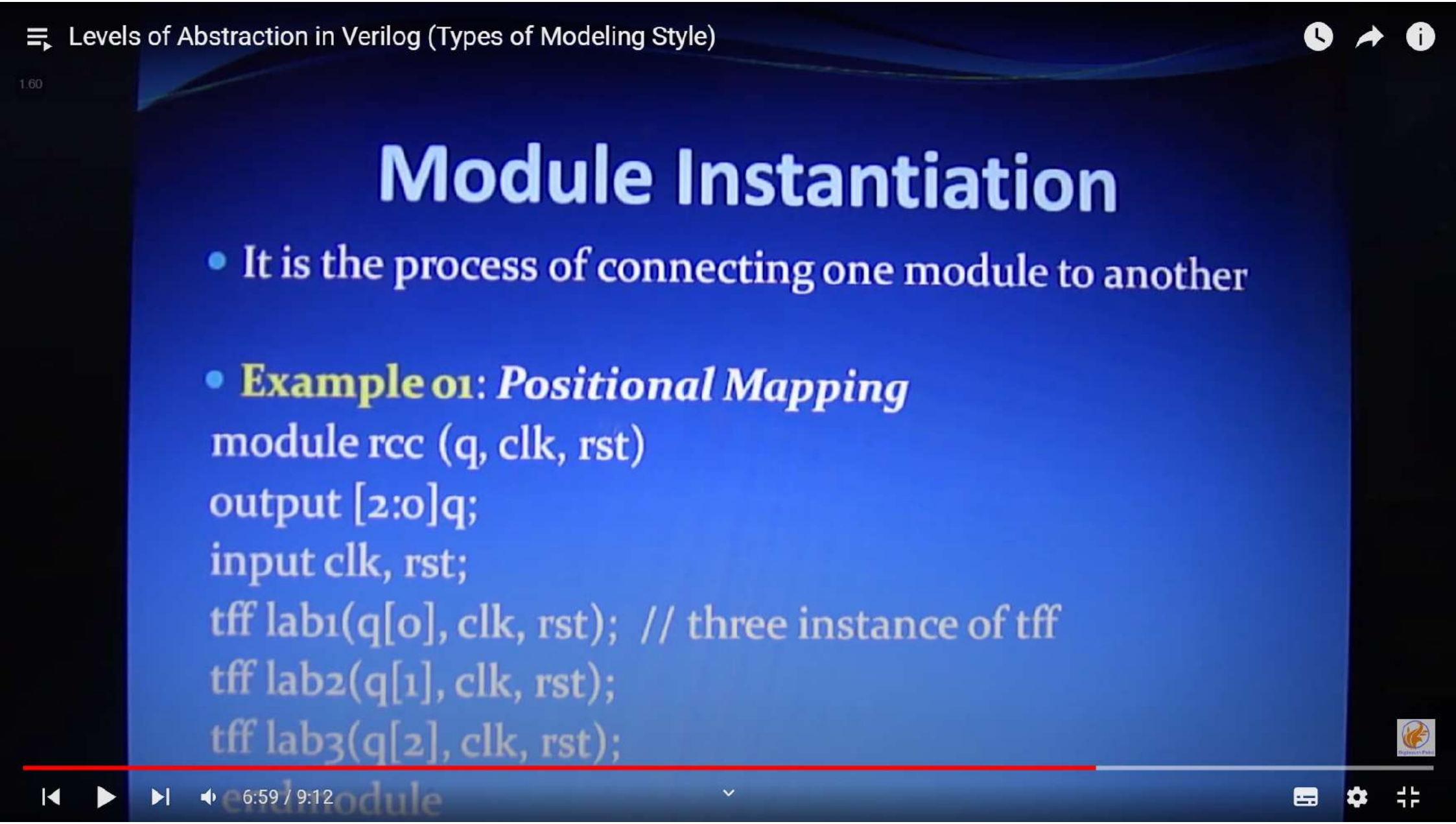
