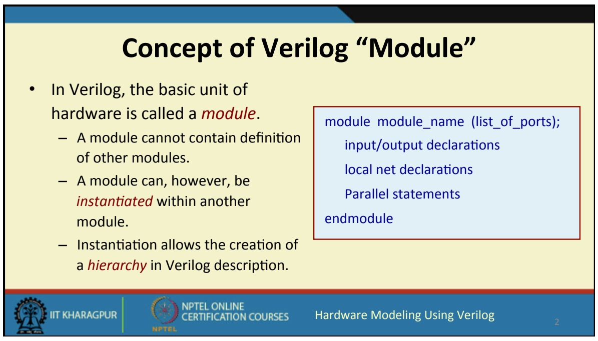
IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Lecture 06: VERILOG LANGUAGE FEATURES (PART 1)

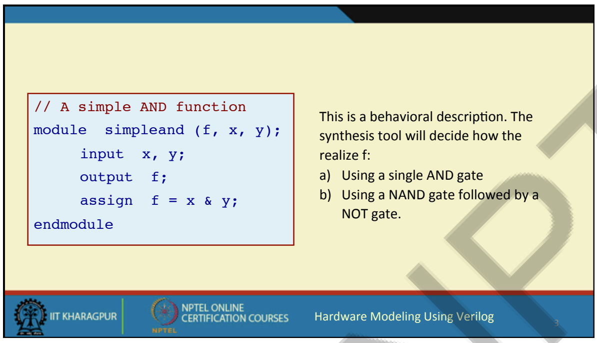
PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Concept of Verilog "Module"

- In Verilog, the basic unit of hardware is called a *module*.
  - A module cannot contain definition of other modules.
  - A module can, however, be *instantiated* within another module.
  - Instantiation allows the creation of a *hierarchy* in Verilog description.

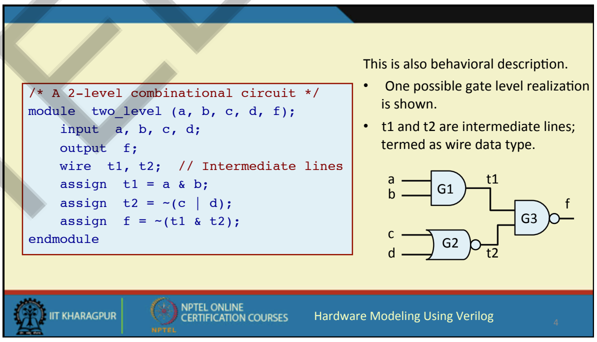
```
module  module_name  (list_of_ports);
    input/output declarations
    local net declarations
    Parallel statements
endmodule
```



```
// A simple AND function
module  simpleand (f, x, y);
     input  x, y;
     output  f;
     assign  f = x & y;
endmodule
```

This is a behavioral description. The synthesis tool will decide how the realize f:

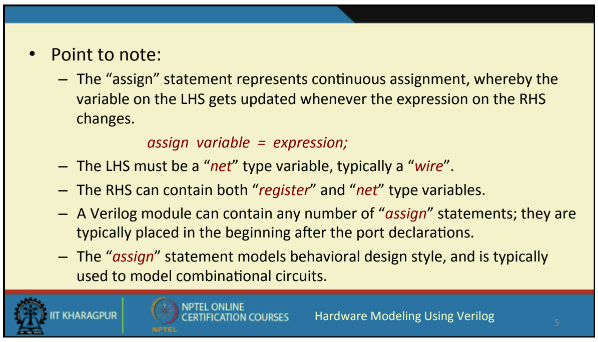
a) Using a single AND gate
b) Using a NAND gate followed by a NOT gate.



```
/* A 2-level combinational circuit */
module  two_level (a, b, c, d, f);
    input  a, b, c, d;
    output  f;
    wire  t1, t2;  // Intermediate lines
    assign  t1 = a & b;
    assign  t2 = ~(c | d);
    assign  f = ~(t1 & t2);
endmodule
```

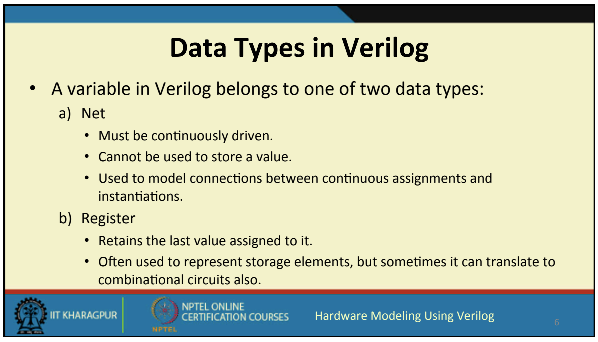
This is also behavioral description.

- One possible gate level realization is shown.
- t1 and t2 are intermediate lines; termed as wire data type.



- Point to note:
  - The "assign" statement represents continuous assignment, whereby the variable on the LHS gets updated whenever the expression on the RHS changes.

    assign variable = expression;
  - The LHS must be a "*net*" type variable, typically a "*wire*".
  - The RHS can contain both "*register*" and "*net*" type variables.
  - A Verilog module can contain any number of "*assign*" statements; they are typically placed in the beginning after the port declarations.
  - The "*assign*" statement models behavioral design style, and is typically used to model combinational circuits.

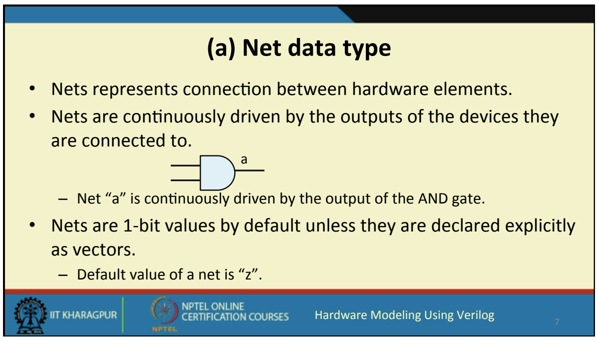


Data Types in Verilog

- A variable in Verilog belongs to one of two data types:

  a) Net
  - Must be continuously driven.
  - Cannot be used to store a value.
  - Used to model connections between continuous assignments and instantiations.

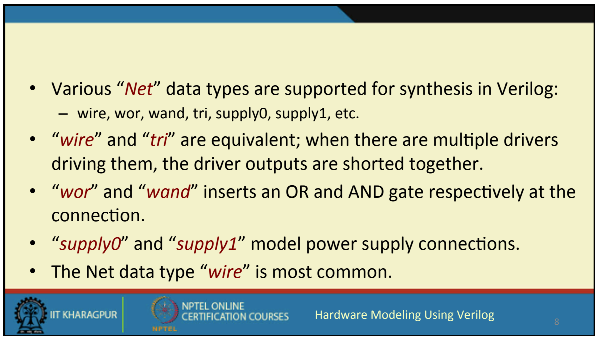
  b) Register
  - Retains the last value assigned to it.
  - Often used to represent storage elements, but sometimes it can translate to combinational circuits also.



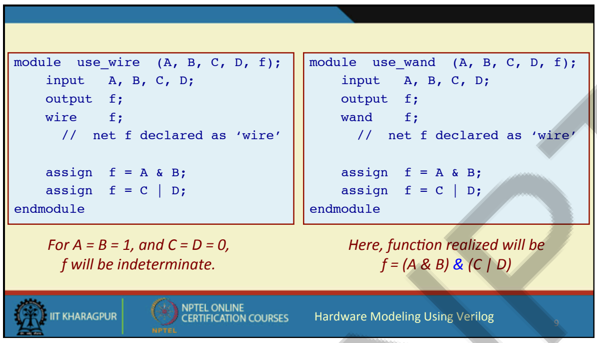
(a) Net data type

- Nets represents connection between hardware elements.
- Nets are continuously driven by the outputs of the devices they are connected to.

  - Net "a" is continuously driven by the output of the AND gate.
- Nets are 1-bit values by default unless they are declared explicitly as vectors.
  - Default value of a net is "z".



- Various "*Net*" data types are supported for synthesis in Verilog:
  - wire, wor, wand, tri, supply0, supply1, etc.
- "*wire*" and "*tri*" are equivalent; when there are multiple drivers driving them, the driver outputs are shorted together.
- "*wor*" and "*wand*" inserts an OR and AND gate respectively at the connection.
- "*supply0*" and "*supply1*" model power supply connections.
- The Net data type "*wire*" is most common.

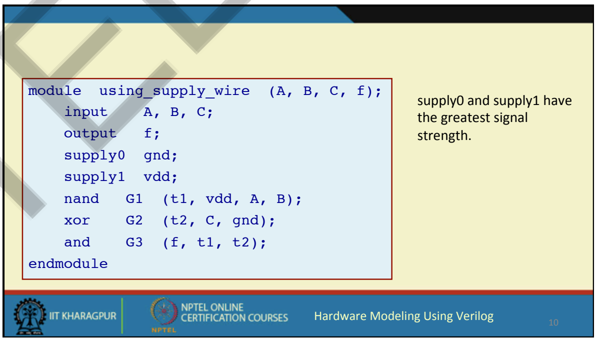


```
module  use_wire  (A, B, C, D, f);
    input   A, B, C, D;
    output  f;
    wire    f;
      //  net f declared as 'wire'

    assign  f = A & B;
    assign  f = C | D;
endmodule
```

For A = B = 1, and C = D = 0, f will be indeterminate.

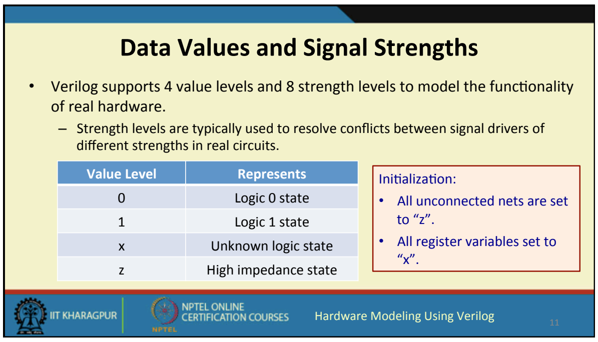
```
module  use_wand  (A, B, C, D, f);
    input   A, B, C, D;
    output  f;
    wand    f;
      //  net f declared as 'wire'

    assign  f = A & B;
    assign  f = C | D;
endmodule
```

Here, function realized will be f = (A & B) & (C | D)



```
module  using_supply_wire  (A, B, C, f);
    input     A, B, C;
    output    f;
    supply0   gnd;
    supply1   vdd;
    nand   G1  (t1, vdd, A, B);
    xor    G2  (t2, C, gnd);
    and    G3  (f, t1, t2);
endmodule
```

supply0 and supply1 have the greatest signal strength.

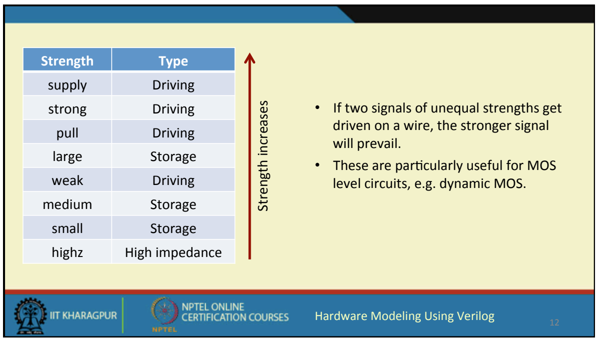


Data Values and Signal Strengths

- Verilog supports 4 value levels and 8 strength levels to model the functionality of real hardware.
  - Strength levels are typically used to resolve conflicts between signal drivers of different strengths in real circuits.

