BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI - K. K. BIRLA GOA CAMPUS

VERILOG IMPORTANT NOTES

Module declaration

```
module module_name(port list separated by comma);
  input ports;
  output ports;
  wire list;
  circuit implementation block;
endmodule
```

- Verilog is case sensitive
- Wires are used for internal connections
- The port list of a module is the interface between the module and its environment.
- Circuit description block can be written with
  - Gate-level modeling (structural modeling)

```
module Full_Adder_G(S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;
  wire w1, w2, w3;

  xor G1(w1, A, B);
  and G2(w2, A, B);
  xor G3(S, w1, Ci);
  and G4(w3, w1, Ci);
  or G5(Co, w2, w3);
endmodule
```

  - Dataflow modeling

```
module Full_Adder_D(S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;

  assign S=(A&B&Ci)|((A)&(~B)&Ci)|((~A)&B&(~Ci))|(A&(~B)
&(~Ci));
  assign Co=(A&B)|(B&Ci)|(A&Ci);

endmodule
```

- Behavioral modeling

```
module Full_Adder_B(S,Co, A, B, Ci);
  output  S, Co;
  input A, B, Ci;
  reg  S, Co;

  always @(A or B or Ci)
  begin
    {Co,S}=A+B+Ci;
  end
endmodule
```

- Predefined primitives : **and**, **or**, **not**, **buf**, **xor**, **xnor**, **nand**, **nor**, **bufif0**, **bufif1**, **notif0**, **notif1**
- The output of a primitive gate is always listed first, followed by the inputs.
- The output of a primitive must be listed first, but the inputs and outputs of a module may be listed in any order.
- User Defined Primitives (UDPs) :
  - It is declared with the keyword **primitive** , followed by a name and port list.
  - There can be only **one output**, and it must be **listed first** in the port list and declared with keyword **output**.
  - There can be any number of inputs. The order in which they are listed in the input declaration must conform to the order in which they are given values in the table that follows.
  - The truth table is enclosed within the keywords **table** and **endtable**.
  - The values of the inputs are listed in order, ending with a colon (:). The output is always the last entry in a row and is followed by a semicolon (;).
  - The declaration of a UDP ends with the keyword **endprimitive**.

```
primitive UDP_1247(S, A, B, Ci);
  output S;
  input A, B, Ci;

  table
    0 0 0 : 0;
    0 0 1 : 1;
    0 1 0 : 1;
    0 1 1 : 0;
    1 0 0 : 1;
    1 0 1 : 0;
    1 1 0 : 0;
    1 1 1 : 1;
  endtable
endprimitive


primitive UDP_3567(Co, A, B, Ci);
  output Co;
  input A, B, Ci;

  table
    0 0 0 : 0;
    0 0 1 : 0;
    0 1 0 : 0;
    0 1 1 : 1;
    1 0 0 : 0;
    1 0 1 : 1;
    1 1 0 : 1;
    1 1 1 : 1;
  endtable
endprimitive
```

– Full Adder using above UDPs

```
module FA_UDP(S, Co, A, B, Ci);
  output S, Co;
  input A, B, Ci;

  UDP_3567 Carry(Co, A, B, Ci);

  UDP_1247 Sum(S, A, B, Ci);

endmodule
```

Gate-Level Modeling

- It is done using instantiations of predefined and user-defined primitive gates. In this type of representation, a circuit is specified by its logic gates and their interconnections.
- There are 12 basic gates as predefined primitives: **and**, **or**, **not**, **buf**, **xor**, **xnor**, **nand**, **nor**, **bufif0**, **bufif1**, **notif0**, **notif1**
- Four of these primitive gates are of the three-state type : **bufif0**, **bufif1**, **notif0**, **notif1**

- The **bufif1** gate behaves like a normal buffer if control = 1. The output goes to a high-impedance state z when control = 0.
- The **bufif0** gate behaves in a similar fashion, except that the high-impedance state occurs when control = 1.
- The two **notif** gates operate in a similar manner, except that the output is the complement of the input when the gate is not in a high-impedance state.
- The gates are instantiated with the statement: **gate name(output, input, control);**
- The outputs of three-state gates can be connected together to form a common output line. To identify such a connection, Verilog HDL uses the keyword **tri** (for tristate) to indicate that the output has multiple drivers.

```
// Mux with three-state output

module mux_tri (m_out, A, B, select);
  output m_out;
  input A, B, select;
  tri m_out;

  bufif1 (m_out, A, select);
  bufif0 (m_out, B, select);
endmodule
```

- Primitives such as **and** are n-input primitives. They can have any number of scalar inputs.
- The **buf** and **not** primitives are n-output primitives. A single input can drive multiple output lines distinguished by their identifiers.
- Outputs are always listed first.
- The logic of each gate is based on a four-valued system : 0, 1, **x** (unknown), **z** (high impedance)
- Truth table for primitive gates

and	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

or	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

xor	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

not	Input	Output
	0	1
	1	0
	x	x
	z	x

- Ports, **wire** and **reg** can be declared having multiple bit width. e.g. **output** [3:0] SUM, **wire** [0:7] A. Here first number in square bracket represent index of MSB and second number represent index of LSB.

- Note that the keywords **not** and **nand** can be written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

```
// Gate-level description of two-to-four-line decoder

module decoder_2x4_gates (D, A, B, enable);
  output [0: 3] D;
  input A, B;
  input enable;
  wire A_not,B_not, enable_not;
  not
    G1 (A_not, A),
    G2 (B_not, B),
    G3 (enable_not, enable);
  nand
    G4 (D[0], A_not, B_not, enable_not),
    G5 (D[1], A_not, B, enable_not),
    G6 (D[2], A, B_not, enable_not),
    G7 (D[3], A, B, enable_not);
endmodule
```

- Two or more modules can be combined to build a hierarchical description of a design. There are two basic types of design methodologies:

  - In a top-down design, the top-level block is defined and then the subblocks necessary to build the top-level block are identified.

