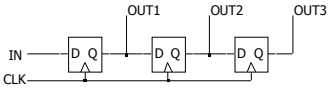
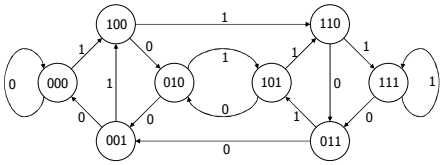
Sequential logic implementation

- Sequential circuits
  - primitive sequential elements
  - combinational logic
- Models for representing sequential circuits
  - finite-state machines (Moore and Mealy)
  - representation of memory (states)
  - changes in state (transitions)
- Basic sequential circuits
  - shift registers
  - counters
- Design procedure
  - state diagrams
  - state transition table
  - next state functions

Any sequential system can be represented with a state diagram

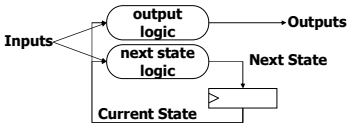
- Shift register
  - input value shown on transition arcs
  - output values shown within state node





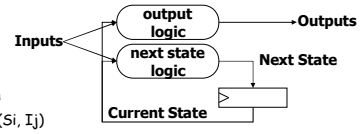
State machine model

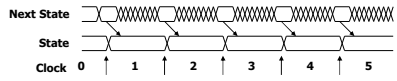
- Values stored in registers are the state of the circuit
- Combinational logic computes:
  - next state
  - outputs
    - function of current state and inputs (Mealy machine)
    - function of current state only (Moore machine)



State Machine Model

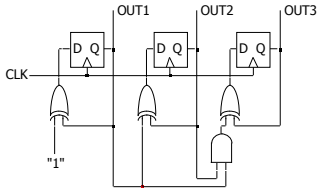
- States: S1, S2, ..., Sk
- Inputs: I1, I2, ..., Im
- Outputs: O1, O2, ..., On
- Transition function: Fs(Si, Ij)
- Output function: Fo(Si) or Fo(Si, Ij)





How do we turn a state diagram into logic?

- e.g. counter
  - flip-flops to hold state
  - logic to compute next state
  - clock signal controls when flip-flop memory can change
    - wait just long enough for combinational logic to compute new value

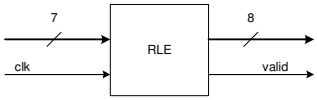


FSM design procedure

- Describe FSM behavior, e.g. state diagram
  - Inputs and Outputs
  - States (symbolic)
  - State transitions
- State diagram to state transition table, i.e. truth table
  - Inputs: inputs and current state
  - Outputs: outputs and next state
- State encoding
  - decide on representation of states
  - lots of choices
- Implementation
  - flip-flop for each state bit
  - synthesize combinational logic from encoded state table

Design Problem – Run-Length Encoder

- 7-bit input stream
- 8-bit output stream
  - high-order bit == 0: Data value
  - high-order bit == 1: Repeat count of previous data value
- Valid bit set when 8-bit output is data or count

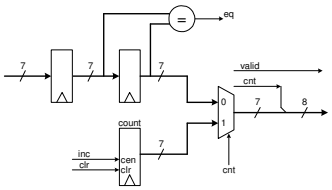


RLE Design

- Split design into datapath and control
- Datapath
  - Registers for data values, count
  - Multiplexors
- Control
  - Keep track of what's happening
  - clear count, increment count, send data value, send count
- Control will be an FSM

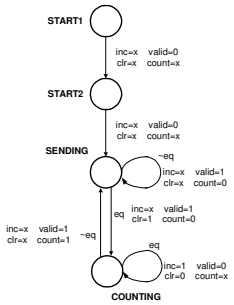
Start with Datapath

- We need to know what to control



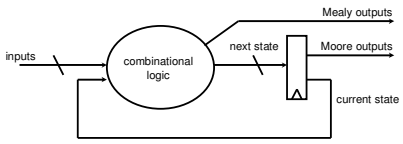
- FSM inputs
  - eq
- FSM outputs
  - clr, inc, valid, cnt,