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

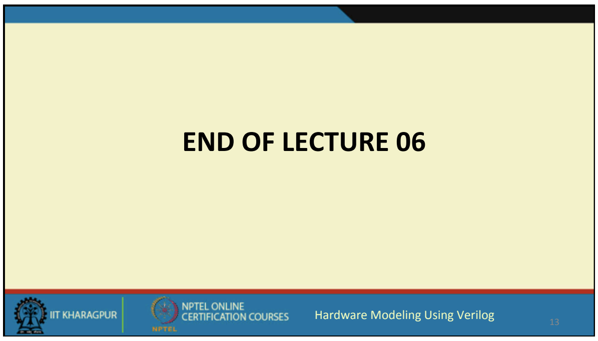
Initialization:
- All unconnected nets are set to "z".
- All register variables set to "x".



Strength	Type
supply	Driving
strong	Driving
pull	Driving
large	Storage
weak	Driving
medium	Storage
small	Storage
highz	High impedance

Strength increases

- If two signals of unequal strengths get driven on a wire, the stronger signal will prevail.
- These are particularly useful for MOS level circuits, e.g. dynamic MOS.

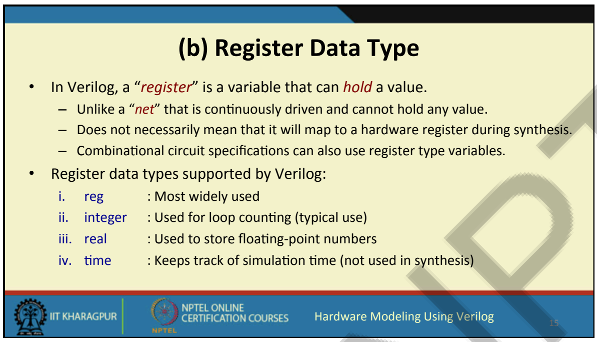


END OF LECTURE 06

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES NPTEL

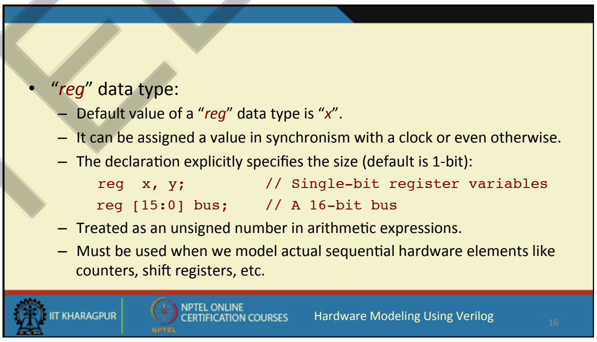
Lecture 07: VERILOG LANGUAGE FEATURES (PART 2)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



(b) Register Data Type

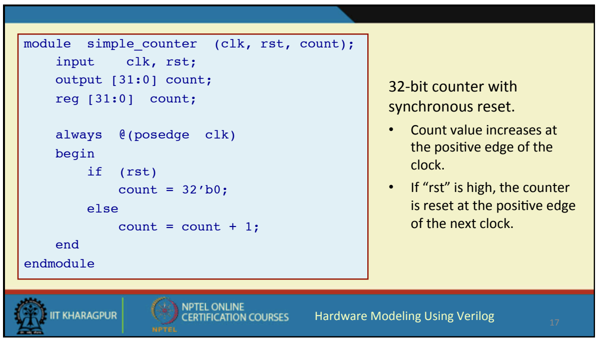
- In Verilog, a "*register*" is a variable that can *hold* a value.
  - Unlike a "*net*" that is continuously driven and cannot hold any value.
  - Does not necessarily mean that it will map to a hardware register during synthesis.
  - Combinational circuit specifications can also use register type variables.
- Register data types supported by Verilog:
  1. reg : Most widely used
  2. integer : Used for loop counting (typical use)
  3. real : Used to store floating-point numbers
  4. time : Keeps track of simulation time (not used in synthesis)



- "*reg*" data type:
  - Default value of a "*reg*" data type is "*x*".
  - It can be assigned a value in synchronism with a clock or even otherwise.
  - The declaration explicitly specifies the size (default is 1-bit):

```
reg  x, y;          // Single-bit register variables
reg [15:0] bus;     // A 16-bit bus
```

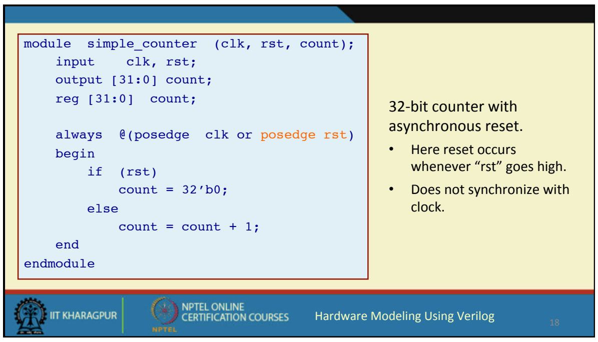
  - Treated as an unsigned number in arithmetic expressions.
  - Must be used when we model actual sequential hardware elements like counters, shift registers, etc.



```
module  simple_counter  (clk, rst, count);
    input    clk, rst;
    output [31:0] count;
    reg [31:0]  count;

    always  @(posedge  clk)
    begin
        if  (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

32-bit counter with synchronous reset.

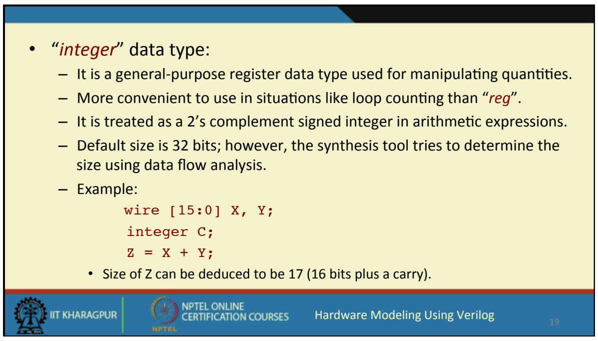
- Count value increases at the positive edge of the clock.
- If "rst" is high, the counter is reset at the positive edge of the next clock.



```
module  simple_counter  (clk, rst, count);
    input    clk, rst;
    output [31:0] count;
    reg [31:0]  count;

    always  @(posedge  clk or posedge rst)
    begin
        if  (rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

32-bit counter with asynchronous reset.

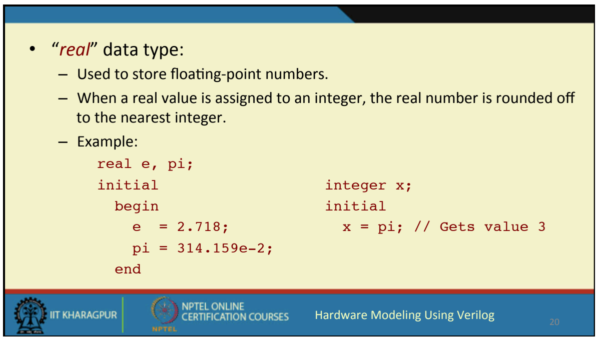
- Here reset occurs whenever "rst" goes high.
- Does not synchronize with clock.



- "*integer*" data type:
  - It is a general-purpose register data type used for manipulating quantities.
  - More convenient to use in situations like loop counting than "*reg*".
  - It is treated as a 2's complement signed integer in arithmetic expressions.
  - Default size is 32 bits; however, the synthesis tool tries to determine the size using data flow analysis.
  - Example:

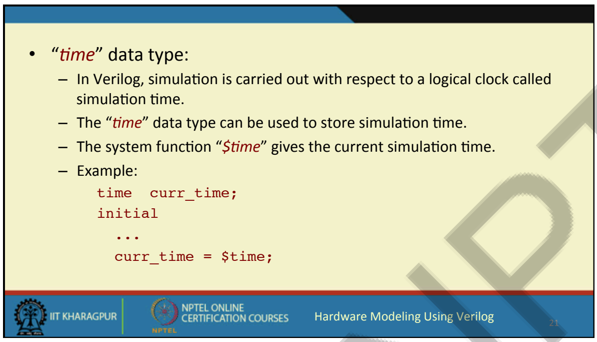
```
wire [15:0] X, Y;
 integer C;
 Z = X + Y;
```

- Size of Z can be deduced to be 17 (16 bits plus a carry).



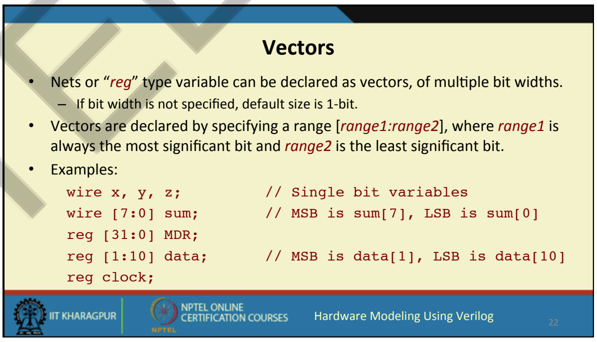
- "*real*" data type:
  - Used to store floating-point numbers.
  - When a real value is assigned to an integer, the real number is rounded off to the nearest integer.
  - Example:

```
real e, pi;
initial                    integer x;
  begin                    initial
    e  = 2.718;              x = pi; // Gets value 3
    pi = 314.159e-2;
  end
```



- "*time*" data type:
  - In Verilog, simulation is carried out with respect to a logical clock called simulation time.
  - The "*time*" data type can be used to store simulation time.
  - The system function "*$time*" gives the current simulation time.
  - Example:

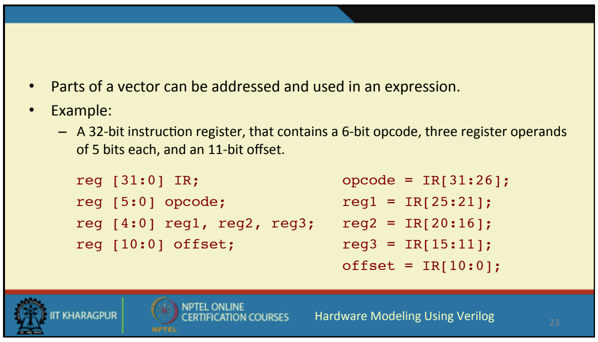
```
time  curr_time;
initial
  ...
  curr_time = $time;
```



Vectors

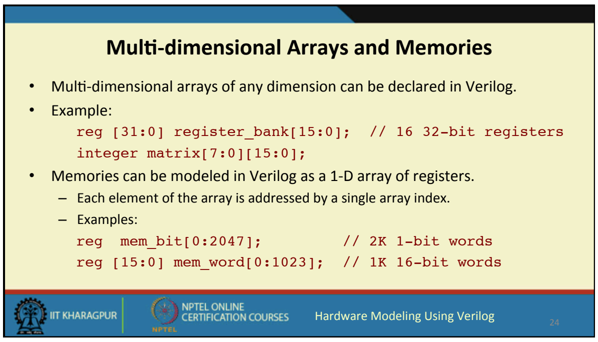
- Nets or "*reg*" type variable can be declared as vectors, of multiple bit widths.
  - If bit width is not specified, default size is 1-bit.
- Vectors are declared by specifying a range [*range1:range2*], where *range1* is always the most significant bit and *range2* is the least significant bit.
- Examples:

```
wire x, y, z;         // Single bit variables
wire [7:0] sum;       // MSB is sum[7], LSB is sum[0]
reg [31:0] MDR;
reg [1:10] data;      // MSB is data[1], LSB is data[10]
reg clock;
```



