

Verilog - Part II

Presentation Outline

- ❖ Modeling Latches and Flip-Flops
- ❖ Blocking versus Non-Blocking Assignments
- ❖ Modeling Sequential Circuit Diagrams
- ❖ Modeling Mealy and Moore State Diagrams
- ❖ Modeling Registers and Counters

Recall: Sensitivity List of always block

❖ Syntax:

```
always @(sensitivity list) begin
```

```
    procedural statements
```

```
end
```

❖ Sensitivity list is a list of signals: @(signal1, signal2, ...)

❖ The sensitivity list triggers the execution of the **always** block

When there is a ***change of value in any listed signal***

Otherwise, the **always** block does nothing until another change occurs on a signal in the sensitivity list

Guidelines for Sensitivity List

- ❖ For combinational logic, the sensitivity list must include **ALL** the signals that are read inside the **always** block

Combinational logic can also use: `@(*)` or `@*`

- ❖ For sequential logic, the sensitivity list may not include all the signals that are read inside the **always** block

- ❖ For **edge-triggered** sequential logic use:

always `@(posedge signal1, negedge signal2, ...)`

- ❖ The **positive edge** or **negative edge** of each signal can be specified in the sensitivity list

Modeling a D Latch with Enable

```
// Modeling a D Latch with Enable and output Q
// Output Q must be of type reg
// Notice that the if statement does NOT have else
// If Enable is 0, then value of Q does not change
// The D_latch stores the old value of Q

module D_latch (input D, Enable, output reg Q);
    always @(D, Enable)
        if (Enable) Q <= D; // Non-blocking assignment
endmodule
```

Modeling a D-type Flip-Flop

```
// Modeling a D Flip-Flop with outputs Q and Qbar
module D_FF (input D, Clk, output reg Q, Qbar);
    // Q and Qbar change at the positive edge of Clk
    // Notice that always is NOT sensitive to D
    always @(posedge Clk)
    begin
        Q <= D;           // Non-blocking assignment
        Qbar <= ~D;        // Non-blocking assignment
    end
endmodule
```

Negative-Edge Triggered D-type Flip-Flop

```
// Modeling a Negative-Edge Triggered D Flip-Flop
// The only difference is the negative edge of Clk
module D_FF2 (input D, Clk, output reg Q, Qbar);
    // Q and Qbar change at the negative edge of Clk
    always @(negedge Clk)
    begin
        Q <= D;           // Non-blocking assignment
        Qbar <= ~D;        // Non-blocking assignment
    end
endmodule
```

D-type Flip-Flop with Synchronous Reset

```
// Modeling a D Flip-Flop with Synchronous Reset input
module D_FF3 (input D, Clk, Reset, output reg Q, Qbar);
    // always block is NOT sensitive to Reset or D
    // Updates happen only at positive edge of Clk
    // Reset is Synchronized with Clk
    always @(posedge Clk)
        if (Reset)
            {Q, Qbar} <= 2'b01;
        else
            {Q, Qbar} <= {D, ~D};
endmodule
```


D-type Flip-Flop with Asynchronous Reset

```
// Modeling a D Flip-Flop with Asynchronous Reset input
module D_FF4 (input D, Clk, Reset, output reg Q, Qbar);
    // Q and Qbar change at the positive edge of Clk
    // Or, at the positive edge of Reset
    // Reset is NOT synchronized with Clk
    always @(posedge Clk, posedge Reset)
        if (Reset)
            {Q, Qbar} <= 2'b01;
        else
            {Q, Qbar} <= {D, ~D};
endmodule
```

JK flip-flop

```
module JK_FF (input J, K, Clk, output reg Q, output Q_b);  
    assign Bib = ~Q;  
    always @ (posedge Clk)  
        case ({J,K})  
            2'b00: Q <= Q;  
            2'b01: Q <= 1'b0;  
            2'b10: Q <= 1'b1;  
            2'b11: Q <= !Q;  
        encase;  
endmodule;
```

