Digital System Design

Design of Datapath Controllers and Sequential Logic

Lecturer: Chihhao Chao
Date: 2009.03.18
Digital System Design

Sequential Circuit Model & Timing Parameters
Combinational Logic Review

- Combinational logic circuits are memoryless
- No feedback path
- Output can have multiple logical transitions before settling to correct value

Inputs ➔ Combinational Circuit ➔ Outputs
Model of Sequential Circuits

- Sequential circuits have memory (i.e. remember the past)
- The output is
  - depend on inputs
  - depend on current state
- In synchronous system
  - clock orchestrates the sequence of events
- Fundamental components
  - Combinational circuits
  - Memory elements
Example

Adding $N$ inputs ($N-1$ Adders)

Using a sequential (serial) approach
D-FF vs. D-Latch

- FF is *edge sensitive* (can be either positive or negative edge)
  - At trigger edge of clock, input transferred to output
- Latch is *level sensitive* (can be either active-high or active-low)
  - When clock is active, input passes to output (transparent)
  - When clock is not active, output stays unchanged
Important Timing Parameters (1)

There is a **timing window** around the clocking event during which the input must remain stable and unchanged in order to be recognized by flip/flop or latch.

**Clock:**
Periodic Event, causes state of memory element to change

**Setup Time** \( (T_{su}) \)
Minimum time before the clocking event by which the input must be stable

**Hold Time** \( (T_h) \)
Minimum time after the clocking event during which the input must remain stable

**Propagation Delay** \( (T_{cq} \text{ for edge-triggered flip/flop or } T_{dq} \text{ for latch}) \)
Delay overhead of the memory element
Important Timing Parameters (2)

Register Timing Parameters

- $T_{eq}$: worst case rising edge clock to q delay
- $T_{eq,cd}$: contamination or minimum delay from clock to q
- $T_{su}$: setup time
- $T_{h}$: hold time

Logic Timing Parameters

- $T_{logic}$: worst case delay through the combinational logic network
- $T_{logic,cd}$: contamination or minimum delay through logic network
System Timing: Minimum Period

\[ T > T_{cq} + T_{logic} + T_{su2} \]
System Timing: Minimum Delay

\[ T_{cq,cd} + T_{\text{logic,cd}} > T_{\text{hold}} \]

Make FF2 satisfy the hold time on rising edge
From The View of Dataflow

At some $t$

These computation are simultaneous:

\[
B(t) = f_{AB}(A(t-1)) \\
C(t) = f_{BC}(B(t-1)) \\
D(t) = f_{CD}(C(t-1))
\]
Digital System Design

Simple Sequential Circuit and Verilog Coding
The Sequential always Block

- Edge-triggered circuits are described using a sequential always block.

**Combinational**

```verilog
module combinational(a, b, sel, out);
    input a, b;
    input sel;
    output out;
    reg out;
    always @ (a or b or sel)
    begin
        if (sel) out = a;
        else out = b;
    end
endmodule
```

**Sequential**

```verilog
module sequential(a, b, sel, clk, out);
    input a, b;
    input sel, clk;
    output out;
    reg out;
    always @ (posedge clk)
    begin
        if (sel) out <= a;
        else out <= b;
    end
endmodule
```

![Combinational Circuit](image1)

![Sequential Circuit](image2)
Sensitivity List

- The use of `posedge` and `negedge` makes an `always` block sequential (edge-triggered).
- Unlike combinational always block, the sensitivity list does determine the behavior of synthesis.

