Incomplete sensitivity list

- The sensitivity list of a combinational always block must include all the inputs used in that block.
- If you forget some signals from the sensitivity list, synthesis will silently assume they are included and generate gates.
- However, the Verilog simulator gives results as per the sensitivity list.

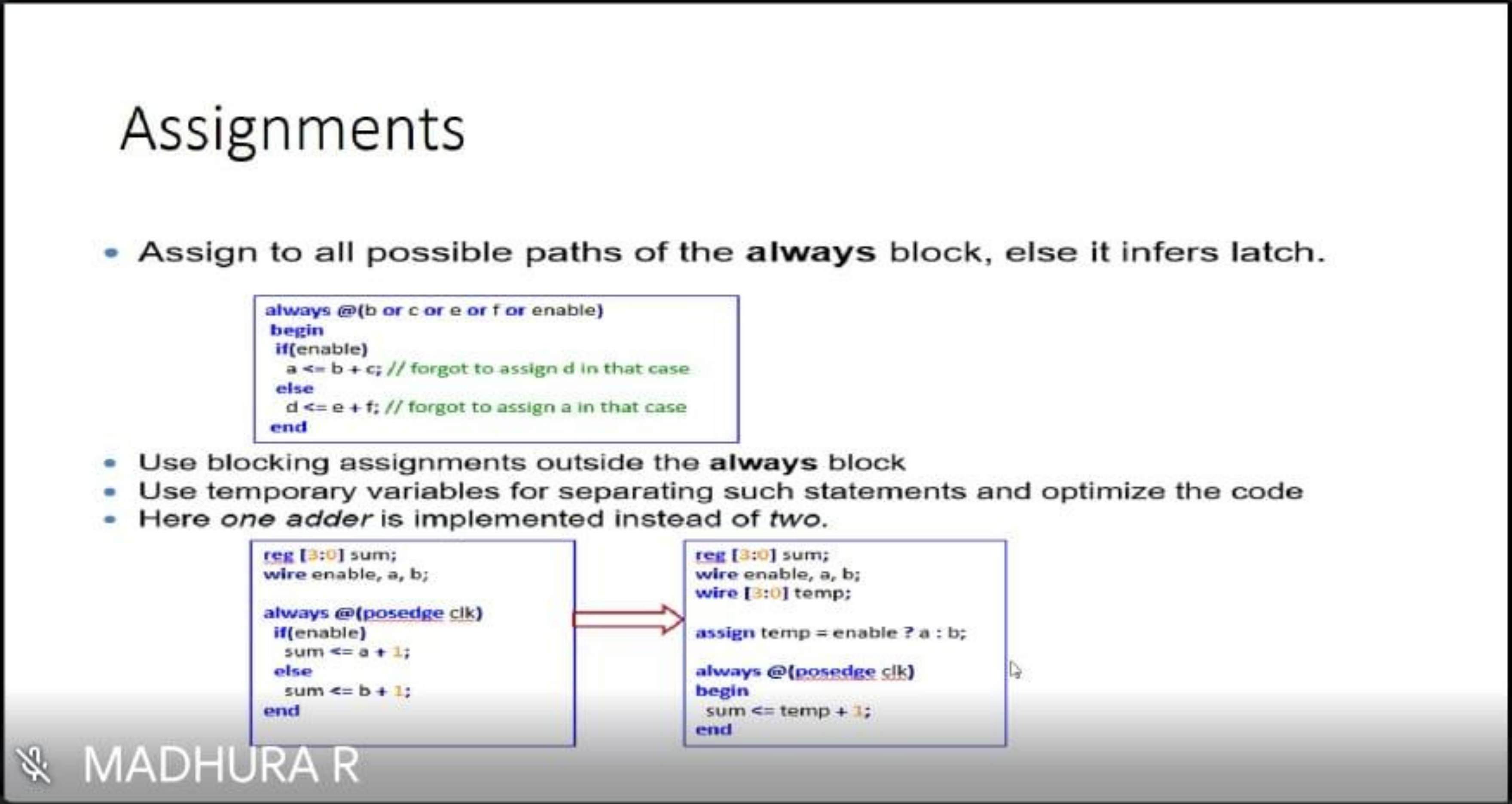
```
always @(b or c or e or f)
 if(enable)
  a = b + c;
 else
  a = e + f;
end
```

✗

```
always @*
 if(enable)
  a = b + c;
 else
  a = e + f;
end
```

✓

MADHURA R



Assignments

- Assign to all possible paths of the **always** block, else it infers latch.

```
always @(b or c or e or f or enable)
 begin
  if(enable)
    a <= b + c; // forgot to assign d in that case
  else
    d <= e + f; // forgot to assign a in that case
 end
```

