Sequential Logic Examples

- Finite State Machine Concept
- FSMs are the decision making logic of digital designs
- Partitions designs into datapath and control elements
- When inputs are sampled and outputs asserted
- Basic Design Approach: 4-step Design Process
- Implementation Examples and Case Studies
  - Finite-string pattern recognizer
  - Complex counter
  - Traffic light controller
  - Door combination lock

General FSM Design Procedure

- (1) Determine inputs and outputs
- (2) Determine possible states of machine
  - State minimization
- (3) Encode states and outputs into a binary code
  - Output encoding
    - Possibly input encoding (if under our control)
- (4) Realize logic to implement functions for states and outputs
  - Combinational logic implementation and optimization
  - Choices in steps 2 and 3 have large effect on resulting logic

Finite String Pattern Recognizer

Finite String Pattern Recognizer (Step 1)

- Finite String Pattern Recognizer
  - One input (X) and one output (Z)
  - Output is asserted whenever the input sequence .0...0 has been observed as long as the sequence 100 has never been seen

Step 1: Understanding the Problem Statement
  - Sample input/output behavior:
    - X: 0 0 1 0 1 0 0 0 1 0 ...
    - Z: 0 0 0 1 1 0 1 0 0 0 ...
    - X: 1 1 0 1 1 1 0 1 0 ...
    - Z: 0 0 0 0 0 0 0 0 0 ...

Finite String Pattern Recognizer (Step 2)

Step 2: Draw State Diagram
  - For the strings that must be recognized, i.e., 010 and 100
  - Moore implementation

Finite String Pattern Recognizer

Finite String Pattern Recognizer (Step 2, cont’d)

- Exit conditions from state S3: have recognized .010
  - If next input is 0 then have .0010...100 (state S6)
  - If next input is 1 then have .0001 = .01 (state S7)

Exit conditions from S1: recognize strings of form .0 (to 1 seen)
  - Loop back to S1 if input is 0

Exit conditions from S4: recognize strings of form .1 (to 0 seen)
  - Loop back to S4 if input is 1

Finite String Pattern Recognizer

Finite String Pattern Recognizer (Step 2, cont’d)

- S2 and S5 still have incomplete transitions
  - S2 = 01; If next input is 1, then string could be prefix of (01)X(00)
  - S4 handles just this case
  - S5 = 10; If next input is 1, then string could be prefix of (01)X(00)
  - S2 handles just this case

Reuse states as much as possible
- Look for some meaning
  - State minimization leads to smaller number of bits to represent states
- Once all states have complete set of transitions we have final state diagram
Finite String Pattern Recognizer (Step 3)

Verilog description including state assignment (or state encoding):

```verilog
module string (clk, X, rst, Q0, Q1, Q2, Z);
    always @(posedge clk)
    begin
        state = state + 1'b0;
        if (X) state = state + 1'b0;
        if (Z) state = state + 1'b0;
    end
    assign Z2 = state[2];
    assign Z1 = state[1];
    assign Z0 = state[0];
endmodule
```

Finite String Pattern Recognizer

Review of Process

- Review individual inputs and outputs to understand specification
- Derive a state diagram
- Write down sequenced of states and transitions for sequential
- To be recognized
- Minimize number of states
- Addking transitions, remove states at much as possible
- State assignment or encoding
- Encode states with unique patterns
- Simulate realization
- Verify if O behavior of your state diagram to ensure it matches specification

Complex Counter

Synchronous 3-bit counter has a mode control M.
- When M = 0, the counter counts in the binary sequence.
- When M = 1, the counter advances through the Gray code sequence.
- Binary: 000, 001, 010, 011, 100, 101, 110, 111
- Gray: 000, 011, 110, 101
- Valid 1/0 behavior (partial):

<table>
<thead>
<tr>
<th>Field (bit)</th>
<th>Gray State</th>
<th>Binary State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>110</td>
<td>101</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>4</td>
<td>010</td>
<td>011</td>
</tr>
<tr>
<td>5</td>
<td>011</td>
<td>000</td>
</tr>
</tbody>
</table>

Complex Counter (State Diagram)

Deriving State Diagram
- One state for each output combination
- Add appropriate arcs for the mode control