  - In a bottom-up design, the building blocks are first identified and then combined to build the top-level block.

```
// Gate-level description of four-bit ripple carry adder
// Description of half adder

// module half_adder (S, C, x, y); // Verilog 1995 syntax
  // output S, C;
  // input x, y;
module half_adder ( output S, C, input x, y); // Verilog
 2001,2005 syntax
// Instantiate primitive gates
  xor (S, x, y);
  and (C, x, y);
```

```
endmodule

// Description of full adder  // Verilog 1995 syntax

// module full_adder (S, C, x, y, z);
  // output S, C;
  // input x, y, z;

module full_adder (output S, C,input x, y, z); // Verilog
 2001, 2005 syntax
  wire S1, C1, C2;
// Instantiate half adders
  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule

// Description of four-bit adder  // Verilog 1995 syntax

// module ripple_carry_4_bit_adder (Sum, C4, A, B, C0);
  // output [3: 0] Sum;
  // output C4;
  // input [3: 0] A, B;
  // input C0;

// Alternative Verilog 2001, 2005 syntax:

module ripple_carry_4_bit_adder (output [3:0] Sum, output
 C4, input [3:0] A, B, input C0);
  wire C1, C2, C3; // Intermediate carries
// Instantiate chain of full adders
  full_adder  FA0 (Sum[0], C1, A[0], B[0], C0),
              FA1 (Sum[1], C2, A[1], B[1], C1),
              FA2 (Sum[2], C3, A[2], B[2], C2),
              FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```

- Modules can be instantiated (nested) within other modules, but module declarations cannot be nested; that is, a module definition (declaration) cannot be placed within another module declaration.
- A description of a module is said to be a structural description if it is composed of instantiations of other modules. Note that **instance names must be specified when defined modules are instantiated**, but using a name is optional when instantiating primitive gates.

Dataflow Modeling

- Dataflow modeling of combinational logic uses a number of operators that act on binary operands to produce a binary result.

Symbol	Operation	Symbol	Operation
+	binary addition		
−	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
∧	bitwise XOR		
∼	bitwise NOT	!	logical NOT
==	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

- Dataflow modeling uses continuous assignments and the keyword assign. A continuous assignment is a statement that assigns a value to a net.

```
// Dataflow description of two-to-four-line decoder

module decoder_2x4_df(output [0:3] D, input A, B, enable);
  assign  D[0] = !((!A) && (!B) && (!enable)),
          D[1] = !(*!A) && B && (!enable)),
          D[2] = !(A && B && (!enable),
          D[3] = !(A && B && (!enable));
endmodule


// Dataflow description of four-bit adder

module binary_adder (
  output [3:0] Sum,
  output C_out,
  input [3: 0] A, B,
  input C_in
  );

  assign {C_out, Sum} = A+B+C_in;
endmodule
```

- Dataflow HDL models describe combinational circuits by their function rather than by their gate structure.

```
// Dataflow description of a four-bit comparator

module mag_compare (
  output A_lt_B, A_eq_B, A_gt_B,
  input [3: 0] A, B
  );
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```

- **Use of conditional operator (? :)**
  - format is : **condition ? true-expression : false-expression;**
  - The condition is evaluated. If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement. If the result is logic 0, the false expression is evaluated.

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output  m_out;
  input   A, B;
  input   select;

  assign m_out = (select)? A : B;
endmodule
```

Behavioral Modeling

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- Behavioral descriptions use the keyword **always**, followed by an optional event control expression and a list of procedural assignment statements.
- The event control expression specifies when the statements will execute.
- The target output of a procedural assignment statement must be of the **reg** data type.
- A **reg** data type retains its value until a new value is assigned.

- The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variables listed after the @ symbol.

```
// Behavioral description of two-to-one-line multiplexer
module mux_2x1_beh (m_out , A, B, select);
  output  m_out;
  input A, B, select;
  reg  m_out;

  always @(A or B or select)
    if (select == 1) m_out = A;
    else m_out = B;
endmodule
```

- Use of **case**:
  - The **case** statement is a multiway conditional branch construct.
  - The statement associated with the first **case** item that matches the **case** expression is executed.
  - In the absence of a match, no statement is executed.
  - The list of **case** items need not be complete. If the list of **case** items does not include all possible bit patterns of the case expression, no match can be detected. Unlisted case items, i.e., bit patterns that are not explicitly decoded can be treated by using the **default** keyword as the last item in the list of **case** items.

```
// Behavioral description of four-to-one line multiplexer

module mux_4x1_beh(
  output reg m_out,
  input in_0, in_1, in_2, in_3,
  input [1: 0] select
  );

  always @ (in_0, in_1, in_2, in_3, select)
    case (select)
      2'b00:  m_out = in_0;
      2'b01:  m_out = in_1;
      2'b10:  m_out = in_2;
      2'b11:  m_out = in_3;
    endcase
endmodule
```

- Binary numbers in Verilog are specified and interpreted with the letter **b** preceded by a prime. The size of the number is written first and then its value. Thus, 2'b01 specifies a two-bit binary number whose value is 01.
- Numbers are stored as a bit pattern in memory, but they can be referenced in decimal, octal, or hexadecimal formats with the letters d' o' and h' respectively.
- If the base of the number is not specified, its interpretation defaults to decimal.

Writing a Simple Test Bench

- A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit in order to test it and observe its response during simulation.
- The general format of test bench is

```
module test_module_name;
   Declare local reg and wire identifiers.

   Instantiate the design module under test.

   Specify a stopwatch, using $finish to terminate the
 simulation.

   Generate stimulus, using initial and always statements.

   Display the output response (text or graphics (or both)).
endmodule
```

- A test module is written like any other module, but it typically has no inputs or outputs.
- The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type.
- The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type.
- The module under test is then instantiated, using the local identifiers in its port list.
- In addition to employing the **always** statement, test benches use the **initial** statement to provide a stimulus to the circuit being tested.
- Actually, **always** is a Verilog language construct specifying how the associated statement is to execute (subject to the event control expression). The **always** statement executes repeatedly in a loop.
- The **initial** statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #.

- For example, consider the **initial** block

```
initial
  begin
    A = 0; B = 0;
    #10 A = 1;
    #20 A = 0; B = 1;
  end
```

  The block is enclosed between the keywords **begin** and **end**. At time 0, A and B are set to 0. Ten time units later, A is changed to 1. Twenty time units after (at t=30), A is changed to 0 and B to 1.

- As an another example, inputs specified by a three-bit truth table can be generated with the initial block:

```
initial
  begin
    D = 3'b000;
    repeat (7)
    #10 D = D + 3'b001;
  end
```

  When the simulator runs, the three-bit vector D is initialized to 000 at time = 0. The keyword repeat specifies a looping statement: D is incremented by 1 seven times, once every 10 time units. The result is a sequence of binary numbers from 000 to 111.

- Test bench and module to be simulated can be in the same file.

