Digital Design with Verilog

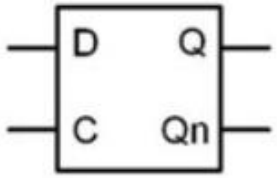
Verilog

Lecture 11: Behavioral Modeling of Sequential Logic

Learning Objectives

- Behavioural modeling of sequential logic storage device
- Behavioural modeling of Finite State Machines (FSMs)
- Behavioural modeling of counters
- Register Transfer Level (RTL) model of a synchronous digital system.
  - Register
  - Shift Register

D-Latch

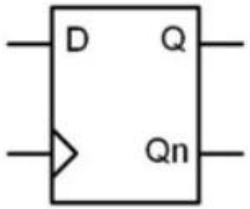


C	D	Q	Qn
0	X	Last Q	Last Qn
1	0	0	1
1	1	1	0

```
module dlatch (output reg  Q, Qn,
               input  wire C, D);


  always @ (C or D)
    if (C == 1'b1)
      begin
        Q  <= D;
        Qn <= ~D;
      end


endmodule
```

Behavioral model of a D-latch in Verilog

D-Flip-Flop

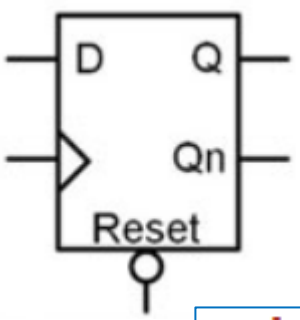


Clk	D	Q	Qn
0	X	Last Q	Last Qn
1	X	Last Q	Last Qn
↓	0	0	1
↓	1	1	0

```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, D);

   always @ (posedge Clock)
      begin
         Q  <= D;
         Qn <= ~D;
      end

endmodule
```

Behavioral model of a D-flip-flop in Verilog

D-Flip-Flop with Asynchronous Reset

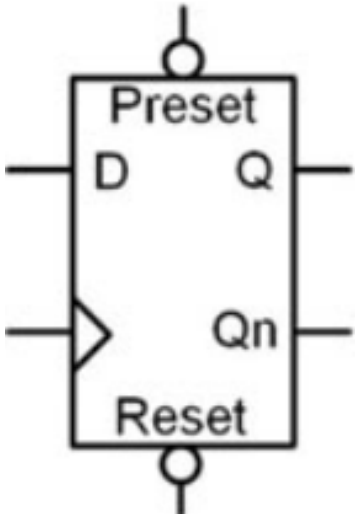


$\overline{R}$	Clk	D	Q	Qn
0	X	X	0	1
1	0	X	Last Q	Last Qn
1	1	X	Last Q	Last Qn
1	↓	0	0	1
1	↓	1	1	0

```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, D);

  always @ (posedge Clock or negedge Reset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else
       begin
         Q  <= D;
         Qn <= ~D;
       end

endmodule
```

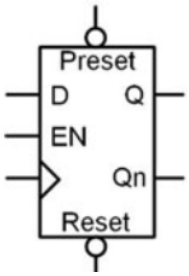
D-FF with Asynchronous Reset and Preset

```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, Preset, D);

  always @ (posedge Clock or negedge Reset or negedge Preset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else if (!Preset)
      begin
        Q  <= 1'b1;
        Qn <= 1'b0;
      end
    else
       begin
         Q  <= D;
         Qn <= ~D;
       end

endmodule
```



R	P	Clk	D	Q	Qn
0	X	X	X	0	1
1	0	X	X	1	0
1	1	0	X	Last Q	Last Qn
1	1	1	X	Last Q	Last Qn
1	1	↓	0	0	1
1	1	↓	1	1	0

D-FF with synchronous enable in Verilog




```
module dflipflop (output reg  Q, Qn,
                  input  wire Clock, Reset, Preset, D, EN);

  always @ (posedge Clock or negedge Reset or negedge Preset)
    if (!Reset)
      begin
        Q  <= 1'b0;
        Qn <= 1'b1;
      end
    else if (!Preset)
      begin
        Q  <= 1'b1;
        Qn <= 1'b0;
      end
    else if (EN)
       begin
         Q  <= D;
         Qn <= ~D;
       end

endmodule
```

Since EN is not listed in the sensitivity list it does not trigger the block when it transitions. This "if" statement is only reached if there is a rising edge of the clock. This models an enable that is synchronous to the clock.