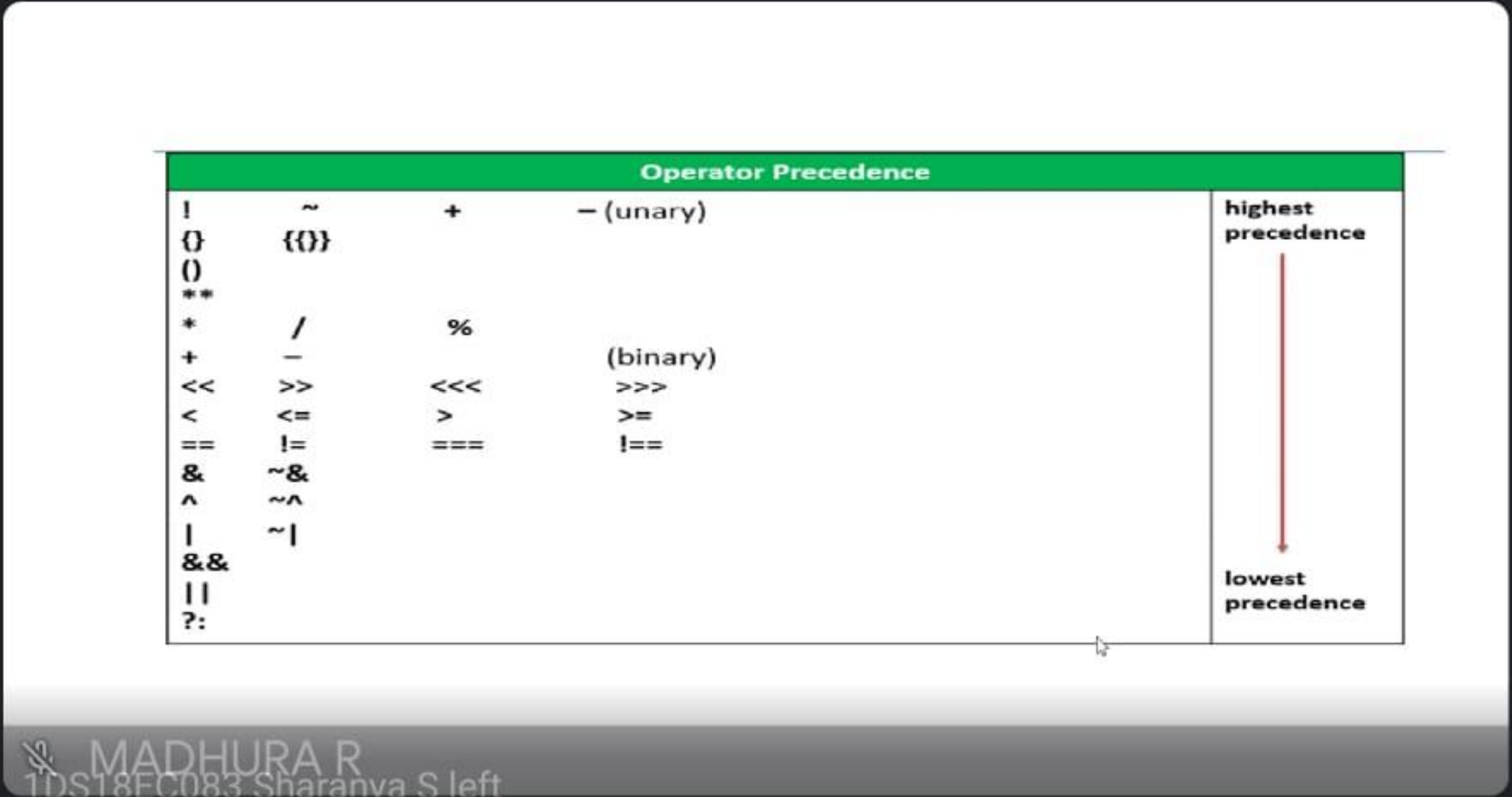
- Use blocking assignments outside the **always** block
- Use temporary variables for separating such statements and optimize the code
- Here *one adder* is implemented instead of *two*.

```
reg [3:0] sum;
wire enable, a, b;

always @(posedge clk)
  if(enable)
    sum <= a + 1;
  else
    sum <= b + 1;
end
```

$\Longrightarrow$

```
reg [3:0] sum;
wire enable, a, b;
wire [3:0] temp;

assign temp = enable ? a : b;

always @(posedge clk)
begin
  sum <= temp + 1;
end
```