- Parts of a vector can be addressed and used in an expression.
- Example:
  - A 32-bit instruction register, that contains a 6-bit opcode, three register operands of 5 bits each, and an 11-bit offset.

```
reg [31:0] IR;                opcode = IR[31:26];
reg [5:0] opcode;             reg1 = IR[25:21];
reg [4:0] reg1, reg2, reg3;   reg2 = IR[20:16];
reg [10:0] offset;            reg3 = IR[15:11];
                              offset = IR[10:0];
```



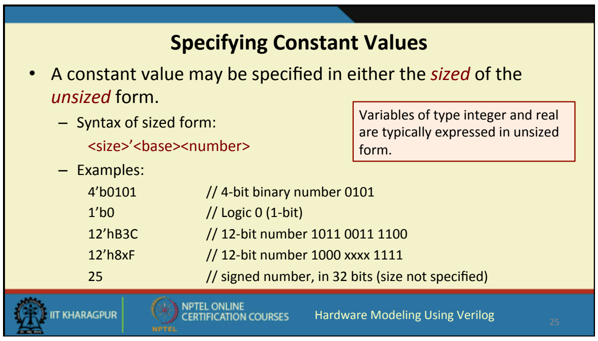
Multi-dimensional Arrays and Memories

- Multi-dimensional arrays of any dimension can be declared in Verilog.
- Example:

```
reg [31:0] register_bank[15:0];  // 16 32-bit registers
integer matrix[7:0][15:0];
```

- Memories can be modeled in Verilog as a 1-D array of registers.
  - Each element of the array is addressed by a single array index.
  - Examples:

```
reg  mem_bit[0:2047];         // 2K 1-bit words
reg [15:0] mem_word[0:1023];  // 1K 16-bit words
```



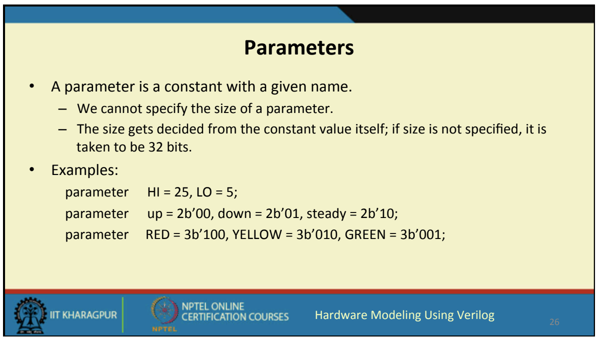
Specifying Constant Values

- A constant value may be specified in either the *sized* of the *unsized* form.
  - Syntax of sized form:

    <size>’<base><number>
  - Examples:

4’b0101	// 4-bit binary number 0101
1’b0	// Logic 0 (1-bit)
12’hB3C	// 12-bit number 1011 0011 1100
12’h8xF	// 12-bit number 1000 xxxx 1111
25	// signed number, in 32 bits (size not specified)

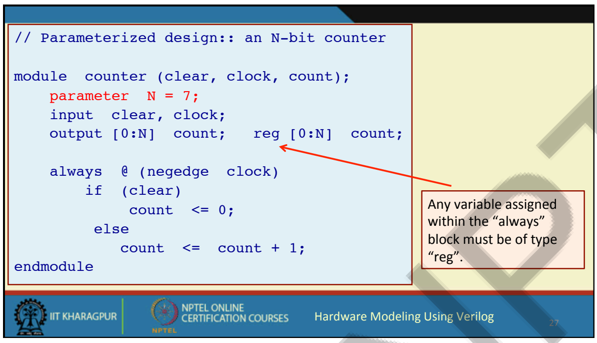
Variables of type integer and real are typically expressed in unsized form.



Parameters

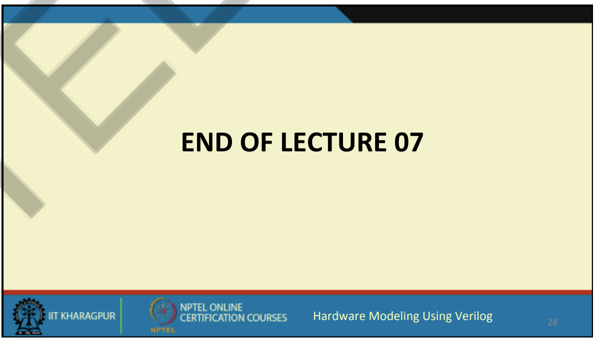
- A parameter is a constant with a given name.
  - We cannot specify the size of a parameter.
  - The size gets decided from the constant value itself; if size is not specified, it is taken to be 32 bits.
- Examples:

```
parameter    HI = 25, LO = 5;
parameter    up = 2b’00, down = 2b’01, steady = 2b’10;
parameter    RED = 3b’100, YELLOW = 3b’010, GREEN = 3b’001;
```

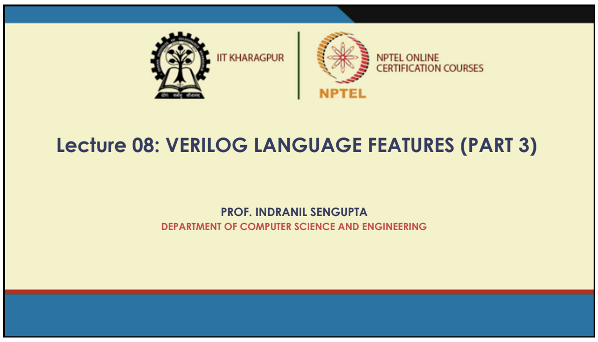


```
// Parameterized design:: an N-bit counter

module  counter (clear, clock, count);
    parameter  N = 7;
    input  clear, clock;
    output [0:N]  count;   reg [0:N]  count;

    always  @ (negedge  clock)
        if  (clear)
             count  <= 0;
         else
            count  <=  count + 1;
endmodule
```

Any variable assigned within the “always” block must be of type “reg”.



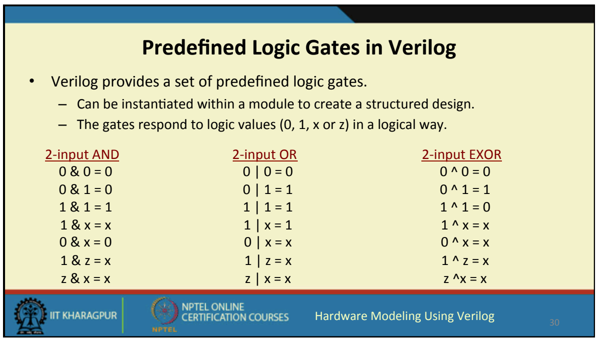
END OF LECTURE 07



IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NPTEL

Lecture 08: VERILOG LANGUAGE FEATURES (PART 3)

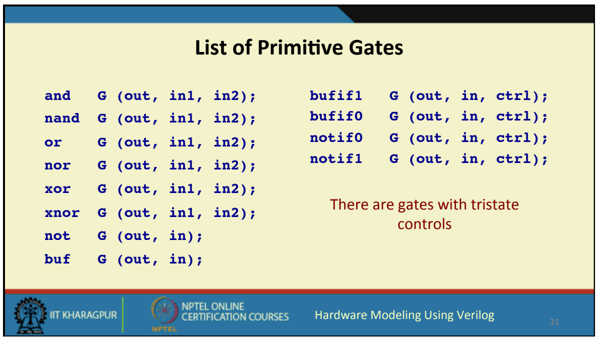
PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Predefined Logic Gates in Verilog

- Verilog provides a set of predefined logic gates.
  - Can be instantiated within a module to create a structured design.
  - The gates respond to logic values (0, 1, x or z) in a logical way.

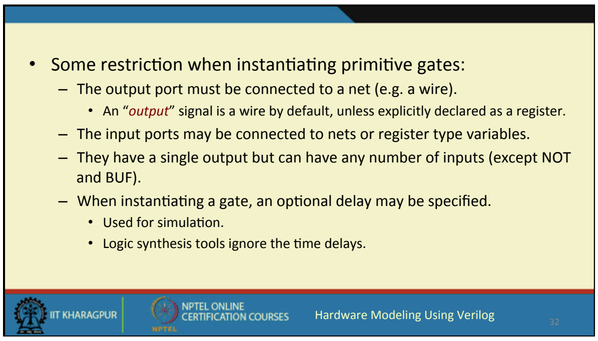
<table>
<tr><th>2-input AND</th><th>2-input OR</th><th>2-input EXOR</th></tr>
<tr><td>0 & 0 = 0</td><td>0 | 0 = 0</td><td>0 ^ 0 = 0</td></tr>
<tr><td>0 & 1 = 0</td><td>0 | 1 = 1</td><td>0 ^ 1 = 1</td></tr>
<tr><td>1 & 1 = 1</td><td>1 | 1 = 1</td><td>1 ^ 1 = 0</td></tr>
<tr><td>1 & x = x</td><td>1 | x = 1</td><td>1 ^ x = x</td></tr>
<tr><td>0 & x = 0</td><td>0 | x = x</td><td>0 ^ x = x</td></tr>
<tr><td>1 & z = x</td><td>1 | z = x</td><td>1 ^ z = x</td></tr>
<tr><td>z & x = x</td><td>z | x = x</td><td>z ^x = x</td></tr>
</table>



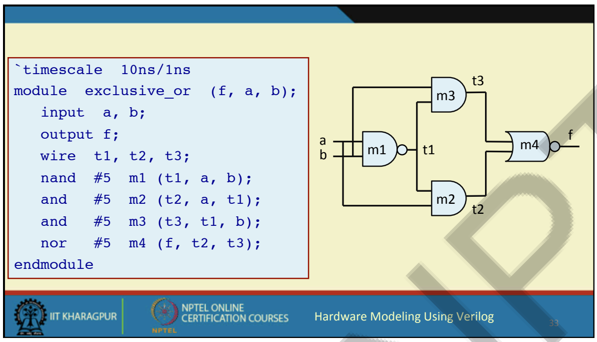
List of Primitive Gates

```
and   G (out, in1, in2);      bufif1   G (out, in, ctrl);
nand  G (out, in1, in2);      bufif0   G (out, in, ctrl);
or    G (out, in1, in2);      notif0   G (out, in, ctrl);
nor   G (out, in1, in2);      notif1   G (out, in, ctrl);
xor   G (out, in1, in2);
xnor  G (out, in1, in2);
not   G (out, in);
buf   G (out, in);
```

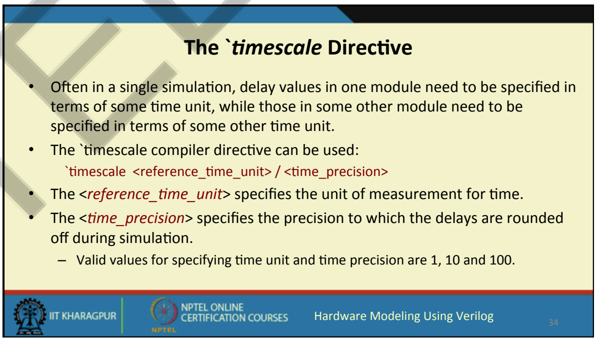
There are gates with tristate controls



- Some restriction when instantiating primitive gates:
  - The output port must be connected to a net (e.g. a wire).
    - An "*output*" signal is a wire by default, unless explicitly declared as a register.
  - The input ports may be connected to nets or register type variables.
  - They have a single output but can have any number of inputs (except NOT and BUF).
  - When instantiating a gate, an optional delay may be specified.
    - Used for simulation.
    - Logic synthesis tools ignore the time delays.