**D Flip-flop with synchronous clear**

```verilog
define_sync_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;
always @(posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

**D Flip-flop with asynchronous clear**

```verilog
define_async_clear(d, clearb, clock, q);
input d, clearb, clock;
output q;
reg q;
always @(negedge clearb or posedge clock)
begin
  if (!clearb) q <= 1'b0;
  else q <= d;
end
endmodule
```

always block entered only at each positive clock edge

always block entered immediately when (active-low) clearb is asserted

Note: The following is **incorrect syntax**: `always @(clear or negedge clock)`

If one signal in the sensitivity list uses `posedge/negedge`, then all signals must.
Synchronous Reset vs. Asynchronous Reset

- **Synchronous Reset**

- **Asynchronous Reset**
Review of Blocking/Nonblocking Assignments

- Verilog supports 2 types of assignments in always block, with subtle different behaviors:

  - Blocking assignment: evaluation and assignment are immediate

    ```
    always @(a or b or c)
    begin
      x = a | b;  // 1. Evaluate a | b, assign result to x
      y = a ^ b ^ c;  // 2. Evaluate a^b^c, assign result to y
      z = b & ~c;  // 3. Evaluate b&(~c), assign result to z
    end
    ```

  - Nonblocking assignment: all assignment deferred until all right-hand sides have been evaluated (end of the virtual timestamp)

    ```
    always @(a or b or c)
    begin
      x <= a | b;  // 1. Evaluate a | b but defer assignment of x
      y <= a ^ b ^ c;  // 2. Evaluate a^b^c but defer assignment of y
      z <= b & ~c;  // 3. Evaluate b&(~c) but defer assignment of z
      4. Assign x, y, and z with their new values
    end
    ```
Assignment Styles for Sequential Logic

Will nonblocking and blocking assignments both produce the desired result?

```verilog
module nonblocking(in, clk, out);
    input in, clk;
    output out;
    reg q1, q2, out;

    always @(posedge clk)
    begin
        q1 <= in;
        q2 <= q1;
        out <= q2;
    end
endmodule
```

```verilog
module blocking(in, clk, out);
    input in, clk;
    output out;
    reg q1, q2, out;

    always @(posedge clk)
    begin
        q1 = in;
        q2 = q1;
        out = q2;
    end
endmodule
```
Use Nonblocking for Sequential Logic

```vhdl
always @(posedge clk)
begin
    q1 <= in;
    q2 <= q1;
    out <= q2;
end
```

“At each rising clock edge, q1, q2, and out simultaneously receive the old values of in, q1, and q2.”

