ISTANBUL TECHNICAL UNIVERSITY

Digital System Design & Applications

Verilog HDL

RES. ASST. FıRAT KULA

Introduction

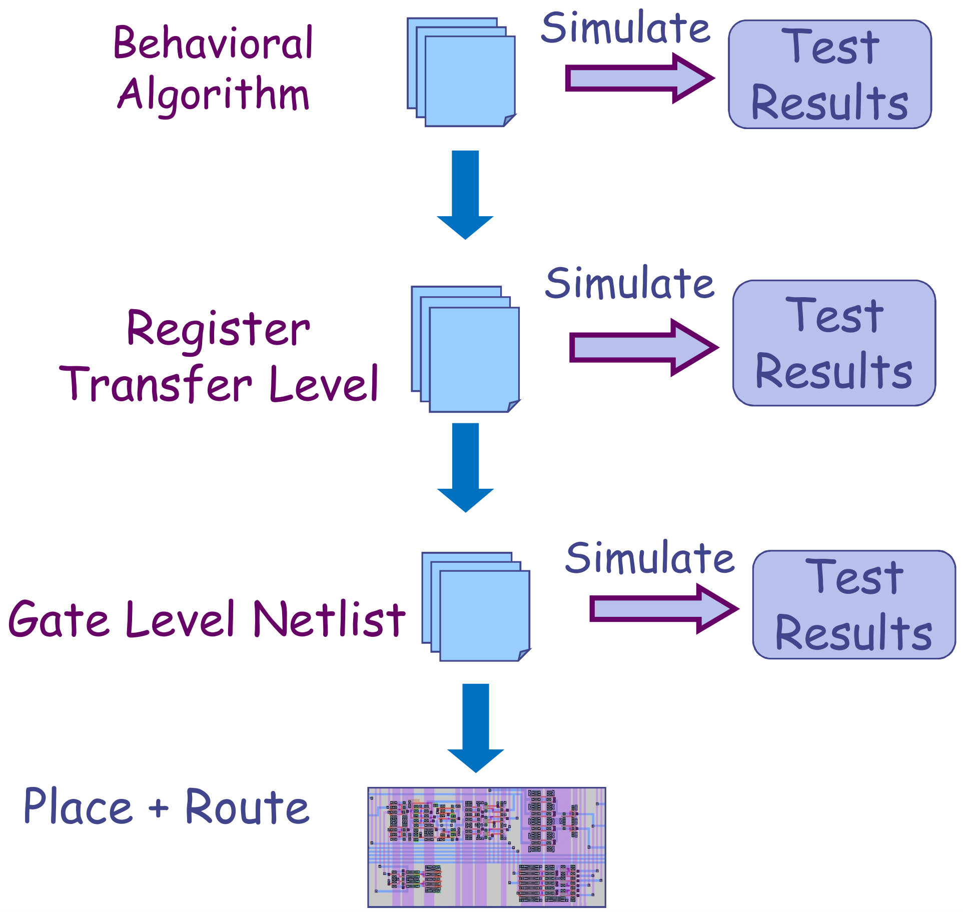
Hardware Description Languages (HDLs) are used to describe digital logic circuits without being tied to a specific electronic technology.

HDLs uses **Register-transfer level (RTL)** abstraction model.

Register-transfer level (RTL) means the flow of digital signals (data) between hardware registers and logical operators.
RTL is a high-level represantation of a digital circuit.
RTL does not consider the physical hardware (real hardware).

The details of gates and their interconnections **(Gate-Level Netlist)** are extracted by logic synthesis tools (e.g. Vivado, Genus) from the RTL description.

Introduction



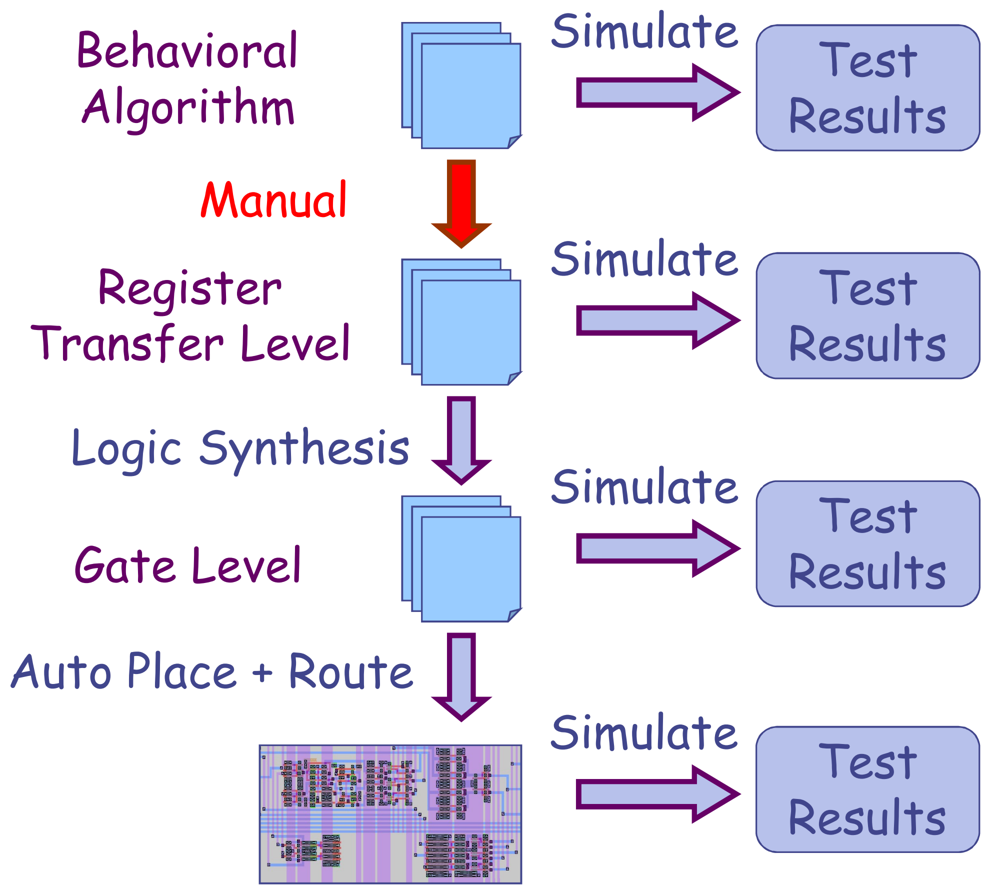
- Coding HDL, C - C++, MATLAB or Python.
- Describing the behavior of the circuit.
- **Textual** represantation.

