Intro to Digital Design
Finite State Machines

Instructor: Justin Hsia

Teaching Assistants:
Emilio Alcantara     Eujean Lee
Naoto Uemura        Pedro Amarante
Wen Li
Relevant Course Information

❖ Quiz 1 grades should be out on Gradescope tonight
  ▪ Both the quiz and solutions will be added to the question bank on the course website

❖ Lab 5 – Verilog implementation of FSMs
  ▪ Step up in difficulty from Labs 1-4 (worth 100 points)
  ▪ Bonus points for minimal logic  \[ 110 \text{ max possible} \]
    • Simplification through design (Verilog does the rest)
Review of Timing Terms

- **Clock**: steady square wave that synchronizes system

- **Flip-flop**: one bit of state that samples every rising edge of CLK (positive edge-triggered)

- **Register**: several bits of state that samples on rising edge of CLK (positive edge-triggered); often has a RESET

- **Setup Time**: when input must be stable *before* CLK trigger

- **Hold Time**: when input must be stable *after* CLK trigger

- **CLK-to-Q Delay**: how long it takes output to change from CLK trigger
SDS Timing Question (all times in ns)

- The circuit below has the following timing parameters:
  - $t_{\text{period}} = 20$, $t_{\text{setup}} = 2$
  - $t_{\text{XOR}} = t_{\text{OR}} = 5$, $t_{\text{NOT}} = 4$
  - Input changes 1 ns after clock trigger

- What is the max $t_{C2Q}$?
  \[
  t_{C2Q} + t_{\text{NOT}} + t_{\text{XOR}} + t_{\text{OR}} \leq t_{\text{period}} - t_{\text{setup}}
  \]
  \[
  \frac{4 + 5 + 5}{20 - 2} \leq 4 \text{ ns}
  \]

- If $t_{C2Q} = 3$, what is the max $t_{\text{hold}}$?
  \[
  1 \text{ ns} + \frac{t_{\text{XOR}}}{5} \geq t_{\text{hold}}
  \]
  \[
  t_{\text{hold}} \leq 6 \text{ ns}
  \]
Outline

❖ Flip-Flop Realities
❖ Finite State Machines
❖ FSMs in Verilog
Flip-Flop Realities: Gating the Clock

- Delay can cause part of circuit to get out of sync with rest
  - More timing headaches!
  - Adds to *clock skew*

- Hard to track non-uniform triggers

- NEVER GATE THE CLOCK!!!
Flip-Flop Realities: External Inputs

- External inputs aren’t synchronized to the clock
  - If not careful, can violate timing constraints

- What happens if input changes around clock trigger?

```
D ___ \sim same\ time! \\
\downarrow \\
Clk ___ \\
Q ___ unknown\ duration
```

"Metastability"

could/will violate setup or hold constraint
Flip-Flop Realities: Metastability

Metastability is the ability of a digital system to persist for an unbounded time in an unstable equilibrium or metastable state

- Circuit may be unable to settle into a stable '0' or '1' logic level within the time required for proper circuit operation
- Unpredictable behavior or random value
- [https://en.wikipedia.org/wiki/Metastability_in_electronics](https://en.wikipedia.org/wiki/Metastability_in_electronics)

State elements can help reject transients
- Longer chains = more rejection, but longer signal delay
Outline

- Flip-Flop Realities
- **Finite State Machines**
- FSMs in Verilog
Finite State Machines (FSMs)

❖ A convenient way to conceptualize computation over time
  ▪ Function can be represented with a state transition diagram
  ▪ You’ve seen these before in CSE311

❖ New for CSE369: Implement FSMs in hardware as synchronous digital systems
  ▪ Flip-flops/registers hold “state”
  ▪ Controller (state update, I/O) implemented in combinational logic
State Diagrams

- An state diagram (in this class) is defined by:
  - A set of *states* \( S \) (circles)
  - An *initial state* \( s_0 \) (only arrow not between states)
  - A *transition function* that maps from the current input and current state to the output and the next state (arrows between states)

  - **Note:** We cover Mealy machines here; Moore machines put outputs on states, not transitions

- State transitions are controlled by the clock:
  - On each clock cycle the machine checks the inputs and generates a new state (could be same) and new output
Example: Buggy 3 Ones FSM

