Modules and Ports

- Each port in the port list is defined as input, output, or inout, based on the direction of the port signal.
- Note that all port declarations are implicitly declared as wire in Verilog.
- Thus, if a port is intended to be a wire, it is sufficient to declare it as output, input, or inout.
- However, if output ports hold their value, they must be declared as reg.

D Flip-flop

```
// module D_FF with asynchronous reset
module D_FF(q, d, clk, reset);
output q;
input d, clk, reset;
reg q;

always @(posedge reset or negedge clk)
  if (reset)
    q <= 1'b0;
  else
    q <= d;
endmodule
```


- Ports of the type input and inout cannot be declared as reg because reg variables store values and input ports should not store values but simply reflect the changes in the external signals they are connected to.

- **Inputs**
  - Internally, input ports must always be of the type net. Externally, the inputs can be connected to a variable which is a reg or a net.
- **Outputs**
  - Internally, outputs ports can be of the type reg or net. Externally, outputs must always be connected to a net. They cannot be connected to a reg.
- **Inouts**
  - Internally, inout ports must always be of the type net. Externally, inout ports must always be connected to a net.

- **Unconnected ports**
- Verilog allows ports to remain unconnected.
  - `fulladd4 fa0(SUM, , A, B, C_IN); // Output port c_out is unconnected`

Connecting Ports to External Signals

Connecting by ordered list

- The signals to be connected must appear in the module instantiation in the same order as the ports in the port list in the module definition.

- **Connecting ports by name**
- For large designs where modules have, many ports, remembering the order of the ports in the module definition is impractical and error-prone.
- Verilog provides the capability to connect external signals to ports by the port names, rather than by position.
- The port connections can be specified in any order as long as the port name in the module definition correctly matches the external signal.

  Eg: fulladd4 fa_byname(.c_out(C_OUT), .sum(SUM), .b(B), .c_in(C_IN), .a(A),);
- Unconnected ports can be dropped.

  fulladd4 fa_byname(.sum(SUM), .b(B), .c_in(C_IN), .a(A),);

Modeling in Verilog

Gate Level Modeling

- Verilog has built in primitives like gates, transmission gates, and switches.
- The gates have one scalar output and multiple scalar inputs.
- The first terminal in the list of gate terminals is an output and the other terminals are inputs.
- The output of a gate is evaluated as soon as one of the inputs changes.

Verilog Primitives

- Just like drawing schematics, circuits of any arbitrary size can be made with primitive components.
- Primitive logic operators are
  - AND
  - OR
  - XOR
  - NOT
  - NAND, NOR, XNOR, BUF
    - Any XOR/XNOR will output an X (unknown) if any input is X or Z.
- Primitives are instantiated in a module
  - Eg: nand (out, in1, in2, in3, in4);
    - nand: primitive type. Verilog keyword

and	i1			
	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

(i2)

or	i1			
	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

(i2)

xor	i1			
	0	1	x	z
0	0	1	x	x
1	1	0	x	x
x	x	x	x	x
z	x	x	x	x

(i2)

nand	i1			
	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

(i2)

nor	i1			
	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

(i2)

xnor	i1			
	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

(i2)

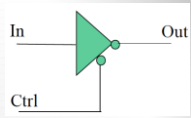
buf	in	out
	0	0
	1	1
	x	x
	z	x

not	in	out
	0	1
	1	0
	x	x
	z	x

- Primitive Pins Are Expandable
  - The number of pins for a primitive gate is defined by the number of nets connected to it.
- All gates except not and buf can have a variable number of inputs, but only one output.
- The not and buf gates can have a variable number of outputs, but only one input.

Primitives with Control Inputs

- Bufif0
- Bufif1
- Notif1
- Notif0



Output is Z (high impedance) if control is not asserted.
Output is X if control IS asserted and input is X or Z.

```
//Instantiation of bufif gates.
        bufif1 b1 (out, in, ctrl);
        bufif0 b0 (out, in, ctrl);
//Instantiation of notif gates
        notif1 n1 (out, in, ctrl);
        notif0 n0 (out, in, ctrl);
```


bufif1		ctrl			
		0	1	x	z
in	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

notif1		i1			
		0	1	x	z
i2	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

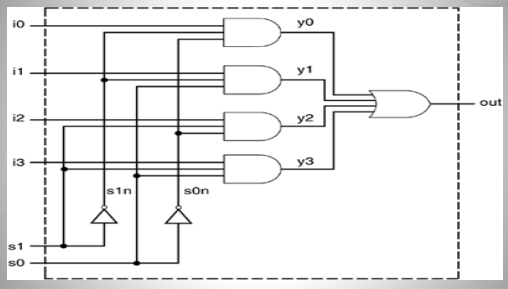
bufif0		i1			
		0	1	x	z
i2	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

