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

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
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 first lecture**

![Combination lock diagram](image)

Can Any Sequential System be Represented with a State Diagram?

- **Shift Register**
  - Input value shown on transition arcs
  - Output values shown within state node

![Shift Register diagram](image)
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, ...

![Diagram of a 3-bit up-counter]

Verilog Upcounter

module binary_cntr (q, clk)
inputs clk;
outputs [2:0] q;
reg [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

![Diagram of a counter circuit](image)

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

```
<table>
<thead>
<tr>
<th>N3</th>
<th>C3</th>
<th>C2</th>
<th>C1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 1</td>
<td>0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
</tr>
</tbody>
</table>

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

CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 11

CS 150 - Spring 2007 – Lec #6: Moore and Mealy Machines - 12
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)

Another Example

- Shift Register
  - Input determines next state
More Complex Counter Example

- **Complex Counter**
  - Repeats five 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**

![State Transition Diagram](image)

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

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**

![K-maps](image)

- **C+ := A**
- **B+ := B' + A'C'**
- **A+ := BC'**
Self-Starting Counters (cont'd)

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

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

Self-Starting Counters

- Start-up States
  - At power-up, counter may be in an unused or invalid state
  - Designer must guarantee 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
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 $F_o(S_i, I_j)$
First Midterm Exam—15 February 2007

Topics to be covered:
- Combinational logic design
  - From spec to truth table to K-map to Boolean Expression
  - Canonical forms of Boolean Expressions
  - Conversions of AND-OR logic to NAND or NOR logic
  - Two level logic implementations using gates, PLA, MUX, DEC, ROM, Xilinx CLB FPGA structures
    - Comparing implementation complexities/figures of merit
    - Combinational Verilog (lab expertise!)
- Basic Sequential logic design
  - Flip flop behavior, analysis, and timing diagrams
  - Using flip flops to design registers, shifters, counters
  - From spec to state diagram to Sequential Verilog
  - Amount of FSM implementation through end of today

Exam mechanics
- Worth ONLY 10% of course grade
- In class, designed for 1 hour, full 80 minutes available
- WILL TAKE PLACE IN 125 CORY LABORATORY!!!
- No Blue Book—all work to be done on the exam paper!
  - Bring pencil and eraser—DUMB to use pen!
  - Cheating = 0 on exam—DO NOT DO IT!
    - F in class plus letter to file for second offense
- Closed Book, Closed Notes BUT
  - 8.5" x 11" two-sided crib sheet OK
    - Developing your crib sheet is a great way to study
    - Don't forget old exams and solutions are all on-line
  - No calculators, PDAs, laptops, camera phones, icq to experts ...
- Write assumptions if problem spec is ambiguous
- Difficult to ask questions during the exam itself
- Written regrade appeals policy
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 Maze](image-url)
**Ant Behavior**

- **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

---

**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**

<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>LOST</td>
<td>1</td>
<td>0</td>
<td>E/G</td>
<td>F</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>B</td>
<td>0</td>
<td>0</td>
<td>C</td>
<td>TR, F</td>
</tr>
<tr>
<td>B</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>(L)</th>
<th>(R)</th>
<th>next state</th>
<th>outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X,Y,Z</td>
<td></td>
<td></td>
<td>(X', Y', Z')</td>
<td>(F)</td>
</tr>
<tr>
<td>000 0 0</td>
<td>0</td>
<td>0</td>
<td>0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>000 0 1</td>
<td>0</td>
<td>1</td>
<td>0 1</td>
<td>0 0</td>
</tr>
<tr>
<td>... ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>010 0 0</td>
<td>0</td>
<td>1</td>
<td>1 1</td>
<td>0 0 1</td>
</tr>
<tr>
<td>010 0 1</td>
<td>0</td>
<td>1</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>010 1 0</td>
<td>0</td>
<td>1</td>
<td>1 0</td>
<td>0 1</td>
</tr>
<tr>
<td>010 1 1</td>
<td>0</td>
<td>1</td>
<td>1 0</td>
<td>1 0</td>
</tr>
<tr>
<td>011 0 0</td>
<td>1</td>
<td>0</td>
<td>0 0</td>
<td>1 0</td>
</tr>
<tr>
<td>011 0 1</td>
<td>1</td>
<td>0</td>
<td>0 1</td>
<td>0 1</td>
</tr>
<tr>
<td>... ...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

it now remains to synthesize these 6 functions

Synthesis of Next State and Output Functions

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

\(e.g.\) \(TR = X + Y Z\)

\(X^+ = X R' + Y Z R' = R' TR\)
**Circuit Implementation**

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

![Circuit Diagram](image)

**Verilog Sketch**

```verilog
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

State Minimization

- Fewer states may mean fewer state variables
- High-level synthesis may generate many redundant states
- Two state 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

<table>
<thead>
<tr>
<th>state inputs</th>
<th>next state outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>X, Y L R</td>
<td>X', Y' F TR TL</td>
</tr>
<tr>
<td>00 0 0</td>
<td>00 1 0 0</td>
</tr>
<tr>
<td>00 - 1</td>
<td>01 1 0 0</td>
</tr>
<tr>
<td>00 1 -</td>
<td>01 1 0 0</td>
</tr>
<tr>
<td>01 0 0</td>
<td>11 0 0 1</td>
</tr>
<tr>
<td>01 - 1</td>
<td>01 0 0 1</td>
</tr>
<tr>
<td>01 1 -</td>
<td>01 0 0 1</td>
</tr>
<tr>
<td>10 0 0</td>
<td>11 1 0 1</td>
</tr>
<tr>
<td>10 0 1</td>
<td>10 1 0 1</td>
</tr>
<tr>
<td>10 1 -</td>
<td>01 1 0 1</td>
</tr>
<tr>
<td>11 - 0</td>
<td>11 1 1 0</td>
</tr>
<tr>
<td>11 - 1</td>
<td>10 1 1 0</td>
</tr>
</tbody>
</table>

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