```vhdl
always @(posedge clk)
begin
    q1 = in;
    q2 = q1;
    out = q2;
end
```

“At each rising clock edge, q1 = in.
After that, q2 = q1 = in.
After that, out = q2 = q1 = in.
Therefore out = in.”

- Blocking assignment do not reflect the intrinsic behavior of multi-stage sequential logic
- Guideline: use nonblocking assignment for sequential always block
Nonblocking vs. Blocking Assignment

• Nonblocking

• Blocking
Use Blocking for Combinational Logic

<table>
<thead>
<tr>
<th>Blocking Behavior</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Given) Initial Condition</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>a changes; always block triggered</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>x = a &amp; b;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>y = x</td>
<td>c;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Nonblocking Behavior

<table>
<thead>
<tr>
<th>Nonblocking Behavior</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>x</th>
<th>y</th>
<th>Deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Given) Initial Condition</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x &lt;= 0</td>
</tr>
<tr>
<td>a changes; always block triggered</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>x &lt;= a &amp; b;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x &lt;= 0, y &lt;= 1</td>
</tr>
<tr>
<td>y &lt;= x</td>
<td>c;</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Assignment completion</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- Nonblocking and blocking will both synthesize correctly. Will both simulate correctly?
- Nonblocking assignment do not reflect the intrinsic behavior of multi-stage combinational logic
- Guideline: use blocking assignment for combinational always block
Digital System Design

Finite State Machine (FSM)
Finite State Machine

- Used to control the circuit core
- Partition FSM and non-FSM part
Finite State Machines

- Synchronous (i.e. clocked) finite state machines (FSMs) have widespread application in digital systems, e.g. as datapath controllers in computational units and processors. Synchronous FSMs are characterized by a finite number of states and by clock-driven state transitions.

- Mealy Machine: The next state and the outputs depend on the present state and the inputs.

- Moore Machine: The next state depends on the present state and the inputs, but the output depends on only the present state.
What is FSM

- A model of computation consisting of
  - a set of states, (limited number)
  - a start state,
  - input symbols,
  - a transition function that maps input symbols and current states to a next state.

State transition diagram
Elements of FSM

- Memory Elements (ME)
  - Memorize Current States (CS)
  - Usually consist of FF or latch
  - N-bit FF have $2^n$ possible states

- Next-state Logic (NL)
  - Combinational Logic
  - Produce next state
    - Based on current state (CS) and input (X)

- Output Logic (OL)
  - Combinational Logic
  - Produce outputs (Z)
    - Based on current state, or
    - Based on current state and input
Mealy Machine

- Output is function of both
- Input and current state
Moore Machine

- Output is function of CS only
- Not function of inputs
Digital System Design

Building Behavioral Models of FSM
Modeling FSM in Verilog

- Sequential Circuits
  - Memory elements of States (CS)

- Combinational Circuits
  - Next-state Logic (NL)
  - Output Logic (OL)

- Three coding styles
  - (1) Separate CS, OL and NL
  - (2) Combines NL+ OL, separate CS
  - (3) Combine CS + NL, separate OL
Coding Style 1
Separate CS, NL, OL

- **CS**
  
  ```vhdl
  always @ (posedge clk)
  current_state <= next_state;
  ```

- **NL**
  
  ```vhdl
  always @ (current_state or In)
  case (current_state)
  S0: case (In)
      In0: next_state = S1;
      In1: next_state = S0;
      ...
      endcase //In
  S1: ...
  S2: ...
  endcase //current_state
  ```

- **OL**
  
  ```vhdl
  // if Moore
  always @ (current_state)
  Z = output_value;
  // if Mealy
  always @ (current_state or In)
  Z = output_value;
  ```
**Coding Style 2**

**Combine NL+OL; Separate CS**

- **CS**
  
  ```
  always @ (posedge clk)
  current_state <= next_state;
  ```

- **NL+OL**
  
  ```
  always @ (current_state or In)
  case (current_state)
  S0: begin
    case (In)
      In0: begin
        next_state = S1;
        Z =values; // Mealy
      end
      In1: . . .
    endcase // In
    Z =values; // Moore
  end //S0
  S1: . . .
  endcase // current_state
  ```
Coding Style 3

Combine CS+NL; Separate OL

- **CS+NL**

```vhdl
always @ (posedge clk)
begin
  case (state)
    S0: case (In)
    In0: state = S1;
    In1: state = S0;
    . . .
    endcase //In
    S1: . . .
  endcase //state
end
```

- **OL**