notif0		i1			
		0	1	x	z
i2	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

Array of Instances

```
wire [3:0] OUT, IN1, IN2;
// basic gate instantiations.
nand n_gate[3:0](OUT, IN1, IN2);
// This is equivalent to the following 4 instantiations
        nand n_gate0(OUT[0], IN1[0], IN2[0]);
        nand n_gate1(OUT[1], IN1[1], IN2[1]);
        nand n_gate2(OUT[2], IN1[2], IN2[2]);
        nand n_gate3(OUT[3], IN1[3], IN2[3]);
```


- Write a gate level model for a 4: 1 multiplexer



```
/* Module 4-to-1 multiplexer. Port list is taken exactly from the I/O diagram.*/
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;
// Gate instantiations
// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);
// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
// 4-input or gate instantiated
or (out, y0, y1, y2, y3);
endmodule
```


- Write the structural code for a 1 bit full adder
- Write the structural code for a 4 bit ripple carry adder using above adders

Gate Delays

- In real circuits, logic gates have delays associated with them.
- Gate delays allow the Verilog user to specify delays through the logic circuits.
- Pin-to-pin delays can also be specified in Verilog.
  - Rise Delay
  - Fall Delay
  - Turn – off Delay

Gate Delays (Cont…)

- **Rise delay:** Associated with a gate output transition to a 1 from another value.

- **Fall Delay:** Associated with a gate output transition to a 0 from another value.

- **Turn-off delay:** Associated with a gate output transition to the high impedance value (z) from another value.
- If the value changes to x, the minimum of the three delays is considered.

Gate Delays (Cont…)

Types of Delay Specification

```
// Delay of delay_time for all transitions
and #(delay_time) a1(out, i1, i2);

// Rise and Fall Delay Specification.
and #(rise_val, fall_val) a2(out, i1, i2);

// Rise, Fall, and Turn-off Delay Specification
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions
and #(4,6) a2(out, i1, i2); // Rise = 4, Fall = 6
bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5
```


Gate Delays (Cont…)

Min /Typ /Max value

- The min/typ/max value is the minimum/typical/max delay value that the designer expects the gate to have.

```
// if only One delay is specified
and #(4:5:6) a1(out, i1, i2);
        // mindelay= 4
        // typdelay= 5
        // maxdelay= 6
```

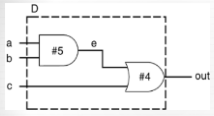
Gate Delays (Cont…)

```
// Two delays are specified
and #(3:4:5, 5:6:7) a2(out, i1, i2);
// mindelays, rise= 3, fall= 5, turn-off = min(3,5)
// typdelays, rise= 4, fall= 6, turn-off = min(4,6)
// maxdelays, rise= 5, fall= 7, turn-off = min(5,7)

// Three delays
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1,i2);
// mindelays, rise= 2 fall= 3 turn-off = 4
// typdelays, rise= 3 fall= 4 turn-off = 5
// maxdelays, rise= 4 fall= 5 turn-off = 6
```


Gate Delays (Cont…)

Verilog Definition for Module D with Delay

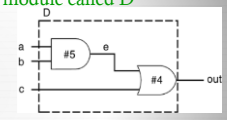


Gate Delays (Cont…)

Verilog Definition for Module D with Delay

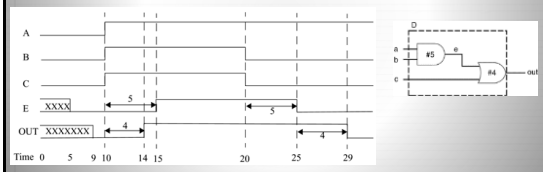
```
// Define a simple combination module called D

module D (out, a, b, c);
output out;
input a,b,c;
wire e;
and #(5) a1(e, a, b); //Delay of 5 on gate a1
or #(4) o1(out, e,c); //Delay of 4 on gate o1
endmodule
```



Gate Delays (Cont…)

1. The outputs E and OUT are initially unknown.
2. At time 10, after A, B, and C all transition to 1, OUT transitions to 1 after a delay of 4 time units and E changes value to 1 after 5 time units.
3. At time 20, B and C transition to 0. E changes value to 0 after 5 time units, and OUT transitions to 0, 4 time units after E changes.