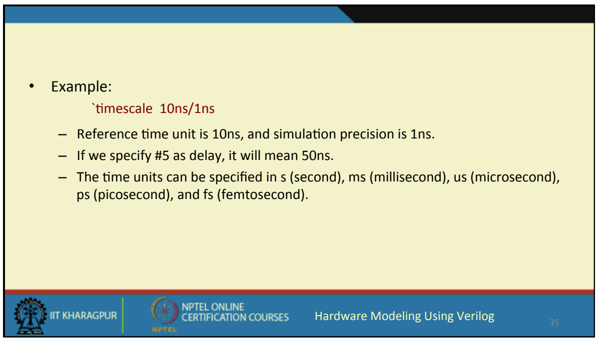


```
`timescale  10ns/1ns
module  exclusive_or  (f, a, b);
   input  a, b;
   output f;
   wire  t1, t2, t3;
   nand  #5  m1 (t1, a, b);
   and   #5  m2 (t2, a, t1);
   and   #5  m3 (t3, t1, b);
   nor   #5  m4 (f, t2, t3);
endmodule
```

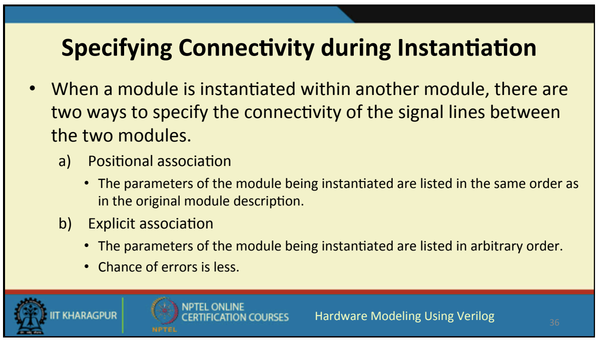


The `*timescale* Directive

- Often in a single simulation, delay values in one module need to be specified in terms of some time unit, while those in some other module need to be specified in terms of some other time unit.
- The `timescale compiler directive can be used:
  `timescale <reference_time_unit> / <time_precision>
- The <*reference_time_unit*> specifies the unit of measurement for time.
- The <*time_precision*> specifies the precision to which the delays are rounded off during simulation.
  - Valid values for specifying time unit and time precision are 1, 10 and 100.

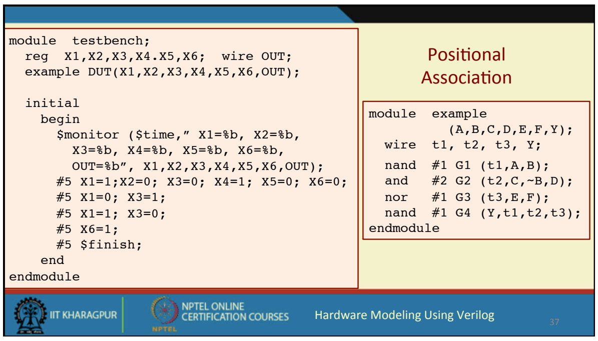


- Example:
  `timescale 10ns/1ns
  - Reference time unit is 10ns, and simulation precision is 1ns.
  - If we specify #5 as delay, it will mean 50ns.
  - The time units can be specified in s (second), ms (millisecond), us (microsecond), ps (picosecond), and fs (femtosecond).



Specifying Connectivity during Instantiation

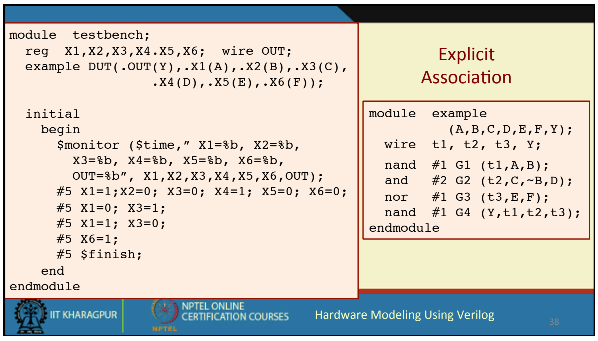
- When a module is instantiated within another module, there are two ways to specify the connectivity of the signal lines between the two modules.
  a) Positional association
     - The parameters of the module being instantiated are listed in the same order as in the original module description.
  b) Explicit association
     - The parameters of the module being instantiated are listed in arbitrary order.
     - Chance of errors is less.



```
module  testbench;
  reg  X1,X2,X3,X4.X5,X6;  wire OUT;
  example DUT(X1,X2,X3,X4,X5,X6,OUT);

  initial
    begin
      $monitor ($time,” X1=%b, X2=%b,
        X3=%b, X4=%b, X5=%b, X6=%b,
        OUT=%b”, X1,X2,X3,X4,X5,X6,OUT);
      #5 X1=1;X2=0; X3=0; X4=1; X5=0; X6=0;
      #5 X1=0; X3=1;
      #5 X1=1; X3=0;
      #5 X6=1;
      #5 $finish;
    end
endmodule
```

Positional Association

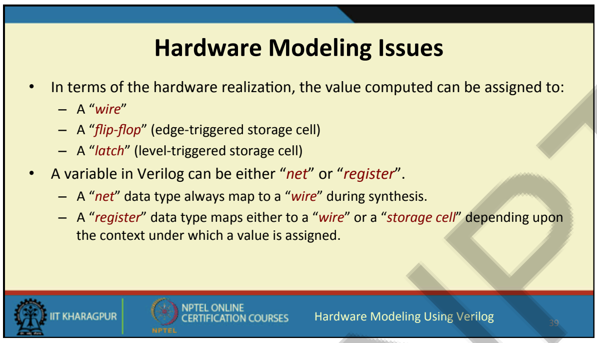
```
module  example
          (A,B,C,D,E,F,Y);
  wire  t1, t2, t3, Y;
  nand  #1 G1 (t1,A,B);
  and   #2 G2 (t2,C,~B,D);
  nor   #1 G3 (t3,E,F);
  nand  #1 G4 (Y,t1,t2,t3);
endmodule
```



```
module  testbench;
  reg  X1,X2,X3,X4.X5,X6;  wire OUT;
  example DUT(.OUT(Y),.X1(A),.X2(B),.X3(C),
              .X4(D),.X5(E),.X6(F));

  initial
    begin
      $monitor ($time,” X1=%b, X2=%b,
        X3=%b, X4=%b, X5=%b, X6=%b,
        OUT=%b”, X1,X2,X3,X4,X5,X6,OUT);
      #5 X1=1;X2=0; X3=0; X4=1; X5=0; X6=0;
      #5 X1=0; X3=1;
      #5 X1=1; X3=0;
      #5 X6=1;
      #5 $finish;
    end
endmodule
```

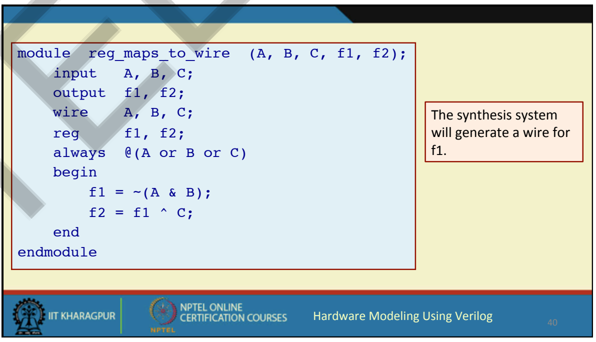
Explicit Association

```
module  example
          (A,B,C,D,E,F,Y);
  wire  t1, t2, t3, Y;
  nand  #1 G1 (t1,A,B);
  and   #2 G2 (t2,C,~B,D);
  nor   #1 G3 (t3,E,F);
  nand  #1 G4 (Y,t1,t2,t3);
endmodule
```



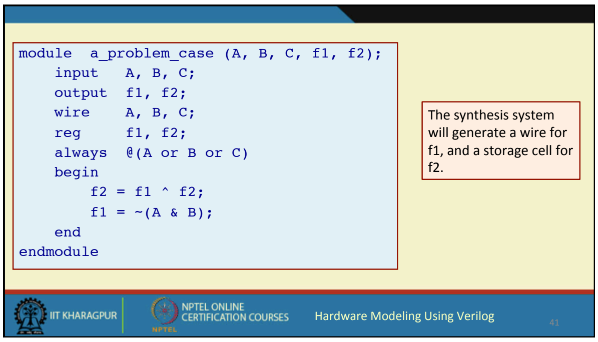
Hardware Modeling Issues

- In terms of the hardware realization, the value computed can be assigned to:
  - A “*wire*”
  - A “*flip-flop*” (edge-triggered storage cell)
  - A “*latch*” (level-triggered storage cell)
- A variable in Verilog can be either “*net*” or “*register*”.
  - A “*net*” data type always map to a “*wire*” during synthesis.
  - A “*register*” data type maps either to a “*wire*” or a “*storage cell*” depending upon the context under which a value is assigned.



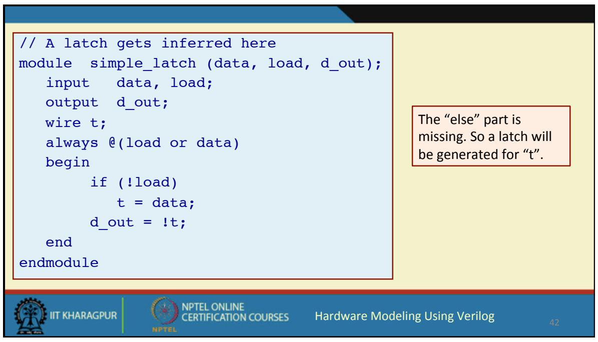
```
module  reg_maps_to_wire  (A, B, C, f1, f2);
    input   A, B, C;
    output  f1, f2;
    wire    A, B, C;
    reg     f1, f2;
    always  @(A or B or C)
    begin
        f1 = ~(A & B);
        f2 = f1 ^ C;
    end
endmodule
```

The synthesis system will generate a wire for f1.



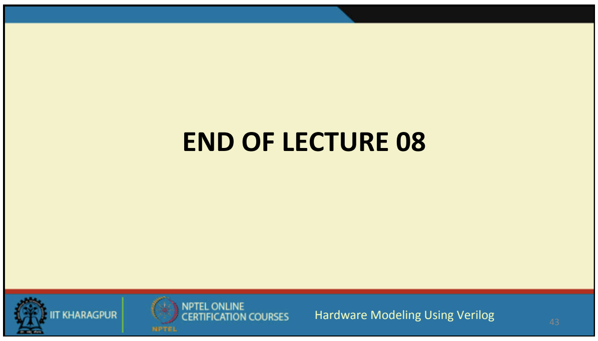
```
module  a_problem_case (A, B, C, f1, f2);
    input   A, B, C;
    output  f1, f2;
    wire    A, B, C;
    reg     f1, f2;
    always  @(A or B or C)
    begin
        f2 = f1 ^ f2;
        f1 = ~(A & B);
    end
endmodule
```

The synthesis system will generate a wire for f1, and a storage cell for f2.

