states
 - ☒ 2) must transition to equivalent states for all input combinations

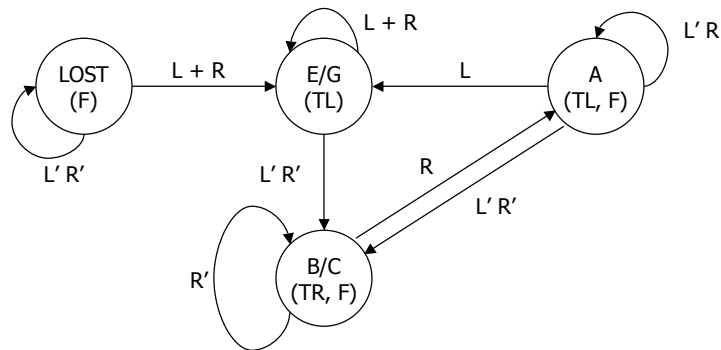
Ant brain revisited

- ⌘ Any equivalent states?



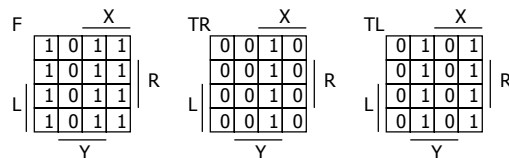
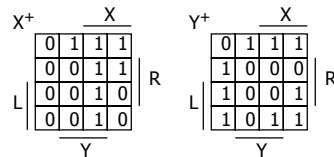
New improved brain

- ⌘ Merge equivalent B and C states
- ⌘ Behavior is exactly the same as the 5-state brain
- ⌘ We now need only 2 state variables rather than 3



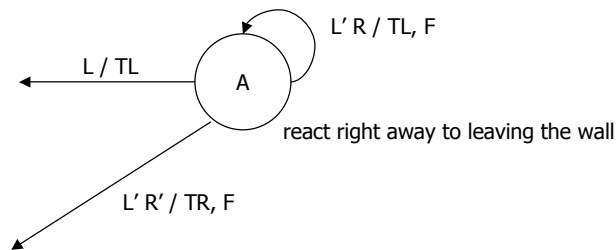
New brain implementation

state	inputs		next state outputs			
X,Y	L	R	X ⁺ ,Y ⁺	F	TR	TL
00	0	0	00	1	0	0
00	-	1	01	1	0	0
00	1	-	01	1	0	0
01	0	0	11	0	0	1
01	-	1	01	0	0	1
01	1	-	01	0	0	1
10	0	0	11	1	0	1
10	0	1	10	1	0	1
10	1	-	01	1	0	1
11	-	0	11	1	1	0
11	-	1	10	1	1	0