```
module t_mux_2x1_df;
  wire t_mux_out;
  reg t_A, t_B;
  reg t_select;
  parameter stop_time = 50;

  mux_2x1_df M1 (t_mux_out, t_A, t_B, t_select); //
 Instantiation of circuit to be tested

  initial #stop_time $finish;

  initial
    begin
    t_select = 1; t_A = 0; t_B = 1;
    #10 t_A = 1; t_B = 0;
    #10 t_select = 0;
    #10 t_A = 0; t_B = 1;
  end
endmodule

// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df (m_out, A, B, select);
  output  m_out;
  input   A, B;
  input   select;

  assign m_out = (select)? A : B;
endmodule
```

Synthesizable HDL Models of Sequential Circuits

There are two kinds of abstract behaviors in the Verilog HDL.

- Single-pass behavior
  - Behavior declared by the keyword **initial** is called single-pass behavior and specifies a single statement or a block statement.
  - A single-pass behavior expires after the associated statement executes.
  - In practice, designers use single-pass behavior primarily to prescribe stimulus signals in a test bench-never to model the behavior of a circuit-because synthesis tools do not accept descriptions that use the **initial** statement.
  - The **initial** statement is useful for generating input signals to simulate a design.
  - In simulating a sequential circuit, it is necessary to generate a clock source for triggering the flip-flops. The following are one of the possible ways to provide a free-running clock that operates for a specified number of cycles:

```
initial
  begin
    clock = 1'b0;
    repeat (30)
    #10 clock = ~clock;
  end
```

```
initial
  begin
    clock = 1'b0;
  end

initial #300 $finish;
always #10 clock = ~clock;
```

```
initial
  begin
    clock = 0;
    forever #10 clock = ~clock;
  end
```

- Cyclic behavior
  - Behavior declared by the keyword **always** is called cyclic behavior.
  - The **always** behavior executes and re-executes indefinitely, until the simulation is stopped.
- A module may contain an arbitrary number of **initial** or **always** behavioral statements.
- They execute concurrently with respect to each other, starting at time 0, and may interact through common variables.
- The activity associated with either type of behavioral statement can be controlled by a delay operator that waits for a certain time or by an event control operator that waits for certain conditions to become true or for specified events (changes in signals) to occur.
- Time delays specified with the # delay control operator are commonly used in single-pass behaviors. The delay control operator suspends execution of statements until a specified time has elapsed.
- Another operator @ is called the event control operator and is used to suspend activity until an event occurs. An event can be an unconditional change in a signal value (e.g., @ A) or a specified transition of a signal value (e.g., @ (**posedge** clock)).
- The general form of this type of statement is

```
always @ (event control expression)
  begin
    // Procedural assignment statements that execute when the
   condition is met
  end
```

- The event control expression specifies the condition that must occur to launch execution of the procedural assignment statements. The variables in the left-hand side of the procedural statements must be of the **reg** data type and must be declared as such. The right-hand side can be any expression that produces a value using Verilog-defined operators.
- The event control expression (also called the sensitivity list) specifies the events that must occur to initiate execution of the procedural statements associated with the **always** block.
- Statements within the block execute sequentially from top to bottom. After the last statement executes, the behavior waits for the event control expression to be satisfied. Then the statements are executed again.
- The sensitivity list can specify level-sensitive events, edge-sensitive events, or a combination of the two. In practice, designers do not make use of the third option, because this third form is not one that synthesis tools are able to translate into physical hardware.
- Level-sensitive events occur in combinational circuits and in latches. For example, the statement

```
always @ (A or B or C)
```

will initiate execution of the procedural statements in the associated **always** block if a change occurs in A, B, or C.

- In synchronous sequential circuits, changes in flip-flops occur only in response to a transition of a clock pulse. The transition may be either a positive edge or a negative edge of the clock, but not both. Verilog HDL takes care of these conditions by providing two keywords: **posedge** and **negedge**. For example, the expression

  ```
  always @(posedge clock or negedge reset)
  ```

  will initiate execution of the associated procedural statements only if the clock goes through a positive transition or if reset goes through a negative transition.

- There are two kinds of procedural assignments: blocking and nonblocking.
  - Blocking assignment
    * Blocking assignments use the symbol (=) as the assignment operator.
    * Blocking assignment statements are executed sequentially in the order they are listed in a block of statements. For example

      ```
      B=A
      C=B+1
      ```

      The first statement transfers the value of A into B. The second statement increments the value of B and transfers the new value to C. At the completion of the assignments, C contains the value of A + 1.
    * **Use blocking assignments when sequential ordering is imperative and in cyclic behavior that is level sensitive** (i.e., in combinational logic).
  - Nonblocking assignment
    * Nonblocking assignments use (<=) as the operator.
    * Nonblocking assignments are executed concurrently by evaluating the set of expressions on the right-hand side of the list of statements; they do not make assignments to their left-hand sides until all of the expressions are evaluated. For example

      ```
      B <= A
      C <= B + 1
      ```

      When the statements are executed, the expressions on the right-hand side are evaluated and stored in a temporary location. The value of A is kept in one storage location and the value of B + 1 in another. After all the expressions in the block are evaluated and stored, the assignment to the targets on the left-hand side is made. In this case, C will contain the original value of B, plus 1.

∗ **Use nonblocking assignments when modeling concurrent execution** (e.g., edge-sensitive behavior such as synchronous, concurrent register transfers) **and when modeling latched behavior**.

∗ Nonblocking assignments are imperative in dealing with register transfer level design. They model the concurrent operations of physical hardware synchronized by a common clock.

HDL Models of Flip-Flops and Latches

- **D-Latch**

```
// Description of D latch
  module D_latch (Q, D, enable);
    output Q;
    input D, enable;
    reg  Q;
    always @(enable or D)
      if (enable) Q <= D; // Same as: if (enable == 1)
  endmodule



// Alternative syntax (Verilog 2001, 2005)
  module D_latch (output reg Q, input enable, D);
    always @(enable, D)
      if (enable) Q <= D;  // No action if enable not
asserted
  endmodule
```

- **D-Type Flip-Flop**

```
// D flip-flop without reset
  module D_FF (Q, D, Clk);
    output Q;
    input D, Clk;
    reg Q;
    always @ (posedge Clk)
      Q <= D;
  endmodule
```

```
// D flip-flop with asynchronous reset (V2001, V2005)
  module DFF (output reg Q, input D, Clk, rst);
    always @ (posedge Clk, negedge rst)
      if (!rst) Q <= 1'b0;  // Same as: if (rst == 0)
      else Q <= D;
  endmodule
```

- The second module includes an asynchronous reset input in addition to the synchronous clock.
- A specific form of an **if** statement is used to describe such a flip-flop so that the model can be synthesized by a software tool.
- The event expression after the @ symbol in the **always** statement may have any number of edge events, either **posedge** or **negedge**.
- For modeling hardware, one of the events must be a clock event. The remaining events specify conditions under which asynchronous logic is to be executed.
- The designer knows which signal is the clock, but clock is not an identifier that software tools automatically recognize as the synchronizing signal of a circuit. The tool must be able to infer which signal is the clock, so you need to write the description in a way that enables the tool to infer the clock correctly. The rules are simple to follow:
  1. Each **if** or **else if** statement in the procedural assignment statements is to correspond to an asynchronous event,
  2. The last **else** statement corresponds to the clock event, and
  3. The asynchronous events are tested first.

  There are two edge events in the second module of HDL. The **negedge** rst (reset) event is asynchronous, since it matches the **if** (! rst) statement. As long as rst is 0, Q is cleared to 0. If Clk has a positive transition, its effect is blocked. Only if rst = 1 can the **posedge** clock event synchronously transfer D into Q.
- Hardware always has a reset signal. It is strongly recommended that all models of edge-sensitive behavior include a reset (or preset) input signal; otherwise, the initial state of the flip-flops of the sequential circuit cannot be determined. A sequential circuit cannot be tested with HDL simulation unless an initial state can be assigned with an input signal.

• **Alternative Flip-Flop Models**

- Following examples describes the construction of a T or JK flip-flop from a D flip-flop and gates. The circuit is described with the characteristic equations of the flip-flops:

$$Q(t+1) = Q \oplus T \qquad \text{for a T flip-flop}$$

$$Q(t+1) = JQ' + K'Q \qquad \text{for a JK flip-flop}$$