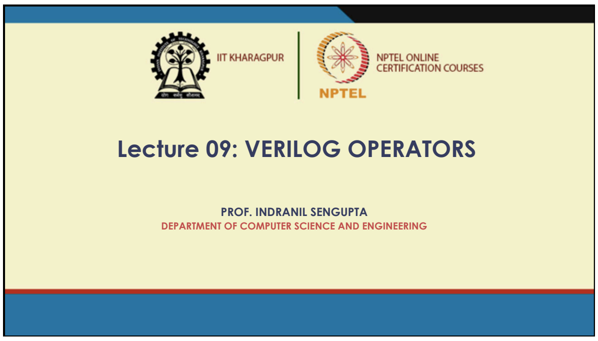


```
// A latch gets inferred here
module  simple_latch (data, load, d_out);
   input   data, load;
   output  d_out;
   wire t;
   always @(load or data)
   begin
        if (!load)
           t = data;
        d_out = !t;
   end
endmodule
```

The “else” part is missing. So a latch will be generated for “t”.



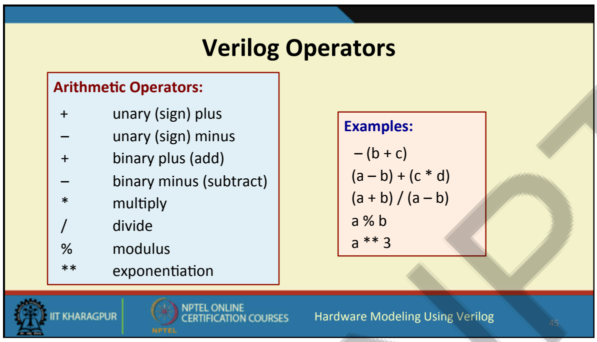
END OF LECTURE 08



IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES NPTEL

Lecture 09: VERILOG OPERATORS

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



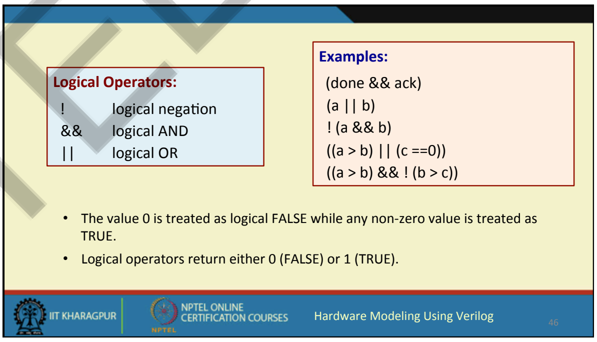
Verilog Operators

Arithmetic Operators:

+	unary (sign) plus
–	unary (sign) minus
+	binary plus (add)
–	binary minus (subtract)
*	multiply
/	divide
%	modulus
**	exponentiation

Examples:

```
– (b + c)
(a – b) + (c * d)
(a + b) / (a – b)
a % b
a ** 3
```



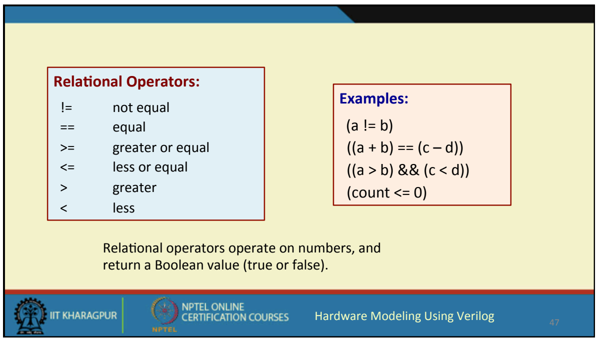
Logical Operators:

!	logical negation
&&	logical AND
||	logical OR

Examples:

```
(done && ack)
(a || b)
! (a && b)
((a > b) || (c ==0))
((a > b) && ! (b > c))
```

- The value 0 is treated as logical FALSE while any non-zero value is treated as TRUE.
- Logical operators return either 0 (FALSE) or 1 (TRUE).



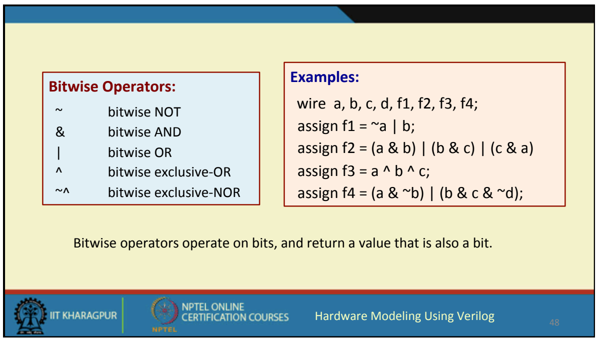
Relational Operators:

!=	not equal
==	equal
>=	greater or equal
<=	less or equal
>	greater
<	less

Examples:

```
(a != b)
((a + b) == (c – d))
((a > b) && (c < d))
(count <= 0)
```

Relational operators operate on numbers, and return a Boolean value (true or false).



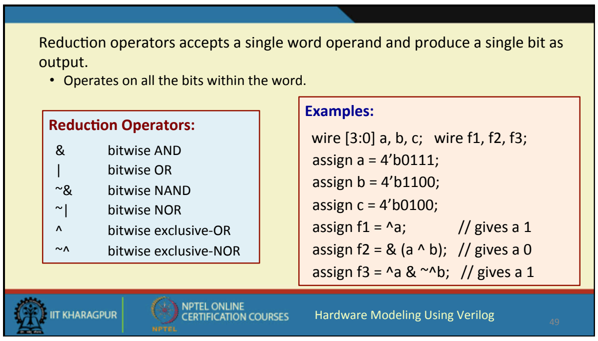
Bitwise Operators:

~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Examples:

```
wire  a, b, c, d, f1, f2, f3, f4;
assign f1 = ~a | b;
assign f2 = (a & b) | (b & c) | (c & a)
assign f3 = a ^ b ^ c;
assign f4 = (a & ~b) | (b & c & ~d);
```

Bitwise operators operate on bits, and return a value that is also a bit.



Reduction operators accepts a single word operand and produce a single bit as output.

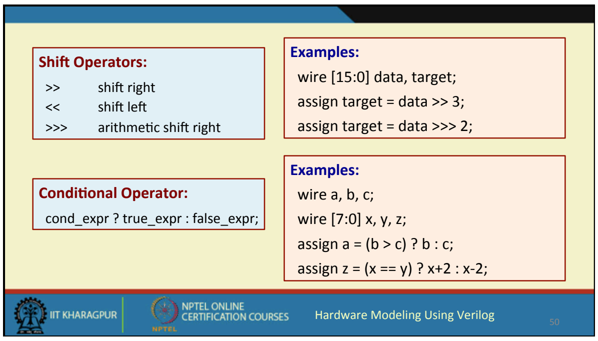
- Operates on all the bits within the word.

Reduction Operators:

&	bitwise AND
|	bitwise OR
~&	bitwise NAND
~|	bitwise NOR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Examples:

```
wire [3:0] a, b, c;   wire f1, f2, f3;
assign a = 4'b0111;
assign b = 4'b1100;
assign c = 4'b0100;
assign f1 = ^a;            // gives a 1
assign f2 = & (a ^ b);  // gives a 0
assign f3 = ^a & ~^b;   // gives a 1
```



Shift Operators:

>>	shift right
<<	shift left
>>>	arithmetic shift right

Examples:

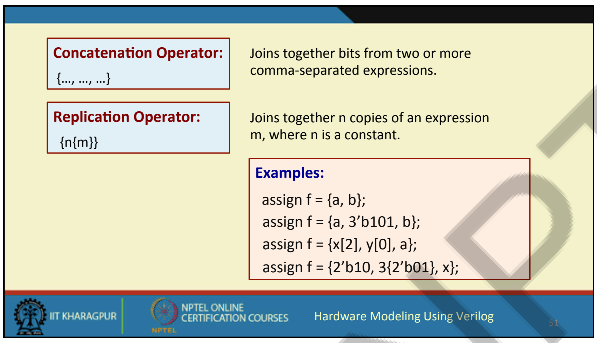
```
wire [15:0] data, target;
assign target = data >> 3;
assign target = data >>> 2;
```

Conditional Operator:

cond_expr ? true_expr : false_expr;

Examples:

```
wire a, b, c;
wire [7:0] x, y, z;
assign a = (b > c) ? b : c;
assign z = (x == y) ? x+2 : x-2;
```



Concatenation Operator:

{…, …, …}

Joins together bits from two or more comma-separated expressions.

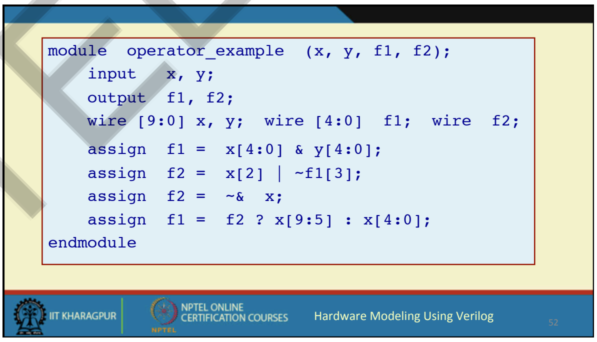
Replication Operator:

{n{m}}

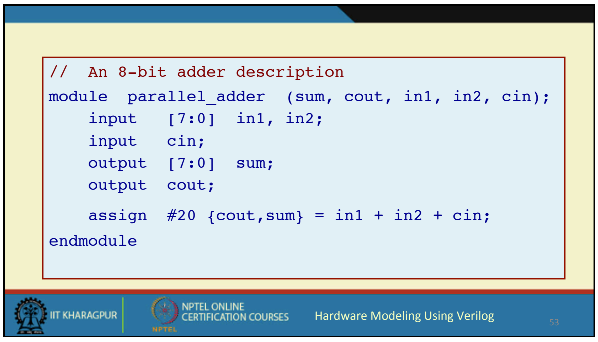
Joins together n copies of an expression m, where n is a constant.

Examples:

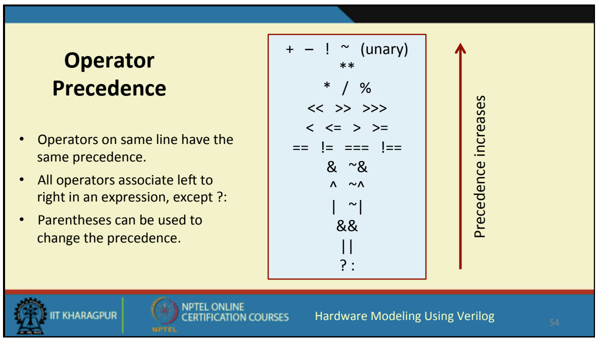
```
assign f = {a, b};
assign f = {a, 3'b101, b};
assign f = {x[2], y[0], a};
assign f = {2'b10, 3{2'b01}, x};
```



```
module  operator_example  (x, y, f1, f2);
    input   x, y;
    output  f1, f2;
    wire [9:0] x, y;  wire [4:0]  f1;  wire  f2;
    assign  f1 =  x[4:0] & y[4:0];
    assign  f2 =  x[2] | ~f1[3];
    assign  f2 =  ~&  x;
    assign  f1 =  f2 ? x[9:5] : x[4:0];
endmodule
```



```
//  An 8-bit adder description
module  parallel_adder  (sum, cout, in1, in2, cin);
    input   [7:0]  in1, in2;
    input   cin;
    output  [7:0]  sum;
    output  cout;

    assign  #20 {cout,sum} = in1 + in2 + cin;
endmodule
```

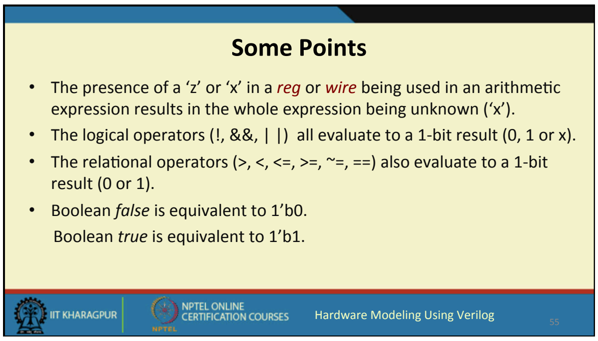


Operator Precedence

- Operators on same line have the same precedence.
- All operators associate left to right in an expression, except ?:
- Parentheses can be used to change the precedence.