T flip-flop

```
module T_FF (Q,T,CLK,RST);  
    output Q;  
    input T,CLK,RST;  
    reg Q;  
    always @ (posedge CLK or negedge RST)  
        if (~RST) Q = 1'b0;  
        else Q = Q ^ T;  
endmodule
```

Procedural Assignment

- ❖ Procedural assignment is used inside a **procedural block** only
- ❖ Two types of procedural assignments:

- ❖ **Blocking assignment:**

variable = *expression*; // = operator

Variable is updated **before** executing next statement

Similar to an assignment statement in programming languages

- ❖ **Non-Blocking assignment:**

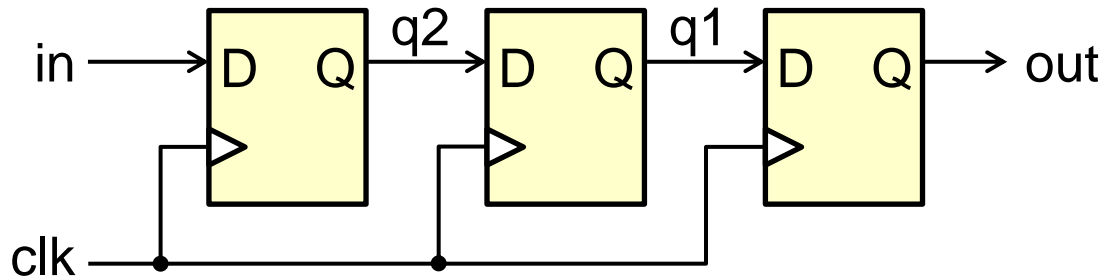
variable <= *expression*; // <= operator

Variable is updated **at the end** of the procedural block

Does not block the execution of next statements

Blocking versus Non-Blocking Assignment

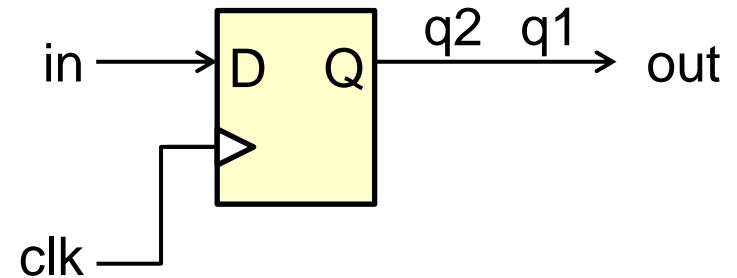
Guideline: Use Non-Blocking Assignment for Sequential Logic



```
module nonblocking
(input in, clk, output reg out);
  reg q1, q2;
  always @ (posedge clk) begin
    q2 <= in;
    q1 <= q2;
    out <= q1;
  end
endmodule
```

Read: in, q2, q1

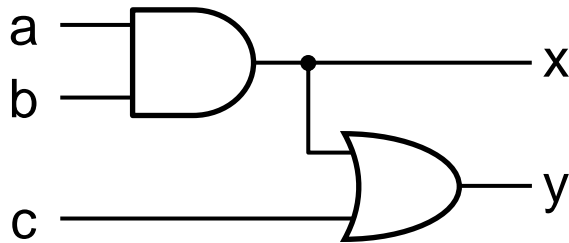
Parallel Assignment at the end



```
module blocking
(input in, clk, output reg out);
  reg q1, q2;
  always @ (posedge clk) begin
    q2 = in;
    q1 = q2; // q1 = in
    out = q1; // out = in
  end
endmodule
```