Modeling Finite State Machines in Verilog

FSMs in Verilog

- Finite state machines can be easily modeled using the behavioral constructs.
- The most common modeling practice for FSMs is to declare two signals of type reg that are called current_state and next_state.
- Then a parameter is declared for each descriptive state name in the state diagram.
- A parameter also requires a value, so the state encoding can be accomplished during the parameter declaration.

FSMs in Verilog

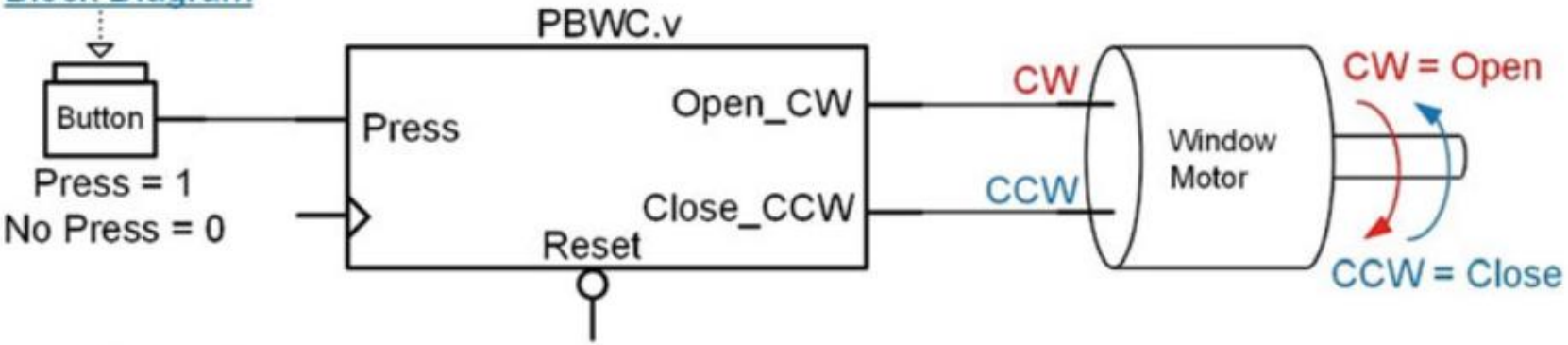
- Once the signals and parameters are created, all of the procedural assignments in the state machine model can use the descriptive state names in their signal assignments.

- Within the Verilog state machine model, three separate procedural blocks are used to describe each of the functional blocks, state memory, next state logic, and output logic

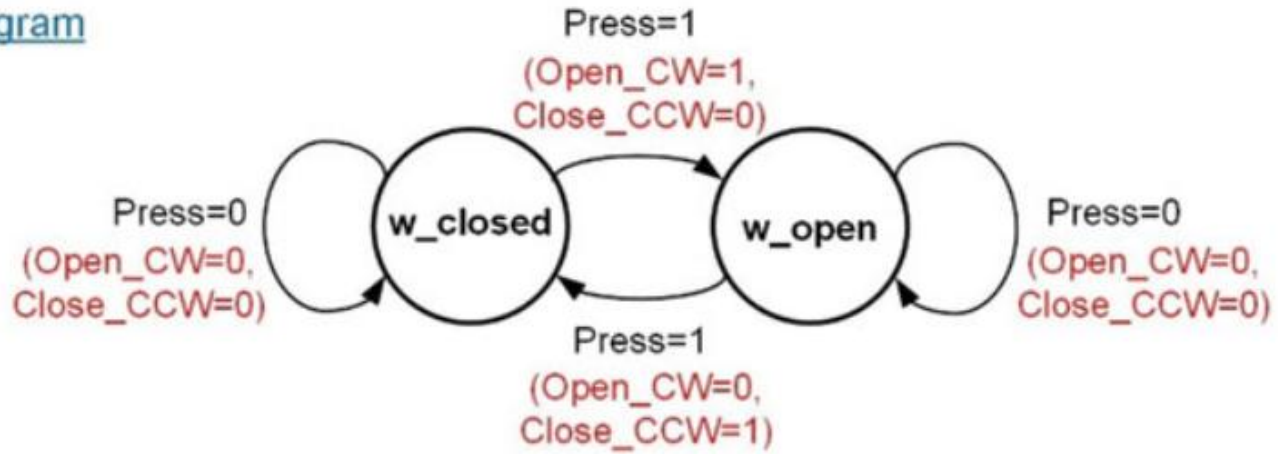
Example: Push-Button Window Controller

The window controller will send the appropriate control signals to a motor to open or close it whenever a button is pressed. The system must keep track whether the window is open or closed in order to send the correct signal, thus a state machine is needed. The block diagram and state diagram for this system is shown below.