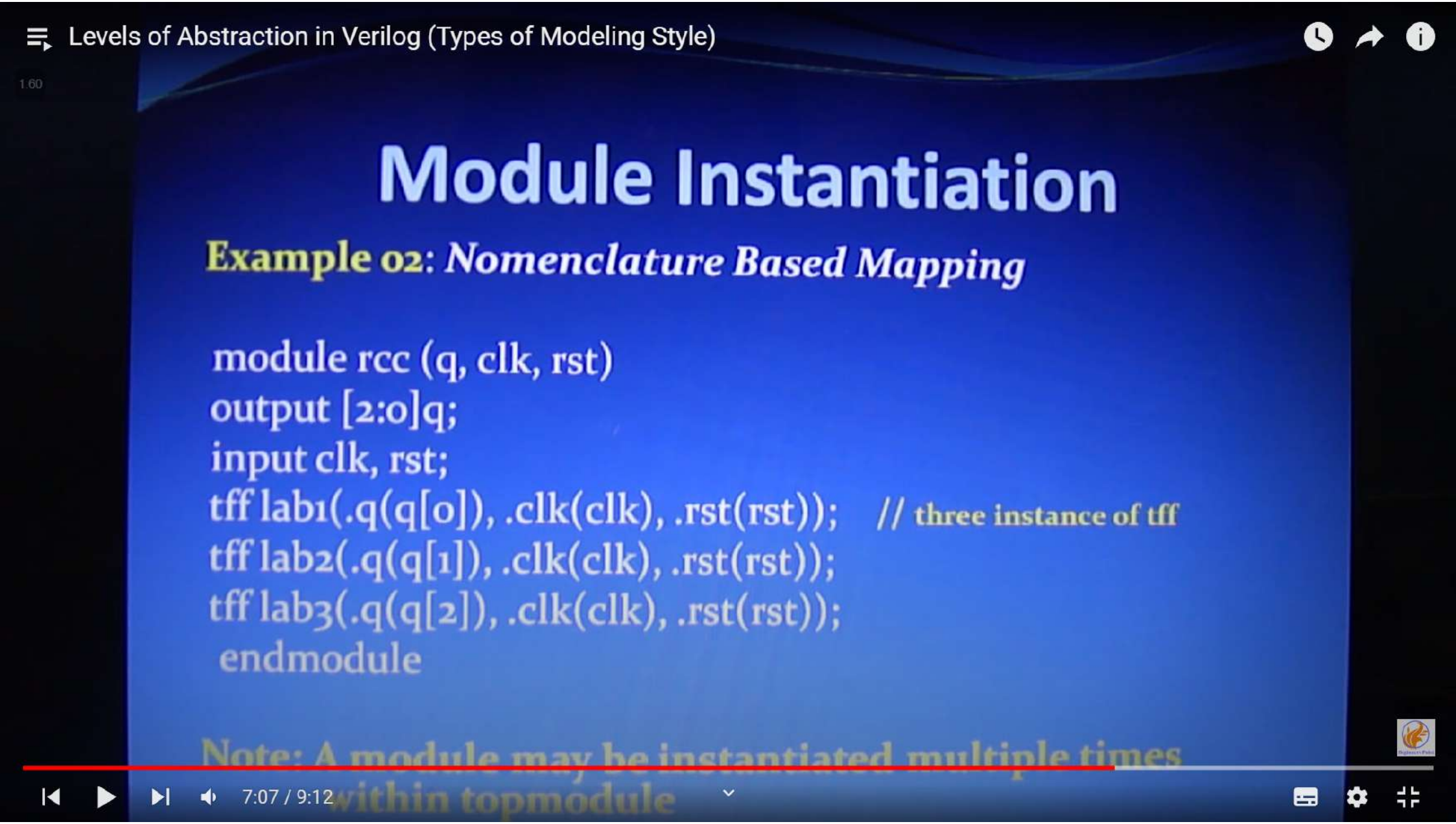