Dataflow (RTL) Modeling

- For small circuits, the gate-level modeling approach works very well because the number of gates is limited and the designer can instantiate and connect every gate individually.
- In complex designs the number of gates is very large.
- Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates.

Continuous Assignments

- A continuous assignment is the most basic statement in dataflow modeling, used to ***drive a value onto a net***.
- The assignment statement starts with the keyword ***assign***.

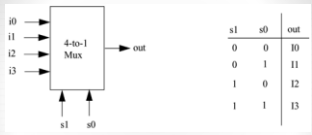
  Syntax: assign [drive_strength][delay] list_of_net_assignments;
  - The default value for drive strength is strong1 and strong0
- Continuous assignments are always active.
- The assignment expression is evaluated as soon as one of the right-hand-side operands changes and the value is assigned to the left-hand-side net.
- The operands on the right-hand side can be registers or nets or function calls.
- Delay values are used to control the time when a net is assigned the evaluated value.

```
// Continuous assign. out is a net. i1 and i2 are nets.
assign out = i1 & i2;

// Continuous assign for vector nets. addr is a 16-bit vector net
// addr1 and addr2 are 16-bit vector registers.
assign addr[15:0] = addr1_bits[15:0] ^ addr2_bits[15:0];

// Concatenation. Left-hand side is a concatenation of a scalar
// net and a vector net.
assign {co, s[3:0]} = a[3:0] + b[3:0] + cn;
```


Dataflow model multiplexer



Dataflow model multiplexer

// 4-to-1 Multiplexer, Using Logic Equations

```
// Module 4-to-1 multiplexer using data flow_ logic equation
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
//Logic equation for out
assign out = (~s1 & ~s0 & i0) | (~s1 & s0 & i1) |
                        (s1 & ~s0 & i2) | (s1 & s0 & i3) ;
endmodule
```


Dataflow model multiplexer

- **4-to-1 Multiplexer, Using Conditional Operators**

```
// Module 4-to-1 multiplexer using data flow. Conditional operator.
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```


Implicit Continuous Assignment

- Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.
- There can be only one implicit declaration assignment per net because a net is declared only once.

```
//Regular continuous assignment
wire out;
assign out = in1 & in2;
//Same effect is achieved by an implicit continuous assignment
wire out = in1 & in2;
```


Implicit Net Declaration

- If a signal name is used to the left of the continuous assignment, an implicit net declaration will be inferred for that signal name.
- If the net is connected to a module port, the width of the inferred net is equal to the width of the module port.

```
// Continuous assign. out is a net.
wire i1, i2;
assign out = i1 & i2; //Note that out was not declared as a wire
                      //but an implicit wire declaration for out
                      //is done by the simulator
```

Delays

- A delay control expression specifies the time duration between initially encountering the statement and when the statement actually executes.
  e.g. #10 A = A + 1;
- There are different ways to specify delays in continuous assignments.

- **Regular Assignment Delay:** This is the most commonly used method.
  e.g.- assign #10 q = x + y;
- **Implicit Continuous Assignment Delay:**
  e.g. – wire #10 out = a ^ b;
  // which is equivalent to the following:
  wire out;
  assign #10 out = a ^ b;
- **Net Declaration Delay:** The delay can be put on the net in declaration itself.
  e.g. – wire #10 out;
  assign out = a & b;
  // which is equivalent to the following:
  wire out;
  assign #10 out = a & b;

Behavioral Modeling

- Behavioral modeling represents the circuit at a very high level of abstraction.
- Verilog provides designers the ability to describe design functionality in an algorithmic manner.
- The designer can describe the behavior of the circuit.

Structured Procedures

- There are two structured procedure statements in Verilog: *always* and *initial*.
- These statements are the two most basic statements in behavioral modeling.
- All other behavioral statements can appear only inside these structured procedure statements.
- Group of statements coming under always and initial blocks are called ***procedural blocks***.
- Assignment inside procedural blocks are called ***procedural assignment***.

Procedural Blocks

- Procedural blocks are the basic components for behavioral modeling.

