Modeling Sequential Circuits and FSMs with Verilog

Prof. Chien-Nan Liu
TEL: 03-4227151 ext:34534
Email: jimmy@ee.ncu.edu.tw

Sequential Circuit Design

- Typical design procedure
  1. Obtain either the state diagram or the state table from the statement of the problem
  2. Reduce the number of states if necessary
  3. Assign binary codes to the states
  4. Obtain the binary coded state table
  5. Choose the type of flip-flop to be used
  6. Derive the simplified flip-flop input equations and output equations from the state table
  7. Draw the logic diagram with D flip-flops and combinational gates according to those equations
Build the State Diagram

Example 4-1:
Implement a circuit that recognizes the occurrence of the sequence of bits $1101$ on $X$ by making $Z$ equal to 1 when the previous three inputs to the circuit were 110 and current input is a 1.

Build the state diagram:

Obtain the State Table

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State $X = 0$</th>
<th>Next State $X = 1$</th>
<th>Output $Z$ $X = 0$</th>
<th>Output $Z$ $X = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>C</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 4-5: State Table for State Diagram in Figure 4-21
Derive the Equations

### TABLE 4-8
**State Table for Design Example**

<table>
<thead>
<tr>
<th>Present State</th>
<th>Input</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

### Equations

- \( D_A = AB + BX \)
- \( D_B = AX + BX + ABX \)
- \( Y = BX \)

---

The Final Circuit

![The Final Circuit Diagram](image)

*Fig. 4-25 Logic Diagram for Sequential Circuit with D Flip-Flops*
Designing with Unused States

Only 5 states are used!!
Unused states are treated as don’t cares.

State Assignment

- State assignment: determine the binary representations of the states in a circuit
- For a circuit with \( n \) state registers, there are \( 2^n \) possible state assignments
- State assignments have large impacts on the resultant circuit area, performance, ...
- Many coding methods exist
  - One-hot: \( n \) bits for \( n \) states (one-to-one mapping)
  - Gray code: change only one bit between adjacent states
  - Minimum length encoding: require \( (n^{1/2} + 1) \) registers
  - Ad-hoc: determine by experience
- Some general guidelines can be referenced
Guidelines for State Assignment

- Assignments for two states are adjacent if they differ in only one variable
  - 010 and 011 are adjacent
  - 010 and 001 are not adjacent
- **Guideline 1**: states which have the same next state for a given input could be given adjacent assignments
- **Guideline 2**: states which are the next states of the same state could be given adjacent assignments
- **Guideline 3**: states which have the same output for a given input could be given adjacent assignments

State Assignment Example

- **Guideline 1**: (S0,S1,S3,S5) (S3,S5) (S4,S6) (S0,S2,S4,S6)
- **Guideline 2**: (S1, S2) (S2,S3) (S1,S4) (S2,S5)X2 (S1,S6)X2
- Try to fulfill as many of these adjacency conditions as possible
  - It's hard to satisfy all those conditions
  - The conditions that appear more times have higher priority
  - K-map can help us to check those conditions
- Two possible state assignments are demonstrated
Effects of the Guidelines

Finite State Machine

- **FSM**: Finite State Machine
  - Sequential circuits with “finite” states
  - Most sequential circuits can be classified as FSMs

- Two primary categories:
  - **Mealy machine**
    - Outputs depend on current states and inputs
  - **Moore machine**
    - Outputs depend on current states only
FSM Structures (1/3)

Mealy

Inputs → Next State Logic (combinational) → Current State Register (sequential) → Output Logic (combinational) → Mealy Outputs

Synchronous reset → Asynchronous reset

FSM Structures (2/3)

Moore

Inputs → Next State Logic (combinational) → Current State Register (sequential) → Output Logic (combinational) → Moore Outputs

Synchronous reset → Asynchronous reset
FSM Structures (3/3)

Next State Logic (combinational)
Current State Register (sequential)
Output Logic (combinational)

Synchronous reset
Asynchronous reset

Mealy/Moore

Coding for FSM

- Separate the combination part and the sequential part
- Use parameters to define the state variables

```verbatim
always@(state) begin
  case(state)
    stateA: nextstate=stateB;
    stateB: nextstate=stateC;
    stateC: nextstate=stateD;
    stateD: nextstate=stateA;
  endcase
end
```

```verbatim
parameter stateA=2'b00,
stateB=2'b01,
stateC=2'b10,
stateD=2'b11;
```

```verbatim
always@(posedge clk) begin
  if (reset) state<=stateA;
  else state<=nextstate;
end
```

Sequential part

Initial state

Combinational part
Blocking v.s Non-Blocking

- Use non-blocking assignments for sequential block
  - Store values until the end of the time slice
  - Avoid simulation race conditions or ambiguity of results
- Use blocking assignments for combinational block
  - Blocking assignments occur immediate in nature

```
always @(posedge clk)
begin
  outa = in;
  outb = outa;
  outc = outb;
end
```

