I. Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.
- **Blocking assignment:** evaluation and assignment are immediate
  ```verilog
  always @ (a or b or c)
  begin
    x = a | b;
    y = a ^ b ^ c;
    z = b & ~c;
  end
  ```
  - 1. Evaluate `a | b`, assign result to `x`
  - 2. Evaluate `a^b^c`, assign result to `y`
  - 3. Evaluate `b&(~c)`, assign result to `z`

- **Nonblocking assignment:** all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)
  ```verilog
  always @ (a or b or c)
  begin
    x <= a | b;
    y <= a ^ b ^ c;
    z <= b & ~c;
  end
  ```
  - 1. Evaluate `a | b` but defer assignment of `x`
  - 2. Evaluate `a^b^c` but defer assignment of `y`
  - 3. Evaluate `b&(~c)` but defer assignment of `z`
  - 4. Assign `x`, `y`, and `z` with their new values

- Sometimes, as above, both produce the same result. Sometimes, not!
Why two ways of assigning values?

Conceptual need for two kinds of assignment (in always blocks):

<table>
<thead>
<tr>
<th>Blocking: Evaluation and assignment are immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
</tr>
<tr>
<td>b = a</td>
</tr>
<tr>
<td>x = a &amp; b</td>
</tr>
<tr>
<td>y = x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-Blocking: Assignment is postponed until all r.h.s. evaluations are done</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt;= b</td>
</tr>
<tr>
<td>b &lt;= a</td>
</tr>
<tr>
<td>x &lt;= a &amp; b</td>
</tr>
<tr>
<td>y &lt;= x</td>
</tr>
</tbody>
</table>

When to use: (only in always blocks!)

<table>
<thead>
<tr>
<th>Sequential Circuits</th>
<th>Combinational Circuits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.111 Fall 2007
Assignment Styles for Sequential Logic

- Will nonblocking and blocking assignments both produce the desired result?

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

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

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.”

• Blocking assignments do not reflect the intrinsic behavior of multi-stage sequential logic

• **Guideline:** use **nonblocking** assignments for sequential *always* blocks
### Use Blocking for Combinational Logic

#### Blocking Behavior

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

- **Nonblocking assignments do not reflect the intrinsic behavior of multi-stage combinational logic**
- **While nonblocking assignments can be hacked to simulate correctly (expand the sensitivity list), it’s not elegant**

**Guideline:** use **blocking assignments for combinational always blocks**

#### Nonblocking Behavior

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

- **always @ (a or b or c)**
  ```
  begin
  x = a & b;
  y = x | c;
  end
  ```

- **Deferred**

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

- **always @ (a or b or c)**
  ```
  begin
  x <= a & b;
  y <= x | c;
  end
  ```
II. Single-clock Synchronous Circuits

We’ll use Flip Flops and Registers – groups of FFs sharing a clock input – in a highly constrained way to build digital systems.

Single-clock Synchronous Discipline:

- No combinational cycles
- Single clock signal shared among all clocked devices
- Only care about value of combinational circuits just before rising edge of clock
- Period greater than every combinational delay
- Change saved state after noise-inducing logic transitions have stopped!
Clocked circuit for on/off button

module onoff(clk, button, light);
    input clk, button;
    output light;
    reg light;
    always @ (posedge clk)
    begin
        if (button) light <= ~light;
    end
endmodule
Asynchronous Inputs in Sequential Systems

What about external signals?

Sequential System

Can’t guarantee setup and hold times will be met!

When an asynchronous signal causes a setup/hold violation...

<table>
<thead>
<tr>
<th>Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Transition is missed on first clock cycle, but caught on next clock cycle.</td>
</tr>
<tr>
<td>II</td>
<td>Transition is caught on first clock cycle.</td>
</tr>
<tr>
<td>III</td>
<td>Output is metastable for an indeterminate amount of time.</td>
</tr>
</tbody>
</table>

Q: Which cases are problematic?
Asynchronous Inputs in Sequential Systems

All of them can be, if more than one happens simultaneously within the same circuit.