Block Diagram

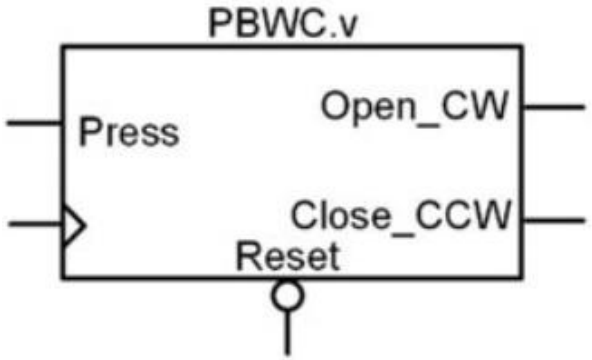


State Diagram



Example: Push-Button Window Controller

Example: Push-Button Window Controller in Verilog – Port Definition




```
module PBWC (output reg  Open_CW, Close_CCW,
             input  wire Clock, Reset, Press);

                    :
                    :
```

Outputs are defined as type reg while inputs are defined of type wire.

Example: Push-Button Window Controller

Modeling States

```
reg       current_state, next_state;
parameter w_closed = 1'b0,
          w_open   = 1'b1;
```

The State Memory Block

```
always @ (posedge Clock or negedge Reset)
  begin: STATE_MEMORY
    if (!Reset)
      current_state <= w_closed;
    else
      current_state <= next_state;
  end
```

Example: Push-Button Window Controller

The Next State Logic Block

```
always @ (current_state or Press)
  begin: NEXT_STATE_LOGIC
    case (current_state)
      w_closed : if (Press == 1'b1)
                    next_state = w_open;
                 else
                    next_state = w_closed;
      w_open   : if (Press == 1'b1)
                    next_state = w_closed;
                 else
                    next_state = w_open;
      default  : next_state = w_closed;
    endcase
  end
```

Example: Push-Button Window Controller

The Output Logic Block

```
always @ (current_state or Press)
  begin: OUTPUT_LOGIC
    case (current_state)
      w_closed : if (Press == 1'b1)
                   begin
                     Open_CW    = 1'b1;
                     Close_CCW = 1'b0;
                   end
                 else
                   begin
                     Open_CW    = 1'b0;
                     Close_CCW = 1'b0;
                   end

      w_open   : if (Press == 1'b1)
                   begin
                     Open_CW    = 1'b0;
                     Close_CCW = 1'b1;
                   end
                 else
                   begin
                     Open_CW    = 1'b0;
                     Close_CCW = 1'b0;
                   end

       default : begin
                     Open_CW    = 1'b0;
                     Close_CCW = 1'b0;
                 end
    endcase

  end
```

Example: Push-Button Window Controller

```
module PBWC (output reg  Open_CW, Close_CCW,
     input  wire Clock, Reset, Press);

 reg       current_state, next_state;
 parameter w_closed = 1'b0,
           w_open   = 1'b1;
//-------------------------------------------------
// STATE MEMORY

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= w_closed;
      else
        current_state <= next_state;
    end
//-------------------------------------------------
// NEXT STATE LOGIC

  always @ (current_state or Press)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        w_closed : if (Press == 1'b1)
                     next_state = w_open;
                   else
                     next_state = w_closed;
        w_open   : if (Press == 1'b1)
                     next_state = w_closed;
                   else
                     next_state = w_open;
        default  : next_state = w_closed;
      endcase
    end
//-------------------------------------------------
// OUTPUT LOGIC

   always @ (current_state or Press)
     begin: OUTPUT_LOGIC
       case (current_state)
         w_closed : if (Press == 1'b1)
                      begin
                        Open_CW  = 1'b1;
                        Close_CCW = 1'b0;
                      end
                    else
                      begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b0;
                      end
         w_open   : if (Press == 1'b1)
                      begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b1;
                      end
                    else
                      begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b0;
                      end

          default : begin
                        Open_CW  = 1'b0;
                        Close_CCW = 1'b0;
                    end
       endcase
     end
endmodule
```

Declaration of state variables and state encoding.