```vhdl
// if Moore
always @ (state)
  Z = output_value;
// if Mealy
always @ (state or In)
  Z = output_value;
```
Behavioral Models of FSM

Example 1

Note:
1. brake has higher priority
2. Moore machine
module speed_machine
( clock, accelerator, brake, speed );
input clock, accelerator, brake;
output [2:0] speed;
reg [2:0] speed;
reg [1:0] curstate, nxtstate;

// state encoding
parameter stopped  = 2`b00;
parameter s_slow   = 2`b01;
parameter s_medium = 2`b10;
parameter s_high   = 2`b11;

// CS
always @ ( posedge clock )
curstate <= nxtstate;

// NL, Next-state Logic
always@( curstate or accelerator or brake )
if ( brake == 1`b1 )
case ( curstate )
  stopped: nxtstate = stopped;
  s_low:   nxtstate = stopped;
  s_medium:nxtstate = s_low;
  s_high:  nxtstate = s_medium;
endcase
else if ( accelerator == 1`b1 )
case ( curstate )
  stopped: nxtstate = s_low;
  s_low:   nxtstate = s_medium;
  s_medium:nxtstate = s_high;
  s_high:  nxtstate = s_high;
endcase
else
  nxtstate = curstate;
endmodule

// OL, Output Logic
always @ ( state )
case( state )
  stopped: speed = 3’d0;
  s_low:   speed = 3’d1;
  s_medium: speed = 3’d3;
  s_high:  speed = 3’d7;
endcase
endmodule
module speed_machine
  ( clock, accelerator, brake, speed );
  input clock, accelerator, brake;
  output [2:0] speed;
  reg [2:0] speed;
  reg [1:0] curstate, nxtstate;
  // state encoding
  parameter stopped  = 2’b00;
  parameter s_slow   = 2’b01;
  parameter s_medium = 2’b10;
  parameter s_high   = 2’b11;
  // CS
  always @(posedge clock )
    current_state <= next_state;
  // NL + OL, Next-state Logic + Output Logic
  always@( curstate or accelerator or brake )
    if ( brake == 1’b1 )
      case ( curstate )
        stopped: {speed,nxtstate} = {3’d0,stopped};
        s_low:   {speed,nxtstate} = {3’d1,s_low};
        s_medium:{speed,nxtstate} = {3’d3,s_medium};
        s_high:  {speed,nxtstate} = {3’d7,s_high};
        endcase
    else if ( accelerator == 1’b1 )
      case ( curstate )
        stopped: {speed,nxtstate} = {3’d0,s_low};
        s_low:   {speed,nxtstate} = {3’d1,s_medium};
        s_medium:{speed,nxtstate} = {3’d3,s_high};
        s_high:  {speed,nxtstate} = {3’d7,s_high};
        endcase
    else
      case ( curstate )
        stopped: {speed,nxtstate} = {3’d0,curstate};
        s_low:   {speed,nxtstate} = {3’d1,curstate};
        s_medium:{speed,nxtstate} = {3’d3,curstate};
        s_high:  {speed,nxtstate} = {3’d7,curstate};
      endcase
  endmodule
module speed_machine  
( clock, accelerator, brake, speed );  
input clock, accelerator, brake;  
output [2:0] speed;  
reg [2:0] speed;  
reg [1:0] curstate;  
/* nxtstate is removed */  

// state encoding  
parameter stopped = 2`b00;  
parameter s_slow = 2`b01;  
parameter s_medium = 2`b10;  
parameter s_high = 2`b11;  

// OL, Output Logic  
always @ ( state )  
case( state )  
  stopped: speed = 3’d0;  
  s_low: speed = 3’d1;  
  s_medium: speed = 3’d3;  
  s_high: speed = 3’d7;  
endcase  
endmodule

// CS + NL, Current State + Next-state Logic  
always @( posedge clock )  
if ( brake == 1`b1 )  
case ( curstate )  
  stopped: curstate <= stopped;  
  s_low: curstate <= stopped;  
  s_medium: curstate <= s_low;  
  s_high: curstate <= s_medium;  
endcase  
else if ( accelerator == 1`b1 )  
case ( curstate )  
  stopped: curstate <= s_low;  
  s_low: curstate <= s_medium;  
  s_medium: curstate <= s_high;  
  s_high: curstate <= s_high;  
endcase  
/* else is removed  
   since it’s edge-trigger block (FF) */  
endmodule
module speed_machine
(input clock, accelerator, brake, speed);
output [2:0] speed;
reg [2:0] speed;
reg [1:0] curstate; /* nxtstate is removed */

// state encoding
parameter stopped = 2'b00;
parameter s_slow = 2'b01;
parameter s_medium = 2'b10;
parameter s_high = 2'b11;

always@( posedge clock )
if ( brake == 1'b1 )
  case ( curstate )
   stopped: {speed,curstate} <= {3’d0,stopped};
   s_low:   {speed,curstate} <= {3’d1,stopped};
   s_medium: {speed,curstate} <= {3’d3,s_low};
   s_high:  {speed,curstate} <= {3’d7,s_medium};
  endcase
else if ( accelerator == 1'b1 )
  case ( curstate )
   stopped: {speed,curstate} <= {3’d0,s_slow};
   s_low:   {speed,curstate} <= {3’d1,s_medium};
   s_medium: {speed,curstate} <= {3’d3,s_high};
   s_high:  {speed,curstate} <= {3’d7,s_high};
  endcase
/* else is removed since it’s edge-trigger block (FF) */
endmodule
# State encoding

- Different state encoding codes will affect