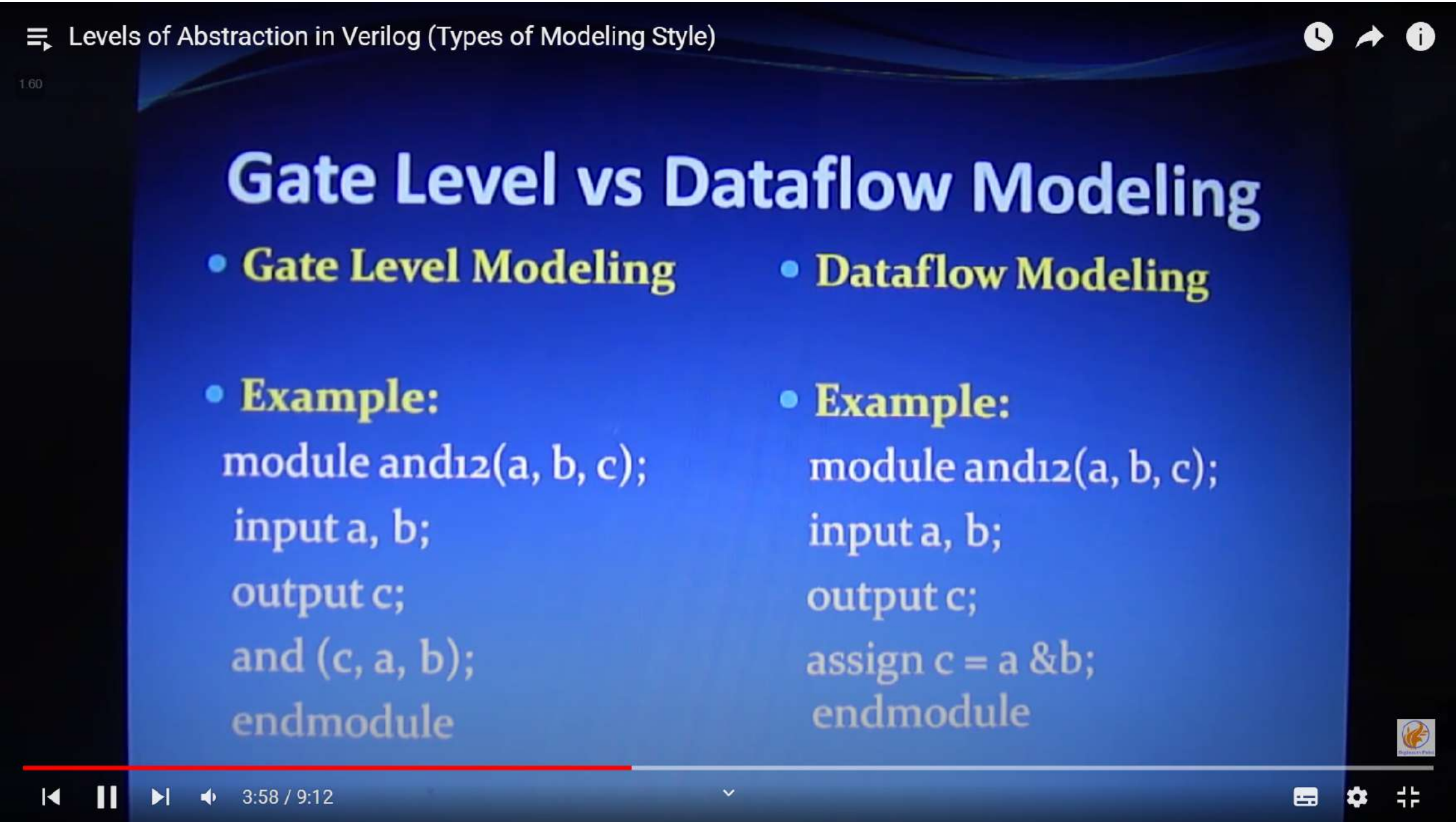
Levels of Abstraction