- High-level represantation of the circuit.
- Describing registers and combinatorial logic.
- **Schematical** represantation at high-level.

- Generating gates and their interconnections using synthesis tools.
- **Schematical** represantation.

- Generate the layout (physical chip/circuit) by using placement/routing tools.
- **Physical** represantation.

Introduction



❖ **HDL tools are used for automatic translation.**

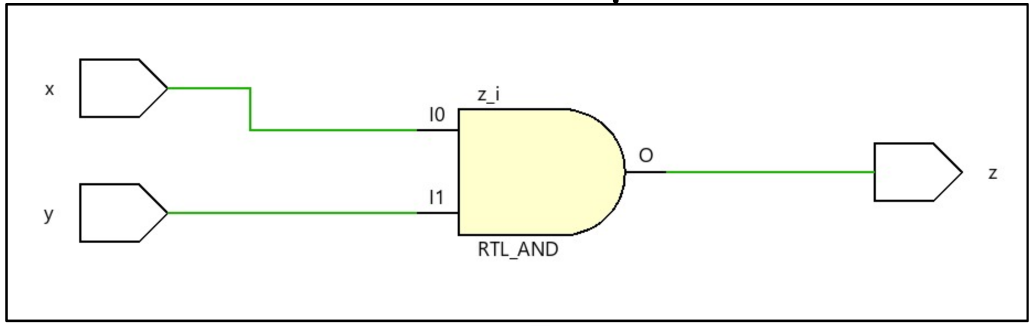
❖ In each step we need to simulate and verify our design.
- Behavioral Simulation
- RTL Sim.
- Post Synthesis Sim.
- Post Imp Sim.

Introduction

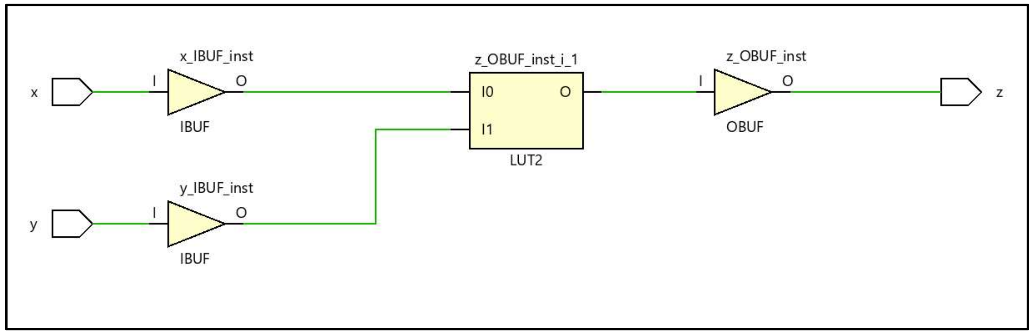
Verilog Code

```
module and2( output z, input x, input y);

    assign z = x&y;

endmodule
```

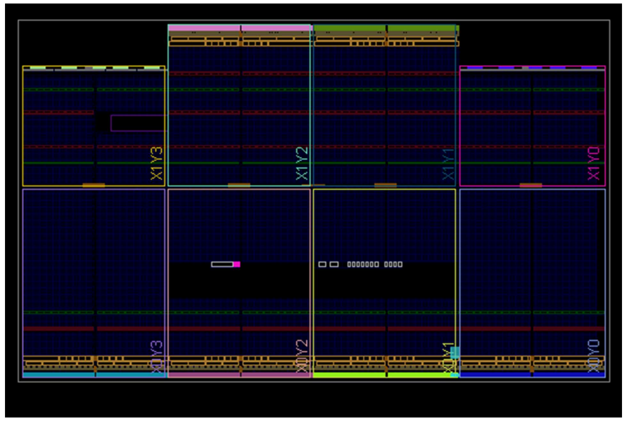
RTL Description



Gate-Level Netlist



Layout



Taken from Vivado, Xilinx Nexsys 4 DDR FPGA Chip
Artix-7 XC7A100T-1CSG324C

Introduction

The most commonly used HDLs are **Verilog HDL** and **VHDL**.

.

For simulation and Verification:

VHDL,SystemVerilog,Verilog,UVM

Intel Altera FPGAs: Quartus, Modelsim (old)

Xilinx FPGAs: Vitis-Vivado

For Asic Designs: Cadence Virtuoso

Xcelium (RTL Sim)

Genus (Synthesis)

Innovus (Place/Route)

Calibre (DRC Check), Quantus(Noise), Tempus(Temperature), Voltus(Power Integrity) and others.