```
// T flip-flop from D flip-flop and gates
 module TFF (Q, T, Clk, rst);
   output Q;
   input T, Clk, rst;
   wire DT;
   assign DT = Q ^ T ;        // Continuous assignment
   // Instantiate the D flip-flop
   DFF TF1 (Q, DT, Clk, rst);
 endmodule
```

```
// JK flip-flop from D flip-flop and gates (V2001, 2005)
 module JKFF (output reg Q, input J, K, Clk, rst);
   wire JK;
   assign JK = (J & ~Q)|(~K & Q);
   // Instantiate D flip-flop
   DFF JK1 (Q, JK, Clk, rst);
 endmodule

// D flip-flop (V2001, V2005)
 module DFF (output reg Q, input D, Clk, rst);
   always @ (posedge Clk, negedge rst)
     if (!rst) Q <= 1'b0;
     else Q <= D;
 endmodule
```

– Following example shows another way to describe a JK flip-flop. Here, we describe the flip-flop by using the characteristic table rather than the characteristic equation. The case multiway branch condition checks the two-bit number obtained by concatenating the bits of J and K. The case expression ({J, K}) is evaluated and compared with the values in the list of statements that follows. The first value that matches the true condition is executed. Since the concatenation of J and K produces a two-bit number, it can be equal to 00, 01, 10, or 11. The first bit gives the value of J and the second the value of K. The four possible conditions specify the value of the next state of Q after the application of a positive-edge clock.

```
// Functional description of JK flip-flop (V2001, 2005)
 module JK_FF (input J, K, Clk, output reg Q, output Q_b);
   assign Q_b = ~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;
     endcase
 endmodule
```

State diagram-Based HDL Models

- **Example : Zero detector (Mealy Model)**
  - This is Mealy Model.
  - This example detects a 0 following a sequence of 1s in a serial bit stream.

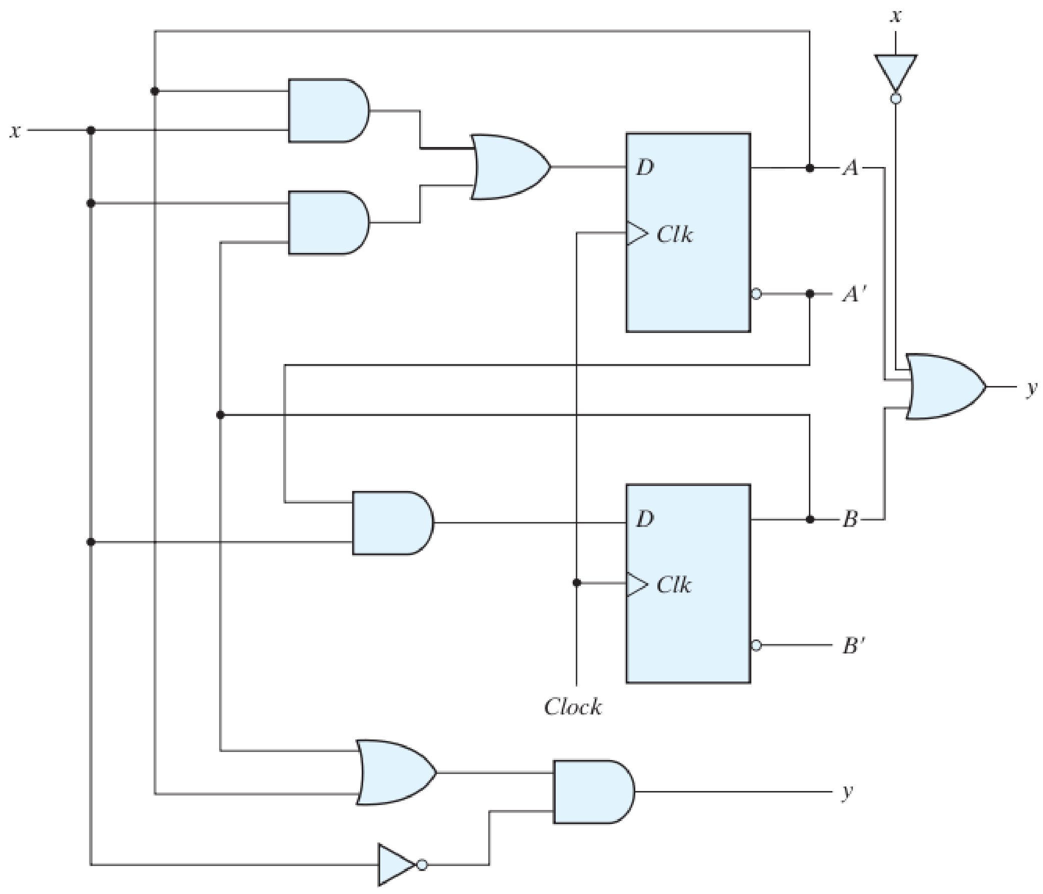

Figure 1: Zero detector circuit diagram

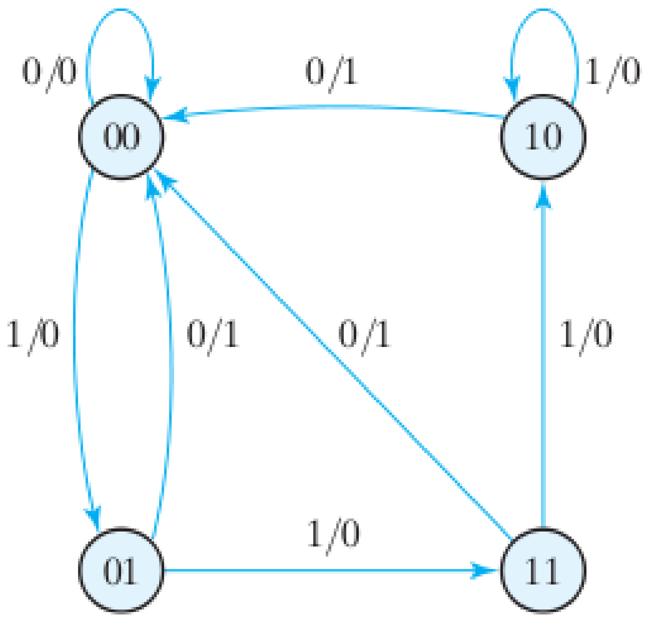

Figure 2: Zero detector state diagram