State memory block. This is sequential logic so non-blocking assignments are used. This block only makes assignments to the signal "current_state".

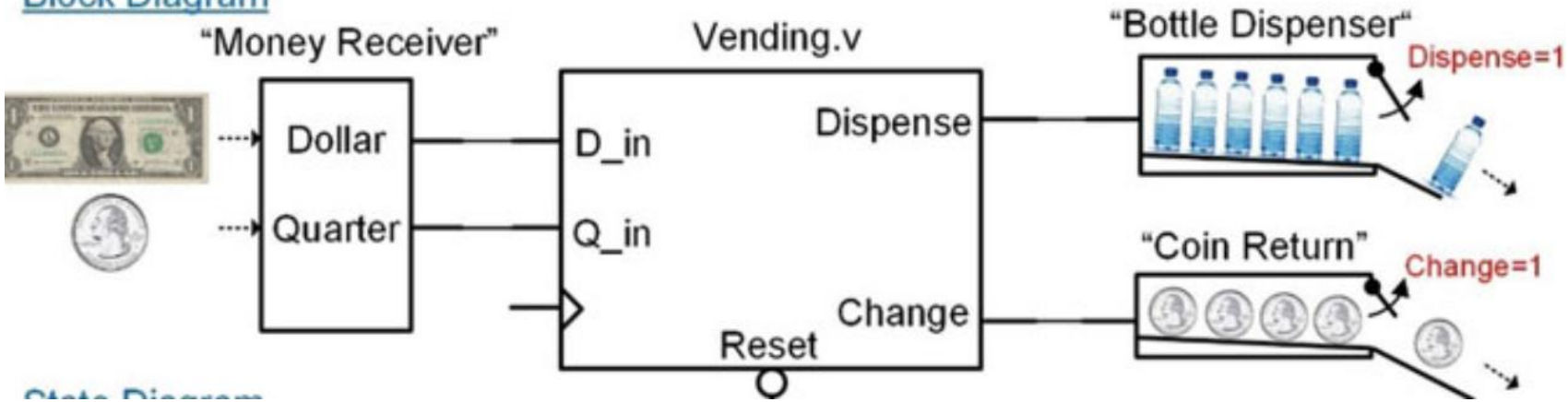
Next state logic block. This is combinational logic so blocking assignments are used. This block only makes assignments to the signal "next_state".

Output logic block. This is combinational logic so blocking assignments are used. Since this is a Mealy machine, the current state and input are listed in the sensitivity list. This block only makes assignments to the outputs "Open_CW" and "Close_CCW".

Example: Vending Machine Controller

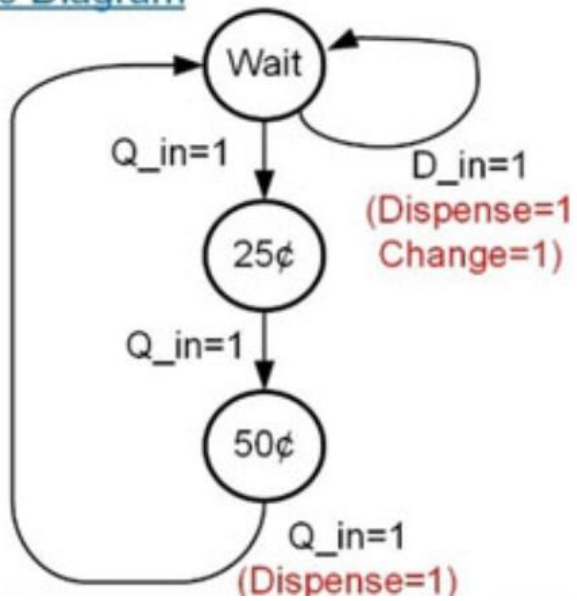
The vending machine sells bottles of water for 75¢. Customers can enter either a dollar bill or quarters. Once a sufficient amount of money is entered, the vending machine will dispense a bottle of water and, if the user entered a dollar, return one quarter in change.

