Recall: Parallel to Serial Converter

```verilog
module ParToSer(LD, X, out, CLK);
    input [1:0] X;
    input LD, CLK;
    output out;
    reg [3:0] Q;
    assign out = Q[0];
    always @(posedge CLK) begin
        if (LD) Q <= X;
        else Q <= (1'b0, Q[3:1]);
    end
endmodule // ParToSer
```

One common use of FSMs is in adapters from one subsystem to another.
- different data widths
- different bit rates
- different protocols,...
Example: Byte-bit stream

- **Byte FIFO**
- **Shift register**
- **Controller**
- **Serial link**
- **Pop**

Example: Byte-bit stream with Rate Matching

- **Byte FIFO**
- **Shift register**
- **Controller**
- **Serial link**
- **Pop**
- **Rdy**
- **How would you implement this FSM?**

Another example: bus protocols

- **A bus is:**
  - shared communication link
  - single set of wires used to connect multiple subsystems

- **A Bus is also a fundamental tool for composing large, complex systems (more later in the term)**
  - systematic means of abstraction

Example: Pentium System Organization

- **Processor/Memory Bus**
- **PCI Bus**
- **I/O Busses**

Arbitration for the bus...

- **Device 1**
- **Device 2**
- **Device N**

- **Central arbitration shown here**
  - Used in essentially all processor-memory busses and in high-speed I/O busses

Simple Synchronous Protocol

- **BReq**
- **BG**
- **CMD Address**
- **Data**

- **Even memory busses are more complex than this**
  - Memory (slave) may take time to respond
  - It needs to control data rate
• Memory waits?
• Additional outputs?
• Memory side?

Fundamental Design Principle
• 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)

General Model of Synchronous Circuit
• All wires, except clock, may be multiple bits wide.
• Registers (reg)
  • collections of flip-flops
  • clock
    • distributed to all flip-flops
    • typical rate?
  • Combinational Logic Blocks (CL)
    • no internal state (no feedback)
    • output only a function of inputs
    • Particular inputs/outputs are optional
    • Optional Feedback
    • ALL CYCLES GO THROUGH A REG!

Composing FSMs into larger designs
Composing Moore FSMs

- Synchronous design methodology preserved

Composing Mealy FSMs ...

- Synchronous design methodology violated!!!
- Why do designers used them?
  - Few states, often more natural in isolation
  - Safe if latch all the outputs
  - Looks like a mealy machine, but isn’t really
  - What happens to the timing?

Finite State Machines in Verilog

FSM timing

- What determines min FSM cycle time (max clock rate)?

Announcements

- Reading 8.4, 7.4, 8.1
- We touched on ideas from chapter 11, not in reader. Will be available on line.

Verilog FSM - Reduce 1s example

- Change the first 1 to 0 in each string of 1’s
  - Example Moore machine implementation

module Reduce(Out, Clock, Reset, In);
input Clock, Reset, In;
reg Out;
reg [1:0] CurrentState; // state register
reg [1:0] NextState;
// State assignment
localparam STATE_00 = 2'b00,
STATE_01 = 2'b01,
STATE_10 = 2'b10,
STATE_11 = 2'b11;
// Current state
CurrentState = In;
if (CurrentState == STATE_00) NextState = STATE_00;
if (CurrentState == STATE_01) NextState = STATE_10;
if (CurrentState == STATE_10) NextState = STATE_01;
if (CurrentState == STATE_11) NextState = STATE_00;
// Next state
NextState = In;
// Output logic
Out = NextState;
endmodule
Moore Verilog FSM: combinational part

always @ (posedge clock) begin
    NextState = case (CurrentState)
        STATE_Zero: begin // we've seen zero
            NextState = STATE_Zero;
        end
        STATE_One: begin // we've seen one
            NextState = STATE_One;
        end
        default: begin // we've seen both
            NextState = STATE_Two;
        end
    endcase
end

Moore Verilog FSM: state part

// Implement the state register
always @ (posedge clock) begin
    CurrentState <= NextState;
end

endmodule

Note: posedge Clock requires NONBLOCKING ASSIGNMENT.