```
// Mealy FSM zero detector Verilog 2001, 2005 syntax
  module Mealy_Zero_Detector (
    output reg y_out,
    input x_in, clock, reset
    );
    reg [1: 0] state, next_state;
    parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'
b11;

    always @ (posedge clock, negedge reset) // Verilog
2001, 2005 syntax
      if (reset == 0) state <= S0;
      else state <= next_state;

    always @ (state, x_in)      // Form the next state
      case (state)
        S0:if (x_in) next_state = S1; else next_state = S0;
        S1:if (x_in) next_state = S3; else next_state = S0;
        S2:if (~x_in) next_state = S0; else next_state = S2
;
        S3:if (x_in) next_state = S2; else next_state = S0;
      endcase

    always @ (state, x_in) // Form the Mealy output
      case (state)
        S0:y_out = 0;
        S1, S2, S3: y_out = ~x_in;
    endcase
  endmodule

  module t_Mealy_Zero_Detector;
    wire t_y_out;
    reg t_x_in, t_clock, t_reset;

    Mealy_Zero_Detector M0(t_y_out, t_x_in, t_clock,
t_reset);

    initial #200 $finish;

    initial
      begin
        t_clock = 0;
        forever #5 t_clock = ~t_clock;
      end

    initial
      fork
        t_reset = 0;
        #2 t_reset = 1;
        #87 t_reset = 0;
        #89 t_reset = 1;
        #10 t_x_in = 1;
        #30 t_x_in = 0;
        #40 t_x_in = 1;
        #50 t_x_in = 0;
        #52 t_x_in = 1;
        #54 t_x_in = 0;
        #70 t_x_in = 1;
        #80 t_x_in = 1;
        #70 t_x_in = 0;
        #90 t_x_in = 1;
        #100 t_x_in = 0;
        #120 t_x_in = 1;
        #160 t_x_in = 0;
        #170 t_x_in = 1;
      join
  endmodule
```

- Its Verilog model uses three always blocks that execute concurrently and interact through common variables.
- The first **always** statement resets the circuit to the initial state S0 = 00 and specifies the synchronous clocked operation. The statement state <= next_state is synchronized to a positive-edge transition of the clock. This means that any change in the value of next_state in the second always block can affect the value of state only as a result of a **posedge** event of clock.
- The second **always** block determines the value of the next state transition as a function of the present state and input. The value assigned to state by the nonblocking assignment is the value of next_state immediately before the rising edge of clock.
- The third **always** block specifies the output as a function of the present state and the input. Note that the value of output y_out may change if the value of input x_in changes while the circuit is in any given state.
- Summary
  * At every rising edge of clock, if reset is not asserted, the state of the machine is updated by the first **always** block; when state is updated by the first **always** block, the change in state is detected by the sensitivity list mechanism of the second **always** block; then the second **always** block updates the value of next_state (it will be used by the first **always** block at the

next tick of the clock); the third **always** block also detects the change in state and updates the value of the output.

    * In addition, the second and third **always** blocks detect changes in x_in and update next_state and y_out accordingly.

- The description of waveforms in the test bench uses the **fork** . . . **join** construct. Statements with the **fork** . . . **join** block execute in parallel, so the time delays are relative to a common reference of t = 0, the time at which the block begins execution.
- It is usually more convenient to use the **fork** . . . **join** block instead of the **begin** . . . **end** block in describing waveforms.
- How does Verilog model Mealy_Zero_Detector correspond to hardware?
    * The first **always** block corresponds to a D flip-flop implementation of the state register.
    * The second **always** block is the combinational logic block describing the next state.
    * The third **always** block describes the output combinational logic of the zero-detecting Mealy machine.
    * The register operation of the state transition uses the nonblocking assignment operator (<=) because the (edge-sensitive) flip-flops of a sequential machine are updated concurrently by a common clock.
    * The second and third **always** blocks describe combinational logic, which is level sensitive, so they use the blocking (=) assignment operator. Their sensitivity lists include both the state and the input because their logic must respond to a change in either or both of them.

- **Example : Moore Model**

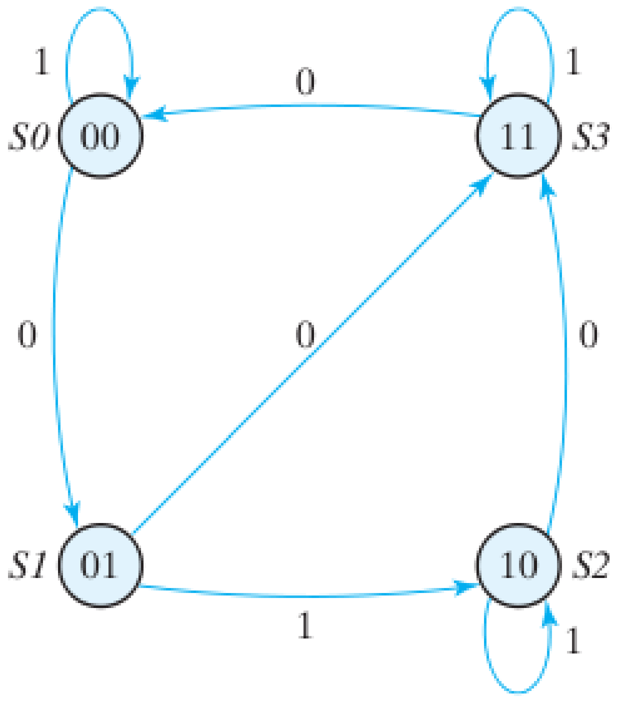

Figure 3: Moore Model state diagram

–

```
// Moore model FSM  Verilog 2001, 2005 syntax
 module Moore_Model (
   output [1: 0] y_out,
   input  x_in, clock, reset
   );
   reg [1: 0] state;
   parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'
b11;

   always @ (posedge clock, negedge reset)
     if (reset == 0) state <= S0;     // Initialize to
state S0
     else  case (state)
           S0:if (~x_in) state <= S1; else state <= S0;
           S1:if (x_in) state <= S2; else state <= S3;
           S2:if (~x_in) state <= S3; else state <= S2;
           S3:if (~x_in) state <= S0; else state <= S3;
         endcase