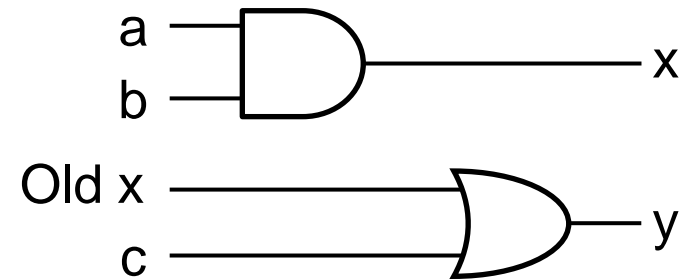
Blocking versus Non-Blocking Assignment

Guideline: Use Blocking Assignment for Combinational Logic



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

Old x is
Latched



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

Evaluate all expressions

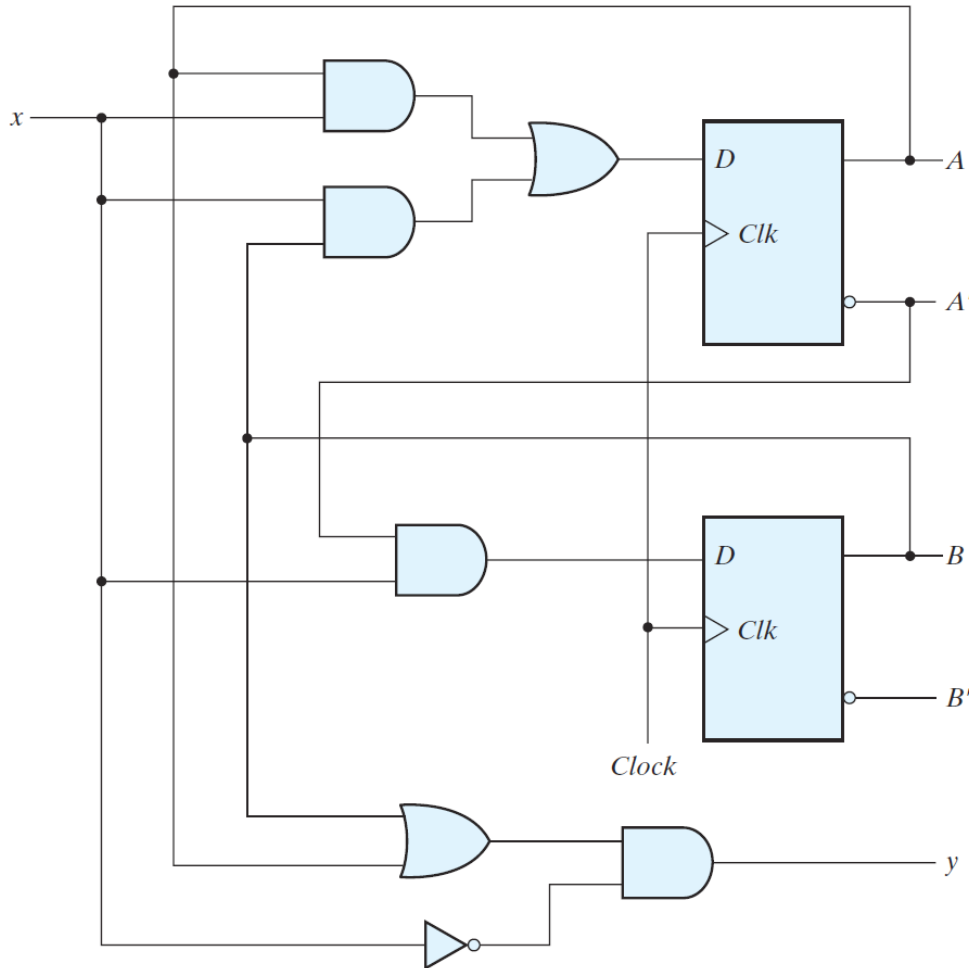
Parallel Assignment at the end

Verilog Coding Guidelines

1. When modeling **combinational** logic, use **blocking** assignments
2. When modeling **sequential** logic, use **non-blocking** assignments
3. When modeling **both sequential** and **combinational logic** within the same always block, use **non-blocking** assignments
4. Do **NOT** mix blocking with non-blocking assignments in the same always block
5. Do **NOT** make assignments to the same variable from more than one always block