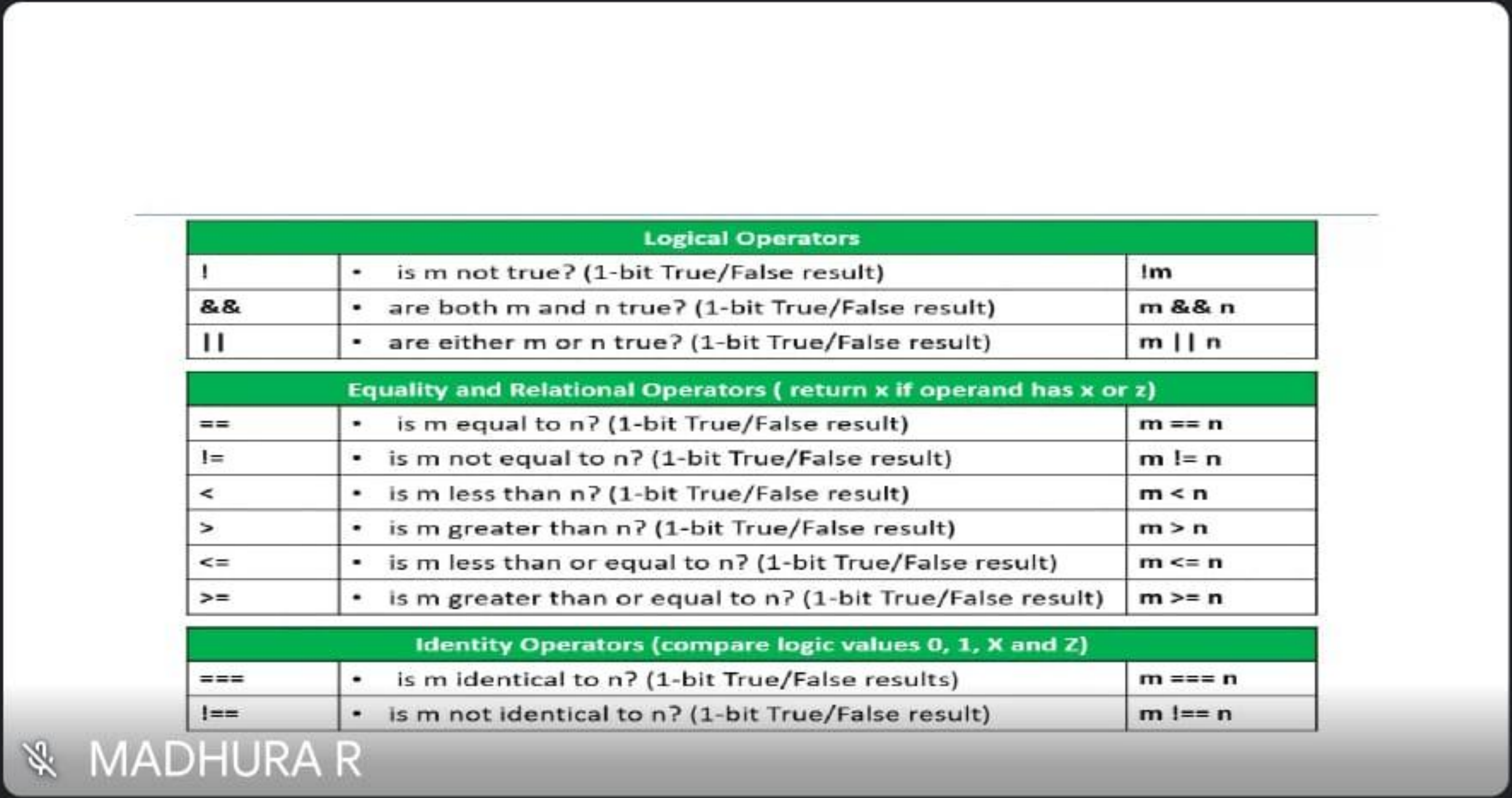
MADHURA R



Operator Precedence				
!	~	+	− (unary)	highest precedence
{}	{{}}			
()				
**				
*	/	%		
+	−		(binary)	
<<	>>	<<<	>>>	
<	<=	>	>=	
==	!=	===	!==	
&	~&			
^	~^			
|	~|			
&&				
||				lowest precedence
?:				