```
initial
begin
… procedural statements …
end
```

- Runs when simulation starts
- Terminates when control reaches the end
- Good for providing stimulus

```
always
begin
… procedural statements …
end
```

- Runs when simulation starts
- Restarts when control reaches the end
- Good for modeling / specifying hardware

Procedural Blocks (Cont..)

- Procedural blocks are like concurrent processes.
- All blocks execute in parallel.
- Statements in a block are executed sequentially, but all within one unit of simulated time. (unless delay is specified)
- **initial block**
  - Executes only once.
- **always block**
  - Executes repeatedly.
  - It must have timing control, otherwise it become **INFINITE LOOP**

initial Statement

- All statements inside an *initial* statement constitute an initial block.
- An initial block starts at time 0, executes exactly once during a simulation, and then ***does not execute*** again.
- If there are multiple initial blocks, each block starts to execute concurrently at time 0.
- The initial blocks are typically used for *initialization.*

initial Statement - Example

```
module stimulus;
reg x,y, a,b, m;
initial
    m = 1'b0;
initial
begin
    #5 a = 1'b1;
    #25 b = 1'b0;
end
initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end
Initial
    #50 $finish;
endmodule
```


initial Statement - Example

```
module stimulus;
reg x,y, a,b, m;
initial
    m = 1'b0;
initial
begin
    #5 a = 1'b1;
    #25 b = 1'b0;
end
initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end
Initial
    #50 $finish;
endmodule
```

Time	Statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish

Combined Variable Declaration and Initialization

Variables can be initialized when they are declared.

```
//The clock variable is defined first
reg clock;
//The value of clock is set to 0
initial clock = 0;

//Instead of the above method, clock variable can be initialized
//at the time of declaration
//This is allowed only for variables declared at module level.
reg clock = 0;
```


Combined Port/Data Declaration and Initialization

- The combined port/data declaration can also be combined with an initialization

```
module adder (sum, co, a, b, ci);
output reg [7:0] sum = 0; //Initialize 8 bit output sum
output reg co = 0; //Initialize 1 bit output co
input [7:0] a, b;
input ci;
--
--
endmodule
```



```
module adder (output reg [7:0] sum = 0, //Initialize 8 bit output
              output reg co = 0, //Initialize 1 bit output co
              input [7:0] a, b,
              input ci);
--
--
endmodule
```

always Statement

- All behavioral statements inside an always statement constitute an always block.
- The always statement starts at time 0 and *executes* the statements in the always block *continuously* in a looping fashion.
- This statement is used to model a block of activity that is repeated continuously in a digital circuit.

always Statement - Example

```
module clock_gen (clock);
output reg clock;
//Initialize clock at time zero
initial
    clock = 1'b0;
//Toggle clock every half-cycle (time period = 20)
always
    #10 clock = ~clock;
initial
    #1000 $finish;
endmodule
```


Procedural Assignments

- Procedural assignments update values of reg, integer, real, or time variables.
- The syntax for the simplest form of procedural assignment is shown below.
- assignment ::= variable_lvalue = [delay_or_event_control] expression
- There are two types of assignment statements are there in Verilog:
  - Blocking statements
  - Non-blocking statements.

Blocking Assignments

- It is a way of "blocking" the further statements until the current statement execution is completed.
- The blocking assignment operator is an equal sign (=).
- Evaluated and assigned in a single step.
- A blocking assignment must evaluate the RHS arguments and update the LHS expression of the blocking assignment without interruption from any other Verilog statement.
- The blocking assignment with timing delays on the RHS of the blocking operator, is considered to be a poor coding style.

Blocking Assignments (Cont..)

- A problem with blocking assignments occurs when
  - The RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block, and
  - Both equations are scheduled to execute in the same simulation time step, such as on the same clock edge.
- If blocking assignments are not properly ordered, a race condition can occur.
- When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

Blocking Assignments - Example

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
   x = 0; y = 1; z = 1;   //Scalar assignments
   count = 0;             //Assignment to integer variables
   reg_a = 16'b0; reg_b = reg_a;        //initialize vectors
   #15 reg_a[2] = 1'b1;          //Bit select assignment with delay
  #10 reg_b[15:13] = {x, y, z} //Assign result of concatenation to
                               // part select of a vector
   count = count+ 1;   //Assignment to an integer (increment)
end
```

Blocking Assignments - Example

- All statements x = 0 through reg_b = reg_a are executed at time 0
- Statement reg_a[2] = 0 at time = 15
- Statement reg_b[15:13] = {x, y, z} at time = 25
- Statement count = count + 1 at time = 25
- Since there is a delay of 15 and 10 in the preceding statements, count = count + 1 will be executed at time = 25 units

Non-blocking Assignments

- Non-blocking assignments allow scheduling of assignments without blocking execution of the statements that follow in a sequential block.
- A <= operator is used to specify nonblocking assignments.
- Evaluated and assigned in two steps:
  - Evaluate the RHS of non-blocking statements at the beginning of the time step.
  - The assignment to the left-hand side is postponed until other evaluations in the current time step are completed.
- Also, the RHS expression of other Verilog non-blocking assignments can also be evaluated and LHS updates scheduled. The non-blocking assignment does not block other.