Structural Modeling of Sequential Circuits

Modeling the Circuit Structure



```
// Mixed Structural and Dataflow
```

```
module Seq_Circuit_Structure
```

```
(input x, Clock, output y);
```

```
wire DA, DB, A, Ab, B, Bb;
```

```
// Instantiate two D Flip-Flops
```

```
D_FF FFA(DA, Clock, A, Ab);
```

```
D_FF FFB(DB, Clock, B, Bb);
```

```
// Modeling logic
```

```
assign DA = (A & x) | (B & x);
```

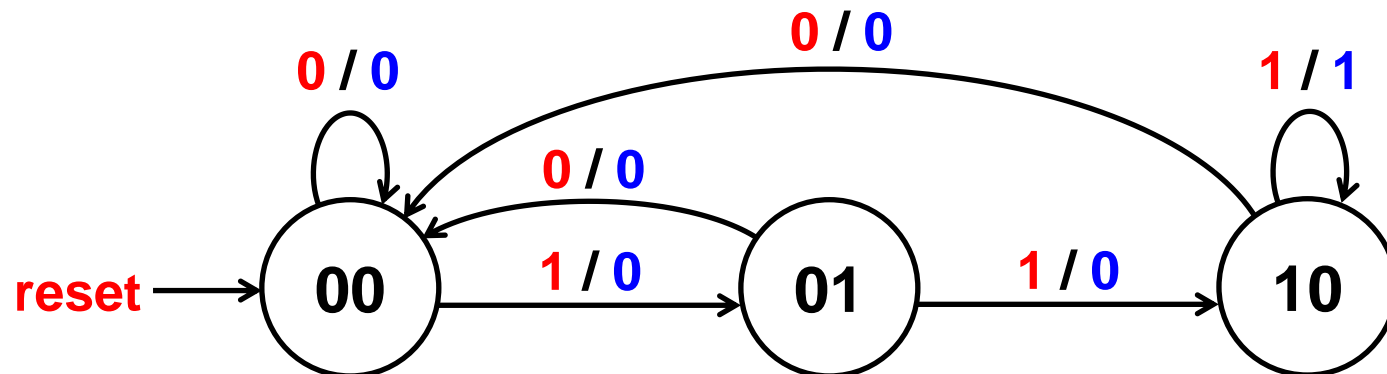
```
assign DB = Ab & x;
```

```
assign y = (A | B) & ~x;
```

```
endmodule
```

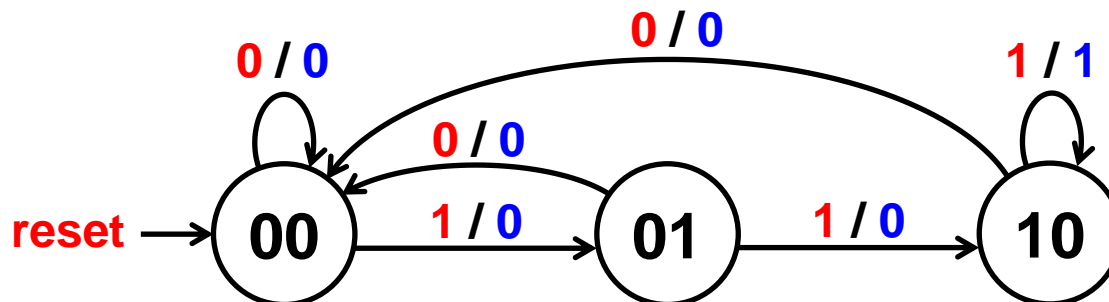

Modeling a State Diagram

- ❖ A state diagram can be modeled directly in Verilog
- ❖ Without the need of having the circuit implementation
- ❖ An example of a Mealy state diagram is shown below
- ❖ This is the state diagram of the **111** sequence detector
- ❖ State assignment: $S_0 = 00$, $S_1 = 01$, and $S_2 = 10$