FSM Controller



Verilog For State Machines

- State machine has two parts
  - State register
  - Combinational Logic
    - Next state function
    - Output function
- Each in a different always block



RLE Module

- rleFSM is one module
- The datapath is specified as a schematic
  - rleFSM is part of schematic
- Possible to describe datapath in the Verilog
  - Text description of inherently graphical design

```
module rleFSM (clk, reset, eq, clr, inc, valid, cnttag);
  input clk, reset;
  input eq;             // current data value == previous data value
  output clr;           // clear count value (0 means 2, . . .)
  output inc;           // increment count value (clr overrides)
  output valid;         // output value is valid
  output cnttag;        // select the count for the output value

  // Use parameter to define symbolic states
  parameter START1 = 0, START2 = 1, SENDING = 2, COUNTING = 3;
  reg [1:0]  state,     // current state
             nextState; // next state
```

Verilog for Registers

- Triggering on "posedge CLK" generates a positive, edge-triggered register
  - The only kind of register we will use

```
reg  [7:0]  state,  // current state
            nextState;     // next state

always @(posedge CLK) begin
  if (reset)
    state = START1;
  else
    state = nextState;
end
```


Verilog for FSM Logic

```
always @(state or eq) begin
    // Set defaults
    valid = 0; inc = x; clr = x; cnttag = x;
    case (state)
    START1:
      nextState = START2;
    START2:
      nextState = SENDING;
    SENDING: begin
      valid = 1;
      cnttag = 0;
      if (eq) begin
        nextState = COUNTING;
      end else begin
        nextState = SENDING;
      end
    end
    COUNTING: begin
      if (eq) begin
        clr = 0;
        inc = 1;
        nextState = COUNTING;
      end else begin
        valid = 1;
        cnttag = 1;
        nextState = SENDING;
      end
    end
end
```


Implementing an FSM

- 1. Perform state assignment
  - different assignments may give very different results
  - no really good heuristics
  - using an extra bit or two for state works well
    - FPGAs often us a 1-hot encoding
- 2. Convert state diagram to state table
  - equivalent representation
  - mechanical
- 3. State table gives truth table for next state and output functions
  - synthesize into logic circuit
  - e.g. 2-level logic implementation

RLE State Table

- reset is implicit – resets state register

eq	state	next state	clr	inc	valid	cnttag
-	START1	START2	x	x	0	x
-	START2	SENDING	x	x	0	x
0	SENDING	SENDING	x	x	1	0
1	SENDING	COUNTING	1	x	1	0
0	COUNTING	SENDING	x	x	1	1
1	COUNTING	COUNTING	0	1	0	x

RLE State Table

- minimal binary encoding
- there are 6 output functions

eq	state	next state	clr	inc	valid	cnttag
-	00	01	x	x	0	x
-	01	10	x	x	0	x
0	10	10	x	x	1	0
1	10	11	1	x	1	0
0	11	10	x	x	1	1
1	11	11	0	1	0	x

Logic Synthesis

Q1

eq \ state	00	01	11	10
0	0	1	1	1
1	0	1	1	1

Q0

eq \ state	00	01	11	10
0	1	0	0	0
1	1	0	1	1

clr

eq \ state	00	01	11	10
0	X	X	X	X
1	X	X	0	1

inc

eq \ state	00	01	11	10
0	X	X	X	X
1	X	X	1	X

valid

eq \ state	00	01	11	10
0	0	0	1	1
1	0	0	0	1

cnttag

eq \ state	00	01	11	10
0	X	X	X	0
1	X	X	1	0

Q1 = Q1 + Q0
Q0 = Q1' Q0' + eq Q1
clr = Q0'
inc = 1
valid = eq' Q1 + Q1 Q0' = Q1 (eq' + Q0')
cnttag = Q0