Non-blocking Assignments – Eg.

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;
//All behavioral statements must be inside an initial or always block
initial
begin
   x = 0; y = 1; z = 1;  //Scalar assignments
   count = 0;            //Assignment to integer variables
   reg_a = 16'b0; reg_b = reg_a; //Initialize vectors
   reg_a[2] <= #15 1'b1; //Bit select assignment with delay
   reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
                                  //to part select of a vector
   count <= count + 1; //Assignment to an integer (increment)
end
```


Non-blocking Assignments – Eg.

- The statements x = 0 through reg_b = reg_a are executed sequentially at time 0.
- Then the three nonblocking assignments are processed at the same simulation time.
  1. reg_a[2] = 0 is scheduled to execute after 15 units (i.e., time = 15)
  2. reg_b[15:13] = {x, y, z} is scheduled to execute after 10 time units (i.e., time = 10)
  3. count = count + 1 is scheduled to be executed without any delay (i.e., time = 0)

- The simulator schedules a nonblocking assignment statement to execute and continues to the next statement in the block without waiting for the nonblocking statement to complete execution.
- Typically, nonblocking assignment statements are executed last in the time step in which they are scheduled, that is, after all the blocking assignments in that time step are executed.
- However, it is recommended that blocking and nonblocking assignments not be mixed in the same always block.

Each Verilog simulation time step is divided into different queues

Time 0:

- Q1 — (in any order) :
  - Evaluate RHS of all non-blocking assignments
  - Evaluate RHS and change LHS of all blocking assignments
  - Evaluate RHS and change LHS of all continuous assignments
  - Evaluate inputs and change outputs of all primitives
  - Evaluate and print output from $display and $write
- Q2 — (in any order) :
  - Change LHS of all non-blocking assignments
- Q3 — (in any order) :
  - Evaluate and print output from $monitor and $strobe

- Concurrent blocking assignments have unpredictable results

```
always @(posedge clk)
#5 A = A + 1;
always @(posedge clk)
#5 B = A + 1;
```

Unpredictable Result:
(new value of B could be evaluated before or after A changes)

- Concurrent non-blocking assignments have predictable results

```
always @(posedge clk)
#5 A <= A + 1;
always @(posedge clk)
#5 B <= A + 1;
```

Predictable Result:
(new value of B will always be evaluated before A changes)

Application of nonblocking assignments

- They are used as a method to model several concurrent data transfers that take place after a common event.

```
always @(posedge clock)
begin
   reg1 <= #1 in1;
   reg2 <= @(negedge clock) in2 ^ in3;
   reg3 <= #1 reg1; //The old value of reg1
end
```


Nonblocking Statements to Eliminate Race Conditions

```
//Two concurrent always blocks with blocking statements
always @(posedge clock)
   a = b;
always @(posedge clock)
   b = a;

//Eg 2: Two concurrent always blocks with nonblocking statements
always @(posedge clock)
   a <= b;
always @(posedge clock)
   b <= a;
```


Timing Controls

- **Delay-Based Timing Control**
- **Level-Sensitive Timing Control**
- **Event-Based Timing Control**

Delay-Based Timing Control

- Delay-based timing control in an expression specifies the time duration between when the statement is encountered and when it is executed.
- **Regular delay control:** used when a non-zero delay is specified to the left of a procedural assignment.
- A timing control before an assignment statement will postpone when the next assignment is evaluated
  - Evaluation is delayed for the amount of time specified


```
begin
#5 A = 1;          -> delay for 5, then evaluate and assign
#5 A = A + 1;      ->delay 5 more, then evaluate and assign
B = A + 1;         ->no delay; evaluate and assign
end
```

What values do A and B contain after 10 time units?

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;
initial begin
    x = 0; // no delay control
    #10 y = 1; // delay control with a number. y = 1 by 10 units
    #latency z = 0; // Delay control with identifier.
    #(latency + delta) p = 1; // Delay control with expression
    #y x = x + 1; // Delay control with identifier.
    #(4:5:6) q = 0; // Minimum, typical and maximum delay.
end
```


Delay-Based Timing Control (Cont..)

