Sequential Logic Implementation

- Models for representing sequential circuits
  - Finite state machines (Moore and Mealy)
  - Representation of memory (states)
  - Changes in state (transitions)

- Design procedure
  - State diagrams
  - State transition table
  - Next state functions

Registers

- Collections of flip-flops with similar controls and logic
  - Stored values somehow related (e.g., form binary value)
  - Share clock, reset, and set lines
  - Similar logic at each stage

- Examples
  - Shift registers
  - Counters

Shift Register

- Holds samples of input
  - Store last 4 input values in sequence
  - 4-bit shift register:
Shift Register Verilog

```verilog
module shift_reg (out4, out3, out2, out1, in, clk);
  output out4, out3, out2, out1;
  input  in, clk;
  reg    out4, out3, out2, out1;
always@(posedge clk)
  begin
    out4 <= out3;
    out3 <= out2;
    out2 <= out1;
    out1 <= in;
  end
endmodule
```

Shift Register Verilog

```verilog
module shift_reg (out, in, clk);
  output [4:1] out;
  input  in, clk;
  reg    [4:1] out;
always@(posedge clk)
  begin
    out <= {out[3:1], in};
  end
endmodule
```

Universal Shift Register

- **Holds 4 values**
- **Serial or parallel inputs**
- **Serial or parallel outputs**
- **Permits shift left or right**
- **Shift in new values from left or right**

<table>
<thead>
<tr>
<th>s0</th>
<th>s1</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>hold state</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>shift right</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>shift left</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>load new input</td>
</tr>
</tbody>
</table>

clear sets the register contents and output to 0
s1 and s0 determine the shift function
left_in left_out right_in right_out clock
Design of Universal Shift Register

Consider one of the four flip-flops

- New value at next clock cycle:

<table>
<thead>
<tr>
<th>Old value</th>
<th>New value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Universal Shift Register Verilog

```verilog
module univ_shift (out, in, lo, ro, in, li, ri, s, clr, clk);
  output [3:0] out;
  output lo, ro;
  input  [3:0] in;
  input  [1:0] s;
  input  li, ri, clr, clk;
  reg    [3:0] out;

assign lo = out[3];
assign ro = out[0];
always @(posedge clk or clr)
  begin
    if (clr) out <= 0;
    else
      case (s)
        3: out <= in;
        2: out <= {out[2:0], ri};
        1: out <= {li, out[3:1]};
        0: out <= out;
      endcase
  end
endmodule
```

Counters

- Sequences through a fixed set of patterns
  - In this case, 1000, 0100, 0010, 0001
  - If one of the patterns is its initial state (by loading or reset)

- Mobius (or Johnson) counter
  - In this case, 1000, 1000, 1110, 1111, 0111, 0011, 0001, 0000
Binary Counter

Logic between registers (not just multiplexer)
- XOR decides when bit should be toggled
- Always for low-order bit, only when first bit is true for second bit, and so on

Binary Counter Verilog

module shift_reg (out4, out3, out2, out1, clk);
output out4, out3, out2, out1;
input  in, clk;
reg    out4, out3, out2, out1;
always @ (posedge clk)
begin
  out4 <= (out1 & out2 & out3) ^ out4;
  out3 <= (out1 & out2) ^ out3;
  out2 <= out1 ^ out2;
  out1 <= out1 ^ 1b'1;
end
endmodule

Binary Counter Verilog

module shift_reg (out4, out3, out2, out1, clk);
output [4:1] out;
input  in, clk;
reg    [4:1] out;
always @ (posedge clk)
  out <= out + 1;
endmodule
Abstraction of State Elements

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

![Diagram of combinational logic and state](image)

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)

![Diagram of asynchronous and synchronous sequential logic](image)

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

![Finite State Machine diagram](image)
Example Finite State Machine Diagram

Combination lock from first lecture

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 thru well-defined state sequence 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, ...
Verilog Upcounter

```verilog
module binary_cntr (q, clk);
    inputs     clk;
    outputs    [2:0] q;
    reg        [2:0] p;
    always @(q)            //Calculate next state
        case (q)
            3'b000: p = 3'b001;
            3'b001: p = 3'b010;
            ...  
            3'b111: p = 3'b000;
        endcase
    always @(posedge clk)  //next becomes current state
        q <= p;
endmodule
```

How Do We Turn a State Diagram into Logic?

- Counter
  - Three 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

How Do We Turn a State Diagram into Logic?