Introduction

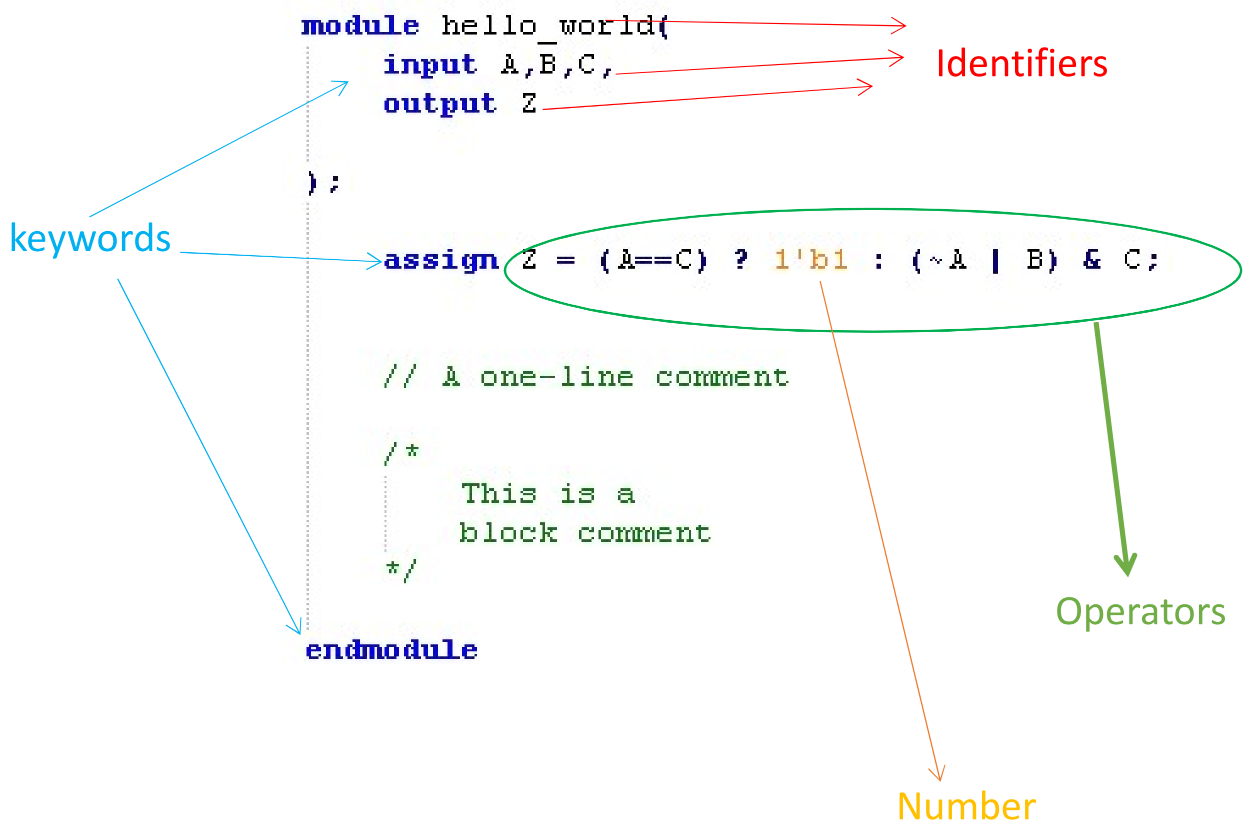
Verilog is **NOT** a programming language.

In terms of hardware description, Verilog **ONLY** describes the behavior of digital circuits. Testbenches are antoher story.

Verilog code is inherently **concurrent** contrary to regular programming languages, which are sequential (C, C++, Python).

Verilog Basics - Lexical Conventions

- ❑ White space: Spaces, tabs, newlines…
- ❑ Comments: One-line and block comments, same as C,C++
- ❑ Operators
- ❑ Numbers
- ❑ Strings
- ❑ Identifiers
- ❑ Keywords
- ❑ System Tasks/Functions
- ❑ Compiler Directives
- ❑ Attributes



```
module hello_world(
    input A,B,C,
    output Z

);

    assign Z = (A==C) ? 1'b1 : (~A | B) & C;

    // A one-line comment

    /*
        This is a
        block comment
    */

endmodule
```

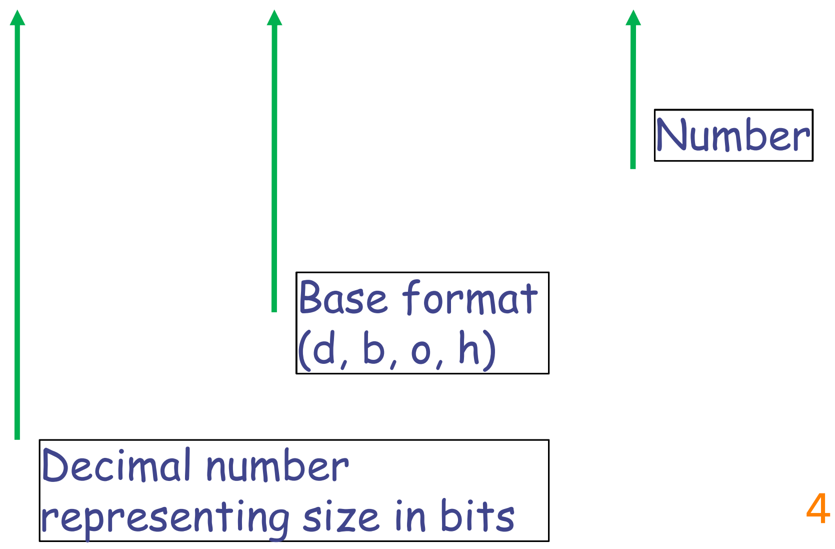
Verilog Basics - Value Set

- In hardware modeling aspect Verilog HDL consists of four basic values:

Value	What it represents
0	Logic zero, or a false condition
1	Logic one, or a true condition
x	Unknown logic value
z	High-impedance state