- FSM to detect 3 consecutive 1’s in the Input

States: S0, S1, S2
Initial State: S0
Transitions of form: input/output

```
INPUT 0 1 0 1 0 1 0 1 0 1 0 1 0
OUTPUT 1 0 1 0 1 0 1 0 1 0 1 0
```
Hardware Implementation of FSM

- Register holds a representation of the FSM’s state
  - Must assign a *unique* bit pattern for each state
  - Output is *present/current state* (PS/CS)
  - Input is *next state* (NS)

- Combinational Logic implements transition function (state transitions + output)

![Diagram showing the interaction between a register, combinational logic, and output]

On clock trigger, old NS becomes PS
FSM: Combinational Logic

- Read off transitions into Truth Table!
  - **Inputs:** Present State (PS) and Input (In)
  - **Outputs:** Next State (NS) and Output (Out)

- Implement logic for *EACH* output (2 for NS, 1 for Out)
# FSM: Logic Simplification

<table>
<thead>
<tr>
<th>PS</th>
<th>In</th>
<th>NS</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>00</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>XX</td>
<td>X</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>XX</td>
<td>X</td>
</tr>
</tbody>
</table>

FSM diagrams showing state transitions and outputs for different inputs.
FSM: Implementation

- $\text{NS}_1 = \text{PS}_0 \cdot \text{In}$
- $\text{NS}_0 = \overline{\text{PS}_1} \cdot \overline{\text{PS}_0} \cdot \text{In}$
- $\text{Out} = \text{PS}_1 \cdot \text{In}$

- How do we test the FSM?
  - “Take” every transition that we care about!

![Diagram of FSM implementation]
State Diagram Properties

- For $S$ states, how many state bits do I use?
  \[ s = \log_2 S \]

- For $I$ inputs, what is the maximum number of transition arrows on the state diagram?
  \[ S \times 2^I \]
  - Can sometimes combine transition arrows:
  - Can sometimes omit transitions (don’t cares)

- For $s$ state bits and $I$ inputs, how big is the truth table?
  \[ 2^{I+s} \]
Vending Machine Example

- Vending machine description/behavior:
  - Single coin slot for dimes and nickels
  - Releases gumball after ≥ 10 cents deposited
  - Gives no change

- State Diagram:
Vending Machine State Table

<table>
<thead>
<tr>
<th>PS</th>
<th>N</th>
<th>D</th>
<th>NS</th>
<th>Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<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>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

\[ NS = \overline{PS} \cdot N + PS \cdot \overline{N} \cdot D \]

\[ Open = D + PS \cdot N \]
Vending Machine Implementation

- Open = D + PS \cdot N
- NS = \overline{PS} \cdot N + PS \cdot \overline{N} \cdot \overline{D}
Outline

❖ Flip-Flop Realities
❖ Finite State Machines
❖ FSMs in Verilog
FSMs in Verilog: Overview

- FSMs follow a very particular organizational structure:

![Diagram of FSM]

- They can be implemented using the following design pattern:
  1. Define states and state variables
  2. Next state logic (ns)
  3. Output logic
  4. State update logic (ps)
FSMs in Verilog: Example

- Arbitrary 3-state FSM that outputs 1 when two consecutive 1’s are seen on the input
  - 2-bit state `ps`
  - `clk` and `reset` inputs
  - 1-bit input `w`
  - 1-bit output `out`

![Finite State Machine Diagram](image)
FSMs in Verilog: (0) Module Outline

module simpleFSM (clk, reset, w, out);
    input  logic clk, reset, w;
    output logic out;
endmodule  // simpleFSM
FSMs in Verilog: (1) State Declarations

```verilog
module simpleFSM (clk, reset, w, out);
    input logic clk, reset, w;
    output logic out;

    // State Encodings and variables
    // (ps = Present State, ns = Next State)
    enum logic [1:0] {S0=2’b00, S1=2’b01, S11=2’b10} ps, ns;

endmodule // simpleFSM
```
FSMs in Verilog: (2) Next State Logic

```verilog
module simpleFSM (clk, reset, w, out);
    ... // (0) port declarations
    ... // (1) state encodings and variables

    // Next State Logic (ns)
    always_comb
        case (ps)
            S0: if (w) ns = S1;
                else ns = S0;
            S1: if (w) ns = S11;
                else ns = S0;
            S11: if (w) ns = S11;
                else ns = S0;
        endcase
endmodule // simpleFSM
```
FSMs in Verilog: (2) Output Logic

module simpleFSM (clk, reset, w, out);
  ... // (0) port declarations
  ... // (1) state encodings and variables
  ... // (2) Next State Logic (ns)