MADHURA R

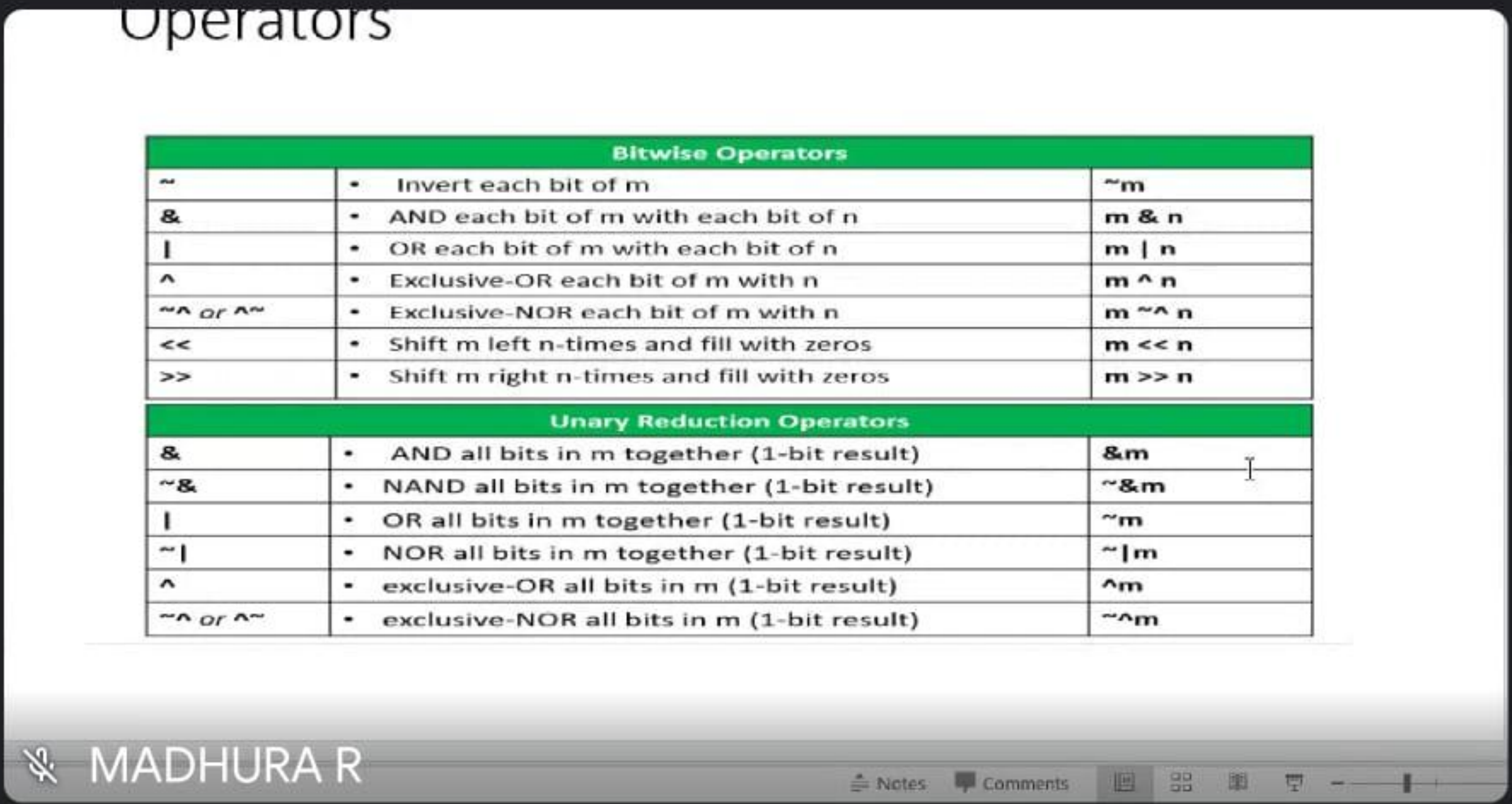
1DS18EC083 Sharanya S left



Logical Operators		
!	• is m not true? (1-bit True/False result)	!m
&&	• are both m and n true? (1-bit True/False result)	m && n
||	• are either m or n true? (1-bit True/False result)	m || n

Equality and Relational Operators (return x if operand has x or z)		
==	• is m equal to n? (1-bit True/False result)	m == n
!=	• is m not equal to n? (1-bit True/False result)	m != n
<	• is m less than n? (1-bit True/False result)	m < n
>	• is m greater than n? (1-bit True/False result)	m > n
<=	• is m less than or equal to n? (1-bit True/False result)	m <= n
>=	• is m greater than or equal to n? (1-bit True/False result)	m >= n