- Behavioral Level (Design of Algorithm)
- Dataflow Level (Design of Equation)
- Gate Level (Interconnection with Logic Gates)
- Switch Level (Implementation in terms of Switches)


Levels of Abstraction in Verilog (Types of Modeling Style)
1:60
Module Instantiation
It is the process of connecting one module to another
Example 01: Positional Mapping
module rcc (q, clk, rst)
output [2:0]q;
input clk, rst;
tff lab1(q[0], clk, rst); // three instance of tff
tff lab2(q[1], clk, rst);
tff lab3(q[2], clk, rst);
endmodule
6:59 / 9:12


Levels of Abstraction in Verilog (Types of Modeling Style)
1.60
Module Instantiation
Example 02: Nomenclature Based Mapping
module rcc (q, clk, rst)
output [2:0]q;
input clk, rst;
tff lab1(.q(q[0]), .clk(clk), .rst(rst));
// three instance of tff
tff lab2(.q(q[1]), .clk(clk), .rst(rst));
tff lab3(.q(q[2]), .clk(clk), .rst(rst));
endmodule
Note: A module may be instantiated multiple times
within topmodule
7:07 / 9:12


Levels of Abstraction in Verilog (Types of Modeling Style)
1.60
Gate Level vs Dataflow Modeling
Gate Level Modeling
Example:
module and12(a, b, c);
input a, b;
output c;
and (c, a, b);
endmodule
Dataflow Modeling
Example:
module and12(a, b, c);
input a, b;
output c;
assign c = a &b;
endmodule
3:58 / 9:12

Data Types

- Verilog has two primary data types
- **Nets :** Represent connection between components

  wire a;
- **Register:** Represent variable used to store data

  reg reset;
- Register store the last value assigned to them until another assignment statement changes their value

Data Types

- **Vectors :** 'Net' and 'reg' data types can be declared as vectors (multiple bit widths)

```
wire [4:0] a;
wire [31:0] b;
reg [0:31] c;
reg [7:0] d;
```

Data Types

- Some other data types are

1) Integer
2) Arrays
3) Memories
4) Parameters : constants defined in module. They cannot be used as variables
5) Strings

Blocking Assignment

- Blocking assignment are executed in the order they are coded. Hence, they are sequential in nature
- They block the execution of next statement, till the current statement is executed completely
- Used to model combinational circuits
- The '=' operator is used to specify the blocking assignments

Example: a = b;

Non - Blocking Assignment

- They are executed in parallel (i.e. at the same time all registers and flip – flops must update their outputs). They are concurrent in nature
- Since the execution of next statement is not blocked due to execution of current statement, they are called non-blocking assignments
- Used to model sequential circuits
- The '<=' operator is used to specify the non-blocking assignments

Example: a <= b;

Example of Blocking & Non-Blocking Statment

- **Blocking Assignment**

```
always @(posedge i_clock)
begin
 r_Test_1 = 1'b1;
 r_Test_2 = r_Test_1;
 r_Test_3 = r_Test_2;
end
```

- **Non-Blocking Assignment**

```
always @(posedge i_clock)
begin
 r_Test_1 <= 1'b1;
 r_Test_2 <= r_Test_1;
 r_Test_3 <= r_Test_2;
end
```

Example : 02

Blocking Assignment

```
initial
begin
a = #10 1'b1;// The simulator assigns 1 to a at time 10
b = #20 1'b0;// The simulator assigns 0 to b at time 30
c = #40 1'b1;// The simulator assigns 1 to c at time 70
```

Nonblocking assignments

```
Initial
begin
d <= #10 1'b1;// The simulator assigns 1 to d at time 10
e <= #20 1'b0;// The simulator assigns 0 to e at time 20
f <= #40 1'b1;// The simulator assigns 1 to f at time 40
```

Why we use Non-Blocking assignment for sequential instead of Blocking?