  - Current state register word length
  - Input format of state transition logic
  - Input format of output logic

<table>
<thead>
<tr>
<th>NO.</th>
<th>Sequential</th>
<th>Gray</th>
<th>Johnson</th>
<th>One-Hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>00000000</td>
<td>000000000000001</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0001</td>
<td>00000001</td>
<td>000000000000010</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0011</td>
<td>00000011</td>
<td>000000000000100</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0010</td>
<td>00000111</td>
<td>000000000010000</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0110</td>
<td>00011111</td>
<td>000000000100000</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>0111</td>
<td>00111111</td>
<td>000000000100000</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>0101</td>
<td>01111111</td>
<td>000000010000000</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>0100</td>
<td>01111111</td>
<td>000000010000000</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1100</td>
<td>11111111</td>
<td>000000010000000</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1101</td>
<td>11111110</td>
<td>000001000000000</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>1111</td>
<td>11111110</td>
<td>000001000000000</td>
</tr>
<tr>
<td>11</td>
<td>1011</td>
<td>1110</td>
<td>11111000</td>
<td>000100000000000</td>
</tr>
<tr>
<td>12</td>
<td>1100</td>
<td>1010</td>
<td>11110000</td>
<td>000100000000000</td>
</tr>
<tr>
<td>13</td>
<td>1101</td>
<td>1011</td>
<td>11100000</td>
<td>001000000000000</td>
</tr>
<tr>
<td>14</td>
<td>1110</td>
<td>1001</td>
<td>11000000</td>
<td>010000000000000</td>
</tr>
<tr>
<td>15</td>
<td>1111</td>
<td>1000</td>
<td>10000000</td>
<td>100000000000000</td>
</tr>
</tbody>
</table>
Common State Encoding Styles

Binary (Highly Encoded) FSM

One-Hot FSM

IDLE

READ

B1

B2

B3

000

001

010

011

110

00001

10000

01000

00010

00100
module fsm_oc4_2
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE = 2'ba00,
    BUSY = 2'ba01,
    BWAIT = 2'b10,
    BFREE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    gnt = 1'b0;
    case (state)
      IDLE : if (req) next = BUSY;
        else next = IDLE;
      BUSY: begin
        gnt = 1'b1;
        if (!done) next = BUSY;
        else if (dly) next = BWAIT;
        else next = BFREE;
      end
      BWAIT: begin
        gnt = 1'b1;
        if (!dly) next = BFREE;
        else next = BWAIT;
      end
      BFREE: if (req) next = BUSY;
        else next = IDLE;
    endcase
  end
endmodule

Binary FSM
module fsm_co4_fc
(output reg gnt,
  input dly, done, req, clk, rst_n);

parameter [3:0] IDLE = 0,
  BUSY = 1,
  EWAIT = 2,
  BFREE = 3;

reg [3:0] state, next;

always @(posedge clk or negedge rst_n)
  if (!rst_n) begin
    state <= 4'b0;
    state[IDLE] <= 1'b1;
  end
  else state <= next;

always @(state or dly or done or req) begin
  next = 4'b0;
  gnt = 1'b0;
  case (1'b1)
    // ambit synthesis case = full, parallel
    state[IDLE] : if (req) next[BUSY] = 1'b1;
                   else next[IDLE] = 1'b1;
    state[BUSY] : begin
      gnt = 1'b1;
      if (!done) next[BUSY] = 1'b1;
      else if (dly) next[EWAIT] = 1'b1;
      else next[BFREE] = 1'b1;
    end
    state[EWAIT] : begin
      gnt = 1'b1;
      if (!dly) next[BFREE] = 1'b1;
      else next[EWAIT] = 1'b1;
    end
    state[BFREE] : begin
      if (req) next[BUSY] = 1'b1;
      else next[IDLE] = 1'b1;
    end
  endcase
end
endmodule

Onehot FSM
Comparison of Binary and Onehot Style

- **Binary-encoded FSM**
  - fewer flip-flops for state register
  - \( = \log_2(\text{state number}) \)

- **Onehot-encoded FSM**
  - more flip-flops for state register
  - \( = \text{state number} \)

- FPGA vendor frequently recommend using onehot encoding style because flip-flops are plentiful in FPGA and the combinational logic cells required to implement is less for onehot style.

- i.e. Onehot style FSM usually runs faster than binary style FSM on FPGA
A Simple Design Example: Level-to-Pulse Converter

- A level-to-pulse converter produces a single-cycle pulse each time its input goes high.
  - In other words, it’s a synchronous rising edge detector.

- Sample application:
  - Button and switches (may need de-bounce processing)
  - Single-cycle enable signal for counters.

Diagram:
```
Whenever input L goes from low to high...