- **Intra-assignment delay control:** Instead of specifying delay control to the left of the assignment, it is possible to assign a delay to the right of the assignment operator.
  - The right-hand side is evaluated before the delay
  - The left-hand side is assigned after the delay

```
always @(A)
B = #5 A;          //A is evaluated at the time it changes, but
                   //is not assigned to B until after 5 time units
```

  - always @(negedge clk)
  - Q <=@(posedge clk) D; //D is evaluated at the negative edge of CLK, Q is changed on the //positive edge of CLK

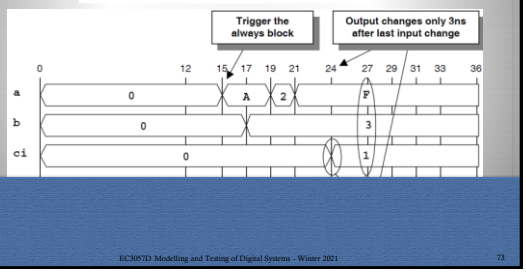

```
//intra assignment delays
reg x, y, z;
initial begin
    x = 0; z = 0;
    y = #5 x + z; //Take value of x and z at the time=0, evaluate
                  //x + z and then wait 5 time units to assign value to y.
End

//Equivalent method with temporary variables and regular delay
    control
initial begin
    x = 0; z = 0;
    temp_xz = x + z;
    #5 y = temp_xz;   //Take value of x + z at the current time and store it
        //in a temporary variable. Even though x and z might change
        //between 0 and 5, the value assigned to y at time 5 is unaffected.
end
```


- Regular delays defer the execution of the entire assignment.
- Intra-assignment delays compute the right hand-side expression at the current time and defer the assignment of the computed value to the left-hand-side variable.
- Intra-assignment delays are like using regular delays with a temporary variable to store the current value of a right-hand-side expression.

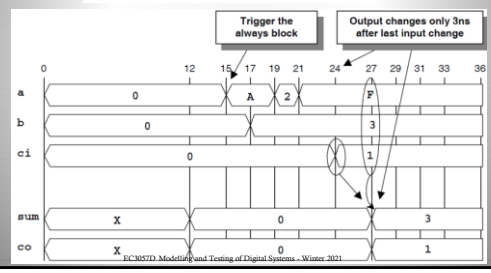

```
module adder_t1 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    #12 {co, sum} = a + b + ci;
endmodule
```




```
module adder_t1 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    #12 {co, sum} = a + b + ci;
endmodule
```



Delay-Based Timing Control (Cont..)

Zero delay control

- Procedural statements in different always-initial blocks may be evaluated at the same simulation time.
- The order of execution of these statements in different always-initial blocks is nondeterministic.
- Zero delay control is a method to ensure that a statement is executed last, after all other statements in that simulation time are executed.


```
initial begin
    x = 0;
    y = 0;
end
initial begin
    #0 x = 1; //zero delay control
    #0 y = 1;
end
```

- In Example, four statements x = 0, y = 0, x = 1, y = 1 are to be executed at simulation time 0. However, since x = 1 and y = 1 have #0, they will be executed last.
- Thus, at the end of time 0, x will have value 1 and y will have value 1. The order in which x = 1 and y = 1 are executed is not deterministic.
- using #0 is not a recommended practice.

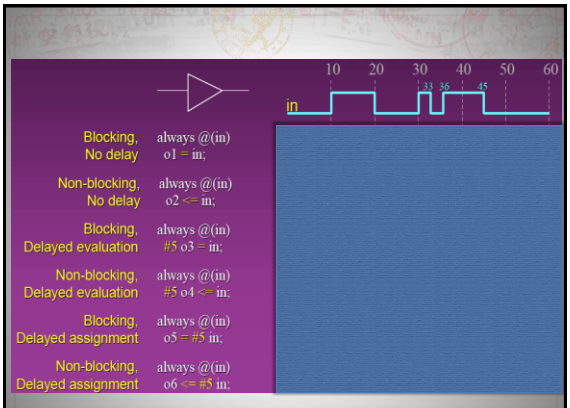
Event-Based Timing Control

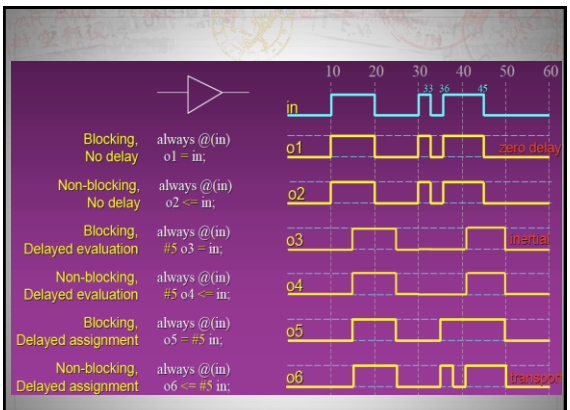
Regular event control (Event Control)

The @ symbol is used to specify an event control.