   assign y_out = state; // Output of flip-flops
endmodule
```

- **Structural Description of Clocked Sequential Circuits**

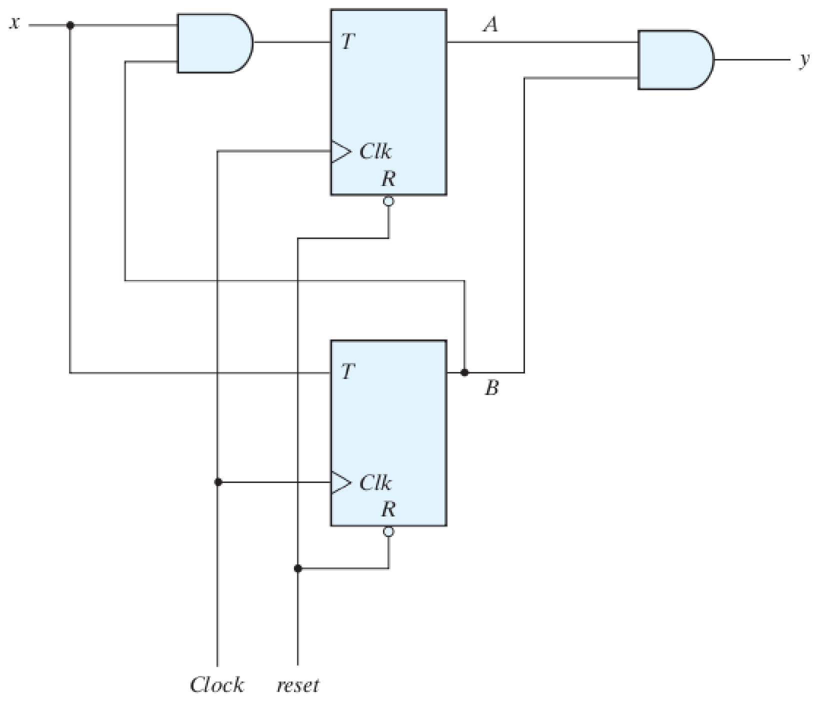

Figure 4: Binary counter circuit diagram

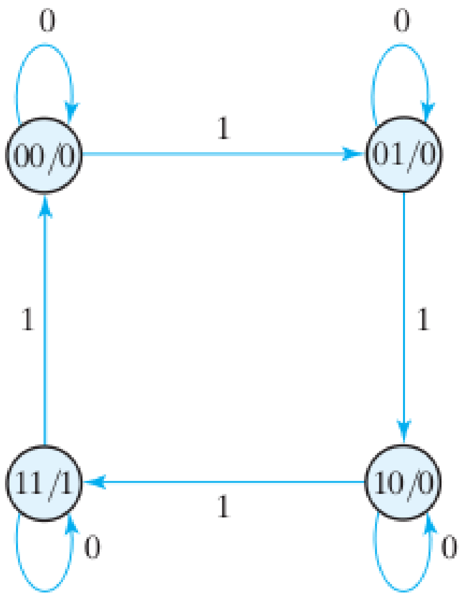

Figure 5: Binary counter state diagram

- Combinational logic circuits can be described in Verilog by a connection of gates (primitives and UDPs), by dataflow statements (continuous assignments), or by level- sensitive cyclic behaviors (**always** blocks).
- Sequential circuits are composed of combinational logic and flip-flops, and their HDL models use sequential UDPs and behavioral statements (edge-sensitive cyclic behaviors) to describe the operation of flip-flops.
- One way to describe a sequential circuit uses a combination of dataflow and behavioral statements. The flip-flops are described with an **always** statement. The combinational part can be described with **assign** statements and Boolean equations.
- The separate modules can be combined to form a structural model by instantiation within a **module**.

```
// State-diagram-based model (V2001, 2005)
  module Moore_Model_counter (
    output y_out,
    input x_in, clock, reset
    );
    reg [1: 0]  state;
    parameter   S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'
b11;

    always @ (posedge clock, negedge reset)
      if (reset == 0) state <= S0;  // Initialize to state
S0
      else case (state)
        S0: if (x_in) state <= S1; else state <= S0;
        S1: if (x_in) state <= S2; else state <= S1;
        S2: if (x_in) state <= S3; else state <= S2;
        S3: if (x_in) state <= S0; else state <= S3;
      endcase

    assign y_out = (state == S3); // Output of flip-flops

  endmodule

// Structural model
  module Moore_Model_STR_counter (
    output  y_out, A, B,
    input   x_in, clock, reset
    );
    wire TA, TB;

// Flip-flop input equations
    assign TA = x_in & B;
    assign TB = x_in;

// Output equation
    assign y_out = A & B;

// Instantiate Toggle flip-flops
    Toggle_flip_flop_3 M_A (A, TA, clock, reset);
    Toggle_flip_flop_3 M_B (B, TB, clock, reset);

  endmodule

  module Toggle_flip_flop (Q, T, CLK, RST_b);
    output  Q;
    input   T, CLK, RST_b;
    reg   Q;
    always @ (posedge CLK, negedge RST_b)
      if (RST_b == 0) Q <= 1'b0;
      else if (T) Q <= ~Q;
  endmodule

// Alternative model using characteristic equation
  // module Toggle_flip_flop (Q, T, CLK, RST_b);
    // output Q;
    // input T, CLK, RST_b;
    // reg  Q;
    // always @ (posedge CLK, negedge RST)
      // if (RST_b == 0) Q <= 1'b0;
      // else Q <= Q ^ T;
  // endmodule


  module t_Moore_counter;
    wire t_y_out_2, t_y_out_1;
    reg t_x_in, t_clock, t_reset;
    Moore_Model_counter M1(t_y_out_1, t_x_in, t_clock,
t_reset);
    Moore_Model_STR_counter M2 (t_y_out_2, A, B, t_x_in,
t_clock, t_reset);

    initial #200 $finish;

    initial
      begin
        t_reset = 0;
        t_clock = 0;
        #5 t_reset = 1;
        repeat (16)
          #5 t_clock = ~t_clock;
      end