Block Diagram



Example: Vending Machine Controller

State Diagram



Port Definition

```
module Vending
  (output reg  Dispense, Change,
   input  wire Clock, Reset, D_in, Q_in);
                        :
                        :
```

Example: Vending Machine Controller

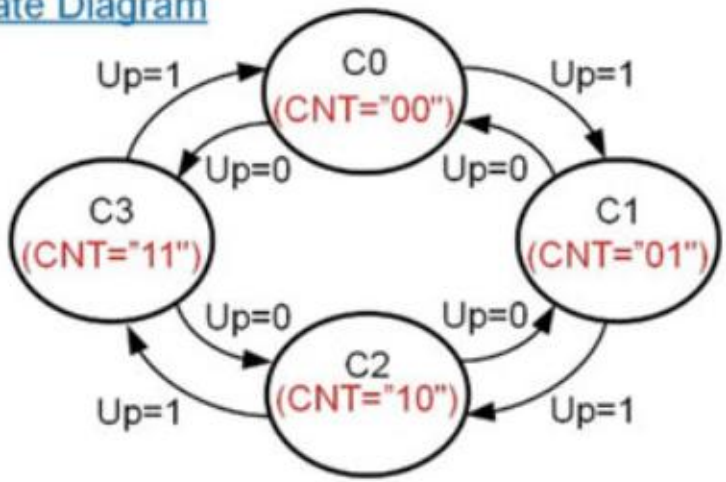
```
module Vending (output reg  Dispense, Change,
                input  wire Clock, Reset, D_in, Q_in);

  reg [1:0] current_state, next_state;
  parameter sWait = 2'b00,
            s25   = 2'b01,
            s50   = 2'b10;

//----------------------------------------------------------
// STATE MEMORY
  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= sWait;
      else
        current_state <= next_state;
    end
//----------------------------------------------------------
// NEXT STATE LOGIC
  always @ (current_state or D_in or Q_in)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        sWait    : if (Q_in == 1'b1)
                     next_state = s25;
                   else
                     next_state = sWait;
        s25      : if (Q_in == 1'b1)
                     next_state = s50;
                   else
                     next_state = s25;
        s50      : if (Q_in == 1'b1)
                     next_state = sWait;
                   else
                     next_state = s50;
        default  : next_state = sWait;
      endcase
    end
//----------------------------------------------------------
// OUTPUT LOGIC
   always @ (current_state or D_in or Q_in)
     begin: OUTPUT_LOGIC
      case (current_state)
        sWait    : if (D_in == 1'b1)
                     begin
                       Dispense = 1'b1; Change = 1'b1;
                     end
                   else
                     begin
                       Dispense = 1'b0; Change = 1'b0;
                     end

        s25      : begin
                     Dispense = 1'b0; Change = 1'b0;
                   end

        s50      : if (Q_in == 1'b1)
                     begin
                       Dispense = 1'b1; Change = 1'b0;
                     end
                   else
                     begin
                       Dispense = 1'b0; Change = 1'b0;
                     end

        default  : begin
                     Dispense = 1'b0; Change = 1'b0;
                   end
      endcase
   end
endmodule
```

2-Bit, Binary Up/Down Counter

Example: 2-Bit Up/Down Counter in Verilog – Design Description and Port Definition

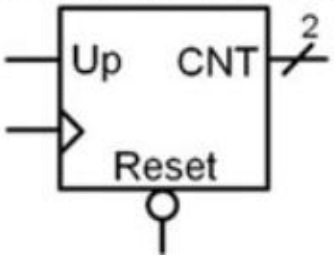
This system will output a synchronous, 2-bit, binary counter. When the system input Up=1, the system will count up. When Up=0, the sytem will count down. The output of the counter is called CNT.