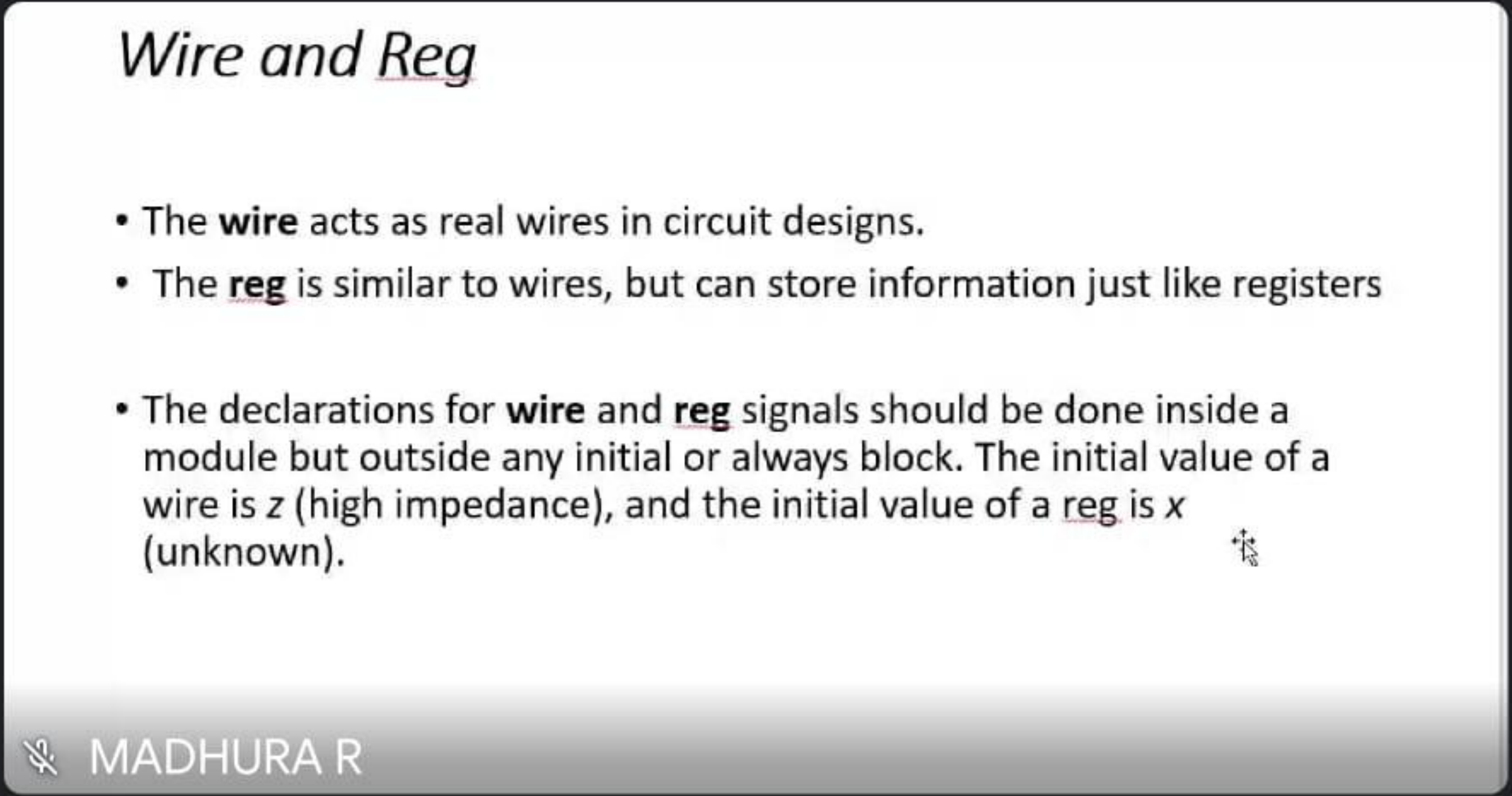
Identity Operators (compare logic values 0, 1, X and Z)		
===	• is m identical to n? (1-bit True/False results)	m === n
!==	• is m not identical to n? (1-bit True/False result)	m !== n



Operators

Bitwise Operators		
~	• Invert each bit of m	~m
&	• AND each bit of m with each bit of n	m & n
|	• OR each bit of m with each bit of n	m | n
^	• Exclusive-OR each bit of m with n	m ^ n
~^ *or* ^~	• Exclusive-NOR each bit of m with n	m ~^ n
<<	• Shift m left n-times and fill with zeros	m << n
>>	• Shift m right n-times and fill with zeros	m >> n

Unary Reduction Operators		
&	• AND all bits in m together (1-bit result)	&m
~&	• NAND all bits in m together (1-bit result)	~&m
|	• OR all bits in m together (1-bit result)	~m
~|	• NOR all bits in m together (1-bit result)	~|m
^	• exclusive-OR all bits in m (1-bit result)	^m
~^ *or* ^~	• exclusive-NOR all bits in m (1-bit result)	~^m

