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

START1

- inc=x  valid=0
- clr=x  count=x

START2

- inc=x  valid=0
- clr=x  count=x

SENDING

- ~eq

COUNTING

- eq
- ~eq
- eq
- eq

FSMs 10
**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

```verilog
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
```

FSMs 12
Verilog for Registers

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

```verilog
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
      valid = 1; cnttag = 1; 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 use 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

<table>
<thead>
<tr>
<th>eq</th>
<th>state</th>
<th>next state</th>
<th>clr</th>
<th>inc</th>
<th>valid</th>
<th>cnttag</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>START1</td>
<td>START2</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>-</td>
<td>START2</td>
<td>SENDING</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>SENDING</td>
<td>SENDING</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>SENDING</td>
<td>COUNTING</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>COUNTING</td>
<td>SENDING</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>COUNTING</td>
<td>COUNTING</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
RLE State Table

- minimal binary encoding
- there are 6 output functions

<table>
<thead>
<tr>
<th>eq</th>
<th>state</th>
<th>next state</th>
<th>clr</th>
<th>inc</th>
<th>valid</th>
<th>cnttag</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>00</td>
<td>01</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>-</td>
<td>01</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>11</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>11</td>
<td>10</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
### Logic Synthesis

<table>
<thead>
<tr>
<th>eq</th>
<th>Q1 state</th>
<th>Q0 state</th>
<th>clr state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 01 11 10</td>
<td>00 01 11 10</td>
<td>00 01 11 10</td>
</tr>
<tr>
<td>1</td>
<td>00 01 11 10</td>
<td>01 00 11 10</td>
<td>11 00 11 10</td>
</tr>
<tr>
<td>inc</td>
<td>00 01 11 10</td>
<td>11 00 11 10</td>
<td>11 00 11 10</td>
</tr>
<tr>
<td>1</td>
<td>11 00 11 10</td>
<td>11 00 11 10</td>
<td>11 00 11 10</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>eq</th>
<th>state</th>
<th>next state</th>
<th>clr</th>
<th>inc</th>
<th>valid</th>
<th>cnttag</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0001</td>
<td>0010</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>-</td>
<td>0010</td>
<td>0100</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>0</td>
<td>0100</td>
<td>0100</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0100</td>
<td>1000</td>
<td>1</td>
<td>x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1000</td>
<td>0100</td>
<td>x</td>
<td>x</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1000</td>
<td>1000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
</tbody>
</table>
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

A: Following wall, touching
   Go forward, turning
   left slightly

B: Following wall, not touching
   Go forward, turning right
   slightly

D: Hit wall again
   Back to state A

C: Break in wall
   Go forward, turning
   right slightly

E: Wall in front
   Turn left until...

F: ...we are here, same as
   state B

G: Turn left until...

LOST: Forward until we
   touch something

FSMs
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

<table>
<thead>
<tr>
<th>state</th>
<th>L</th>
<th>R</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOST</td>
<td>0</td>
<td>0</td>
<td>LOST</td>
<td>F</td>
</tr>
<tr>
<td>LOST</td>
<td>0</td>
<td>1</td>
<td>E/G</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>B</td>
<td>TL, F</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>TL, F</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
<td>E/G</td>
<td>TL, F</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>E/G</td>
<td>TL, F</td>
</tr>
<tr>
<td>B/C</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>TR, F</td>
</tr>
<tr>
<td>B/C</td>
<td>0</td>
<td>1</td>
<td>A</td>
<td>TR, F</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Synthesis

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

<table>
<thead>
<tr>
<th>state</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>L R (X,Y,Z)</td>
<td>(X', Y', Z')</td>
<td>(F), (T), (L)</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>0 0 0</td>
<td>0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>... ... ...</td>
<td>... ...</td>
<td>...</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 0</td>
<td>0 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>0 1 1</td>
<td>1 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>... ... ...</td>
<td>... ...</td>
<td>...</td>
</tr>
</tbody>
</table>

It now remains to synthesize these 6 functions

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

FSMs 25
## Synthesis of Next State and Output Functions

<table>
<thead>
<tr>
<th>state L R</th>
<th>next state X', Y', Z'</th>
<th>outputs F TR TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 0 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>000 - 1</td>
<td>0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>000 1 -</td>
<td>0 0 1</td>
<td>1 0 0</td>
</tr>
<tr>
<td>001 0 0</td>
<td>0 1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>001 - 1</td>
<td>0 1 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>001 1 -</td>
<td>0 1 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>010 0 0</td>
<td>0 1 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>010 0 1</td>
<td>0 1 0</td>
<td>1 0 1</td>
</tr>
<tr>
<td>010 1 -</td>
<td>0 0 1</td>
<td>1 0 1</td>
</tr>
<tr>
<td>011 - 0</td>
<td>1 0 1</td>
<td>1 1 0</td>
</tr>
<tr>
<td>011 - 1</td>
<td>0 1 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>100 - 0</td>
<td>1 0 0</td>
<td>1 1 0</td>
</tr>
<tr>
<td>100 - 1</td>
<td>0 1 0</td>
<td>1 1 0</td>
</tr>
</tbody>
</table>

### Example

- \( TR = X + Y Z \)
- \( X' = X R' + Y Z R' = R' TR \)
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

Ant is in deep trouble if it gets in this state
Don’t Care Example

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

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

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Don’t cares in FSMs (cont’d)

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

\[ C^+ = B \]
\[ B^+ = A + B' C \]
\[ A^+ = A' C' + AC \]
**Self-starting FSMs**

- Deriving state transition table from don't care assignment

<table>
<thead>
<tr>
<th>Present State</th>
<th></th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
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
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

`A`

```
L / TL
```

```
L' R / TL, F
```

**react right away to leaving the wall**

```
L' R' / TR, F
```

FSMs
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

![State Diagram]

<table>
<thead>
<tr>
<th>reset</th>
<th>input</th>
<th>current state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>D</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>D</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>reset</th>
<th>input</th>
<th>current state</th>
<th>next state</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>–</td>
<td>–</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>C</td>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>
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

  ![Serial Adder Diagram]

- Adding 5 serial numbers on 5 separate lines

  ![5 Serial Adder Diagram]

- What is the clock period of this circuit?

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Cin</th>
<th>P.S.</th>
<th>N.S.</th>
<th>Cout</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Moore Implementation - Pipelined

- Example: adding 5 serial numbers

What is the clock period of this circuit?