Verilog Basics - Number Representations

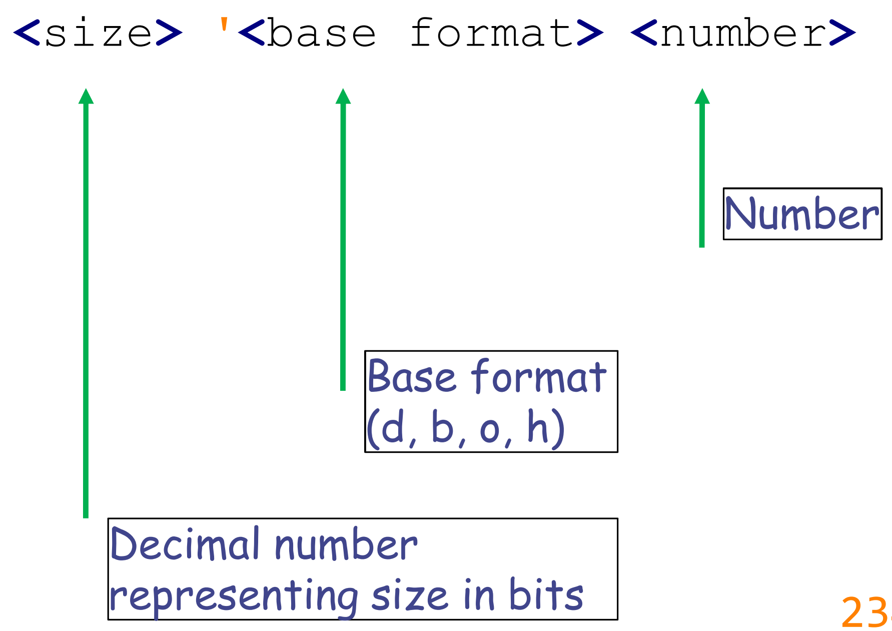
```
<size> '<base format> <number>
```



- <size> is written only in decimal and specifies the number of bits in the number.
- Base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O).

```
4'b1111   // This is a 4-bit binary number
12'habc   // This is a 12-bit hexadecimal number
16'd255   // This is a 16-bit decimal number.
```

Verilog Basics - Number Representations



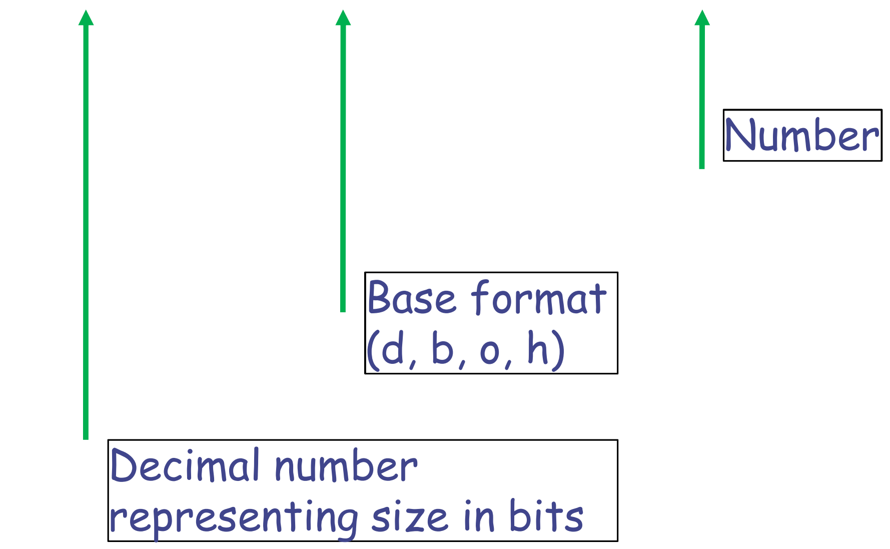
```
<size> '<base format> <number>
```

- Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).
- Numbers that are specified without a <base format> specification are decimal numbers by default.

```
23456 // This is a 32-bit decimal number by default
'hc3  // This is a 32-bit hexadecimal number
'o21  // This is a 32-bit octal number
```

Verilog Basics - Number Representations

```
<size> '<base format> <number>
```



- Negative numbers can be specified by putting a minus sign before the size for a constant number.
- Negative numbers are always specified as 2's complement form of the corresponding number.

```
-6'd3   // 6-bit negative number stored as 2's complement of 3
         //  111101 => -3
```

Verilog Basics - Number Representations

- Verilog has two symbols for **unknown** and **high impedance** values. An unknown value is denoted by an **X**. A high impedance value is denoted by **Z**.
- If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z.

```
12'h13x   // This is a 12-bit hex number; 4 least significant bits unknown
6'hx      // This is a 6-bit unknown hex number
32'bz     // This is a 32-bit high impedance number
```

Verilog Basics - Number Representations

- An underscore character "_" is allowed anywhere in a number except the first character.
- Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.

```
12'b1111_0000_1010  // Use of underline characters for readability
```

Verilog Basics – Modules

- ❑ Modules are the basic building blocks in Verilog
- ❑ Communicates with the outside world via its inputs and outputs (ports)
- ❑ A module has a unique name, a port list and a parameter list (more on this later)

❑ Two alternative module declarations:

```
module <module_name>
( <port_name_list> );

    <declare_port_directions>

    /*
        Functionality of the circuitry
        described by this module
    */

endmodule
```

Verilog-1995 style

```
module <module_name>
( <port_list_with_directions> );

    /*
        Functionality of the circuitry
        described by this module
    */

endmodule
```

Verilog-2001 style

Verilog Basics – Modules

- Inputs and outputs of a module are called PORTS of this module
- In Verilog, a port can have 3 possible directions:

Keyword	Behaviour
input	Input port
output	Output port
inout	Bidirectional port

Verilog Basics - Modules

- Example module declarations:

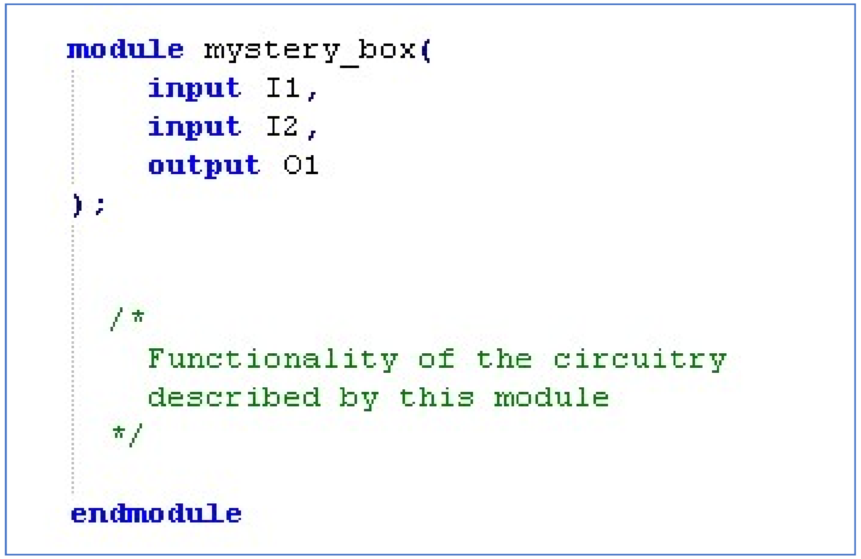
```
module mystery_box( I1, I2, O1);

    input I1,I2;
    output O1;

  /*
      Functionality of the circuitry
      described by this module
  */

endmodule
```