6 gates

RLE State Table

- 1-hot encoding
- there are 8 output functions

eq	state	next state	clr	inc	valid	cnttag
-	0001	0010	x	x	0	x
-	0010	0100	x	x	0	x
0	0100	0100	x	x	1	0
1	0100	1000	1	x	1	0
0	1000	0100	x	x	1	1
1	1000	1000	0	1	0	x

1-Hot Logic Functions

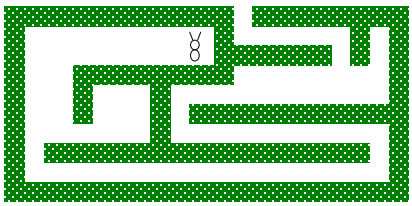
- Synthesize directly from table

Q0 = reset
Q1 = Q0
Q2 = Q1 + eq' (Q2 + Q3)
Q3 = eq (Q2 + Q3)
inc = 1
clr = Q2
valid = Q2 + eq' Q3
cnttag = Q3

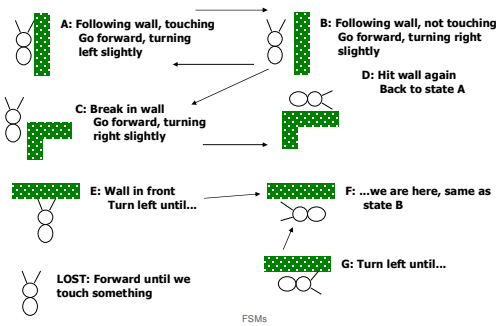
- Same cost as minimal encoding (6 gates)

Another Example: Ant Brain

- Sensors: L and R antennae, 1 if 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

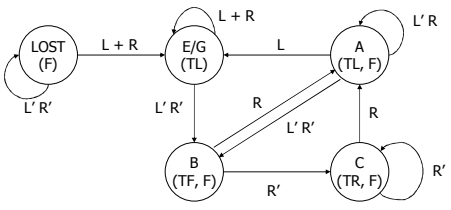


Ant Behavior - Case Analysis



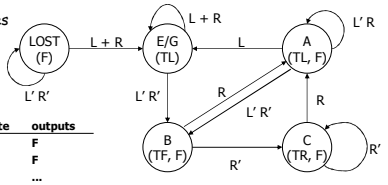
Designing an Ant Brain

- State Diagram
- Once we have state diagram, we just "turn the crank"



State Transition Truth Table

- Use symbolic states and outputs
- Take advantage of extra freedom



state	L	R	next state	outputs
LOST	0	0	LOST	F
LOST	0	1	E/G	F
...	...	...	...	...
A	0	0	B	TL, F
A	0	1	A	TL, F
A	1	0	E/G	TL, F
A	1	1	E/G	TL, F
B/C	0	0	C	TR, F
B/C	0	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	0	0	000	1	0	0
000	0	1	001	1	0	0
...	...	...	...	...		
010	0	0	011	1	0	1
010	0	1	010	1	0	1
010	1	0	001	1	0	1
010	1	1	001	1	0	1
011	0	0	100	1	1	0
011	0	1	010	1	1	0
...	...	...	...	...		

It now remains to synthesize these 6 functions

Synthesis of Next State and Output Functions

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

f(X,Y,Z,L,R)

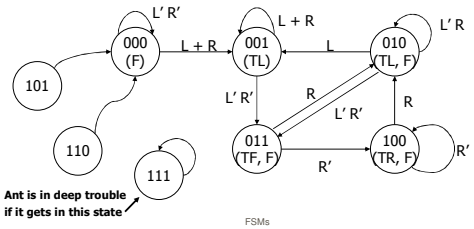
e.g.

