# Advanced Verilog

EECS 270 v10/23/06

# 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**.

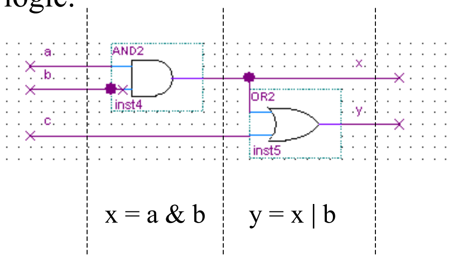
For example:

```
assume a = b = 0 initially;
a <= 1;⎫
       ⎬ Execute  together (in parallel)
b <= a;⎭
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:



x = a & b

y = x | b

logic flow

# To Block or Not to Block ? cont

```
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
```

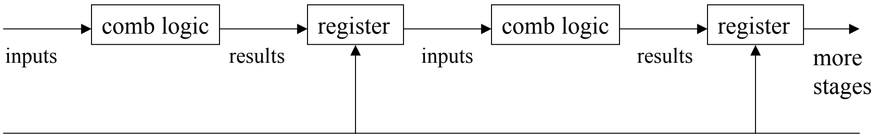
| Blocking behavior | a | b | c | x | y |
|---|---|---|---|---|---|
| Initial values | 1 | 1 | 0 | 1 | 1 |
| a changes→always block execs | 0 | 1 | 0 | 1 | 1 |
| x = a & b; //make assignment | 0 | 1 | 0 | 0 | 1 |
| y = x \| c; //make assignment | 0 | 1 | 0 | 0 | 0 |

| Non-blocking behavior | a | b | c | x | y |
|---|---|---|---|---|---|
| Initial values | 1 | 1 | 0 | 1 | 1 |
| a changes→always block execs | 0 | 1 | 0 | 1 | 1 |
| x = a & b; | 0 | 1 | 0 | 1 | 1 |
| y = x \| c; //x not passed from here | 0 | 1 | 0 | 1 | 1 |
| make x, y assignments | 0 | 1 | 0 | 0 | 1 |

non-blocking behavior does not preserve logic flow!!

# Sequential Logic

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

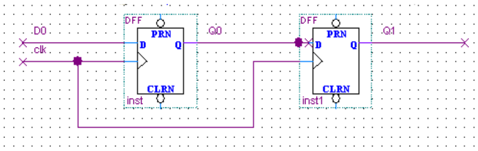


clk

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:



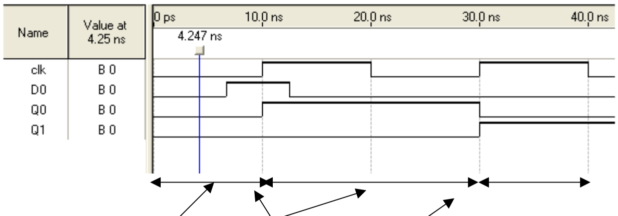
common clock

No combinational logic for simple shift, just simple pass thru

| clk | D0 | Q0 | Q1 |
|---|---|---|---|
| initial | 1 | 0 | 0 |
| rising edge | 1 | 1 | 0 |
| rising edge | 0 | 0 | 1 |

# Sequential Example Cont 1

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

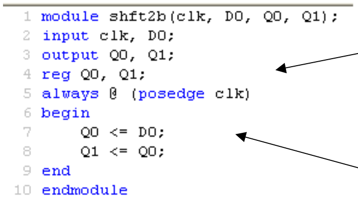


inputs evaluated between rising clk edges

results latched on rising clk edge

# Sequential Example Cont 2

- 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.



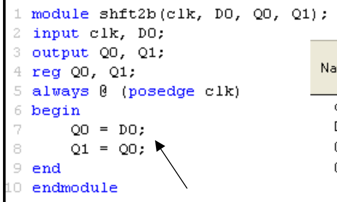
```
module shft2b(clk, D0, Q0, Q1);
input clk, D0;
output Q0, Q1;
reg Q0, Q1;
always @ (posedge clk)
begin
    Q0 <= D0;
    Q1 <= Q0;
end
endmodule
```

Sensitive 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.

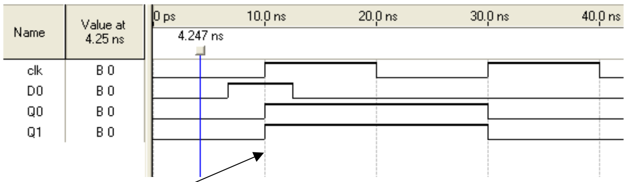
process statements in parallel

# Sequential Example Cont 3

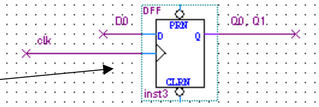
- What if we used blocking statements instead. Notice the following results:



```
module shft2b(clk, D0, Q0, Q1);
input clk, D0;
output Q0, Q1;
reg Q0, Q1;
always @ (posedge clk)
begin
     Q0 = D0;
     Q1 = Q0;
end
endmodule
```



| Name | Value at 4.25 ns |
|---|---|
| clk | B 0 |
| D0 | B 0 |
| Q0 | B 0 |
| Q1 | B 0 |



The logic statements are simplified to Q0 = Q1 = D0. 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)

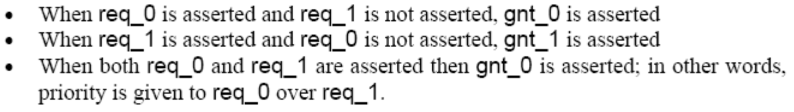
     Combinational logic that determines next state based on current state and inputs.
  2. State Register (sequential)

     Sequential logic that holds the value of the current state.
  3. Output Logic (combinational)

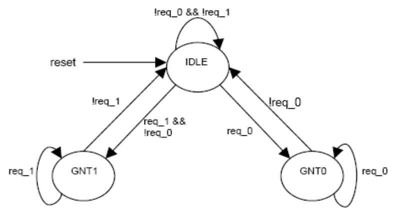
     Combinational logic that sets the output based on current state.

# FSM Example: A Simple Arbiter

- Four inputs: reset, clock req_0 and req_1.
- Two outputs: gnt_0 and gnt_1.

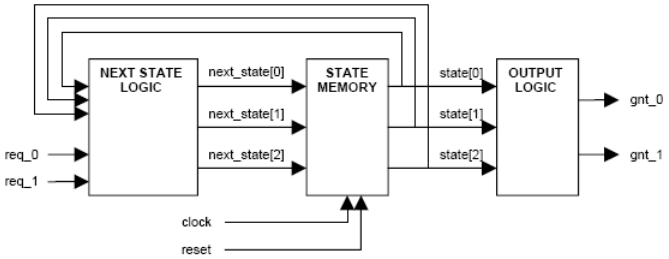


- When req_0 is asserted and req_1 is not asserted, gnt_0 is asserted
- When req_1 is asserted and req_0 is not asserted, gnt_1 is asserted
- When both req_0 and req_1 are asserted then gnt_0 is asserted; in other words, priority is given to req_0 over req_1.



# Modeling the Arbiter in Verilog

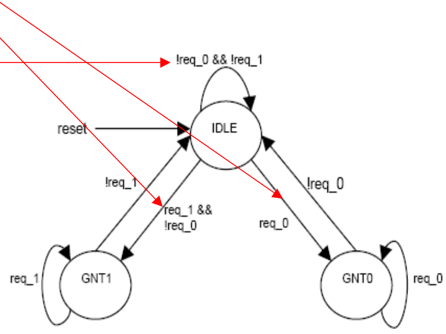
- Identify the combinational and sequential components.
- Express each component as a combinational or sequential always block.



# Arbiter Next State Always Block

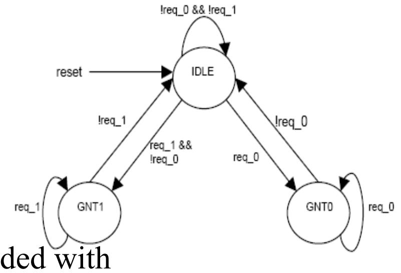
```
always @*
begin
  case(state)
  IDLE :      if (req_0 == 1'b1)
                   next_state = GNT0;
              else if (req_1 == 1'b1)
                   next_state= GNT1;
              else
                   next_state = IDLE;
  GNT0 :      if (req_0 == 1'b1)
                   next_state = GNT0;
              else
                   next_state = IDLE;
  GNT1 :      if (req_1 == 1'b1)
                   next_state = GNT1;
              else
                   next_state = IDLE;
  default:    next_state = IDLE;
  endcase
end
```

Use this combinational always block to implement state transition logic with case and if-then-else constructs



# Arbiter State Register Always Block