Verilog-1995 style

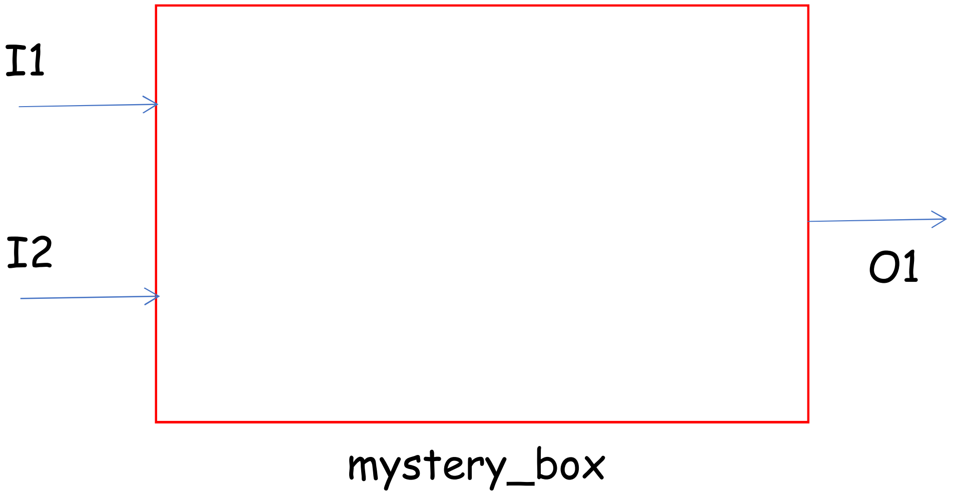
```
module mystery_box(
    input I1,I2,
    output O1
);


   /*
     Functionality of the circuitry
     described by this module
   */

endmodule
```

Verilog-2001 style

Verilog Basics - Modules



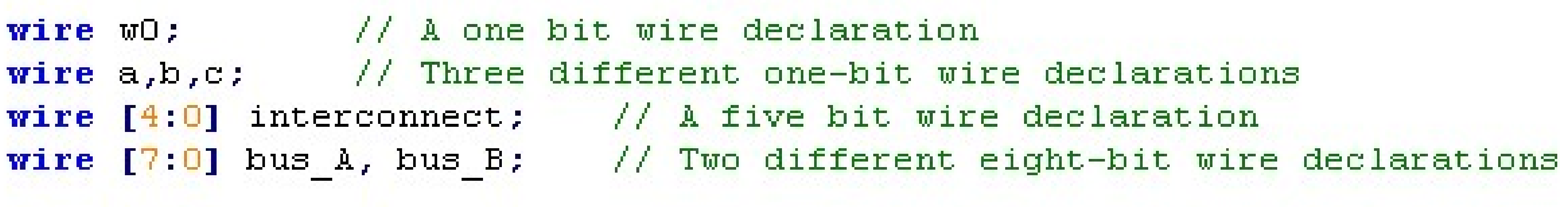
```
module mystery_box(
    input I1,
    input I2,
    output O1
);


  /*
    Functionality of the circuitry
    described by this module
  */

endmodule
```



- This current code represents an empty box with two single-bit inputs named I1,I2; and a single-bit output named O1

Verilog Basics – wire keyword

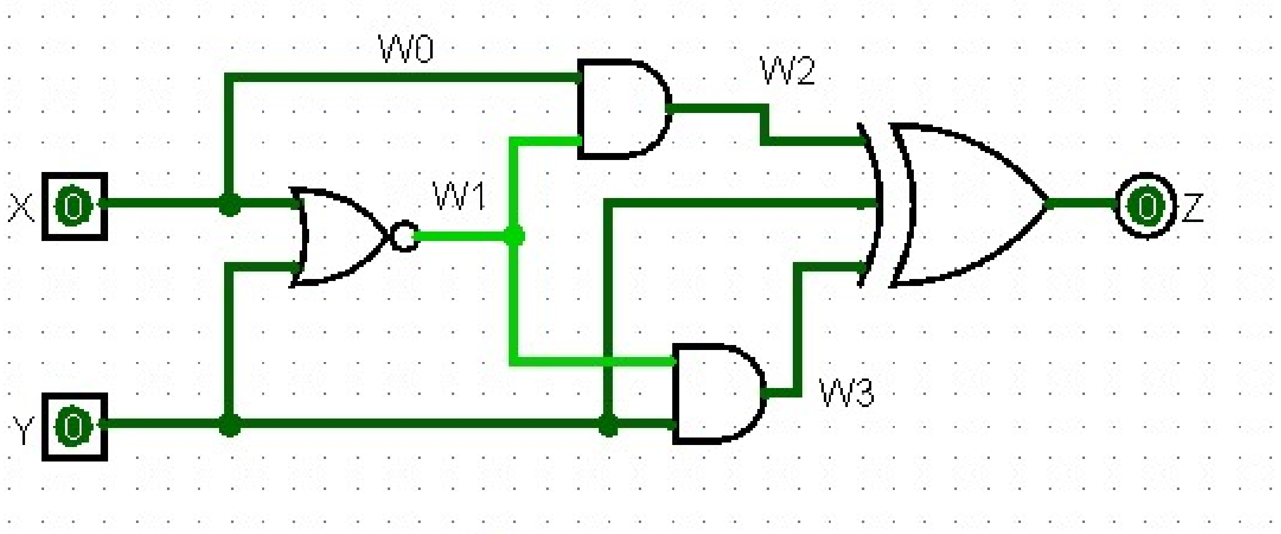
- "wire" keyword represent physical nets in the design.
- Like in real circuits, nets have values that are continuously driven on them
- **Ports** are wires by default (originated from Verilog-1995)
- Wires can be declared inside a module body.



```
wire w0;          // A one bit wire declaration
wire a,b,c;       // Three different one-bit wire declarations
wire [4:0] interconnect;    // A five bit wire declaration
wire [7:0] bus_A, bus_B;    // Two different eight-bit wire declarations
```