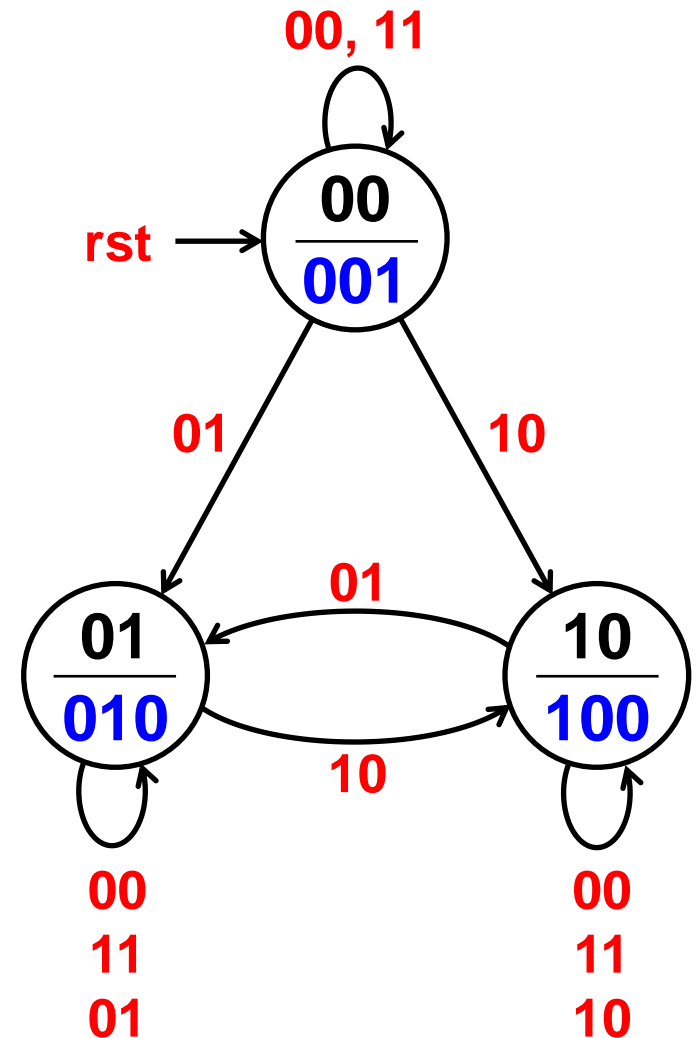
Modeling a Mealy State Diagram

```
module Mealy_111_detector (input x, clock, reset, output z);  
    reg [1:0] state;           // present state  
    always @(posedge clock, posedge reset)  
        if (reset) state <= 'b00;  
        else case(state)  
            'b00: if (x) state <= 'b01; else state <= 'b00;  
            'b01: if (x) state <= 'b10; else state <= 'b00;  
            'b10: if (x) state <= 'b10; else state <= 'b00;  
        endcase  
    // output depends on present state and input x  
    assign z = (state == 'b10) & x;  
endmodule
```



Modeling a Moore State Diagram

```
module Moore_Comparator (input A, B, clk, rst, output GT, LT, EQ);  
    reg [1:0] state;    // present state  
    assign GT = state[1];  
    assign LT = state[0];  
    assign EQ = ~(GT | LT);  
    always @(posedge clk, posedge rst)  
        if (rst) state <= 'b00;  
        else case (state)  
            'b00: state <= ({A,B}=='b01)?'b01:  
                        ({A,B}=='b10)?'b10:'b00;  
            'b01: state <= ({A,B}=='b10)?'b10:'b01;  
            'b10: state <= ({A,B}=='b01)?'b01:'b10;  
        endcase  
    endmodule
```