Blocking Assignment is Sequential Logic (Registers)
Nonblocking Assignment is Combinational Logic

Mealy Verilog FSM for Reduce-1s example

module Reduce (clock, reset, in, out);
    input Clock, Reset, In;
    output Out;
    reg CurrentState; // state register
    reg NextState;
    localparam STATE_Zero = 1'd0;
    always @ (posedge Clock) begin
        if (Reset) CurrentState = STATE_Zero;
        else CurrentState = case (CurrentState)
            STATE_Zero: begin // we've seen zero
                NextState = STATE_Zero;
            end
            STATE_One: begin // we've seen one
                NextState = STATE_One;
            end
            default: begin // we've seen both
                NextState = STATE_Two;
            end
        endcase
        CurrentState <= NextState;
    end
end

Restricted FSM Implementation Style

- Mealy machine requires two always blocks
  - Register needs posedge Clock block
  - Input to output needs combinational block
- Moore machine can be done with one always block, but...
  - E.g., simple counter
  - Very bad idea for general FSMs
    - This will cost you hours of confusion, don't try it
  - We will not accept labs with this style for general FSMs
- Use two always blocks!
- Moore outputs
  - Share with state register, use suitable state encoding

Single-always Moore Machine (Not Allowed!)

module Single_always_Moore (clk, reset, in, out);
    input Clk, Reset, In;
    output Out;
    reg [1:0] State; // state register
    parameter zero = 0, one = 1, twos = 2;
    reg AlwaysState;
    always @ (posedge clk) begin
        if (reset) State = zero;
        else State = case (State)
            zero: begin // we've seen zero
                AlwaysState <= one;
            end
            one: begin // we've seen one
                AlwaysState <= zero;
            end
            default: begin // we've seen both
                AlwaysState <= twos;
            end
        endcase
        if (AlwaysState) begin // we've seen both
            Out <= one;
        end
    end
end

Single-always Moore Machine (Not Allowed!)

always @ (posedge clk) begin
    if (reset) State <= zero;
    else State <= case (State)
        zero: begin // we've seen zero
            NextState <= one;
        end
        one: begin // we've seen one
            NextState <= zero;
        end
        default: begin // we've seen both
            NextState <= twos;
        end
    end
end

This is confusing: the output does not change until the next clock cycle.
Finite State Machines

- Implement combinational logic using one always block
- Implement an explicit state register using a second always block

State Reduction

- State Reduction is based on:
  Two states are equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state.
  If two states are equivalent, one can be eliminated without affecting the behavior of the FSM.
- Several algorithms exist:
  - Row matching method.
  - Implication table method.

  “Row Matching” is based on the state-transition table:
  If two states have the same output, and both transition to the same next state, or both transition to each other, or both self-loop, then they are equivalent.
  Combine the equivalent states into a new renamed state.
  Repeat until no more states are combined.
  - Note: This algorithm is slightly different than the book.

Row Matching Example

<table>
<thead>
<tr>
<th>State Transition Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
</tr>
<tr>
<td>PS</td>
</tr>
<tr>
<td>x=0</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

Row Matching Example (cont)

<table>
<thead>
<tr>
<th>Reduced State Transition Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>NS</td>
</tr>
<tr>
<td>PS</td>
</tr>
<tr>
<td>x=0</td>
</tr>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
</tbody>
</table>

FSM Optimization

- State Reduction:
  Motivation:
  - lower cost
  - fewer flip-flops in one-hot implementations
  - possibly fewer flip-flops in encoded implementations
  - more don’t cares in NS logic
  - fewer gates in NS logic
  - Simpler to design with extra states then reduce later.

- Example: Odd parity checker.
  Two machines - identical behavior.

State Reduction:

- The “row matching” method is not guaranteed to result in the optimal solution in all cases, because it only looks at pairs of states.
- For example:

  - Another (more complicated) method guarantees the optimal solution:
    - ‘Implication table’ method:
      cf. Mano, chapter 9
    - What ‘rule of thumb’ heuristics?
Summary

- FSMs are critical tool in your design toolbox
  - Adapters, Protocols, Datapath Controllers, ...
- They often interact with other FSMs
- Important to design each well and to make them work together well.
- Keep your verilog FSMs clean
  - Separate combinational part from state update
- Good state machine design is an iterative process
  - State encoding
  - Reduction
  - Assignment

"K-maps" are used to help visualize good encodings.
- Adjacent states in the STD should be made adjacent in the map.

State Maps

State Assignment

- Alternative heuristics based on input and output behavior as well as transitions:
- Important to design each well and to make them work together well.
- Keep your verilog FSMs clean
  - Separate combinational part from state update
- Good state machine design is an iterative process
  - State encoding
  - Reduction
  - Assignment