Verilog Basics – wire keyword

- ❑"wire" keyword represent physical nets in the design.
- ❑Like in real circuits, nets have values that are continuously driven on them
- ❑**Ports** are wires by default (originated from Verilog-1995)
- ❑Wires can be declared inside a module body.



```
wire W0,W1,W2,W3;
```

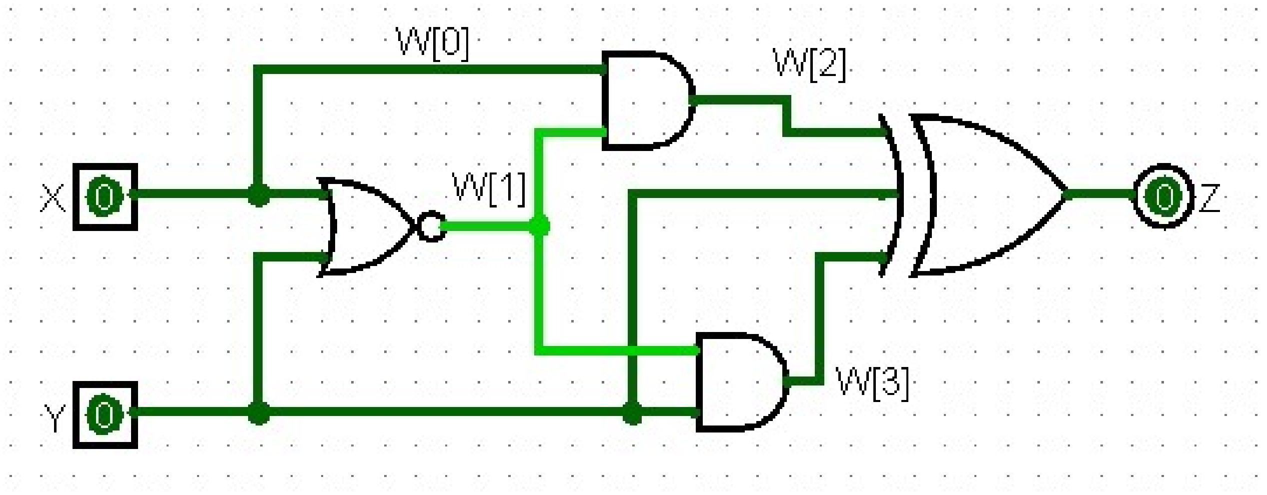
Or…

```
wire W0;
wire W1;
wire W2;
wire W3;
```

Note that ports X,Y,Z are already treated as wires by default, so there's no need to re-identify those

Verilog Basics – wire keyword

- "wire" keyword represent physical nets in the design.
- Like in real circuits, nets have values that are continuously driven on them
- **Ports** are wires by default (originated from Verilog-1995)
- Wires can be declared inside a module body.



Or…

```
wire [3:0] W;
```

Then these wires would be represented by W[0],W[1],W[2],W[3]

Verilog Basics – assign keyword

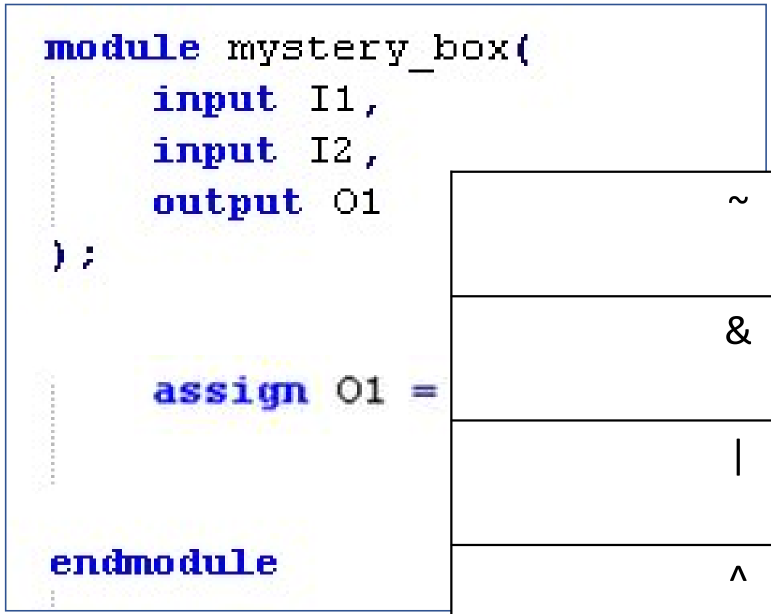
- "assign" keyword is used to specify a driver for a wire
- Continuous assignments, always active.
- The assignment expression is evaluated as soon as the right-hand-side expression changes.
- Assign can be used inside a module body, but outside a procedural block (more on this later)

```
assign <net_name> = <expression>;
```