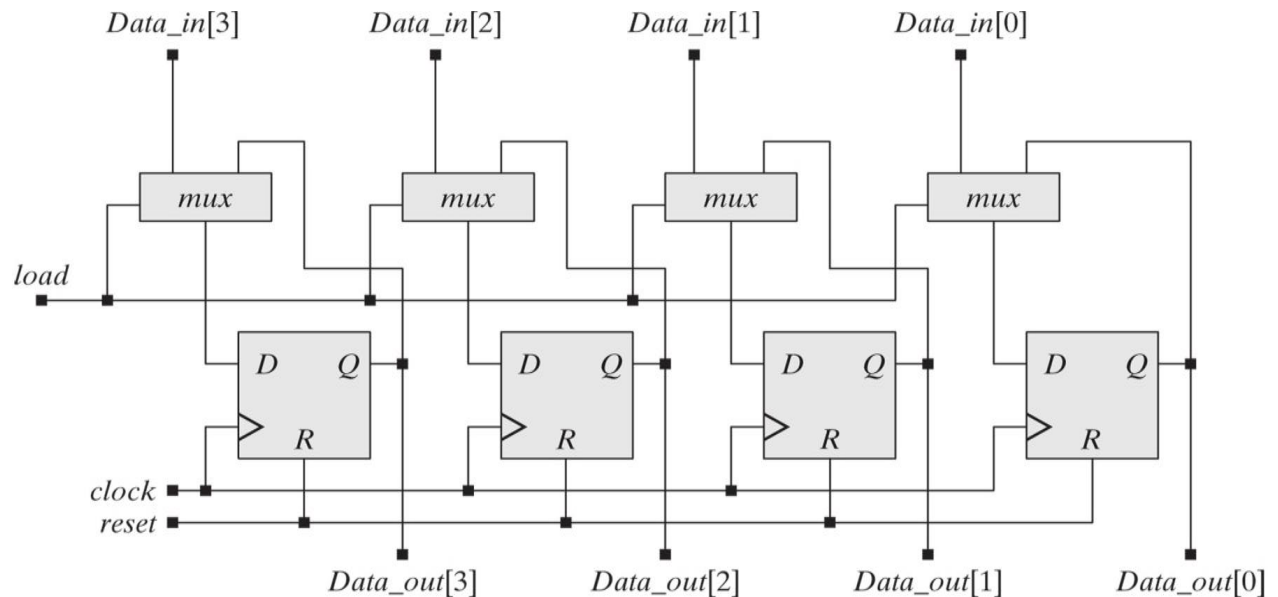


Modeling a Register with Parallel Load

```
module Register #(parameter n = 4)
  (input [n-1:0] Data_in, input load, clock, reset,
   output reg [n-1:0] Data_out);

  always @(posedge clock, posedge reset) // Asynchronous reset
    if (reset) Data_out <= 0;
    else if (load) Data_out <= Data_in;

endmodule
```



Modeling a Shift Register

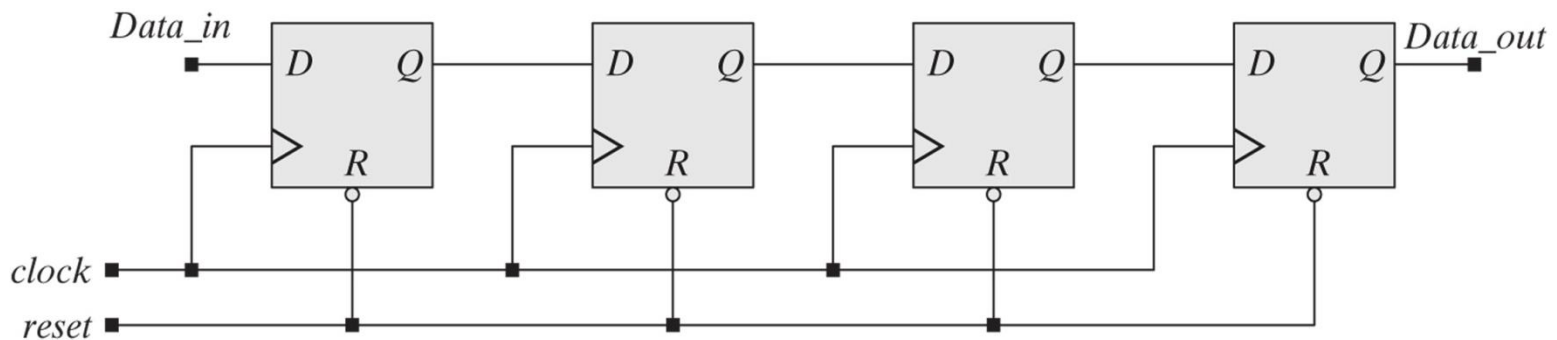
```
module Shift_Register #(parameter n = 4)
    (input Data_in, clock, reset, output Data_out);

    reg [n-1:0] Q;

    assign Data_out = Q[0];

    always @(posedge clock, negedge reset) // Asynchronous reset
        if (!reset) Q <= 0;                // Active Low reset
        else Q <= {Data_in, Q[n-1:1]};    // Shifts to the right

endmodule
```