```
always @ (posedge clock)
begin
  if (reset == 1'b1)
      state <= IDLE;
  else
      state <= next_state;
end
```



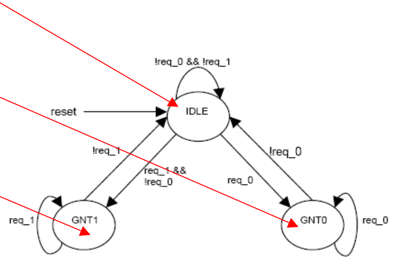
- The state register will be loaded with next_state from the next state logic on the rising edge of clock.

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

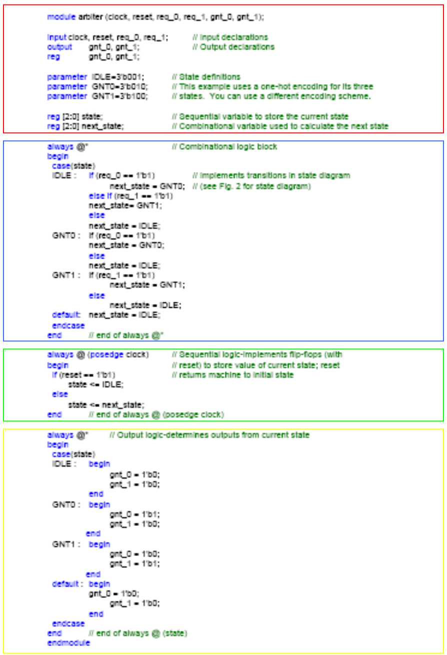
# Arbiter Output Always Block

```
always @*
begin
  case(state)
  IDLE :      begin
              gnt_0 = 1'b0;
              gnt_1 = 1'b0;
              end
  GNT0 :      begin
              gnt_0 = 1'b1;
              gnt_1 = 1'b0;
            end
  GNT1 :      begin
              gnt_0 = 1'b0;
              gnt_1 = 1'b1;
            end
  default :   begin
              gnt_0 = 1'b0;
              gnt_1 = 1'b0;
           end
  endcase
end
```

Use a combinational always block to implement logic output based on state.



## Integrate Into One Module



Module and IO Declaration

```
module arbiter (clock, reset, req_0, req_1, gnt_0, gnt_1);

input clock, reset, req_0, req_1;        // Input declarations
output      gnt_0, gnt_1;                // Output declarations
reg         gnt_0, gnt_1;

parameter  IDLE=3'b001;          // State definitions
parameter  GNT0=3'b010;          // This example uses a one-hot encoding for its three
parameter  GNT1=3'b100;          // states.  You can use a different encoding scheme.

reg [2:0] state;                 // Sequential variable to store the current state
reg [2:0] next_state;            // Combinational variable used to calculate the next state
```

Next State Always Block

```
always @*                        // Combinational logic block
begin
 case(state)
 IDLE :     if (req_0 == 1'b1)        // Implements transitions in state diagram
                 next_state = GNT0;   // (see Fig. 2 for state diagram)
            else if (req_1 == 1'b1)
            next_state= GNT1;
            else
            next_state = IDLE;
 GNT0 :     if (req_0 == 1'b1)
            next_state = GNT0;
            else
            next_state = IDLE;
 GNT1 :     if (req_1 == 1'b1)
                 next_state = GNT1;
            else
                 next_state = IDLE;
 default:   next_state = IDLE;
 endcase
end         // end of always @*
```

State Register Always Block

```
always @ (posedge clock)         // Sequential logic-implements flip-flops (with
begin                            // reset) to store value of current state; reset
 if (reset == 1'b1)              // returns machine to initial state
      state <= IDLE;
 else
      state <= next_state;
end         // end of always @ (posedge clock)
```

Output Always Block

```
always @*        // Output logic-determines outputs from current state
begin
 case(state)
 IDLE :     begin
                 gnt_0 = 1'b0;
                 gnt_1 = 1'b0;
            end
 GNT0 :     begin
                 gnt_0 = 1'b1;
                 gnt_1 = 1'b0;
           end
 GNT1 :     begin
                 gnt_0 = 1'b0;
                 gnt_1 = 1'b1;
           end
 default :  begin
            gnt_0 = 1'b0;
                 gnt_1 = 1'b0;
            end
 endcase
end         // end of always @ (state)
endmodule
```