Idea: ensure that external signals directly feed exactly one flip-flop

This prevents the possibility of I and II occurring in different places in the circuit, but what about metastability?
Handling Metastability

• Preventing metastability turns out to be an impossible problem
• High gain of digital devices makes it likely that metastable conditions will resolve themselves quickly
• Solution to metastability: allow time for signals to stabilize

How many registers are necessary?
• Depends on many design parameters (clock speed, device speeds, …)
• In 6.111, a pair of synchronization registers is sufficient
III. Finite State Machines

- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized “states” of operation.
- At each clock edge, combinational logic computes outputs and next state as a function of inputs and present state.
Example 1: Light Switch

- State transition diagram

- Logic diagram

![State transition diagram for the light switch example.](image-url)

![Logic diagram for the light switch example.](image-url)
Example 2: 4-bit Counter

- Logic diagram

- Verilog

```verilog
# 4-bit counter
module counter(clk, count);
    input clk;
    output [3:0] count;
    reg [3:0] count;

    always @ (posedge clk) begin
        count <= count + 1;
    end
endmodule
```
Example 2: 4-bit Counter

- Logic diagram

- Verilog

```
# 4-bit counter with enable
module counter(clk,enb,count);
    input clk,enb;
    output [3:0] count;
    reg [3:0] count;

    always @(posedge clk) begin
        count <= enb ? count+1 : count;
    end
endmodule
```

Could I use the following instead?
```
if (enb) count <= count+1;
```
Example 2: 4-bit Counter

- Logic diagram

- Verilog

```verilog
# 4-bit counter with enable and synchronous clear
module counter(clk, enb, clr, count);
    input clk, enb, clr;
    output [3:0] count;
    reg [3:0] count;

    always @(posedge clk) begin
        count <= clr ? 4'b0 : (enb ? count+1 : count);
    end
endmodule

Isn't this a lot like Exercise 1 in Lab 2?
```
Two Types of FSMs

Moore and Mealy FSMs: different output generation

- **Moore FSM:**

  - Inputs: $x_0 \ldots x_n$
  - Comb. Logic
  - Flip-Flops
  - Outputs: $y_k = f_k(S)$

- **Mealy FSM:**

  - Inputs: $x_0 \ldots x_n$
  - Comb. Logic
  - Flip-Flops
  - Outputs: $y_k = f_k(S, x_0 \ldots x_n)$

- Direct combinational path!
Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single-cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters

Whenever input $L$ goes from low to high...

...output $P$ produces a single pulse, one clock period wide.
Step 1: State Transition Diagram

- Block diagram of desired system:

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

```
if L=1 at the clock edge, then jump to state 01.

if L=0 at the clock edge, then stay in state 00.
```

This is the output that results from this state. *(Moore or Mealy?)*
Step 2: Logic Derivation

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) (S_0)</td>
<td>(L)</td>
<td>(S_1^+) (S_0^+)</td>
<td>(P)</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0</td>
<td>1</td>
<td>0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 1</td>
<td>0</td>
<td>0 0</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>1</td>
<td>1 1</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>0</td>
<td>0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1</td>
<td>1</td>
<td>1 1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Combinational logic may be derived using Karnaugh maps
Moore Level-to-Pulse Converter

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

Moore FSM circuit implementation of level-to-pulse converter:
Design of a Mealy Level-to-Pulse

- 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. While in state \( S=1 \) and as long as \( L \) remains at 1, this output is asserted.
Mealy Level-to-Pulse Converter

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

- FSM's state simply remembers the previous value of $L$
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions
Moore/Mealy Trade-Offs

- How are they different?
  - **Moore**: outputs = f(state) only
  - **Mealy**: outputs = f(state and input)
  - Mealy outputs generally occur one cycle earlier than a Moore:

  <Diagram>

  **Moore**: delayed assertion of P
  **Mealy**: immediate assertion of P

- Compared to a Moore FSM, a Mealy FSM might...
  - Be more difficult to conceptualize and design
  - Have fewer states
Light Switch Revisited

Level-to-Pulse FSM

Light Switch FSM