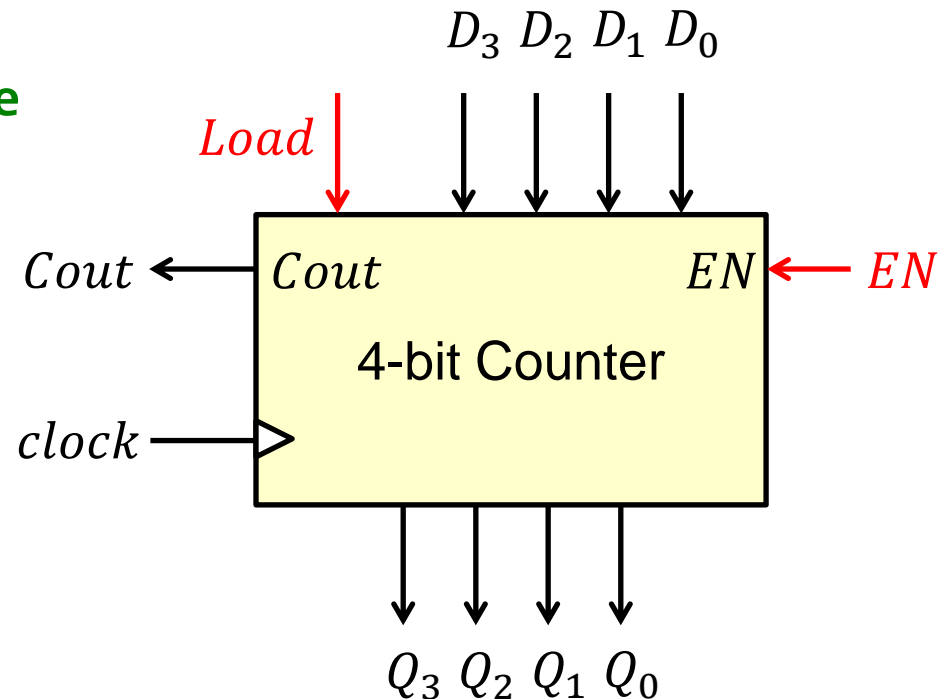
Modeling a Counter with Parallel Load

```
module Counter_with_Load #(parameter n = 4) // n-bit counter
( input [n-1:0] D, input Load, EN, clock,
  output reg [n-1:0] Q, output Cout);

assign Cout = (&Q) & EN;

// Sensitive to Positive-edge
always @(posedge clock)
  if (Load)
    Q <= D;
  else if (EN)
    Q <= Q + 1;

endmodule
```



Modeling a Generic Up-Down Counter

```
module Up_Down_Counter #(parameter n = 16) // n-bit counter
```

```
( input [n-1:0] Data_in,  
  input [1:0] f, input reset, clock,  
  output reg [n-1:0] Count );
```

```
// Asynchronous reset
```

```
always @(posedge clock, posedge reset)
```

```
  if (reset) Count <= 0;
```

```
  else if (f == 1) Count <= Count + 1;
```

```
  else if (f == 2) Count <= Count - 1;
```

```
  else if (f == 3) Count <= Data_in;
```

```
endmodule
```

f = 0 → Disable counter
f = 1 → Count up
f = 2 → Count down
f = 3 → Load counter