Traffic Light Controller as Two Communicating FSMs

Without Separate Timer
- S0 would require 7 states
- S1 would require 3 states
- S2 would require 7 states
- S3 would require 3 states
- S1 and S3 have simple transformation
- S0 and S2 would require many more arcs
- C could change in any of seven states

By Factoring Out Timer
- Greatly reduce number of states
- 4 instead of 20
- Counter only requires seven or eight states
- 12 total instead of 20
Communicating Finite State Machines

- One machine’s output is another machine’s input

![Finite State Machine Diagram](image)

Datapath and Control

- Digital hardware systems = data-path + control
  - Datapath: registers, counters, combinational functional units (e.g., ALU, communication (e.g., buses)
  - Control: FSM generating sequences of control signals that instructs datapath what to do next

![Datapath and Control Diagram](image)

Digital Combinational Lock

- Door Combination Lock:
  - Punch in values in sequence and the door opens; if there is an error the lock must be reset; once the door opens the lock must be reset
  - Inputs: sequence of input values, reset
  - Outputs: door open/closed
  - Memory: must remember combination or always have it available
  - Open questions: how do you test the internal combination?
    - Stored in memory (hashed)
    - Hardwired via switches left by others

![Digital Combinational Lock Diagram](image)

Implementation in Software

```plaintext
integer combination_lock() {  
  integer v1, v2, v3;  
  integer error = 0;  
  static integer o[3] = 3, 4, 2;  
  while (new_value()) {  
    v1 = read_value();  
    if (v1 != o[1]) then error = 1;  
    while (new_value()) {  
      v2 = read_value();  
      if (v2 != o[2]) then error = 1;  
      while (new_value()) {  
        v3 = read_value();  
        if (v3 != o[3]) then error = 1;  
        if (error == 1) then return(1); else return (1);  
      }  
    }  
  }  
return(0);  
```

![Implementation in Software Diagram](image)

Determining Details of the Specification

- How many bits per input value?
- How many values are in sequence?
- How do we know a new input value is entered?
- What are the states and state transitions of the system?

![Determining Details of the Specification Diagram](image)

Digital Combination Lock State Diagram

- States: 5 states
  - Represent points in execution of machine
  - Each state has outputs
  - Transitions: 6 from state to state, 5 self transitions, 1 global
  - Changes of state occur when clock says it’s ok
  - Based on value of inputs
- Inputs: reset, new, results of comparisions
- Output: open/closed

![Digital Combination Lock State Diagram](image)
Datapath and Control Structure

State Table for Combination Lock

Encodings for Combination Lock

Datapath Implementation for Combination Lock

Datapath Implementation (cont'd)

Tri-State Gates
Tri-State and Multiplexing

- When Using Tri-State Logic
  1. Never use more than one "driver" for any one line at any one time (pulling high and low at same time can severely damage circuit)
  2. Only use one line when it's being driven (using a floating value may cause failure)

- Using Tri-State Gates to Implement an Economical Multiplexer

Open-Collector Gates and Wired-AND

- Open collector: another way to connect gate outputs to same wire
  - Can be used if you need to pull output low
  - Cannot actively drive output high (default = pulled high through resistor)

- Wired-AND can be implemented with open collector logic
  1. If A and B are "1", output is actively pulled low
  2. If A and B are "0", output is stuck high
  3. If one gate output is low and the other high, then low
  4. If both outputs are "1", the wire will be "1" if not pulled high by resistor

- Hence, the two NAND functions are ANDed together

Digital Combination Lock (New Datapath)

- Decrease number of inputs
- Remove 3 code digits as inputs
- Use code registers
- Make them readable from value
- Need 3 least significant inputs (most in input (4*3)-3=9)
  - Case 1: done with 3 types of 1 and decoder
    1. Case 1: done with 2's complement decoder

Section Summary

- FSM Design
  - Understanding the problem
  - Generations of state diagram
  - Implementation using synthesis tools
  - Iteration on design/specification to improve quality of mapping
  - Communicating state machines

- 4*4 case studies
  - Understand I/O behavior
  - Draw diagrams
  - Enumerate states for the `get` and `set`
  - Expand with error conditions
  - Reuse states whenever possible