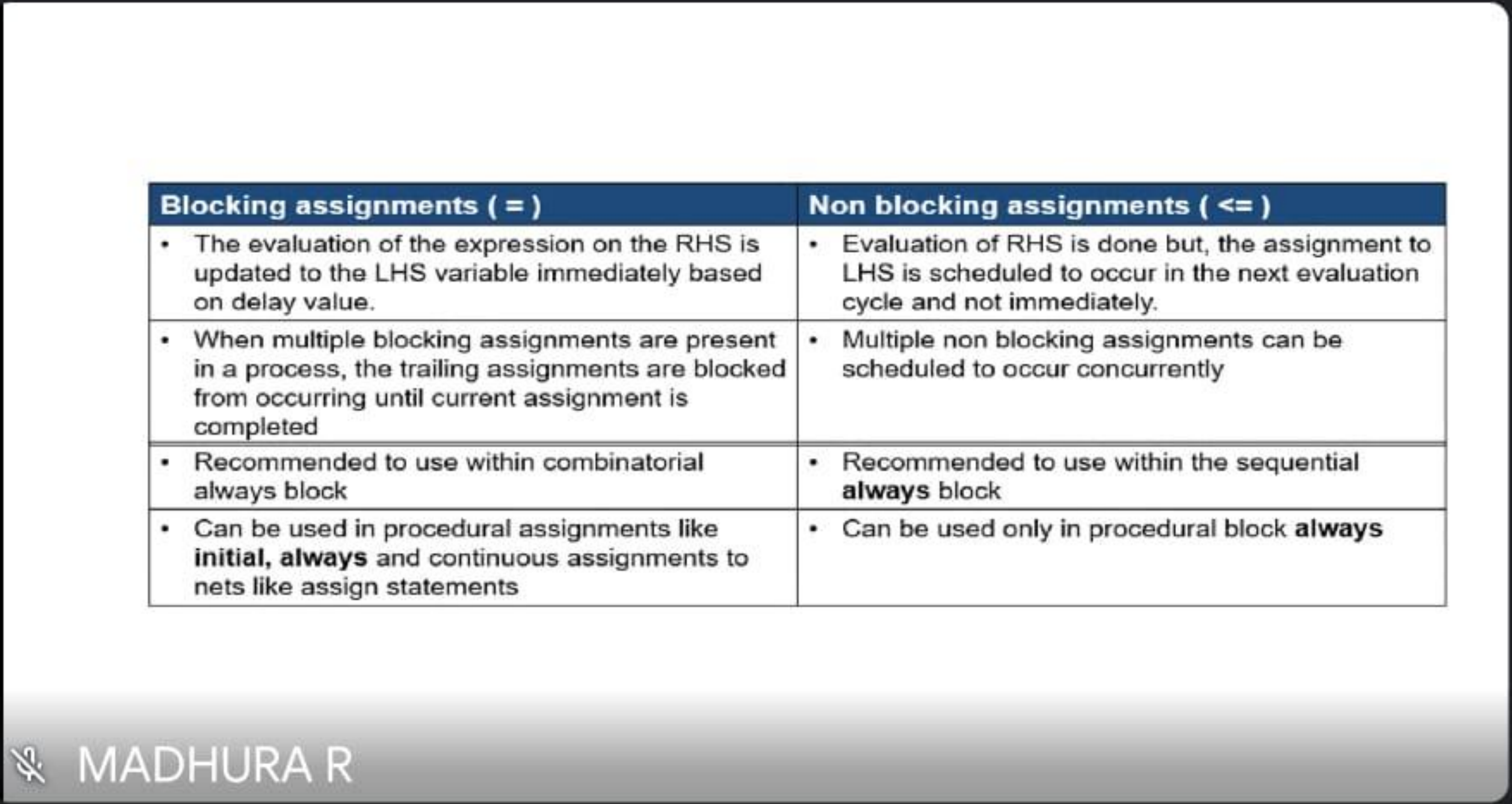


Wire and Reg

- The **wire** acts as real wires in circuit designs.
- The **reg** is similar to wires, but can store information just like registers

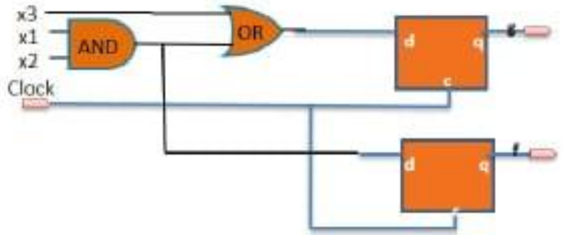
- The declarations for **wire** and **reg** signals should be done inside a module but outside any initial or always block. The initial value of a wire is *z* (high impedance), and the initial value of a reg is *x* (unknown).

MADHURA R

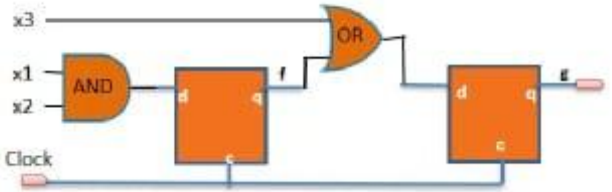


Blocking assignments (=)	Non blocking assignments (<=)
• The evaluation of the expression on the RHS is updated to the LHS variable immediately based on delay value.	• Evaluation of RHS is done but, the assignment to LHS is scheduled to occur in the next evaluation cycle and not immediately.
• When multiple blocking assignments are present in a process, the trailing assignments are blocked from occurring until current assignment is completed	• Multiple non blocking assignments can be scheduled to occur concurrently
• Recommended to use within combinatorial always block	• Recommended to use within the sequential **always** block
• Can be used in procedural assignments like **initial, always** and continuous assignments to nets like assign statements	• Can be used only in procedural block **always**

```
// Example  for blocking
module example3(x1, x2, x3, Clock,  g);
 input x1, x2, x3, Clock;
 output  g;
 reg f, g;

 always @(posedge Clock)
 begin
  f = x1 & x2;
  g = f | x3;
 end
endmodule
```



```
//Example  for non  blocking

 module example3(x1, x2, x3, Clock,  g);
 input x1, x2, x3, Clock;
 output  g;
 reg f, g;

 always @(posedge Clock)
  begin
  f <= x1 & x2;
  g <= f | x3;
 end
endmodule
```