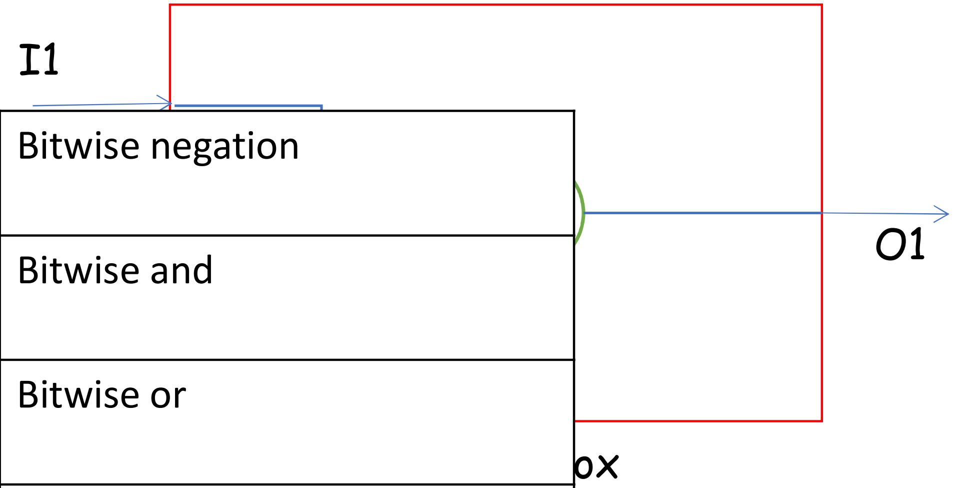
```
wire A1, A2;
wire B1, B2;

assign B1 = A1 & A2    // Assign bitwise and of A1 and A2 to B1
assign B2 = 1;        // Assign constant logic-1 to B2.
```

Verilog Basics – assign keyword

- Let's add some new stuff to our mystery box…

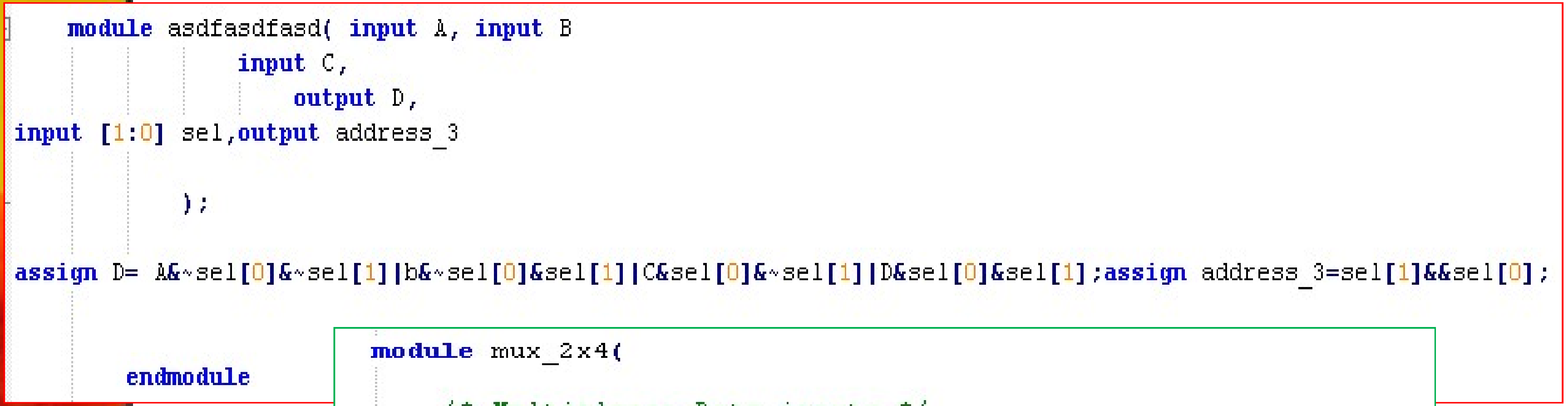


```
module mystery_box(
        input I1,
        input I2,
        output O1
);


        assign O1 =

endmodule
```