State Diagram



Port Definition

Counter_2bit_UpDown.v




```
module Counter_2bit_UpDown
    (output reg [1:0] CNT,
     input  wire      Clock, Reset, Up);
                :
                :
```

2-Bit, Binary Up/Down Counter

```
module Counter_2bit_UpDown (output reg [1:0] CNT,
                            input  wire Clock, Reset, Up);
  reg [1:0] current_state, next_state;
  parameter C0 = 2'b00,
            C1 = 2'b01,
            C2 = 2'b10,
            C3 = 2'b11;
//------------------------------------------------------------
// STATE MEMORY

  always @ (posedge Clock or negedge Reset)
    begin: STATE_MEMORY
      if (!Reset)
        current_state <= C0;
      else
        current_state <= next_state;
    end
//------------------------------------------------------------
// NEXT STATE LOGIC

  always @ (current_state or Up)
    begin: NEXT_STATE_LOGIC
      case (current_state)
        C0      : if (Up == 1'b1) next_state = C1; else next_state = C3;
        C1      : if (Up == 1'b1) next_state = C2; else next_state = C0;
        C2      : if (Up == 1'b1) next_state = C3; else next_state = C1;
        C3      : if (Up == 1'b1) next_state = C0; else next_state = C2;
        default : next_state = C0;
      endcase
    end
//------------------------------------------------------------
// OUTPUT LOGIC

   always @ (current_state)
     begin: OUTPUT_LOGIC
      case (current_state)
        C0      : CNT = 2'b00;
        C1      : CNT = 2'b01;
        C2      : CNT = 2'b10;
        C3      : CNT = 2'b11;
        default : CNT = 2'b00;
      endcase
   end
endmodule
```

Binary Counter- Single Procedural Block

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire Clock, Reset);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        CNT <= CNT + 1;
    end

endmodule
```

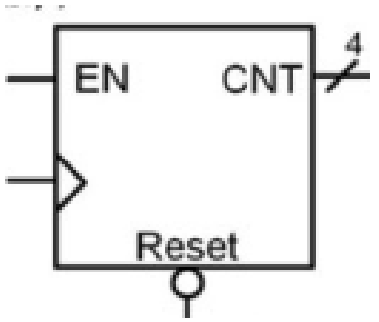
Counters with Range Checking

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire Clock, Reset);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (CNT == 10)
          CNT <= 0;
        else
          CNT <= CNT + 1;
    end

endmodule
```

A nested if-else statement checks if the counter has reached its maximum value. If it has, it is reset back to zero. If it hasn't, it increments.

Binary Counter with Enable in Verilog

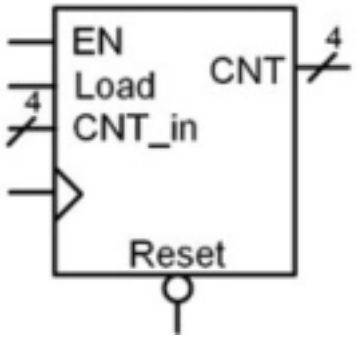
```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire Clock, Reset, EN);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (EN)
          CNT <= CNT + 1;
    end

endmodule
```



The EN is synchronous to the clock, so its logic is nested beneath the portion of the main if-else clause that handles the behavior when the counter receives a rising edge of clock.

Counters with Loads

```
module Counter_4bit_Up (output reg [3:0] CNT,
                        input  wire Clock, Reset, EN, Load,
                        input  wire [3:0] CNT_in);

  always @ (posedge Clock or negedge Reset)
    begin: COUNTER
      if (!Reset)
        CNT <= 0;
      else
        if (EN)
          if (Load)
            CNT <= CNT_in;
          else
            CNT <= CNT + 1;
    end

endmodule
```



A nested if-else statement is used to load CNT with CNT_in when the Load signal is asserted and the counter receives a rising edge of clock.

RTL Modeling

RTL Modeling

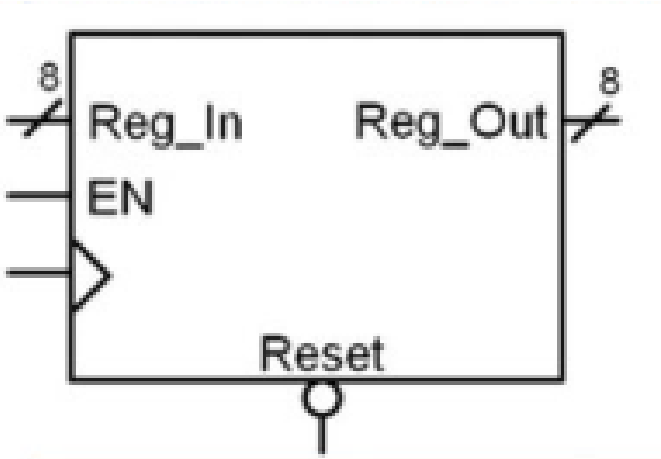
- Register Transfer Level modeling refers to a level of design abstraction in which
  - vector data is moved and operated on in a synchronous manner.