TR = X + Y Z

$X^+ = X R' + Y Z R' = R' TR$

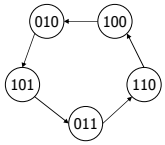
Don't cares in FSM synthesis

- 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



Don't Care Example

- Implement simple count sequence: 000, 010, 011, 101, 110
- Derive the state transition table from the state transition diagram



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

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

Don't cares in FSMs (cont'd)

- Synthesize logic for next state functions derive input equations for flip-flops

C^+

A \ BC	00	01	11	10
0	X	1	1	0
1	X	1	X	0

B^+

A \ BC	00	01	11	10
0	X	0	0	1
1	X	1	X	1

A^+

A \ BC	00	01	11	10
0	X	1	0	0
1	X	0	X	1

$C^+ = B$

$B^+ = A + B' C$

$A^+ = A' C' + AC$

Self-starting FSMs

- Deriving state transition table from don't care assignment

C^+

A \ BC	00	01	11	10
0	0	1	1	0
1	0	1	1	0

B^+

A \ BC	00	01	11	10
0	1	0	0	1
1	1	1	1	1

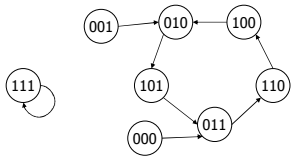
A^+

A \ BC	00	01	11	10
0	1	1	0	0
1	0	0	1	1

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

Self-starting FSMs

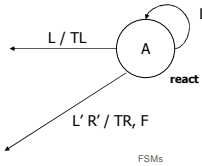
- Start-up states
  - at power-up, FSM may be in an used or invalid state
  - design must guarantee that it (eventually) enters a valid state
- Self-starting solution
  - design FSM so that all the invalid states eventually
  - transition to a valid state may limit exploitation of don't cares



implementation on previous slide

Mealy vs. Moore Machines

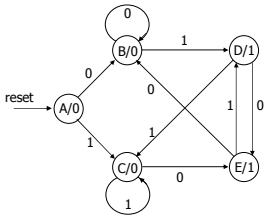
- Moore: outputs depend on current state only
- Mealy: outputs may depend on current state and current inputs
- Our ant brain is a Moore machine
  - output does not react immediately to input change
- We could have specified a Mealy FSM
  - outputs have immediate reaction to inputs (do glitches matter?)
  - as inputs change, so does next_state, but doesn't commit until clock tick



react right away to leaving the wall

Specifying outputs for a Moore machine

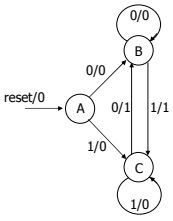
- Output is only function of state
  - specify in state bubble in state diagram
  - example: sequence detector for 01 or 10



reset	input	current state	next 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

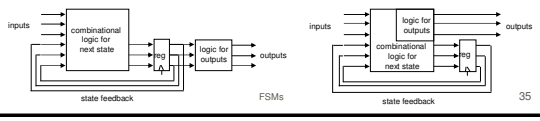
- Output is function of state and inputs
  - specify output on transition arc between states
  - example: sequence detector for 01 or 10



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

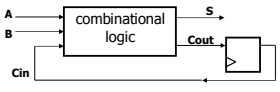
Comparison of Mealy and Moore machines

- Mealy machines tend to have fewer states
  - different outputs on arcs (i*n) rather than states (n)
- Mealy machines react faster to inputs
  - react in same cycle – don't need to wait for clock
  - delay to output depends on arrival of input
- Moore machines are generally 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



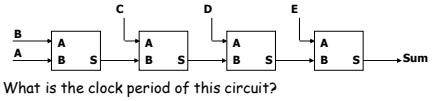
Isn't Mealy Always Better?

- Example: serial adder - add bits arriving serially on two input wires



A	B	P.S. Cin	N.S. Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

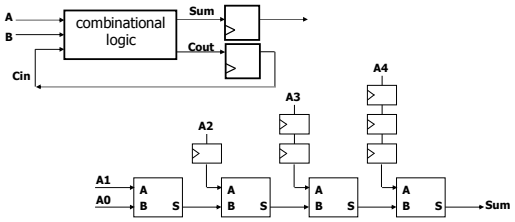
- Adding 5 serial numbers on 5 separate lines



- What is the clock period of this circuit?

Moore Implementation - Pipelined

- Example: adding 5 serial numbers



- What is the clock period of this circuit?