    initial
      begin
        t_x_in = 0;
        #15 t_x_in = 1;
        repeat (8)
          #10 t_x_in = ~t_x_in;
      end
  endmodule
```

HDL for Registers and Counters

- Registers and counters can be described in Verilog at either the behavioral or the structural level.
- Behavioral modeling describes only the operations of the register, as prescribed by a function table, without a preconceived structure.
- A structurallevel description shows the circuit in terms of a collection of components such as gates, flip flops, and multiplexers. The various components are instantiated to form a hierarchical description of the design similar to a representation of a multilevel logic diagram.
- When a machine is complex, a hierarchical description creates a physical partition of the machine into simpler and more easily described units.
- **Shift Register**

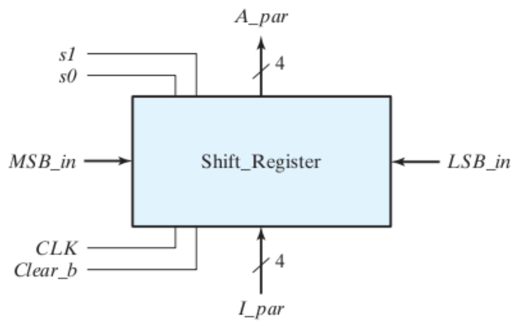

Figure 6: Universal shift register block diagram

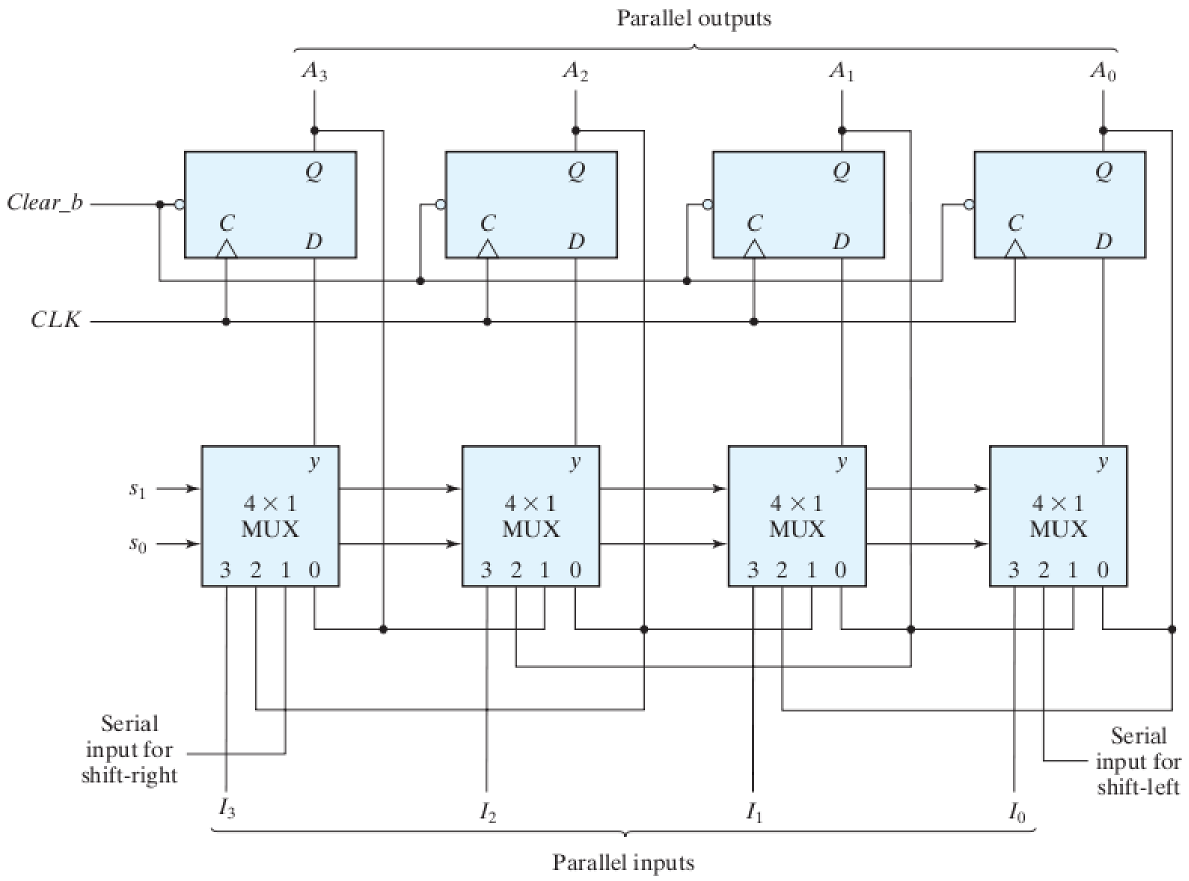

Figure 7: Universal shift register structure

s_1	s_0	**Register Operation**
0	0	No change
0	1	Shift right
1	0	Shift left
1	1	Load parallel data

– **Behavioral model**

```
// Behavioral description of a 4-bit universal shift
 register
module Shift_Register_4_beh (
  output reg  [3: 0]  A_par,  // Register output
  input    [3: 0]  I_par,  // Parallel input
  input    s1, s0,    // Select inputs
  MSB_in, LSB_in,     // Serial inputs
  CLK, Clear_b  // Clock and Clear
  );

  always @ (posedge CLK, negedge Clear_b)
    if (Clear_b == 0) A_par <= 4 b0000;
    else
      case ({s1, s0})
        2'b00: A_par <= A_par;      // No change
        2'b01: A_par <= {MSB_in, A_par[3: 1]}; // Shift
 right
        2'b10: A_par <= {A_par[2: 0], LSB_in}; // Shift
 left
        2'b11: A_par <= I_par; // Parallel load of input
      endcase
endmodule
```

– **Structural model**

```
// Structural description of a 4-bit universal shift
 register
  module Shift_Register_4_str (
    output [3: 0] A_par,  // Parallel output
```

```
      input [3: 0] I_par ,    // Parallel input
    input s1 , s0 , // Mode select
    input MSB_in , LSB_in , CLK , Clear_b // Serial inputs ,
clock , clear
    );

    // bus for mode control
    assign [1:0] select = {s1 , s0};
    // Instantiate the four stages
    stage ST0 (A_par [0] , A_par [1] , LSB_in , I_par [0] , A_par
[0] , select , CLK , Clear_b);
    stage ST1 (A_par [1] , A_par [2] , A_par [0] , I_par [1] ,
A_par [1] , select , CLK , Clear_b);
    stage ST2 (A_par [2] , A_par [3] , A_par [1] , I_par [2] ,
A_par [2] , select , CLK , Clear_b);
    stage ST3 (A_par [3] , MSB_in , A_par [2] , I_par [3] , A_par
[3] , select , CLK , Clear_b);
  endmodule

  // One stage of shift register
  module stage (i0 , i1 , i2 , i3 , Q , select , CLK , Clr_b);
    input   i0 , // circulation bit selection
            i1 , // data from left neighbor or serial input
for shift-right
            i2 , // data from right neighbor or serial input
 for shift-left
            i3; // data from parallel input
    output  Q;
    input [1: 0] select; // stage mode control bus
    input CLK , Clr_b; // Clock , Clear for flip-flops
    wire mux_out;

    // instantiate mux and flip-flop
    Mux_4_x_1 M0 (mux_out , i0 , i1 , i2 , i3 , select);
    D_flip_flop M1 (Q , mux_out , CLK , Clr_b);
  endmodule

  // 4x1 multiplexer      // behavioral model
  module Mux_4_x_1 (mux_out , i0 , i1 , i2 , i3 , select);
    output  mux_out;
    input i0 , i1 , i2 , i3;
    input [1: 0] select;
    reg mux_out;
    always @ (select, i0, i1, i2, i3)
      case (select)
        2'b00:  mux_out = i0;
        2'b01:  mux_out = i1;
        2'b10:  mux_out = i2;
        2'b11:  mux_out = i3;
      endcase
  endmodule