- This design methodology is widely used in data path modeling and computer system design.

Modeling Registers in Verilog

- The term register describes a group of D-Flip-Flops running off of the same clock, reset, and enable inputs.
- Data is moved in and out of the bank of D-flip-flops as a vector.
- Logic operations can be made on the vectors and are latched into the register on a clock edge.
- A register is a higher level of abstraction that allows vector data to be stored without getting into the details of the lower level implementation of the DFlip-Flops and combinational logic.

Modeling Registers in Verilog

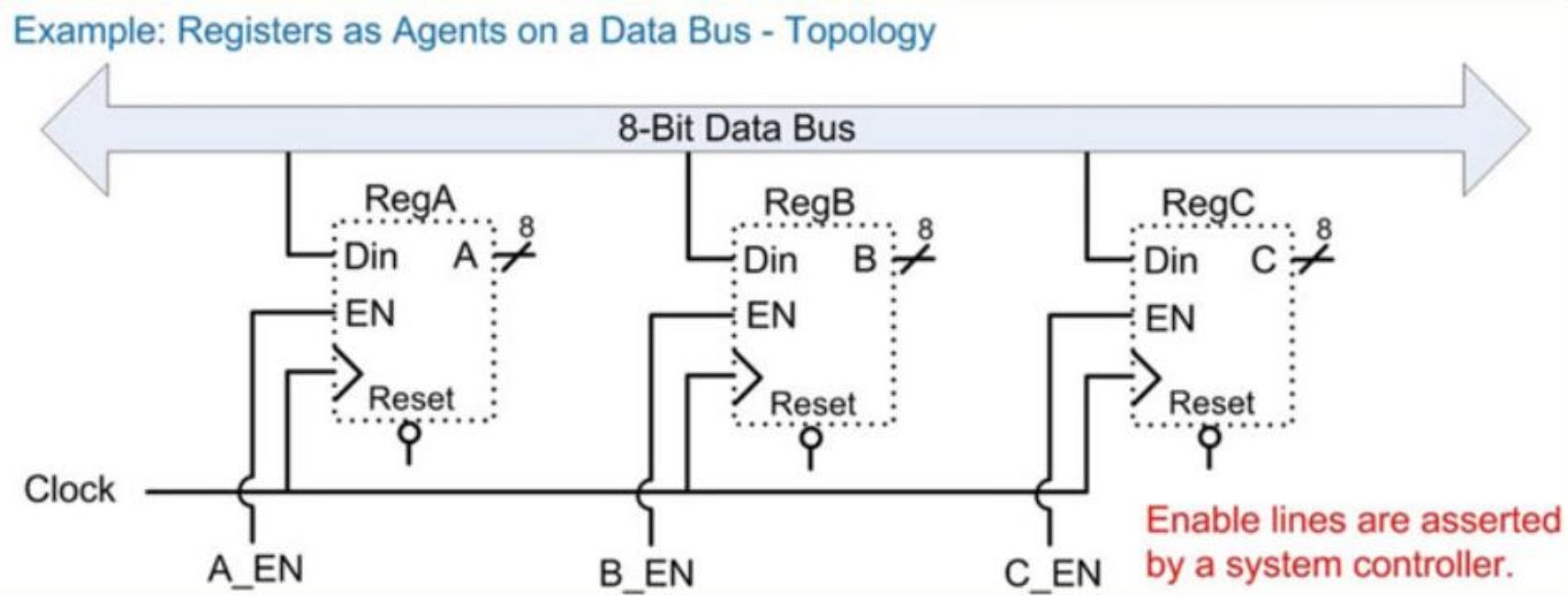


$\overline{R}$	Clk	EN	Reg_Out	
0	X	X	x"00"	Reset
1	X	0	Last Reg_Out	Disabled (ignore clock)
1	0	1	Last Reg_Out	Store
1	1	1	Last Reg_Out	Store
1	↓	1	Reg_In	Update