Level to Pulse Converter

CLK

P

...output P produces a single pulse, one clock period wide.
```
State Transition Diagram

- Block diagram of desired system

- State transition diagram is a useful FSM representation and design aid

![State Transition Diagram](image)

- "if L=1 at the clock edge, then jump to state 01."
- "if L=0 at the clock edge, then stay in state 00."
- Binary values of states
- "This is the output that results from this state. (Moore or Mealy?)"
Logic Derivation for a Moore FSM

- Transition diagram is readily converted to a state transition table (just a truth table)

<table>
<thead>
<tr>
<th>Current State</th>
<th>In</th>
<th>Next State</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 )</td>
<td>( S_0 )</td>
<td>( L )</td>
<td>( S_1^+ )</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>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Combinational logic may be derived through Karnaugh map

\[ S_{1}^+ = L S_{0} \]
\[ S_{0}^+ = L \]
\[ P = S_{1} S_{0} \]
Moore Level-to-Pulse Converter

Moore FSM circuit implementation of level-to-pulse converter:

\[ S_1^+ = L S_0 \]
\[ S_0^+ = L \]
\[ P = \overline{S_1 S_0} \]
Design of Mealy Level-to-Pulse Converter

- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations.

1. When L=1 and S=0, this output is asserted immediately and until the state transition occurs (or L changes).

2. After the transition to S=1 and as long as L remains at 1, this output is asserted.

Output transitions immediately. State transitions at the clock edge.
Mealy Level-to-Pulse Converter

Mealy FSM circuit implementation of level-to-pulse converter:

- FSM’s state simply remember the previous value of $L$
- Circuit benefits from Mealy FSM’s implicit single-cycle assertion of outputs during state transitions
Trade-Off: Mealy or Moore?

**Moore:** delayed assertion of P

- Mealy machine response earlier than Moore machine
- Compare to Moore FSM, Mealy FSM:
  - Be more difficult to conceptualize and design
  - Have fewer states
FSM Timing Requirements

Minimum Clock Period

Minimum Delay

\[ T > T_{\text{eq}} + T_{\text{logic}} + T_{\text{su}} \]

\[ T_{\text{eq,cd}} + T_{\text{logic,cd}} > T_{\text{hold}} \]
Example: Vending Machine

Design spec.

- All selections are 30 cents
- This machine makes change (dimes and nickels only)
- Input: 1 coin per clock
  - Q: quarter (25 cents)
  - D: dime (10 cents)
  - N: nickel (5 cents)
- Output: 1 can or 1 coin per clock
  - DC: dispense can
  - DD: dispense dime
  - DN: dispense nickel
State Definition

- A starting (idle) state:
  - idle

- A state for each possible amount of money captured:
  - got5c
  - got10c
  - got15c
  - ... (continued)

- What’s the maximum amount of money captured before purchase?
  - 25 cents (just shy of a purchase) + one quarter (largest coin)
  - ... got35c
got50c

- States to dispense change (one per coin dispensed):
  - got45c
  - Dispense Dime
  - Dispense Nickel
State Reduction

Duplicate states have:
- The same outputs, and
- The same transitions

There are two duplicates in our original diagram.

17 states
5 state bits

15 states
4 state bits
Verilog Coding for Moore Vender

- State register
  - Sequential always block
- Next-state combinational logic
  - combinational always block with case
- Output combinational logic
  - combinational always block with case
  - assignment (simple output)