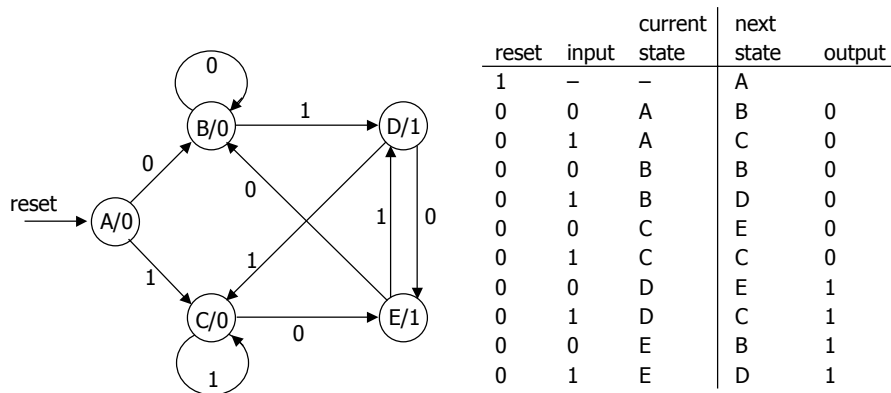
Mealy vs. Moore machines

- ⌘ 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



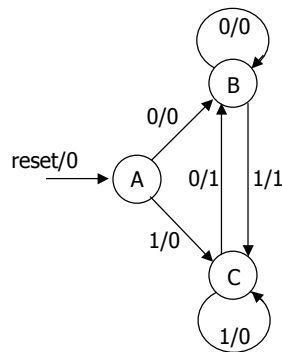
Specifying outputs for a Moore machine

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



Specifying outputs for a Mealy machine

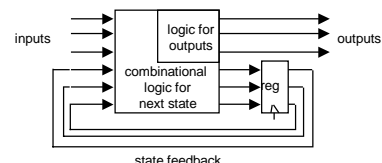
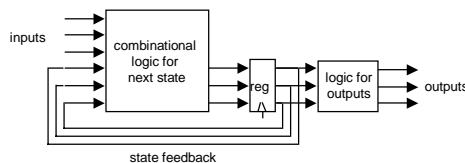
- ⌘ Output is function of state and inputs
 - ☒ specify output on transition arc between states
 - ☒ example: sequence detector for 01 or 10



reset	input	current state	next 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

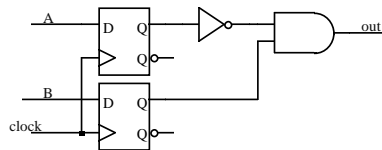
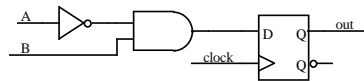
Comparison of Mealy and Moore machines

- ⌘ 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



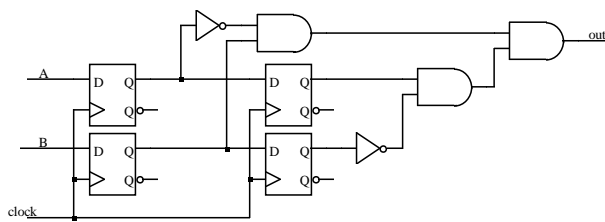
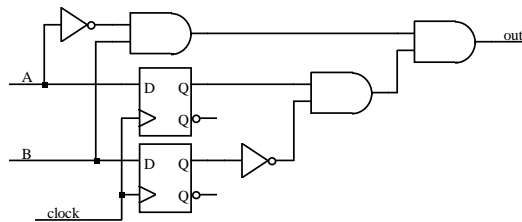
Mealy and Moore examples

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



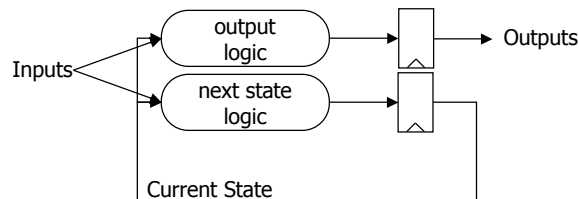
Mealy and Moore examples (cont'd)

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



Registered Mealy machine (really Moore)

- ⌘ 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

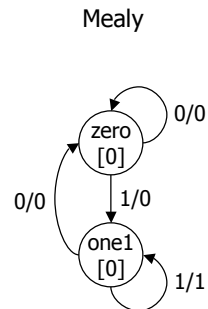
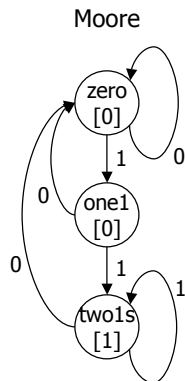


Hardware Description Languages and Sequential Logic

- ⌘ 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



Verilog FSM - Reduce 1s example

⌘ Moore machine

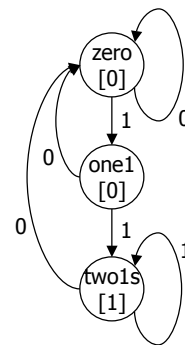
```

`define zero 0
`define one1 1 ← state assignment
`define two1s 2

module reduce (clk, reset, in, out);
  input clk, reset, in;
  output out;
  reg out;
  reg [2:1] state; // state variables
  reg [2:1] next_state;

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

```



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 = `twos;
        else next_state = `zero;
      end
    `twos:
      // we've seen at least 2 ones
      begin
        if (in) next_state = `twos;
        else next_state = `zero;
      end
  endcase
endmodule

```

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;
    `twos: out = 1;
  endcase

```

Mealy Verilog FSM

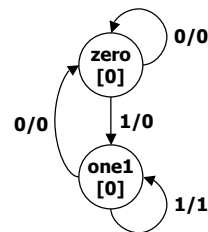
```

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
```

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
- ⌘ Hardware description languages