Anshik Jain left

Example on Non Blocking Assignments

```
module Non_Blocking (a,b,c,d);
 input a, b;
 output reg c, d;
 initial begin
  c <= 0;
 #50 c <= a|b;      // a|b executes at t=0 then wait for 50 units, and execute c=a|b
 d <= c;            // The RHS value 'c' is '0' at time t=0. Assignment d=0 at t=50
 c <= #50 a&b;      // a&b=0 execute at t=0 and assignment  c=a&b at t=100
 end
endmodule
```

Subscribe

Non Blocking Assignments

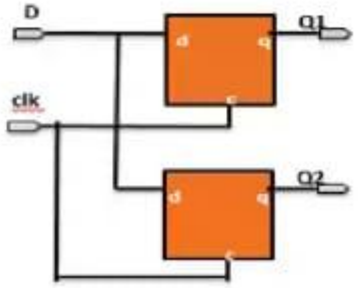
- Non blocking assignments executes in parallel
- The execution of the next statement is not blocked
- Non Blocking assignments are represented with " <= " sign
- For all the non blocking assignments in a block, the right-hand sides are evaluated first and stored in a temporary registers.
- Used to perform Sequential Circuits

```
module nonblocking (a, b, clk, x);          Assume Input a=4,b=10
      inputs a, b, clk;
      output reg x,
     always @(posedge clk)
        x <= a;
        b <= 5;
        x  <= b;
     $display(" x=%b ", x);            Output:-  x=10
       end
    endmodule
```

Double synchronizer example – 2 Flip flops are connected back to back

Use of blocking statements

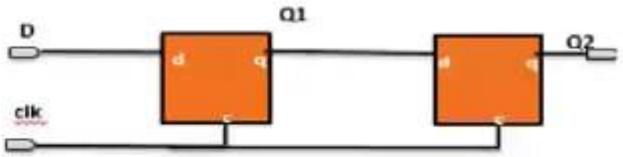
```
// D FFs with blocking assignment
module example1(D, clk,  Q2);
 input D, clk;
 output  Q2;
 reg Q1, Q2;

 always @(posedge clk)
  begin
   Q1 = D;
   Q2 = Q1;
  end
endmodule
```



Combinatorial elements must always use blocking assignments =

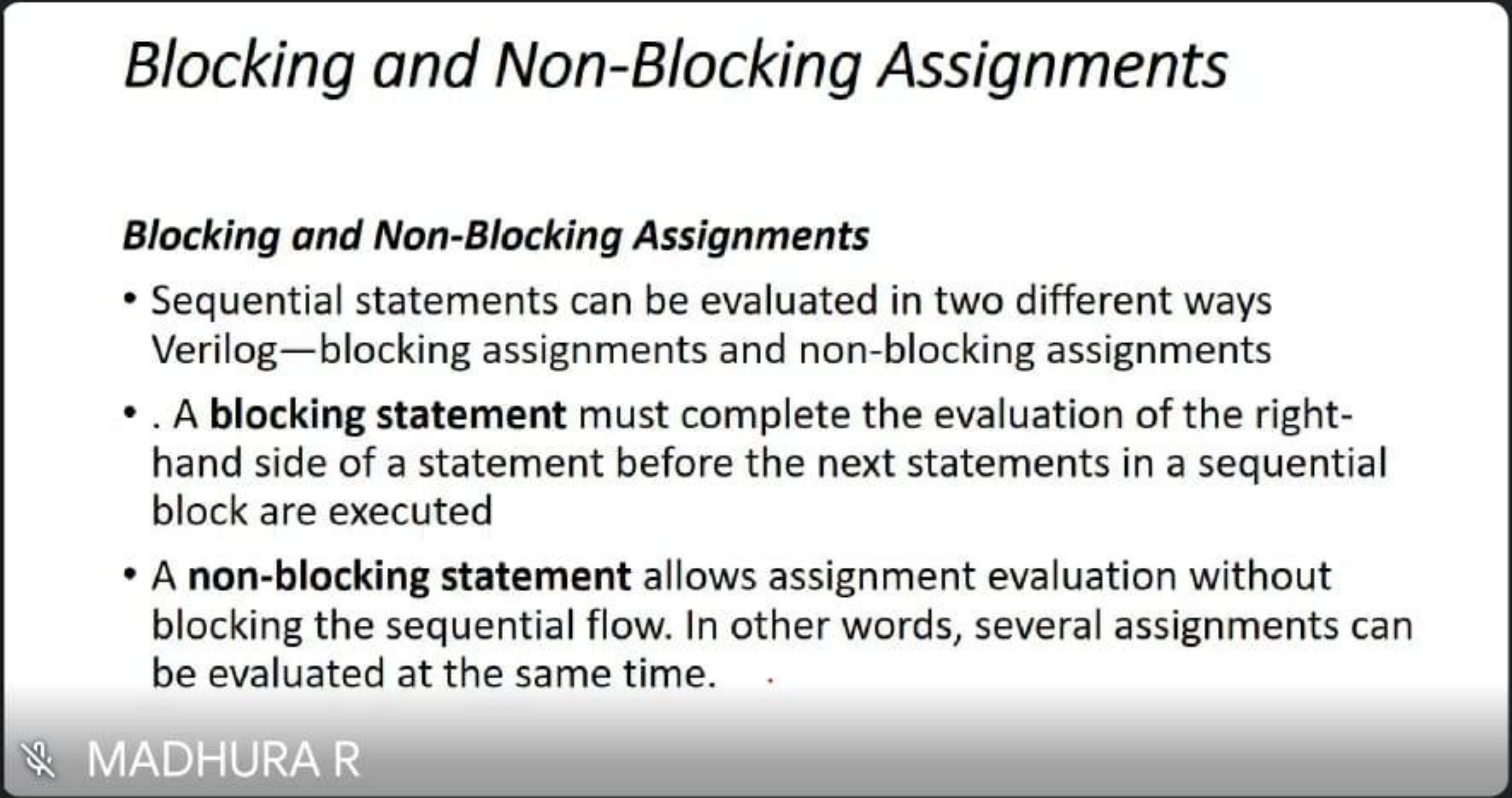
Use of Non blocking statements

```
// D FFs with Non blocking assignment
module example1(D, clk,  Q2);
 input D, clk;
 output  Q2;
 reg Q1, Q2;
 always @(posedge clk)
  begin
   Q1 <= D; // substitute
   Q2 <= Q1; // non-blocking assignments
  end
endmodule
```