- Counter
  - Three 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

<table>
<thead>
<tr>
<th>current state</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 000</td>
<td>001</td>
</tr>
<tr>
<td>1 001</td>
<td>010</td>
</tr>
<tr>
<td>2 010</td>
<td>011</td>
</tr>
<tr>
<td>3 011</td>
<td>100</td>
</tr>
<tr>
<td>4 100</td>
<td>101</td>
</tr>
<tr>
<td>5 101</td>
<td>110</td>
</tr>
<tr>
<td>6 110</td>
<td>111</td>
</tr>
<tr>
<td>7 111</td>
<td>000</td>
</tr>
</tbody>
</table>

Implementation

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

```
C3 C2 C1
0 0 0 1 0 1 0
0 1 0 0 1 1
0 1 1 1 1 0
1 0 1 1 1 0
1 1 0 1 1 1
1 1 1 0 0 0
```

```
N1 := C1'
N2 := C1C2 + C1'C2
N3 := C1C2C3 + C1'C3 + C2'C3
```

Implementation (cont'd)

- Programmable Logic Building Block for Sequential Logic
  - Macro-cell: FF + logic
  - D-FF
  - Two-level logic capability like PAL (e.g., 8 product terms)
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: S₁, S₂, ..., Sₖ
- Inputs: I₁, I₂, ..., Iₘ
- Outputs: O₁, O₂, ..., Oₙ
- Transition function: Fₛ(Sᵢ, Iⱼ)
- Output function: Fₒ(Sᵢ) or Fₒ(Sᵢ, Iⱼ)

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 Brain

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

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

C: Break in wall
Go forward, turning right slightly

D: Hit wall again
Back to state A

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

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

\[
\begin{array}{c|c|c|c|c|c}
\text{state} & \begin{array}{c} L \end{array} & \begin{array}{c} R \end{array} & \begin{array}{c} \text{next state} \end{array} & \begin{array}{c} X \end{array} & \begin{array}{c} Y \end{array} & \begin{array}{c} Z \end{array} \\
\hline
\text{LOST} & 0 & 0 & \text{LOST} & L & E/G & \text{F} \\
\text{LOST} & 1 & 0 & \text{LOST} & L & E/G & \text{F} \\
\text{LOST} & 0 & 1 & \text{LOST} & L & E/G & \text{F} \\
\text{LOST} & 1 & 1 & \text{LOST} & L & E/G & \text{F} \\
A & 0 & 0 & A & \text{B} & \text{TL} & \text{F} \\
A & 0 & 1 & A & \text{E/G} & \text{TL} & \text{F} \\
A & 1 & 0 & A & \text{E/G} & \text{TL} & \text{F} \\
A & 1 & 1 & A & \text{E/G} & \text{TL} & \text{F} \\
B & 0 & 0 & \text{B} & \text{C} & \text{TR} & \text{F} \\
B & 0 & 1 & \text{B} & \text{A} & \text{TR} & \text{F} \\
B & 1 & 0 & \text{B} & \text{A} & \text{TR} & \text{F} \\
B & 1 & 1 & \text{B} & \text{A} & \text{TR} & \text{F} \\
\end{array}
\]

Synthesis

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

\[
\begin{array}{c|c|c|c|c|c|c}
\text{state} & \begin{array}{c} X \end{array} & \begin{array}{c} Y \end{array} & \begin{array}{c} Z \end{array} & \begin{array}{c} \text{next state} \end{array} & \begin{array}{c} X' \end{array} & \begin{array}{c} Y' \end{array} & \begin{array}{c} Z' \end{array} \\
\hline
000 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
000 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
000 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\
000 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
010 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
010 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
010 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\
010 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
011 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
011 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
011 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\
011 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
\end{array}
\]

It now remains to synthesize these 6 functions

LOST
- 000
- 001
E/G
- 010
A
- 011
B
- 100
C
Synthesis of Next State and Output Functions

<table>
<thead>
<tr>
<th>state</th>
<th>inputs X, Y, Z</th>
<th>next state X', Y', Z'</th>
<th>outputs F, TR, TL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0 0</td>
<td>000 1 0 0</td>
<td>TR = X + Y Z</td>
<td></td>
</tr>
<tr>
<td>0001 0 1</td>
<td>001 1 0 0</td>
<td>TR = X' + Y Z' = R' TR</td>
<td></td>
</tr>
<tr>
<td>0010 0 0</td>
<td>001 0 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011 0 1</td>
<td>010 0 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100 0 0</td>
<td>010 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101 0 1</td>
<td>011 0 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110 0 0</td>
<td>011 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111 1 0</td>
<td>100 0 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000 0 0</td>
<td>100 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001 0 1</td>
<td>101 0 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010 0 0</td>
<td>101 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011 1 0</td>
<td>110 0 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100 0 0</td>
<td>110 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101 0 1</td>
<td>111 0 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110 0 0</td>
<td>111 1 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111 1 0</td>
<td>111 1 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Circuit Implementation

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

Verilog Sketch

```
module ant_brain (F, TR, TL, L, R);
inputs     L, R;
outputs    F, TR, TL;
reg        X, Y, Z;
assign F  = function(X, Y, Z, L, R);
assign TR = function(X, Y, Z, L, R);
assign TL = function(X, Y, Z, L, R);
always @(posedge clk)
begin
    X <= function (X, Y, Z, L, R);
    Y <= function (X, Y, Z, L, R);
    Z <= function (X, Y, Z, L, R);
end
endmodule
```
Don’t Cares in FSM Synthesis

- What happens to the "unused" states (101, 110, 111)?
- Exploited as don’t cares to minimize the logic
  - If states can’t happen, then 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

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