```verilog
module mooreVender (N, D, Q, DC, DN, DD, 
                   clk, reset, state); 
  input N, D, Q, clk, reset; 
  output DC, DN, DD; 
  output [3:0] state; 
  reg [3:0] state, next;

  parameter IDLE = 0; 
  parameter GOT_5c = 1; 
  parameter GOT_10c = 2; 
  parameter GOT_15c = 3; 
  parameter GOT_20c = 4; 
  parameter GOT_25c = 5; 
  parameter GOT_30c = 6; 
  parameter GOT_35c = 7; 
  parameter GOT_40c = 8; 
  parameter GOT_45c = 9; 
  parameter GOT_50c = 10; 
  parameter RETURN_20c = 11; 
  parameter RETURN_15c = 12; 
  parameter RETURN_10c = 13; 
  parameter RETURN_5c = 14; 

  always @ (posedge clk or negedge reset) 
  if (!reset) state <= IDLE; 
  else state <= next;
```

Design of Datapath Controllers and Sequential Logic - 2009.3.18
Chihhao Chao
pp. 55
Next-state logic within a combinational always block

always @ (state or N or D or Q) begin

    case (state)
        IDLE: if (Q) next = GOT_25c;
            else if (D) next = GOT_10c;
            else if (N) next = GOT_5c;
            else next = IDLE;
        GOT_5c: if (Q) next = GOT_30c;
                 else if (D) next = GOT_15c;
                 else if (N) next = GOT_10c;
                 else next = GOT_5c;
        GOT_10c: if (Q) next = GOT_35c;
                   else if (D) next = GOT_20c;
                   else if (N) next = GOT_15c;
                   else next = GOT_10c;
        GOT_15c: if (Q) next = GOT_40c;
                   else if (D) next = GOT_25c;
                   else if (N) next = GOT_20c;
                   else next = GOT_15c;
        GOT_20c: if (Q) next = GOT_45c;
                   else if (D) next = GOT_30c;
                   else if (N) next = GOT_25c;
                   else next = GOT_20c;
        GOT_25c: if (Q) next = GOT_50c;
                   else if (D) next = GOT_35c;
                   else if (N) next = GOT_30c;
                   else next = GOT_25c;
        GOT_30c: next = IDLE;
        GOT_35c: next = RETURN_5c;
        GOT_40c: next = RETURN_10c;
        GOT_45c: next = RETURN_15c;
        GOT_50c: next = RETURN_20c;
        RETURN_20c: next = RETURN_10c;
        RETURN_15c: next = RETURN_5c;
        RETURN_10c: next = IDLE;
        RETURN_5c: next = IDLE;
        default: next = IDLE;
    endcase
end

Combinational output assignment

assign DC = (state == GOT_30c || state == GOT_35c ||
            state == GOT_40c || state == GOT_45c ||
            state == GOT_50c);
assign DN = (state == RETURN_5c);
assign DD = (state == RETURN_20c || state == RETURN_15c ||
            state == RETURN_10c);
FSM Output Glitching Problem

- FSM state bits may not transition at precisely the same time
- Combinational logic for outputs may contain hazard, i.e. result in glitch

during this state transition...

...the state registers may transition like this...

...causing the DC output to glitch like this!

assign DC = (state == GOT_30c || state == GOT_35c || state == GOT_40c || state == GOT_45c || state == GOT_50c);

If the soda dispenser is glitch-sensitive, your customers can get a 20-cent soda!
Registered FSM Outputs are Glitch-Free

- Move output generation into sequential always block
- Calculate outputs based on next state
- May increase FSM critical path

```vhdl
reg DC, DN, DD;

// Sequential always block for state assignment
always @ (posedge clk or negedge reset) begin
    if (!reset) state <= IDLE;
    else if (clk) state <= next;
    if (next == GOT_30c || next == GOT_35c ||
        next == GOT_40c || next == GOT_45c ||
        next == GOT_50c);
    DN <= (next == RETURN_15c);
    DD <= (next == RETURN_20c || next == RETURN_15c ||
        next == RETURN_10c);
end
```