  // Output Logic - could have been in "always" block
  // or part of Next State Logic.
  assign out = (ns == S11);

endmodule // simpleFSM
FSMs in Verilog: (2) Output Logic

```verilog
module simpleFSM (clk, reset, w, out);
...
// (0) port declarations
...
// (1) state encodings and variables
...
// (2) Next State Logic (ns)
...
// (3) Output Logic

// State Update Logic (ps)
always_ff @(posedge clk)
  if (reset)
    ps <= S0; // "initial" state
  else
    ps <= ns;
endmodule // simpleFSM
```
Reminder: Blocking vs. Non-blocking

- NEVER mix in one `always` block!
- Each variable written in only one `always` block

### Blocking (=) in CL:

```
// Output Logic
assign out = (ns == S11);

// Next State Logic (ns)
always_comb
  case (ps)
    S0: if (w) ns = S1;
    else   ns = S0;
    S1: if (w) ns = S11;
    else   ns = S0;
    S11: if (w) ns = S11;
    else   ns = S0;
  endcase
```

### Non-blocking (<=) in SL:

```
// State Update Logic (ps)
always_ff @(posedge clk) 
  if (reset)
    ps <= S0;
  else
    ps <= ns;
```
One or Two Blocks?

- We showed the state update in two separate blocks:
  - `always_comb` block that calculates the next state (ns)
  - `always_ff` block that defines the register (ps updates to last ns on clock trigger)

- Can this be done with a single block?
  - If so, which one: `always_comb` or `always_ff`
always_comb
  case (ps)
    S0: if (w) ns = S1;
        else ns = S0;
    S1: if (w) ns = S11;
        else ns = S0;
    S11: if (w) ns = S11;
        else ns = S0;
  endcase

always_ff @(posedge clk)
  if (reset)
    ps <= S0;
  else
    case (ps)
      S0: if (w) ps <= S1;
          else ps <= S0;
      S1: if (w) ps <= S11;
          else ps <= S0;
      S11: if (w) ps <= S11;
          else ps <= S0;
    endcase
FSMs Test Bench (1/2)

module simpleFSM_tb ();
    logic clk, reset, w, out;

    // instantiate device under test
    simpleFSM dut (.clk, .reset, .w, .out);

    // generate simulated clock
    parameter CLOCK_PERIOD=100;
    initial begin
        clk <= 0;
        forever #(CLOCK_PERIOD/2) clk <= ~clk;
    end

    // generate test vectors
    endmodule // simpleFSM_tb
// generate test vectors
initial begin
    reset <= 1; w <= 0; @(posedge clk); // reset
    reset <= 0;
    @(posedge clk); // 4 cycles of 0 input
    @ (posedge clk);
    @ (posedge clk);
    @ (posedge clk);
    w <= 1; @ (posedge clk);
    w <= 0; @ (posedge clk);
    w <= 1; @ (posedge clk); // 4 cycles of 1 input
    @ (posedge clk);
    @ (posedge clk);
    @ (posedge clk);
    w <= 0; @ (posedge clk);
    @ (posedge clk); // extra cycle
    $stop; // pause the simulation
end
endmodule // simpleFSM_tb
Testbench Waveforms

clock cycles: 0 1 2 3 4 5 6 7 8 9 10 11

 clk  reset  w  ps  ns  out
S0  S1  S0  S1  S11  S0

0/0 cycles 1-4
1/0 cycles 5,7
1/1 cycle 8

Reset cycle 0
0/0 cycle 11
0/0 cycles 9,10
Summary

❖ Gating the clock and external inputs can cause timing issues and metastability

❖ FSMs visualize state-based computations
  ▪ Implementations use registers for the state (PS) and combinational logic to compute the next state and output(s)
  ▪ Mealy machines have outputs based on state transitions

❖ FSMs in Verilog usually have separate blocks for state updates and CL
  ▪ Blocking assignments in CL, non-blocking assignments in SL
  ▪ Testbenches need to be carefully designed to test all state transitions