  // Behavioral model of D flip-flop
  module D_flip_flop (Q, D, CLK, Clr_b);
    output  Q;
    input D, CLK, Clr;
    reg Q;

    always @ (posedge CLK, negedge Clr_b)
      if (!Clr_b) Q <= 1'b0; else Q <= D;
  endmodule
```

- **Synchronous Counter**

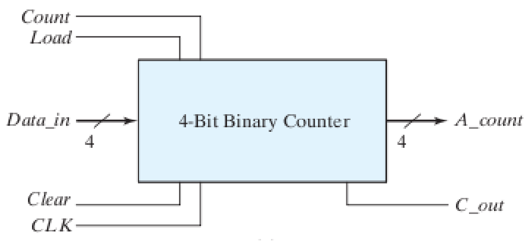

Figure 8: binary counter with parallel load block diagram

Clear	CLK	Load	Count	Function
0	X	X	X	Clear to 0
1	$\uparrow$	1	X	Load inputs
1	$\uparrow$	0	1	Count next binary state
1	$\uparrow$	0	0	No change

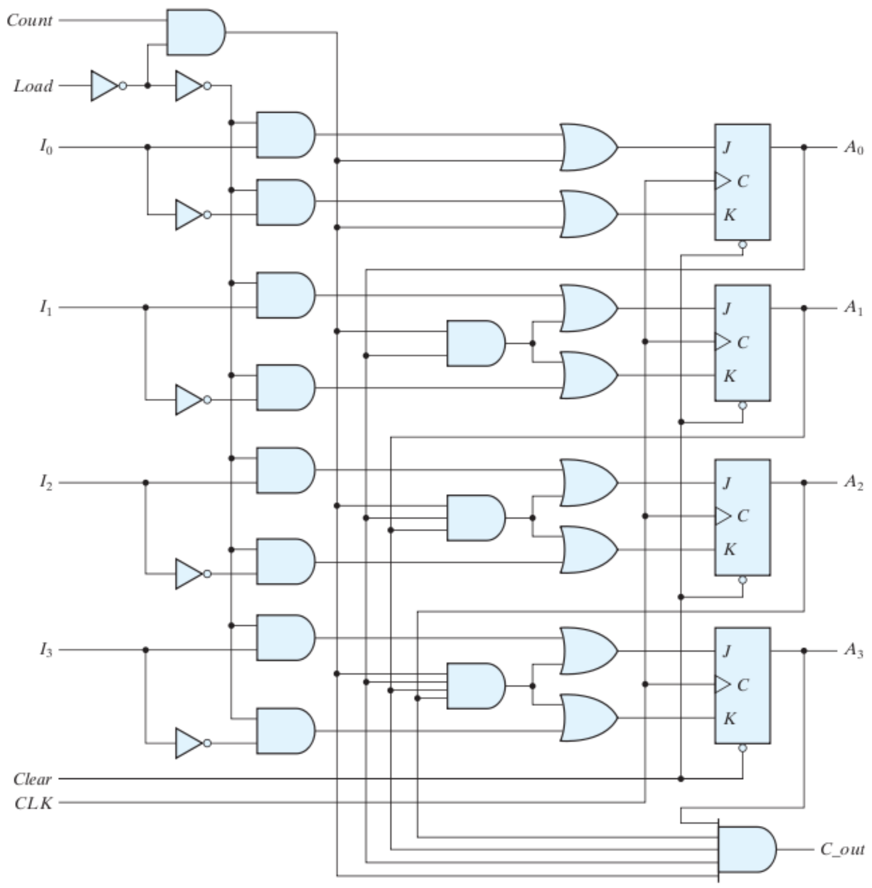

Figure 9: binary counter with parallel load structure

```
// Four-bit binary counter with parallel load (V2001, 2005)
module Binary_Counter_4_Par_Load (
  output reg [3: 0]   A_count,  // Data output
  output  C_out,  // Output carry
  input [3: 0]  Data_in,  // Data input
  input Count,  // Active high to count
        Load,   // Active high to load
        CLK,  // Positive-edge sensitive
        Clear_b   // Active low
  );

  assign C_out = Count && (~Load) && (A_count == 4'b1111);

  always @ (posedge CLK, negedge Clear_b)
    if (~Clear_b)   A_count <= 4'b0000;
    else if (Load)  A_count <= Data_in;
    else if (Count) A_count <= A_count + 1'b1;
    else A_count <= A_count; // redundant statement
endmodule
```

- **Ripple Counter**

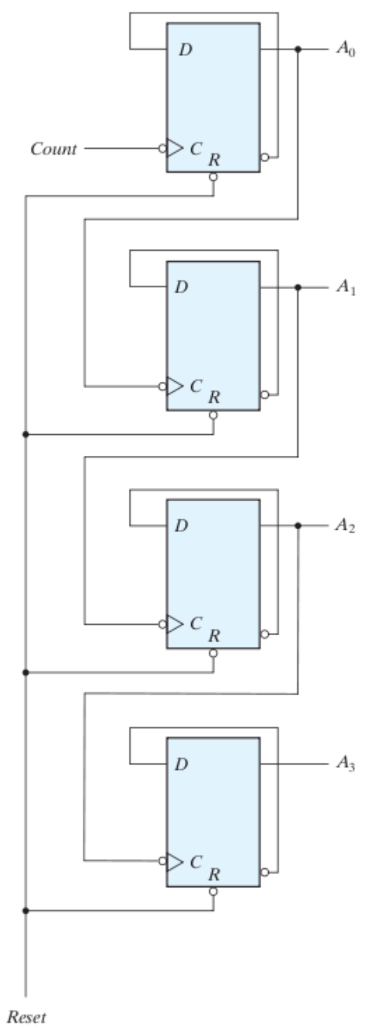

Figure 10: 4-bit binary counter with D flip-flop

```
// Ripple counter

‘timescale 1ns / 100 ps

module Ripple_Counter_4bit (A3, A2, A1, A0, Count, Reset);
  output A3, A2, A1, A0;
  input Count, Reset;

  // Instantiate complementing flip-flop
  Comp_D_flip_flop F0 (A0, Count, Reset);
  Comp_D_flip_flop F1 (A1, A0, Reset);
  Comp_D_flip_flop F2 (A2, A1, Reset);
  Comp_D_flip_flop F3 (A3, A2, Reset);
endmodule
```

```
// Complementing flip-flop with delay
// Input to D flip-flop = Q'
module Comp_D_flip_flop (Q, CLK, Reset);
  output Q;
  input   CLK, Reset;
  reg    Q;

  always @ (negedge CLK, posedge Reset)
    if (Reset) Q <= 1'b0;
    else Q <= #2 ~Q;  // intra-assignment delay
endmodule

// Stimulus for testing ripple counter
module t_Ripple_Counter_4bit;
  reg    Count;
  reg    Reset;
  wire  A0, A1, A2, A3;

  // Instantiate ripple counter
  Ripple_Counter_4bit M0 (A3, A2, A1, A0, Count, Reset);

  always  #5 Count = ~Count;

  initial
    begin
      Count = 1'b0;
      Reset = 1'b1;
      #4 Reset = 1'b0;
    end

  initial #170 $finish;
endmodule
```