Continuous Assignments

review

• Continuously assigns right side of expression to left side.
• Limited to basic Boolean and ? operators. For example a 2:1 mux:
  – ? operator
    assign D = (A==1) ? B : C; // if A then D = B else D = C;

  – Boolean operators
    assign D = (B & A) | (C & ~A); // if A then D = B else D = C;
Procedural Assignments

- Executes a procedure allowing for more powerful constructs such as if-then-else and case statement.
- For example 2:1 mux:
  - if-then-else
    ```
    if (A) then D = B else D = C;
    ```
  - case
    ```
    case(A)
      1'b1 : D = B;
      1'b0 : D = C;
    endcase
    ```
This is obviously much easier to implement and read then Boolean expressions!!

Always Block

- An always block is an example of a procedure.
- The procedure executes a set of assignments when a defined set of inputs change.
2:1 mux Always Block

Module mux_2_1(a, b, out, sel);
input a, b, sel;
output out;
reg out;
always @(a or b or sel)
begin
if (sel) out = a;
else out = b;
end
endmodule

Declare Module and IO as before.
All data types in always blocks must be declared as a ‘reg’ type.
This is required even if the data type is for combinational logic.
The always block ‘executes’ whenever signals named in the sensitivity list change.
Literally: always execute at a or b or sel.
Sensitivity list should include conditional (sel) and right side (a, b) assignment variables.

As Easier Way to Implement the Sensitivity List

- Recent versions of Verilog provides a means to implement the sensitivity list without explicitly listing each potential variable.
- Instead of listing variables as in the previous example
  always @(a or b or sel)
Simply use
always @*
The * operator will automatically identify all sensitive variables.
Blocking vs Non-Blocking Assignments

- Blocking (=) and non-blocking (<=) assignments are provided to control the execution order within an always block.

- Blocking assignments **literally block** the execution of the next statement until the current statement is executed.
  - **Consequently, blocking assignments result in ordered statement execution.**

  For example:

  ```
  assume a = b = 0 initially;
  a = 1;       //executed first
  b = a;       //executed second
  then a = 1, b = 1 after ordered execution
  ```

Blocking vs Non-Blocking Cont

- Non-blocking assignments **literally do not block** the execution of the next statements. The right side of all statements are determined first, then the left sides are assigned together.
  - **Consequently, non-blocking assignments result in simultaneous or parallel statement execution.**

  For example:

  ```
  assume a = b = 0 initially;
  a <= 1;       //executed first
  b <= a;       //executed second
  then a = 1, b = 0 after parallel execution
  ```

  **Result is different from ordered exec!!! Does not preserve logic flow**
To Block or Not to Block?

- Ordered execution mimics the inherent logic flow of combinational logic.
- Hence blocking assignments generally work better for combinational logic.
- For example:

```
Module blocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;
  always @* begin
    x = a & b;
    y = x | c;
  end
endmodule
```

```
Module nonblocking(a,b,c,x,y);
  input a,b,c;
  output x,y;
  reg x,y;
  always @* begin
    x <= a & b;
    y <= x | c;
  end
endmodule
```

non-blocking behavior does not preserve logic flow!!
Sequential Logic

- Can be generalized as a series of combinational blocks with registers to hold results.

```
+-------+    +-------+    +-------+    +-------+
| inputs|  -->| register|  -->| comb logic|  -->| register|  -->| more stages|
+-------+    +-------+    +-------+    +-------+
                 | clk                  |
```

Results of each stage are stored (latched) in a register (D-Flip-Flops) by a common clock.

Sequential Example

- Shift registers are used to implement multiplication/division and other functions.
- Consider a simple 2-bit “right” shift:

```
<table>
<thead>
<tr>
<th>clk</th>
<th>D0</th>
<th>Q0</th>
<th>Q1</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rising edge</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>rising edge</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

common clock
No combinational logic for simple shift, just simple pass thru
Sequential Example Cont 1

- Notice that the inputs of each stage are “evaluated” then latched into the registers at each rising clock edge.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</td>
<td>8'0</td>
</tr>
<tr>
<td>d0</td>
<td>8'0</td>
</tr>
<tr>
<td>d1</td>
<td>8'0</td>
</tr>
<tr>
<td>d0</td>
<td>8'0</td>
</tr>
</tbody>
</table>

Because the shift logic evaluates inputs in parallel and latches result on a rising clk edge, a **non-blocking always** procedure sensitive to a rising clock can be used to implement.

```verilog
module shift3x(clk, d0, d1);
input clk, d0;
output d1;
reg d0, d1;
always @ (posedge clk)
begin
    d1 <= d0;
end
endmodule
```

Sensitivity to rising clock edge. Note that in this case we must explicitly specify sensitivity to the rising edge of clock. Simply using the * will not work.
Sequential Example Cont 3

- What if we used blocking statements instead.
  Notice the following results:

```verilog
module shiftRegister (D0, Q0, Q1);

input clk, D0;
output Q0, Q1;
reg Q0, Q1;
always @(posedge clk)
begin
  Q0 <= D0;
  Q1 <= Q0;
end
endmodule
```

The logic statements are simplified to $Q_0 = Q_1 = D_0$. Logic is evaluated on rising edge of clk. Verilog is synthesized as one stage logic.

Summary

- **Combinational logic**: Use **blocking statements** with always blocks with the `*` operator to mimic logic flow of combinational logic.

- **Sequential logic**: Use **non-blocking statements** with always blocks sensitive to rising clock edge to mimic parallel sequential logic.
Modeling Finite State Machines with Verilog

- Finite State Machines can be modeled with three general functions:
  1. Next State Logic (combinational)
     Combinational logic that determines next state based on current state and inputs.
  2. State Register (sequential)
     Sequential logic that holds the value of the current state.
  3. Output Logic (combinational)
     Combinational logic that sets the output based on current state.

FSM Example: A Simple Arbiter

- Four inputs: reset, clock req_0 and req_1.
- Two outputs: gnt_0 and gnt_1.
- When req_0 is asserted and req_1 is not asserted, gnt_0 is asserted
- When req_1 is asserted and req_0 is not asserted, gnt_1 is asserted
- When both req_0 and req_1 are asserted then gnt_0 is asserted; in other words, priority is given to req_0 over req_1.
Modeling the Arbiter in Verilog

- Identify the combinational and sequential components.
- Express each component as a combinational or sequential always block.

Arbiter Next State Always Block

Use this combinational always block to implement state transition logic with case and if-then-else constructs.
• The state register will be loaded with next_state from the next state logic on the rising edge of clock.

• Reset will set the state register to IDLE state on RESET and the rising edge of clock.

Arbiter Output Always Block

use a combinational always block to implement logic output based on state.
Integrate Into One Module

Module and IO Declaration

Next State Always Block

State Register Always Block

Output Always Block