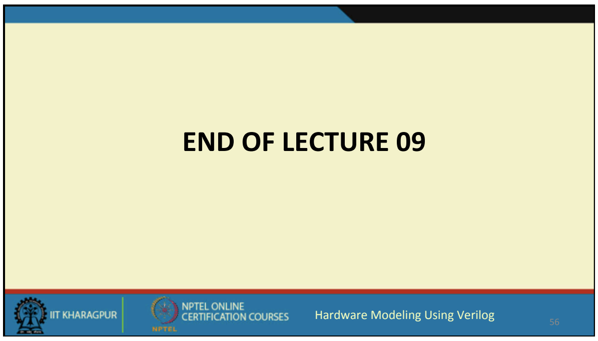
```
+  –  !  ~  (unary)
**
*  /  %
<<  >>  >>>
<  <=  >  >=
==  !=  ===  !==
&  ~&
^  ~^
|  ~|
&&
||
? :
```

Precedence increases

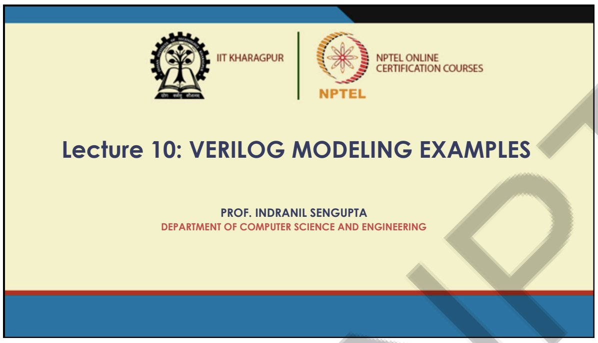


Some Points

- The presence of a 'z' or 'x' in a *reg* or *wire* being used in an arithmetic expression results in the whole expression being unknown ('x').
- The logical operators (!, &&, | |) all evaluate to a 1-bit result (0, 1 or x).
- The relational operators (>, <, <=, >=, ~=, ==) also evaluate to a 1-bit result (0 or 1).
- Boolean *false* is equivalent to 1'b0.
  Boolean *true* is equivalent to 1'b1.



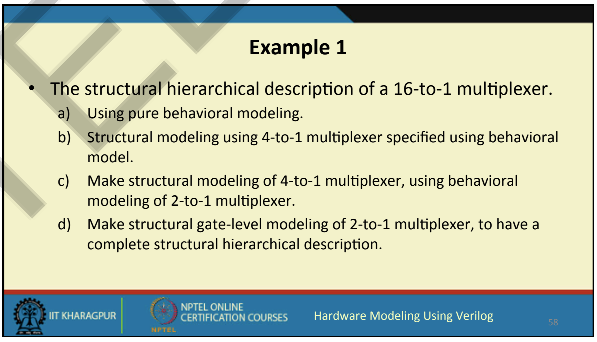
END OF LECTURE 09



IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NPTEL

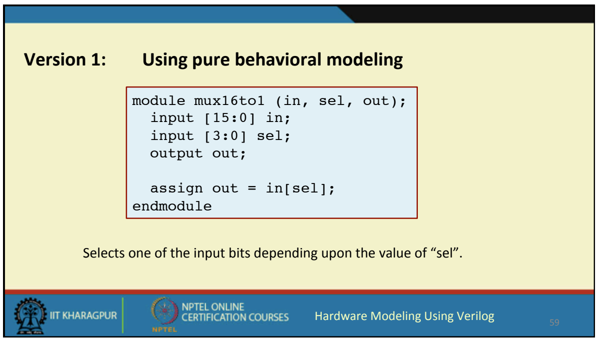
Lecture 10: VERILOG MODELING EXAMPLES

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



Example 1

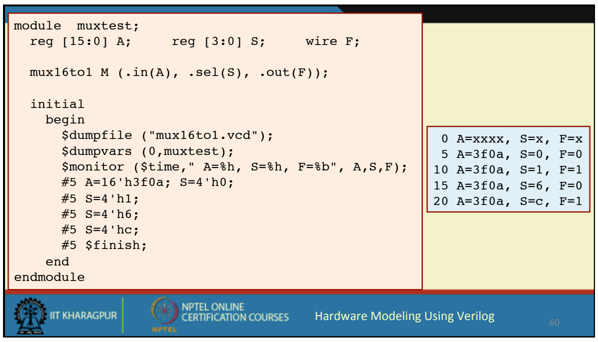
- The structural hierarchical description of a 16-to-1 multiplexer.
  a) Using pure behavioral modeling.
  b) Structural modeling using 4-to-1 multiplexer specified using behavioral model.
  c) Make structural modeling of 4-to-1 multiplexer, using behavioral modeling of 2-to-1 multiplexer.
  d) Make structural gate-level modeling of 2-to-1 multiplexer, to have a complete structural hierarchical description.



Version 1: **Using pure behavioral modeling**

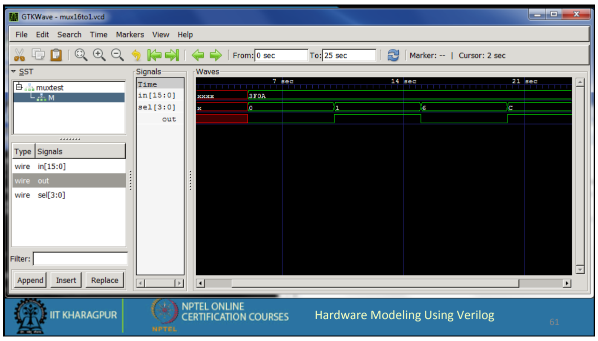
```
module mux16to1 (in, sel, out);
  input [15:0] in;
  input [3:0] sel;
  output out;

  assign out = in[sel];
endmodule
```

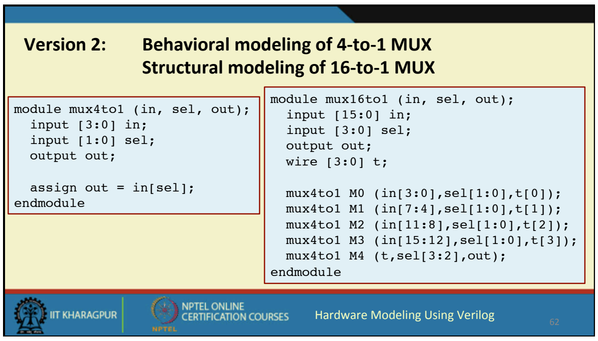
Selects one of the input bits depending upon the value of "sel".



```
module  muxtest;
  reg [15:0] A;     reg [3:0] S;     wire F;

  mux16to1 M (.in(A), .sel(S), .out(F));

  initial
    begin
      $dumpfile ("mux16to1.vcd");
      $dumpvars (0,muxtest);
      $monitor ($time," A=%h, S=%h, F=%b", A,S,F);
      #5 A=16'h3f0a; S=4'h0;
      #5 S=4'h1;
      #5 S=4'h6;
      #5 S=4'hc;
      #5 $finish;
    end
endmodule
```

```
 0 A=xxxx, S=x, F=x
 5 A=3f0a, S=0, F=0
10 A=3f0a, S=1, F=1
15 A=3f0a, S=6, F=0
20 A=3f0a, S=c, F=1
```

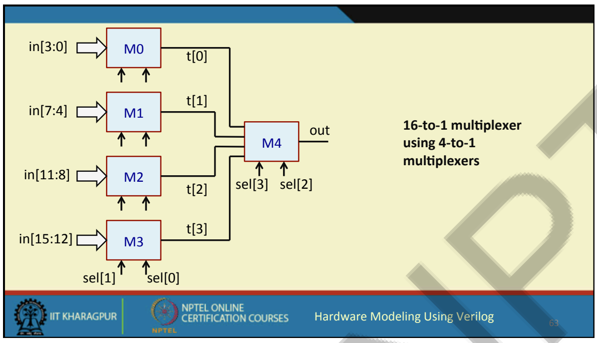




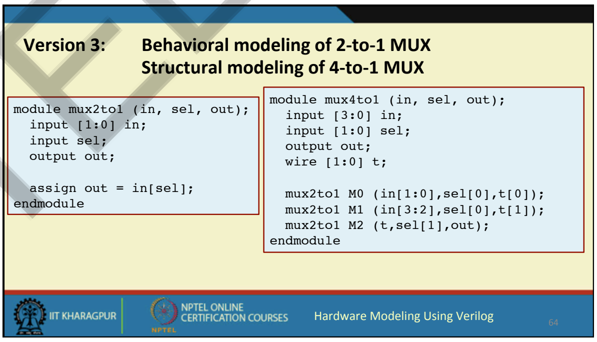
Version 2: Behavioral modeling of 4-to-1 MUX
Structural modeling of 16-to-1 MUX

```
module mux4to1 (in, sel, out);
  input [3:0] in;
  input [1:0] sel;
  output out;

  assign out = in[sel];
endmodule
```

```
module mux16to1 (in, sel, out);
  input [15:0] in;
  input [3:0] sel;
  output out;
  wire [3:0] t;

  mux4to1 M0 (in[3:0],sel[1:0],t[0]);
  mux4to1 M1 (in[7:4],sel[1:0],t[1]);
  mux4to1 M2 (in[11:8],sel[1:0],t[2]);
  mux4to1 M3 (in[15:12],sel[1:0],t[3]);
  mux4to1 M4 (t,sel[3:2],out);
endmodule
```



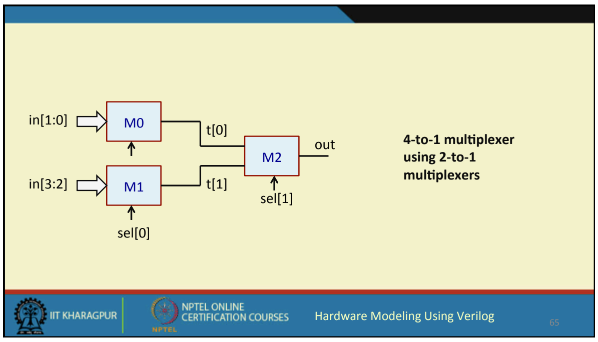
16-to-1 multiplexer using 4-to-1 multiplexers