Statements can be executed on changes in signal value or at a positive or negative transition of the signal value.

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                                        //a positive transition (
    0 to 1,x or z, x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                                        //a negative transition
    ( 1 to 0,x or z,x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
                                        //to q at the positive
    edge of clock
```

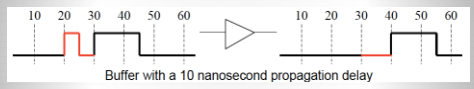





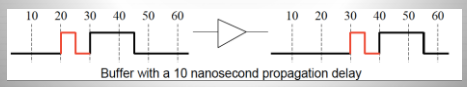
Propagation delay methods

Hardware has two primary propagation delay methods:

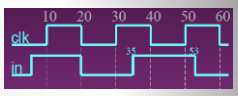
- Inertial delay models devices with finite switching speeds; input glitches do not propagate to the output



- Transport delay models devices with near infinite switching speeds; input glitches propagate to the output

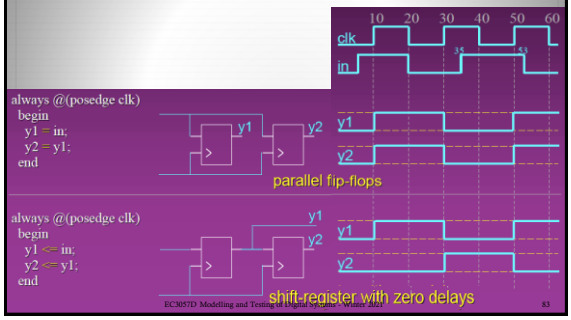


Sequential Logic
Procedural Assignments




```
always @(posedge clk)
 begin
  y1 = in;
  y2 = y1;
 end
```

```
always @(posedge clk)
 begin
  y1 <= in;
  y2 <= y1;
 end
```

Conditional Statements

- Conditional statements are used for making decisions based upon certain conditions.
- These conditions are used to decide whether or not a statement should be executed.
- Keywords ***if*** and ***else*** are used for conditional statements.

```
if (<expression>) true_statement ;
```
← Case 1

```
if (<expression>) true_statement ;
else false_statement ;
```
← Case 2

```
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```
← Case 2

Case 1

```
if(!lock) buffer = data;
if(enable) out = in;
```

Case 2

```
if (number_queued< MAX_Q_DEPTH)
begin
   data_queue = data;
   number_queued = number_queued + 1;
end
else
   $display("Queue Full. Try again");
```

Case 3

```
if (alu_control == 0) y = x + z;
else if(alu_control == 1) y = x - z;
else if(alu_control == 2) y = x * z;
else
$display("Invalid control signal");
```


Multiway Branching

case statement

- The keywords case, endcase, and default are used in the case statement.

```
case (expression)
    alternative1: statement1;
    alternative2: statement2;
    alternative3: statement3;
    ...
    ...
    default:
    default_statement;
endcase
```

```
//Execute statements based on the ALU
control
reg [1:0] alu_control;
...
case (alu_control)
2'd0 : y = x + z;
2'd1 : y = x - z;
2'd2 : y = x * z;
default : $display("Invalid signal");
endcase
```



```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
    output out;
    input i0, i1, i2, i3;
    input s1, s0;
    reg out;
always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ({s1, s0}) //Switch based on concatenation of control signals
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display("Invalid control signals");
    endcase
endmodule
```

Case Statement with x and z

- Considers unknown signals on select.
- If any select signal is x then outputs are x.
- If any select signal is z, outputs are z.
- If one is x and the other is z, x gets higher priority.

```
2'bx0, 2'bx1, 2'bxz, 2'bxx, 2'b0x, 2'b1x, 2'bzx :
begin
    out0 = 1'bx; out1 = 1'bx; out2 = 1'bx; out3 = 1'bx;
end
2'bz0, 2'bz1, 2'bzz, 2'b0z, 2'b1z :
begin
    out0 = 1'bz; out1 = 1'bz; out2 = 1'bz; out3 = 1'bz;
end
```


casex, casez Keywords

- There are two variations of the case statement - casex and casez.
- casez treats all z values in the case alternatives or the case expression as don't cares. All bit positions with z can also represented by ? in that position.
- casex treats all x and z values in the case item or the case expression as don't cares.

```
reg [3:0] encoding;
integer state;
casex (encoding) //logic value x represents a don't care bit.
        4'b1xxx : next_state = 3;
        4'bx1xx : next_state = 2;
        4'bxx1x : next_state = 1;
        default : next_state = 0;
endcase
```