Blocking Assignment
Non-Blocking Assignment

always @(posedge clk)
begin
    outa<=in;
    outb<=outa;
    outc<=outb;
end

Illustration of Execution Order
Example: Sequence Recognizer

```vhdl
// Sequence Recognizer: Verilog Process Description
// (See Figure 4.21 for state diagrams)
module seq_rec_y(CKE, RESET, I, Z);
  input CKE, RESET, I;
  output I;
  reg (1:0) state, next_state;
  parameter A = 2'h00, B = 2'h01, C = 2'h10, D = 2'h11;
  reg Z;
  // state register: implements positive edge-triggered
  // state storage with asynchronous reset.
  always @posedge CKE or posedge RESET
  begin
    if (RESET == 1) 
      state <= A;
    else
      state <= next_state;
  end
  // next state function: implements next state as function
  // of I and state
  always @I or state
  begin
    case (state)
      A: if (I == 1) 
        next_state <= B;
        else 
        next_state <= A;
      B: if (I == 0) 
        next_state <= C;
        else 
        next_state <= B;
      C: if (I == 0) 
        next_state <= D;
        else 
        next_state <= C;
      D: if (I == 1) 
        next_state <= B;
        else 
        next_state <= D;
    endcase
  end
endmodule
```

Various Coding Styles for FSM

- There are many different coding styles can be used to describe a FSM
  - HDL is a very flexible language
- Typical coding styles:
  - 1-process FSM
  - 2-process FSM
  - 3-process (or more) FSM

Just describe state-by-state!!
1-Process FSM

Lump all descriptions into a single process

module counter (clk, rst, load, in, count);
input clk, rst, load;
input [7:0] in;
output [7:0] count;
reg [7:0] count;
always @(posedge clk)
begin
if (rst) count = 0;
else if (load) count = in;
else if (count == 255) count = 0;
else count = count + 1;
end
endmodule

2-Process FSM (1/5)

functionally partitioned

structurally partitioned
module FSM_S2 (Clock, Reset, X, Y);
    input Clock, Reset, X;
    output [2:0] Y;
    reg [2:0] Y;
    reg [1:0] CS, NS;
    parameter ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
    always @(X or CS) begin : COMB
        case (CS)
            ST0 : begin
                Y = 1;
                NS = ST1;
            end
            ST1 : begin
                Y = 2;
                if (X) NS = ST3;
                else NS = ST2;
            end
            ST2 : begin
                Y = 3;
                NS = ST3;
            end
            ST3 : begin
                Y = 4;
                NS = ST0;
            end
            default : begin
                Y = 1;
                NS = ST0;
            end
        endcase
    end
    always @(posedge Clock or posedge Reset)
    begin : SEQ
        if (Reset)
            CS <= ST0;
        else
            CS <= NS;
    end
endmodule
2-Process FSM (4/5)

2-process, functionally partitioning

module FSM_F2 (Clock, Reset, X, Y);
    input Clock, Reset, X;
    output [2:0] Y;
    reg [2:0] Y;
    reg [1:0] STATE;
    parameter [1:0] ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
    always @(posedge Clock or posedge Reset)
        begin : NEXT_STATE
            if (Reset)
                STATE <= ST0;
            else
                case (STATE)
                    ST0 : STATE <= ST1;
                    ST1 : begin
                        if (X)
                            STATE <= ST3;
                        else
                            STATE <= ST2;
                    end
                    ST2 : STATE <= ST3;
                    ST3 : STATE <= ST0;
                endcase
        end
    end
endmodule

2-Process FSM (5/5)

case (STATE)
    ST0 : STATE <= ST1;
    ST1 : begin
        if (X)
            STATE <= ST3;
        else
            STATE <= ST2;
    end
    ST2 : STATE <= ST3;
    ST3 : STATE <= ST0;
endcase
always @(STATE)
    begin : OUT
        case (STATE)
            ST0 : Y = 1;
            ST1 : Y = 2;
            ST2 : Y = 3;
            ST3 : Y = 4;
            default : Y = 1;
        endcase
    end
endmodule
3-Process FSM (1/4)

- Current State Register (sequential)
- Next State Logic (combinational)
- Output Logic (combinational)

3-Process FSM (2/4)

- 3-process, structurally partitioning

```verbatim
module FSM_S3(Clock, Reset, X, Y);
    input Clock, Reset, X;
    output [2:0] Y;
    reg [2:0] Y;
    reg [1:0] CS, NS;
    parameter [1:0] ST0 = 0, ST1 = 1, ST2 = 2, ST3 = 3;
    always @(X or CS)
        begin : COMB
            NS = ST0;
        case (CS)
```
3-Process FSM (3/4)

```
ST0 : begin
    NS = ST1;
end
ST1 : begin
    if (X)
        NS = ST3;
    else
        NS = ST2;
end
ST2 : begin
    NS = ST3;
end
ST3 : begin
    NS = ST0;
end
endcase
end
// end process COMB
```

3-Process FSM (4/4)

```
always @(posedge Clock or posedge Reset)
begin : SEQ
    if (Reset)
        CS <= ST0;
    else
        CS <= NS;
end
```

```
always @(CS)
begin : OUT
    case (CS)
        ST0 : Y = 1;
        ST1 : Y = 2;
        ST2 : Y = 3;
        ST3 : Y = 4;
        default : Y = 1;
    endcase
end
endmodule
```