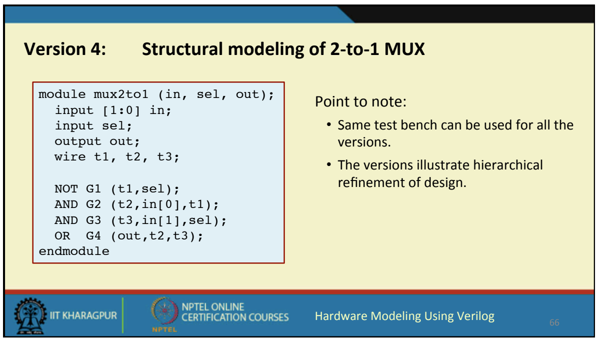
Version 3: Behavioral modeling of 2-to-1 MUX
Structural modeling of 4-to-1 MUX

```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;

  assign out = in[sel];
endmodule
```

```
module mux4to1 (in, sel, out);
  input [3:0] in;
  input [1:0] sel;
  output out;
  wire [1:0] t;

  mux2to1 M0 (in[1:0],sel[0],t[0]);
  mux2to1 M1 (in[3:2],sel[0],t[1]);
  mux2to1 M2 (t,sel[1],out);
endmodule
```



4-to-1 multiplexer using 2-to-1 multiplexers

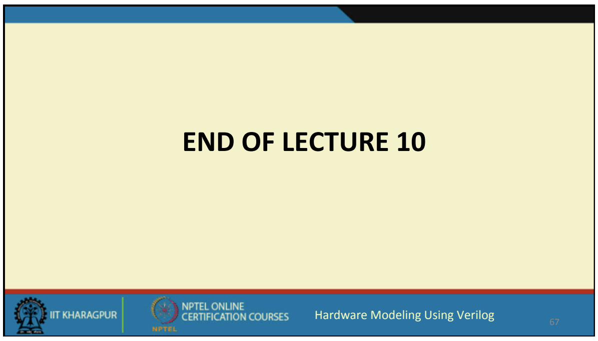


Version 4: Structural modeling of 2-to-1 MUX

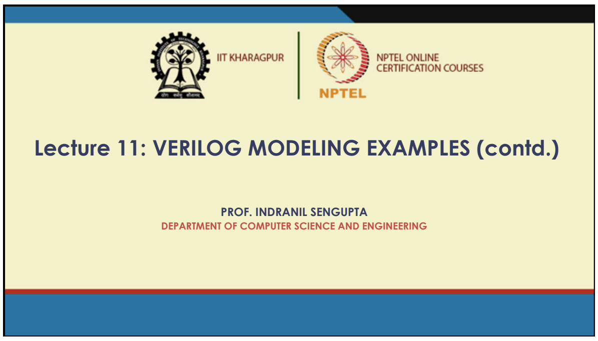
```
module mux2to1 (in, sel, out);
  input [1:0] in;
  input sel;
  output out;
  wire t1, t2, t3;

  NOT G1 (t1,sel);
  AND G2 (t2,in[0],t1);
  AND G3 (t3,in[1],sel);
  OR  G4 (out,t2,t3);
endmodule
```

Point to note:

- Same test bench can be used for all the versions.
- The versions illustrate hierarchical refinement of design.



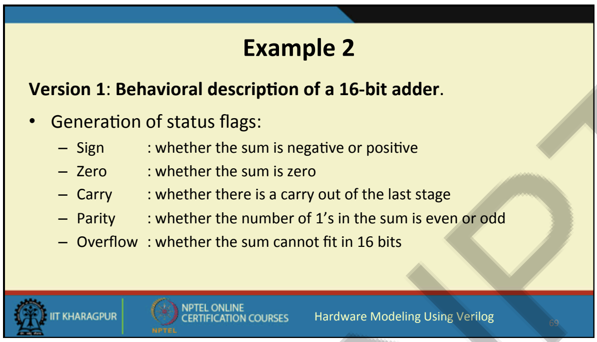
END OF LECTURE 10



IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES NPTEL

Lecture 11: VERILOG MODELING EXAMPLES (contd.)

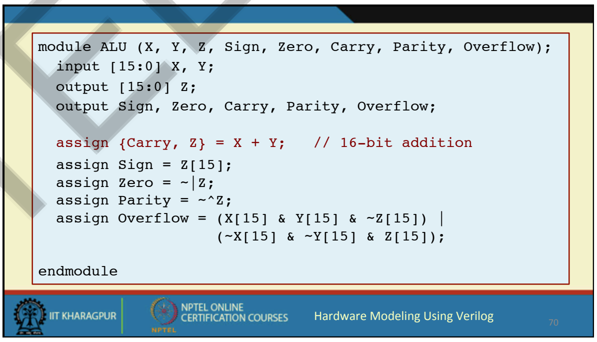
PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



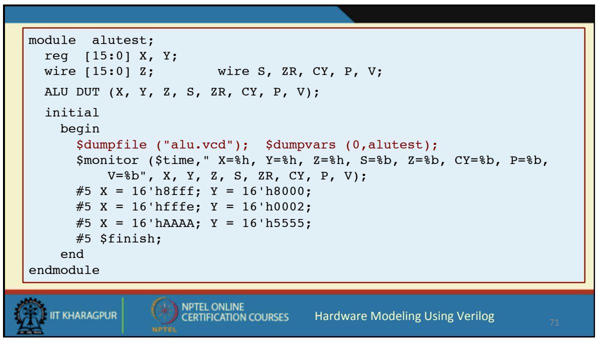
Example 2

Version 1: Behavioral description of a 16-bit adder.

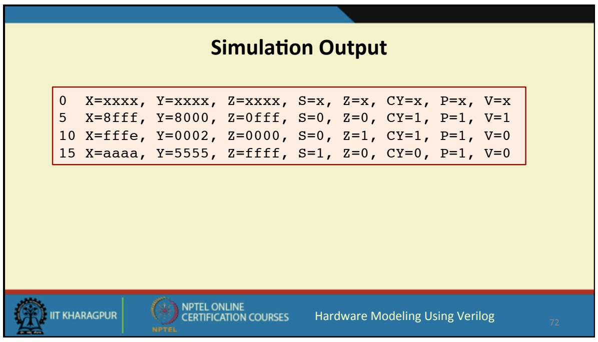
- Generation of status flags:
  - Sign : whether the sum is negative or positive
  - Zero : whether the sum is zero
  - Carry : whether there is a carry out of the last stage
  - Parity : whether the number of 1's in the sum is even or odd
  - Overflow : whether the sum cannot fit in 16 bits



```
module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
  input [15:0] X, Y;
  output [15:0] Z;
  output Sign, Zero, Carry, Parity, Overflow;

  assign {Carry, Z} = X + Y;   // 16-bit addition
  assign Sign = Z[15];
  assign Zero = ~|Z;
  assign Parity = ~^Z;
  assign Overflow = (X[15] & Y[15] & ~Z[15]) |
                    (~X[15] & ~Y[15] & Z[15]);

endmodule
```

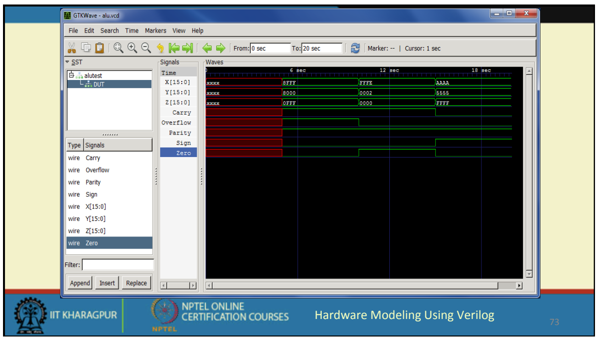


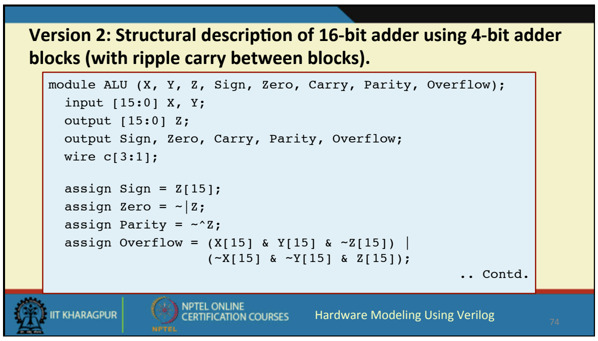
```
module  alutest;
  reg  [15:0] X, Y;
  wire [15:0] Z;        wire S, ZR, CY, P, V;
  ALU DUT (X, Y, Z, S, ZR, CY, P, V);
  initial
    begin
      $dumpfile ("alu.vcd");  $dumpvars (0,alutest);
      $monitor ($time," X=%h, Y=%h, Z=%h, S=%b, Z=%b, CY=%b, P=%b,
          V=%b", X, Y, Z, S, ZR, CY, P, V);
      #5 X = 16'h8fff; Y = 16'h8000;
      #5 X = 16'hfffe; Y = 16'h0002;
      #5 X = 16'hAAAA; Y = 16'h5555;
      #5 $finish;
    end
endmodule
```



Simulation Output

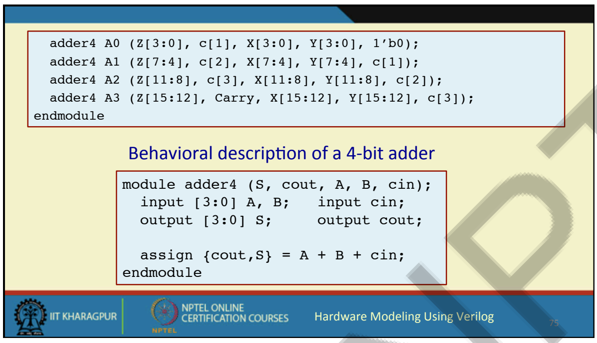
```
0  X=xxxx, Y=xxxx, Z=xxxx, S=x, Z=x, CY=x, P=x, V=x
5  X=8fff, Y=8000, Z=0fff, S=0, Z=0, CY=1, P=1, V=1
10 X=fffe, Y=0002, Z=0000, S=0, Z=1, CY=1, P=1, V=0
15 X=aaaa, Y=5555, Z=ffff, S=1, Z=0, CY=0, P=1, V=0
```





Version 2: Structural description of 16-bit adder using 4-bit adder blocks (with ripple carry between blocks).