To Overcome Race around condition

```
reg r1, r2, r3, r4;
always@(posedge clk)
r2 = r1;
always@(posedge clk)
r3 = r2;
always@(posedge clk)
r4 = r3;
```

These run in some order, but it is difficult to predict

```
always@(posedge clk)
r2 <= r1;
always@(posedge clk)
r3 <= r2;
always@(posedge clk)
r4 <= r3;
```

RHS evaluated when assignment run, LHS updated only after all events for the current instant have run

Verilog Operators

- Arithmetic Operators
- Logical Operators
- Relational Operators
- Equality Operators
- Bitwise Operators
- Reduction Operators
- Shift Operators
- Concatenation Operators
- Replication Operators

Example of Arithmetic Operators

- If A = 4'b0011 and B = 4'b0100 then

A * B = 4'b1100
A + B = 4'b0111
B – A = 4'b0001

- If A = 4 and B = 2 then

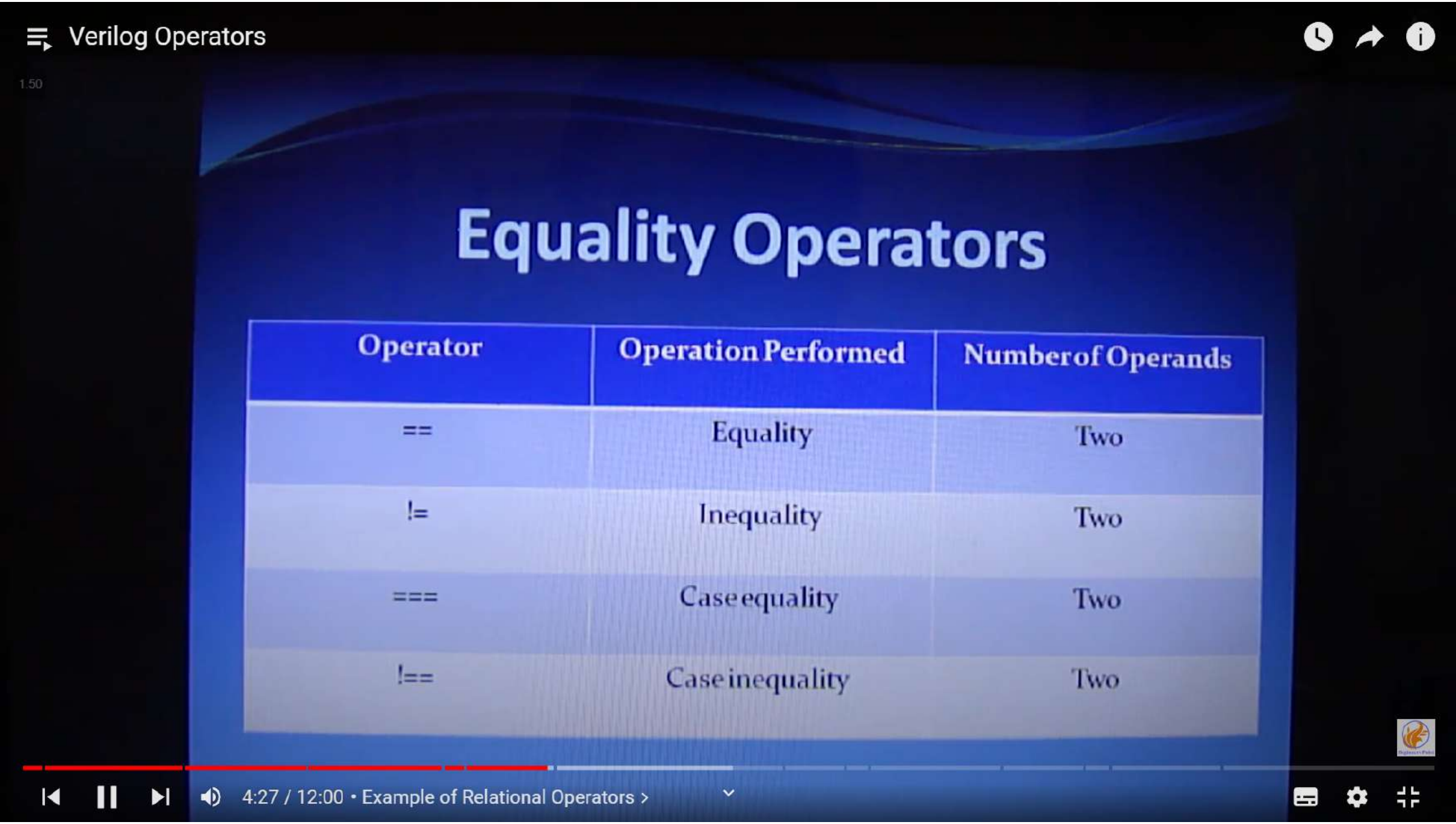
B = A ** B = 4 ** 2 = 16
A % B = 0

Logical Operators

Operator	Operation Performed	Number of Operands
!	Not	One
&&	Logical And	Two
||	Logical OR	Two

Relational Operators

Operator	Operation Performed	Number of Operands
>	Greater than	Two
<	less than	Two
>=	greater than or equal	Two
<=	Less than or equal	Two


Verilog Operators
1:50
Equality Operators
Operator
Operation Performed
Number of Operands
==
Equality
Two
!=
Inequality
Two
===
Case equality
Two
!==
Case inequality
Two
4:27 / 12:00 • Example of Relational Operators >

Bitwise Operators

Operator	Operation Performed	Number of Operands
~	Bitwise Negation	One
&	Bitwise And	Two
|	Bitwise OR	Two
^	Bitwise XOR	Two
^~ or ~^	Bitwise XNOR	Two

Reduction Operators

Operator	Operation Performed	Number of Operands
&	Reduction And	One
~&	Reduction Nand	One
|	Reduction OR	One
~|	Reduction NOR	One
^	Reduction XOR	One
^~ or ~^	Reduction XNOR	One

Shift Operators

Operator	Operation Performed	Number of Operands
>>	Right Shift	Two
<<	Left Shift	Two
>>>	Arithmetic Right Shift	Two
<<<	Arithmetic Left Shift	Two

Concatenation Operators