"an input encoding = 4'b10xz would cause next_state = 3 to be executed"

While Loop

- The keyword while is used to specify this loop.
- The while loop executes until the while expression is not true.
- If the loop is entered when the while-expression is not true, the loop is not executed at all.

```
integer count;
initial
   begin
   count = 0;
     while (count < 128) //Execute loop till count is 127, exit at
                         //count 128
    begin
            $display("Count = %d", count);
            count = count + 1;
     end
   end
```


Example

- Write a code using "while loop" to find the first bit with a value 1 in flag vector variable

- //Initialize array elements
- 'define MAX_STATES 32
- integer state [0: 'MAX_STATES-1]; //Integer array state with elements
- 0:31
- integer i;
- initial
- begin
- for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
- state[i] = 0;


```
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;
initial begin
    flag = 16'b 0010_0000_0000_0000;
    i = 0;
    continue = 'TRUE;
    while((i < 16) && continue ) //Multiple conditions using operators.
    begin
        if (flag[i]) begin
          $display("Encountered a TRUE bit at element number %d", i);
          continue = 'FALSE;
        end
        i = i + 1;
   end
end
```

For Loop

- The keyword for is used to specify this loop.
- The for loop contains three parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable

```
integer count;
initial
    for ( count=0; count < 128; count = count + 1)
        $display("Count = %d", count);
```

- The for loop provides a more compact loop structure than the while loop.
- However, the while loop is more general-purpose than the for loop.

Example

- Write a behavioral code using "for loop" to initialize all the even and odd locations of an array with 0 and 1 respectively.


```
//Initialize array elements
'define MAX_STATES 32
integer state [0: 'MAX_STATES-1];
integer i;
initial begin
    for(i = 0; i < 32; i = i + 2) state[i] = 0;
    for(i = 1; i < 32; i = i + 2) state[i] = 1;
end
```


Repeat Loop

- The keyword repeat is used for this loop.
- The repeat construct executes the loop a fixed number of times. A repeat construct cannot be used to loop on a general logical expression.
- A repeat construct must contain a number, which can be a constant, a variable or a signal value.

```
integer count;
initial
  begin
     count = 0;
     repeat(128)
       begin
          $display("Count = %d", count);
          count = count + 1;
       end
  end
```


Example

- Write the code for a data buffer, which after receiving a data_start signal, reads data for next 8 cycles.


```
module data_buffer(data_start, data, clock);
parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;
reg [15:0] buffer [0:7];
integer i;
always @(posedge clock) begin
   if(data_start) //data start signal is true
   begin
       i = 0;
       repeat(cycles) //Store data at the posedge of next 8 clock cycles
       begin
           @(posedge clock) buffer[i] = data; //waits till next posedge to latch data
           i = i + 1;
       end
   end
end
endmodule
```

Forever Loop

- The keyword ***forever*** is used to express this loop.
- This construct is not synthesizable
- The loop does not contain any expression and executes forever until the $finish task is encountered.
- The loop is equivalent to a while loop with an expression that always evaluates to true.
- A forever loop can be exited by use of the disable statement

Eg: Synchronize two register values at every positive edge of clock

```
reg clock;
reg x, y;
initial
    forever @(posedge clock) x = y;
```


- Example: Clock generation - Use forever loop instead of always block


```
reg clock;
initial
begin
    clock = 1'b0;
    forever #10 clock = ~clock;
end
```


Sequential Statements in Verilog

1. begin
       sequential_statements
   end
2. if (expression)
       sequential_statement
   else
       sequential_statement
3. case (expression)
       expr: sequential_statement
       …….
       default: sequential_statement
   endcase

Sequential Statements in Verilog

4. forever
       sequential_statement
5. repeat (expression)
       sequential_statement
6. while (expression)
       sequential_statement
7. for (expr1; expr2; expr3)
       sequential_statement
8. # (time_value)
       Makes a block suspend for "time_value" time units.
9. @ (event_expression)
       Makes a block suspend until event_expression triggers.