ECE 2300
Digital Logic & Computer Organization
Spring 2024

More Verilog
Finite State Machines
Announcements

• HW 2 due tomorrow
Exercise: Verilog Circuit Modeling

- Which of the following Verilog code snippets infer sequential logic

```verilog
reg q;
always @ (a, b)
begin
    q = a & b;
end
(a)
```

```verilog
reg q;
always @ (posedge clk)
begin
    q = a & b;
end
(b)
```

```verilog
input clk, d;
output q;
assign q = clk & d;
(c)
```

```verilog
reg q;
always @ (clk)
begin
    if ( clk ) // D latch?
        q = d;
end
(d)
```
Recap: Assignments in Verilog

• Continuous assignments apply to combinational logic only

• *Always blocks* contain a set of procedural assignments (blocking or nonblocking)
  – An always block can be used to model either combinational or sequential logic
    • Sequential logic can only be modeled using always blocks

  – Always blocks execute concurrently with other always blocks, instance statements, and continuous assignment statements in a module
Blocking Assignments

- **Blocking assignment: LHS = RHS**

```verilog
type[2]:
    input A, B;
    reg Y, Z;
    always @ (posedge clk)
    begin
        Y = A & B;
        Z = Y;
    end
```

When a reg ("Y" here) is assigned in a blocking assignment ("Y=A&B"), employ its input ("A&B") for connection in RHS of a subsequent assignment ("Z=Y")
Nonblocking Assignments

- **Nonblocking assignment: LHS <= RHS**

  ```vhdl
  input A, B;
  reg Y, Z;
  always @(posedge clk)
  begin
    Y <= A & B;
    Z <= Y;
  end
  ```

  Y and Z are inferred as FFs here, since the always block is sensitive to the clock edge

  - RHS evaluated *in parallel* (order doesn't matter)
  - Assignment to LHS is *delayed* until the end of the always block

**Simulation behavior**

\[ Z_{\text{next}} \leftarrow Y \quad \text{// use “old” Y} \]
\[ Y_{\text{next}} \leftarrow A \& B \]

**Synthesized circuit**

When a reg (“Y” here) is assigned in a *nonblocking assignment* (“Y=A&B”), employ its *output for connection* in RHS of another assignment (“Z=Y”)
Procedural Statements in Always Block

- Procedural statements are similar to conventional programming language statements
  - begin-end blocks
    - begin procedural-statement … procedural-statement end
  - if
    - if ( condition ) procedural-statement else procedural-statement
  - case
    - case ( sel-expression ) choice : procedural-statement … endcase
  - for
    - for(initial_assignment; expression; step_assignment) statement;
  - while
    - while ( logical-expression ) procedural-statement
  - repeat
    - repeat ( integer-expression ) procedural-statement

Mostly used in test bench
(simulation only, not synthesizable)
Conditionals

• Logical operators used in conditional expressions
  
  &&  logical AND
  ||   OR
  !    logical NOT
  ==   logical equality
  !=   logical inequality
  >    greater than
  >=   greater than or equal
  <    less than
  <=   less than or equal

• Don’t confuse with Boolean operators
Net and Variable Types

• **We will mainly use two data type classes**
  
  – *wire*: represents a physical connection (or net) between hardware elements
    
    • A stateless way of connected two elements
    • Can only be used to model combinational logic
    • *Cannot be used in the left-hand side in an always block*

  – *reg*: similar to wires, but can be used to store information (or state) like registers
    
    • This is used in the behavioral style only
    • *Can be used to model both combinational & sequential logic*
    • Cannot be used in the left-hand side of a continuous assignment statement
(Improperly Created) **Inferred Latches**

- To infer combinational logic, you’re recommended to ensure that each variable within an always block gets assigned a value under *all possible conditions*
  - Otherwise, the Verilog compiler assumes that the last value should be used, and will create a latch

```verilog
reg out;
always @(d, sel)
begin
  if (sel == 1'b1)
    out = d;
  else
    out = ~d;
end
```

- *out* is keeping state because it’s not always assigned a value ➔ **latch inferred**

- *out* assigned a value in both conditions ➔ **combinational logic inferred** (not latch)
Finite State Machine

- A Finite State Machine (FSM) is an abstract representation of a sequential circuit
  - The state embodies the condition of the system at this particular time
  - The combinational logic determines the output and next state values
  - The output values may depend only on the current state value (Moore), or on the current state and input values (Mealy)
Elements of an FSM

1. A finite number of inputs
2. A finite number of outputs
3. A finite number of states
4. A specification of all state transitions

Can be described by a state diagram
FSM: General Form

- Inputs and current state determine state transitions
- Output changes determined by changes in
  - Current state, or
  - Current state + inputs
Moore Machine

Inputs

Outputs only depend on current state value

Lecture 8: 14
Mealy Machine

Outputs only depend on input and current state value
State Diagram

- Visual specification of an FSM
  - Bubble for every state
  - Arcs showing state transitions
  - Input values shown on the arcs
  - Output values shown within the bubbles (Moore) or on the arcs (Mealy)
  - Clock input implicit (always present, triggering state transitions)

Moore FSM Example

Mealy FSM Example
Moore State Diagram

- 1 input, 1 output, 3 states
- Bubble for each state
- State transitions (arcs) for each input value
- Input values on the arcs
- Output values within the bubbles
- Starts at S0 when Reset asserted
Lecture 8:

Mealy State Diagram

- 1 input, 1 output, 2 states
- Bubble for each state
- State transitions (arcs) for each input value
- Input values on the arcs (first number)
- Output values on the arcs (second number)
- Starts at S0 when Reset asserted

Mealy FSM Example
FSM Design Procedure

(1) Understand the problem statement and determine inputs and outputs

(2) Identify states and create a state diagram

(3) Determine the number of required FFs

(4) Implement combinational logic for outputs and next state

(5) Simulate the circuit to test its operation
Example FSM: Pattern Detector

- Monitors the input, and outputs a 1 whenever a specified input pattern is detected

- Example: Output a 1 whenever 111 is detected on the input over 3 consecutive clock cycles
  - Overlapping patterns also detected (1111...)

- Input $In$ (one bit)
- Output $Out$ (one bit)
- $Reset$ causes FSM to start in initial state
- $Clock$ input not shown (always present)
111 Pattern Detector: Moore State Diagram

- Output a 1 whenever 111 is detected on the input over 3 consecutive clock cycles (overlapping pattern also detected)
111 Pattern Detector: Mealy State Diagram

- Output a 1 whenever 111 is detected on the input over 3 consecutive clock cycles (overlapping pattern also detected)
Example FSM: Pushbutton Lock

- Two pushbutton inputs, X1 and X2
- One output, UL (“Unlock”)
- UL = 1: the lock is unlocked, when X1 is pushed, followed by X2 being pushed twice (X1, X2, X2)
- Represent X1 and X2 as two-bit input
  - 00: neither button pushed
  - 10: X1 pushed
  - 01: X2 pushed
  - 11: both pushed, reset the lock (to the locked state)
Pushbutton Lock: Moore State Diagram

- Output: UL=1 with (X1, X2, X2)
- Input: 00 (neither), 10 (X1), 01 (X2), 11 (reset)
Pushbutton Lock: Mealy State Diagram

- Output: UL=1 with (X1, X2, X2)
- Input: 00 (neither), 10 (X1), 01 (X2), 11 (reset)
Before Next Class

• H&H 4.9

Next Time

More Finite State Machines