Sequential elements must always use non blocking assignments <=

MADHURA R



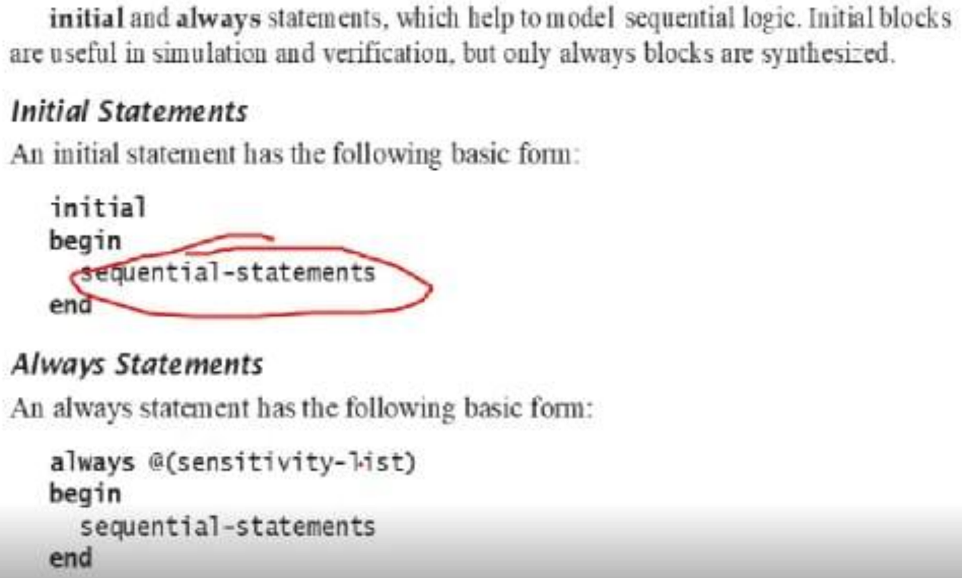
Blocking and Non-Blocking Assignments

Blocking and Non-Blocking Assignments

- Sequential statements can be evaluated in two different ways Verilog—blocking assignments and non-blocking assignments
- . A **blocking statement** must complete the evaluation of the right-hand side of a statement before the next statements in a sequential block are executed
- A **non-blocking statement** allows assignment evaluation without blocking the sequential flow. In other words, several assignments can be evaluated at the same time.

Procedural Assignments

- Procedural assignments are used to model registers and finite state machines using the **always** keyword



initial and **always** statements, which help to model sequential logic. Initial blocks are useful in simulation and verification, but only always blocks are synthesized.

Initial Statements

An initial statement has the following basic form:

```
initial
begin
  sequential-statements
end
```

Always Statements

An always statement has the following basic form:

```
always @(sensitivity-list)
begin
  sequential-statements
end
```

MADHURA R