Digital Design Implementation Guide

Table of Contents

Introduction

This guide provides a structured approach to implementing Finite State Machines (FSMs) in Verilog, covering both Mealy and Moore machine types.

Understanding FSM Types

Mealy Machine

- Outputs depend on both current state AND inputs
- Generally uses fewer states
- Output can change asynchronously with input
- Output equation: Z = f(Present State, Input)

Moore Machine

- Outputs depend ONLY on current state
- May require more states than Mealy
- Output changes synchronously with state transitions
- Output equation: Z = f(Present State)

FSM Design Steps

1. **Problem Analysis**
    - Identify inputs and outputs
    - List all possible states
    - Determine required state transitions
2. **State Encoding**
    - Assign binary codes to states

- Use parameters for readability

```
parameter S0 = 3'b000, S1 = 3'b001, ...;
```

3. **State Register Declaration**

   - Define state register size

```
reg [2:0] state;  // For up to 8 states
```

4. **Implementation Choice**

   - Decide between Mealy or Moore based on requirements
   - Consider timing and state count requirements

Code Templates

Mealy Machine Template

```
module fsm_mealy(
    input clk, reset,
    input w,
    output reg z
);
    // State definitions
    parameter   A = 3'b000,
                B = 3'b001,
                C = 3'b010;

    reg [2:0] state;

    always @(posedge clk, posedge reset) begin
        if (reset) begin
            state <= A;
        end
        else begin
            case(state)
                A: if(w) begin state<=B; z=0; end
                   else begin state<=A; z=1; end
                B: if(w) begin state<=C; z=1; end
                   else begin state<=A; z=0; end
                // Add more states as needed
                default: state <= 3'bxxx;
            endcase
        end
    end
endmodule
```

Moore Machine Template

```
module fsm_moore(
    input clk, reset,
    input w,
    output reg z
);
    // State definitions
    parameter   A = 3'b000,
                B = 3'b001,
                C = 3'b010;

    reg [2:0] state;

    // Output logic - depends only on state
    always @(state) begin
        case(state)
            A: z = 0;
            B: z = 1;
            C: z = 0;
            default: z = 0;
        endcase
    end

    // State transitions
    always @(posedge clk, posedge reset) begin
        if (reset) begin
            state <= A;
        end
        else begin
            case(state)
                A: if(w) state<=B; else state<=A;
                B: if(w) state<=C; else state<=A;
                // Add more states as needed
                default: state <= 3'bxxx;
            endcase
        end
    end
endmodule
```

Implementation Examples

Converting Mealy to Moore

1. Separate output logic from state transitions
2. Move output assignments to a dedicated always block
3. Make output dependent only on state

4. Add additional states if needed

Key Differences in Implementation

- Mealy: Output assignments within state transition logic
- Moore: Separate output logic block
- Mealy: Output can change with input
- Moore: Output changes only with state

Best Practices

1. **State Encoding**
    - Use meaningful state names
    - Consider one-hot encoding for large FSMs
    - Use parameters for state definitions
2. **Reset Logic**
    - Always include synchronous or asynchronous reset
    - Initialize to a known state
3. **Default Cases**
    - Include default cases in case statements
    - Handle undefined states
4. **Documentation**
    - Comment state transitions
    - Include timing diagrams
    - Document input/output behavior

Troubleshooting

Common Issues and Solutions:

1. **Unintended State Transitions**
    - Check reset logic
    - Verify state encoding
    - Review case statement completeness
2. **Timing Issues**
    - Review clock domain crossings
    - Check for combinational loops
    - Verify setup/hold times
3. **Output Glitches**
    - In Mealy: Check input synchronization
    - In Moore: Verify output logic separation

4. **Synthesis Issues**
    - Use proper always block sensitivity lists
    - Avoid latches
    - Check for incomplete assignments

Priority Encoder

Introduction

A priority encoder is a combinational circuit that outputs the binary code of the highest priority input that is active.

8-to-3 Priority Encoder

Code Template

```
module priority_encoder_8to3(
    input [7:0] in,      // 8-bit input
    output reg [2:0] out,// 3-bit output
    output reg valid     // valid output indicator
);

always @(*) begin
    casex(in)
        8'b1xxxxxxx: begin out = 3'b111; valid = 1'b1; end  // Priority 7
        8'b01xxxxxx: begin out = 3'b110; valid = 1'b1; end  // Priority 6
        8'b001xxxxx: begin out = 3'b101; valid = 1'b1; end  // Priority 5
        8'b0001xxxx: begin out = 3'b100; valid = 1'b1; end  // Priority 4
        8'b00001xxx: begin out = 3'b011; valid = 1'b1; end  // Priority 3
        8'b000001xx: begin out = 3'b010; valid = 1'b1; end  // Priority 2
        8'b0000001x: begin out = 3'b001; valid = 1'b1; end  // Priority 1
        8'b00000001: begin out = 3'b000; valid = 1'b1; end  // Priority 0
        8'b00000000: begin out = 3'b000; valid = 1'b0; end  // No input
        default: begin out = 3'b000; valid = 1'b0; end
    endcase
end
endmodule
```