Operator	Operation Performed	Number of Operands
{ }	Concatenation	Any Number
{ { } }	Replication	Any Number

Example of Concatenation Operators

- If a = 1'b1 and b = 2'b00

y = {a, b} gives 3'b100

y = { 4 { a } } gives 4'b1111

y = { 4 { a }, 2 { b } } gives 8'b11110000

Conditional Operators

Operator	Operation Performed	Number of Operands
? :	Conditional	Three

Operator Precedence

- Operators are evaluated in order of their precedence (i.e. highest to lowest)
- Operators of equal precedence are evaluated from left to right
- Use parentheses to control order.

Highest	Precedence Order						Lowest
Bitwise	Arithmetic	Shift	Relational	Equality	Reduction	Logical	Conditional

Case Statement

The keywords are case, endcase, default

Syntax:

```
Case (Expression)
        label 01 : Statement_01;
        label 02 : Statement_02;
        label 03 : Statement_03;
        ......
        .......
        Default: default_statement;
endcase
```

Nand Gate using Case Statement

```
module nand_case(a, b, c);
  input a, b;
  output c;
  reg c;
  always@ (a or b)
  begin
        case ({a,b})
          2'b00:  begin
                    c = 1'b1;
                    end
         2'b01:  begin
                    c = 1'b1;
```

Nand Gate using Case Statement

```
        2'b10:  begin
                c = 1'b1;
                end
        2'b11:  begin
                c = 1'bo;
                end
        default: begin
                 c = 1'bo;
                 end
    endcase
  end
endmodule
```

'always' statement

- It is the heart of hardware modeling using Verilog
- **Syntax :**

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

  /* behavioral constructs that describe needed functionality
(example: if/else, case statement etc.) */

end
```

Sensitivity list : statements in the always block will be executed if and only if these signal changed

Note: We can use 'n' numbers of always block in a module