Modeling Registers in Verilog

```
module RegX (output reg [7:0]   Reg_Out,
             input  wire        Clock, Reset, EN,
             input  wire [7:0]  Reg_In);

  always @ (posedge Clock or negedge Reset)
    begin: REGISTER
      if (!Reset)
        Reg_Out <= 8'h00;
      else
        if (EN)
          Reg_Out <= Reg_In;
    end

endmodule
```

Registers as Agents on a Data Bus



Registers as Agents on a Data Bus

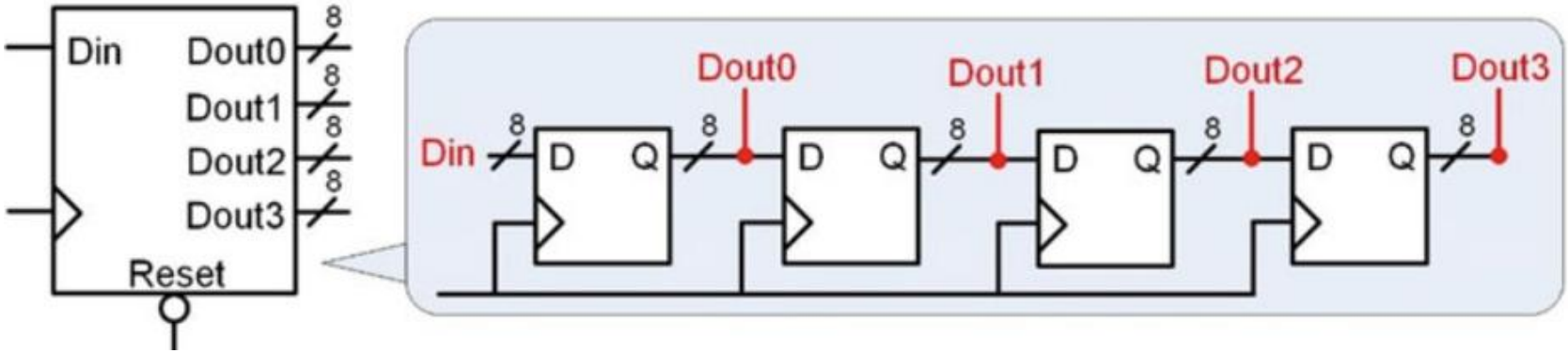
```
module MultiDropBus
   (output reg [7:0]  A, B, C,
    input  wire       Clock, Reset,
    input  wire [7:0] Data_Bus,
    input  wire       A_EN, B_EN, C_EN);

  always @ (posedge Clock or negedge Reset)
    begin: A_REG
      if (!Reset)
        A <= 8'h00;
      else
        if (A_EN ==1)
          A <= Data_Bus;
    end

  always @ (posedge Clock or negedge Reset)
    begin: B_REG
      if (!Reset)
        B <= 8'h00;
      else
        if (B_EN ==1)
          B <= Data_Bus;
    end

  always @ (posedge Clock or negedge Reset)
    begin: C_REG
      if (!Reset)
        C <= 8'h00;
      else
        if (C_EN ==1)
          C <= Data_Bus;
    end

endmodule
```

Each register is modeled as a separate block. The register has a synchronous enable that controls when it acquires data off of the data bus.

All registers are attached to the data bus as receivers.

Shift Registers in Verilog

- A shift register is a circuit which consists of multiple registers connected in series.
- Data is shifted from one register to another on the rising edge of the clock.
- This type of circuit is often used in serial-to-parallel data converters.



Shift Registers in Verilog

```
module Shift_Register
   (output reg [7:0]  Dout0, Dout1, Dout2, Dout3,
    input  wire       Clock, Reset,
    input  wire [7:0] Din);

  always @ (posedge Clock or negedge Reset)
    begin: SHIFT_REGISTER
      if (!Reset)
        begin
          Dout0 <= 8'h00;
          Dout1 <= 8'h00;
          Dout2 <= 8'h00;
          Dout3 <= 8'h00;
        end
      else
        begin
          Dout0 <= Din;
          Dout1 <= Dout0;
          Dout2 <= Dout1;
          Dout3 <= Dout2;
        end
    end

endmodule
```

References

- Chapter 9, Introduction to Logic Circuits & Logic Design with Verilog by Brock J. LaMeres

- Disclaimer: “I don’t claim the ownership of all the slides, some of the material is picked up from various publicly available sources on the internet”.

Thank you