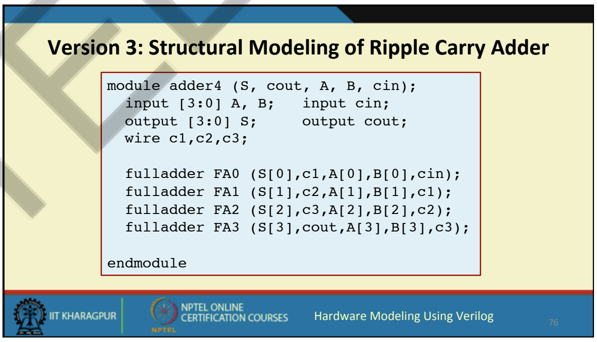
```
module ALU (X, Y, Z, Sign, Zero, Carry, Parity, Overflow);
  input [15:0] X, Y;
  output [15:0] Z;
  output Sign, Zero, Carry, Parity, Overflow;
  wire c[3:1];

  assign Sign = Z[15];
  assign Zero = ~|Z;
  assign Parity = ~^Z;
  assign Overflow = (X[15] & Y[15] & ~Z[15]) |
                    (~X[15] & ~Y[15] & Z[15]);
                                                  .. Contd.
```



```
  adder4 A0 (Z[3:0], c[1], X[3:0], Y[3:0], 1'b0);
  adder4 A1 (Z[7:4], c[2], X[7:4], Y[7:4], c[1]);
  adder4 A2 (Z[11:8], c[3], X[11:8], Y[11:8], c[2]);
  adder4 A3 (Z[15:12], Carry, X[15:12], Y[15:12], c[3]);
endmodule
```

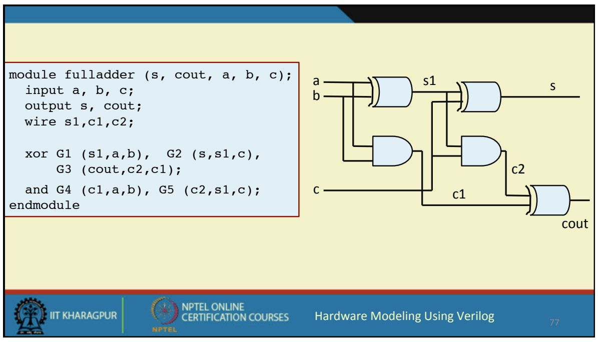
Behavioral description of a 4-bit adder

```
module adder4 (S, cout, A, B, cin);
  input [3:0] A, B;    input cin;
  output [3:0] S;      output cout;

  assign {cout,S} = A + B + cin;
endmodule
```

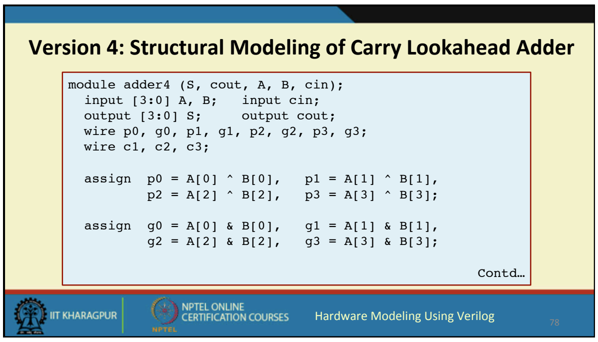


Version 3: Structural Modeling of Ripple Carry Adder

```
module adder4 (S, cout, A, B, cin);
  input [3:0] A, B;    input cin;
  output [3:0] S;      output cout;
  wire c1,c2,c3;

  fulladder FA0 (S[0],c1,A[0],B[0],cin);
  fulladder FA1 (S[1],c2,A[1],B[1],c1);
  fulladder FA2 (S[2],c3,A[2],B[2],c2);
  fulladder FA3 (S[3],cout,A[3],B[3],c3);

endmodule
```

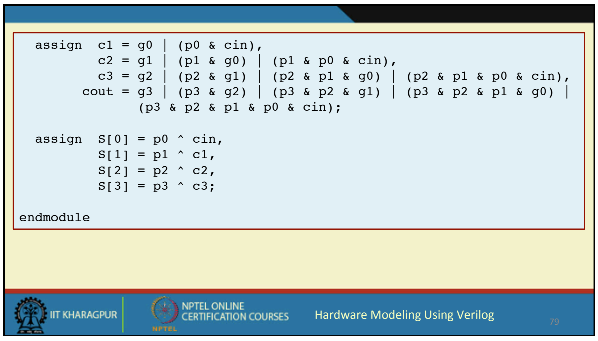


```
module fulladder (s, cout, a, b, c);
  input a, b, c;
  output s, cout;
  wire s1,c1,c2;

  xor G1 (s1,a,b),  G2 (s,s1,c),
      G3 (cout,c2,c1);

  and G4 (c1,a,b), G5 (c2,s1,c);
endmodule
```

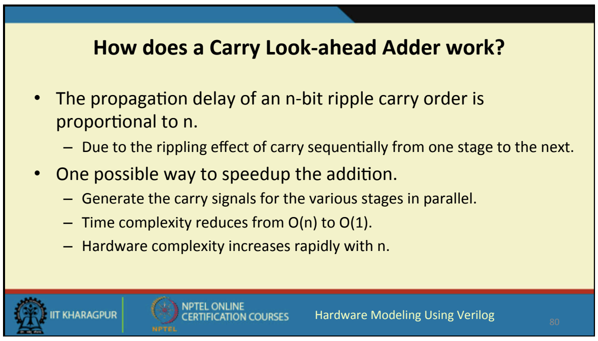


Version 4: Structural Modeling of Carry Lookahead Adder

```
module adder4 (S, cout, A, B, cin);
  input [3:0] A, B;   input cin;
  output [3:0] S;     output cout;
  wire p0, g0, p1, g1, p2, g2, p3, g3;
  wire c1, c2, c3;

  assign  p0 = A[0] ^ B[0],   p1 = A[1] ^ B[1],
          p2 = A[2] ^ B[2],   p3 = A[3] ^ B[3];

  assign  g0 = A[0] & B[0],   g1 = A[1] & B[1],
          g2 = A[2] & B[2],   g3 = A[3] & B[3];

                                           Contd…
```

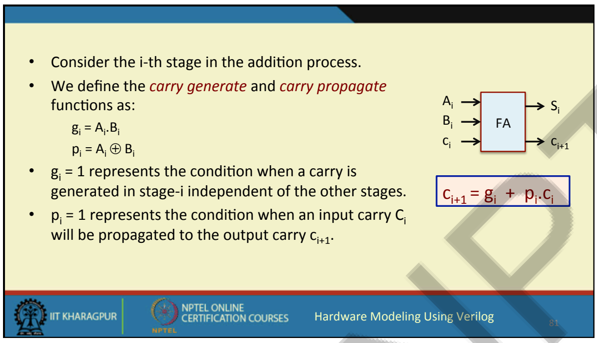


```
  assign  c1 = g0 | (p0 & cin),
          c2 = g1 | (p1 & g0) | (p1 & p0 & cin),
          c3 = g2 | (p2 & g1) | (p2 & p1 & g0) | (p2 & p1 & p0 & cin),
        cout = g3 | (p3 & g2) | (p3 & p2 & g1) | (p3 & p2 & p1 & g0) |
                  (p3 & p2 & p1 & p0 & cin);

  assign  S[0] = p0 ^ cin,
          S[1] = p1 ^ c1,
          S[2] = p2 ^ c2,
          S[3] = p3 ^ c3;

endmodule
```



How does a Carry Look-ahead Adder work?

- The propagation delay of an n-bit ripple carry order is proportional to n.
  - Due to the rippling effect of carry sequentially from one stage to the next.
- One possible way to speedup the addition.
  - Generate the carry signals for the various stages in parallel.
  - Time complexity reduces from O(n) to O(1).
  - Hardware complexity increases rapidly with n.



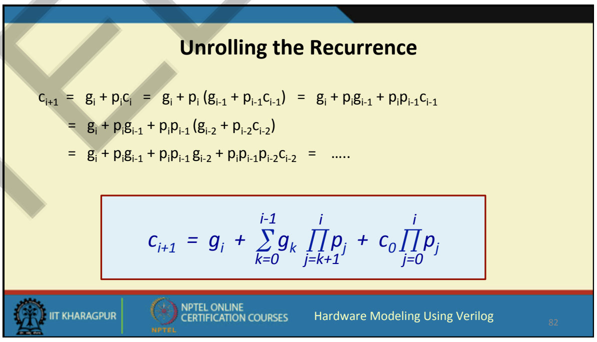
- Consider the i-th stage in the addition process.
- We define the *carry generate* and *carry propagate* functions as:

  $g_i = A_i.B_i$

  $p_i = A_i \oplus B_i$
- $g_i = 1$ represents the condition when a carry is generated in stage-i independent of the other stages.
- $p_i = 1$ represents the condition when an input carry C_i will be propagated to the output carry c_{i+1}.

A_i
B_i
c_i
FA
S_i
c_{i+1}

$\boxed{c_{i+1} = g_i + p_i.c_i}$



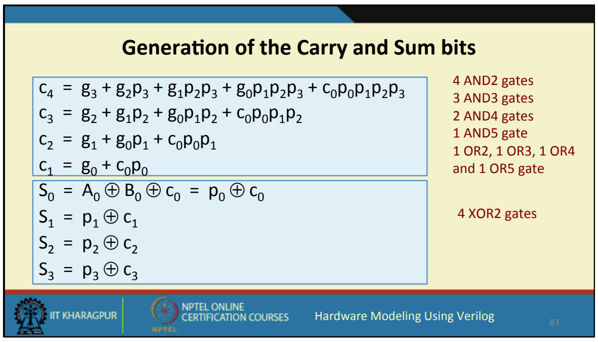
Unrolling the Recurrence

$$c_{i+1} = g_i + p_ic_i = g_i + p_i\,(g_{i-1} + p_{i-1}c_{i-1}) = g_i + p_ig_{i-1} + p_ip_{i-1}c_{i-1}$$

$$= g_i + p_ig_{i-1} + p_ip_{i-1}\,(g_{i-2} + p_{i-2}c_{i-2})$$

$$= g_i + p_ig_{i-1} + p_ip_{i-1}\,g_{i-2} + p_ip_{i-1}p_{i-2}c_{i-2} = \;.....$$

$$\boxed{c_{i+1} = g_i + \sum_{k=0}^{i-1} g_k \prod_{j=k+1}^{i} p_j + c_0 \prod_{j=0}^{i} p_j}$$



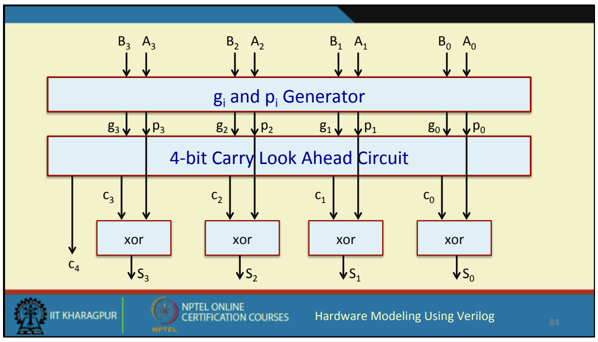
Generation of the Carry and Sum bits

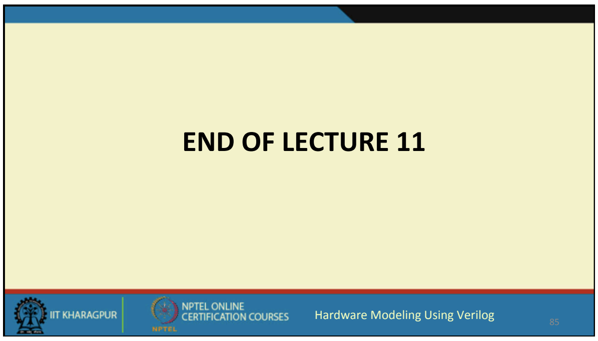
$$c_4 = g_3 + g_2p_3 + g_1p_2p_3 + g_0p_1p_2p_3 + c_0p_0p_1p_2p_3$$
$$c_3 = g_2 + g_1p_2 + g_0p_1p_2 + c_0p_0p_1p_2$$
$$c_2 = g_1 + g_0p_1 + c_0p_0p_1$$
$$c_1 = g_0 + c_0p_0$$

4 AND2 gates
3 AND3 gates
2 AND4 gates
1 AND5 gate
1 OR2, 1 OR3, 1 OR4
and 1 OR5 gate

$$S_0 = A_0 \oplus B_0 \oplus c_0 = p_0 \oplus c_0$$
$$S_1 = p_1 \oplus c_1$$
$$S_2 = p_2 \oplus c_2$$
$$S_3 = p_3 \oplus c_3$$

4 XOR2 gates





END OF LECTURE 11

IIT KHARAGPUR

NPTEL ONLINE CERTIFICATION COURSES

Hardware Modeling Using Verilog

85