Key Features

1. **Input** [**7:0**]
    - 8-bit input where each bit represents a priority level
    - Bit 7 has highest priority, Bit 0 has lowest priority
2. **Output** [**2:0**]
    - 3-bit binary output representing highest active input

- Output ranges from 000 to 111

3. **Valid Signal**
   - Indicates if any input is active
   - High when at least one input is 1
   - Low when all inputs are 0

Truth Table Example

```
Input       | Output | Valid
8'b10100000 | 3'b111 | 1    // Bit 7 is high (highest priority)
8'b00100000 | 3'b101 | 1    // Bit 5 is high
8'b00000010 | 3'b001 | 1    // Bit 1 is high
8'b00000000 | 3'b000 | 0    // No bits high
```

Implementation Notes

1. **Using casex**
   - 'x' represents "don't care" conditions
   - Simplifies priority logic implementation
   - Makes code more readable
2. **Priority Logic**
   - Always checks highest priority bit first
   - Lower priority bits are ignored if higher priority bit is set
   - Uses cascading priority structure
3. **Best Practices**
   - Use meaningful signal names
   - Include valid signal for error checking
   - Document priority levels clearly
   - Consider adding parameter definitions for flexibility
4. **Common Applications**
   - Interrupt handling systems
   - Resource allocation
   - Task scheduling
   - Memory management

Quartus 8.1 Simulation Guide

Creating Waveform File (.vwf)

1. **Create New Waveform**
   - File → New
   - Select "Vector Waveform File" (.vwf)

- Save file (e.g., “fsm_simulation.vwf”)

2. **Basic Setup**
   - Edit → End Time (set to 1000ns)
   - Set Grid size as needed (View → Grid Size)

Adding Signals

1. **For FSM Testing**

```
Right-click → Insert Node or Bus:
- clk    (1-bit)
- reset  (1-bit)
- w      (1-bit)
- z      (1-bit)
- state  (3-bits)
```

2. **For Priority Encoder Testing**

```
Right-click → Insert Node or Bus:
- in[7:0]   (8-bits)
- out[2:0]  (3-bits)
- valid     (1-bit)
```

Setting Test Patterns

1. **FSM Test Pattern**

```
Clock setup:
- Right-click on clk → Clock
- Set period = 20ns

Reset pattern:
- High (0-40ns)
- Low (40ns onwards)

Input 'w' test sequence:
Time(ns)  w    Expected z
40-100    1    0
100-160   0    0
160-220   1    0
220-280   1    1
280-340   0    0
```

2. **Priority Encoder Test Pattern**

```
Input 'in[7:0]' sequence:
Time(ns)  in[7:0]    Expected out[2:0]  valid
0-100     10000000   111                1
```

100-200	01000000	110	1
200-300	00100000	101	1
300-400	00000001	000	1
400-500	00000000	000	0

Running Simulation

1. **Preparation**
    - Processing → Generate Functional Simulation Netlist
    - Assignments → Settings → Simulator: "Quartus II Simulator"
    - File → Save
2. **Execute**
    - Processing → Start Simulation

Analyzing Results

1. **FSM Verification Points**
    - Reset behavior correct
    - State transitions on clock edges
    - Output changes match specification
    - No unexpected state transitions
2. **Priority Encoder Checks**
    - Correct priority handling
    - Valid signal operation
    - Output stability
    - Response to input changes

Common Issues and Solutions

1. **Timing Problems**
    - Check clock period settings
    - Verify signal transition times
    - Ensure adequate test duration
2. **Display Issues**
    - Adjust radix display (Right-click → Radix)
    - Modify grid settings
    - Change time scale if needed
3. **Simulation Errors**
    - Verify node names match design
    - Check signal connections
    - Confirm proper netlist generation

Best Practices

1. **Test Organization**
   - Use clear test patterns
   - Include comments in waveform
   - Test all state transitions
   - Verify edge cases
2. **Documentation**
   - Save waveform configurations
   - Export important results
   - Document test scenarios
3. **Verification Steps**
   - Confirm reset operation
   - Check all state transitions
   - Verify output timing
   - Test boundary conditions