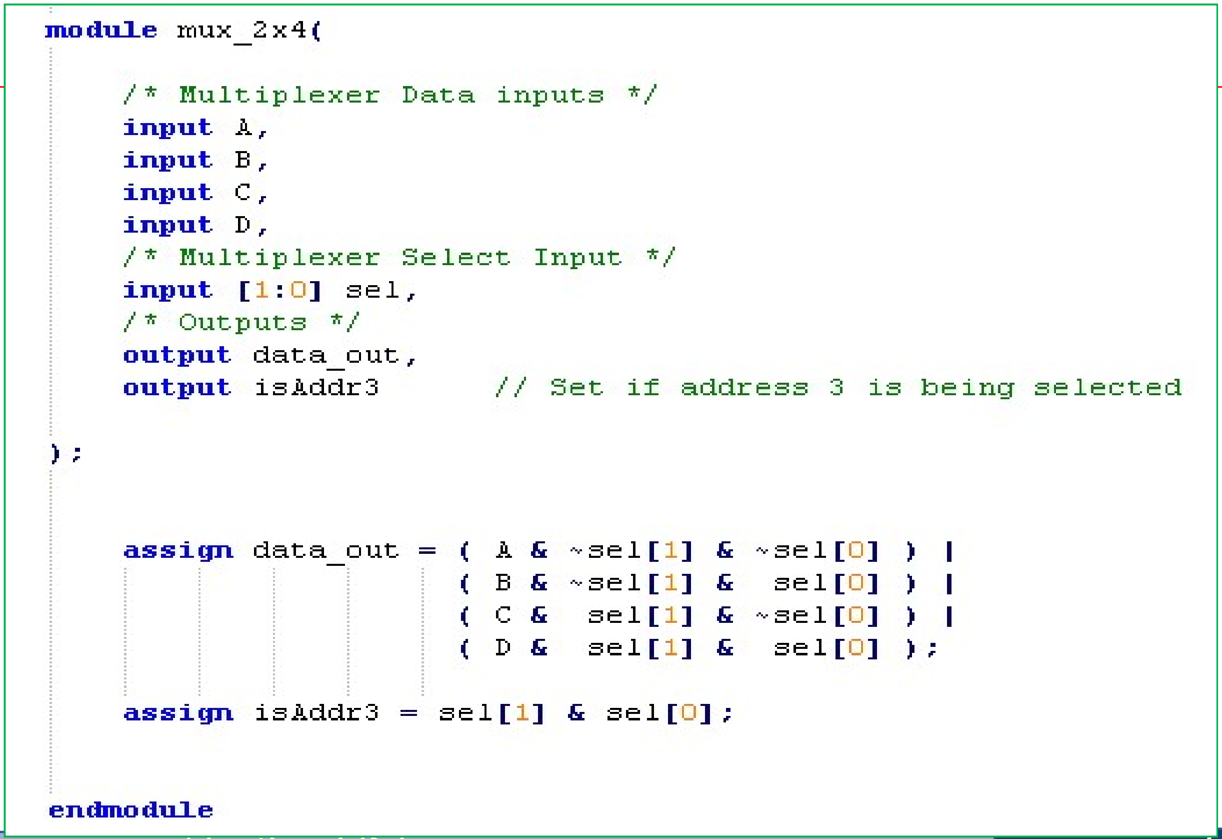


~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exlcusive or

- <u>And operator represented an and gate…</u>



```
module asdfasdfasd( input A, input B
                input C,
                    output D,
input [1:0] sel,output address_3

            );

assign D= A&~sel[0]&~sel[1]|b&~sel[0]&sel[1]|C&sel[0]&~sel[1]|D&sel[0]&sel[1];assign address_3=sel[1]&&sel[0];


        endmodule
```



```
module mux_2x4(

    /* Multiplexer Data inputs */
    input A,
    input B,
    input C,
    input D,
    /* Multiplexer Select Input */
    input [1:0] sel,
    /* Outputs */
    output data_out,
    output isAddr3      // Set if address 3 is being selected

);


    assign data_out = ( A & ~sel[1] & ~sel[0] ) |
                      ( B & ~sel[1] &  sel[0] ) |
                      ( C &  sel[1] & ~sel[0] ) |
                      ( D &  sel[1] &  sel[0] );

    assign isAddr3 = sel[1] & sel[0];


endmodule
```

Modeling Concepts

- In the scope of the course, we will see three types of circuit modeling approaches:
  - Dataflow modeling
  - Structural modeling
  - Behavioural modeling

- All three of them provide different abstractions for forming circuit descirptions, and are usually used together in bigger designs.

Dataflow Modeling

- In dataflow modeling, connections between signals are expressed using Verilog operators,
- What we see so far was a glimpse of this approach.
    - We have seen bitwise operators. Using them we described the circuits from gate level expressions.

~	Bitwise negation
&	Bitwise and
|	Bitwise or
^	Bitwise exlcusive or

More Operators - Conditional Assignment Operator

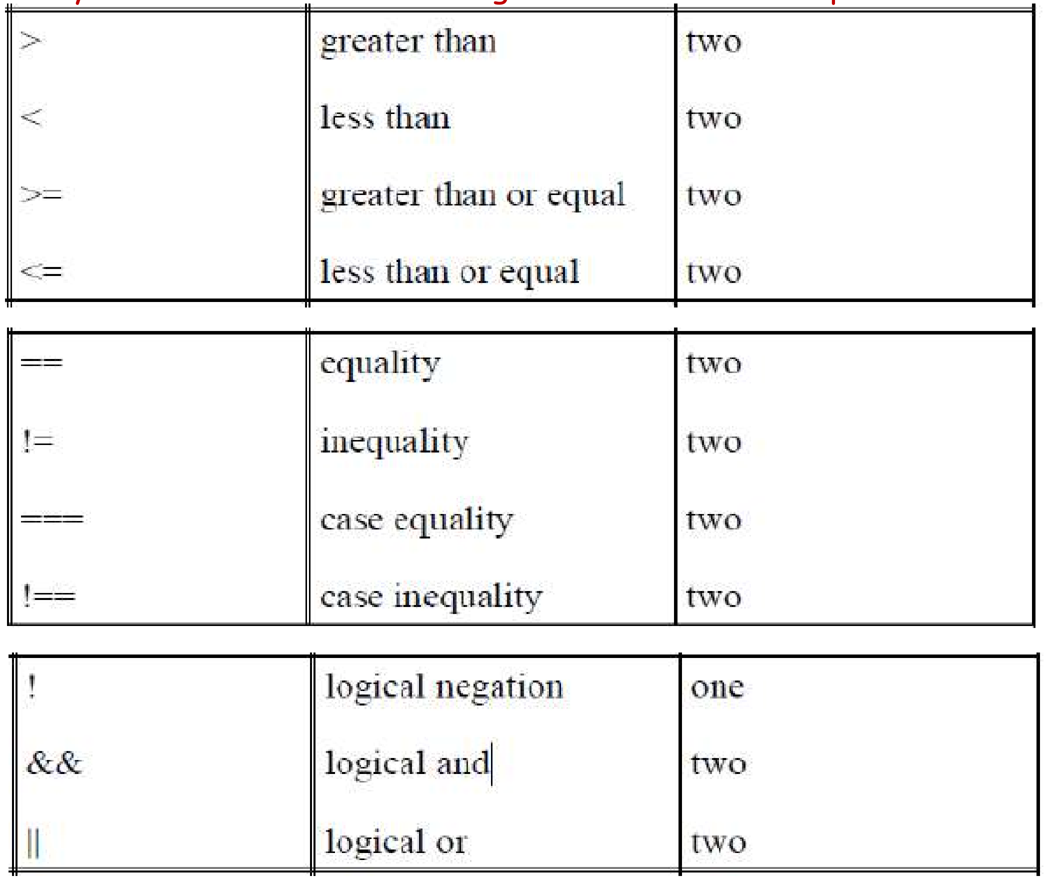
- Also known as (? :) operator

```
assign <target_signal> = (<condition>) ? (<assigned_expression_if_true>) : (<assigned_expression_if_false>);
```

```
assign data_out = (sel==1) ? (I1 ^ I2) : (~I1 & I2);
```

- In terms of hardware, this corresponds to multiplexing

More Operators – Relational & Logical



Relational

Symbol	Meaning	# of Operands
>	greater than	two
<	less than	two
>=	greater than or equal	two
<=	less than or equal	two

Equality

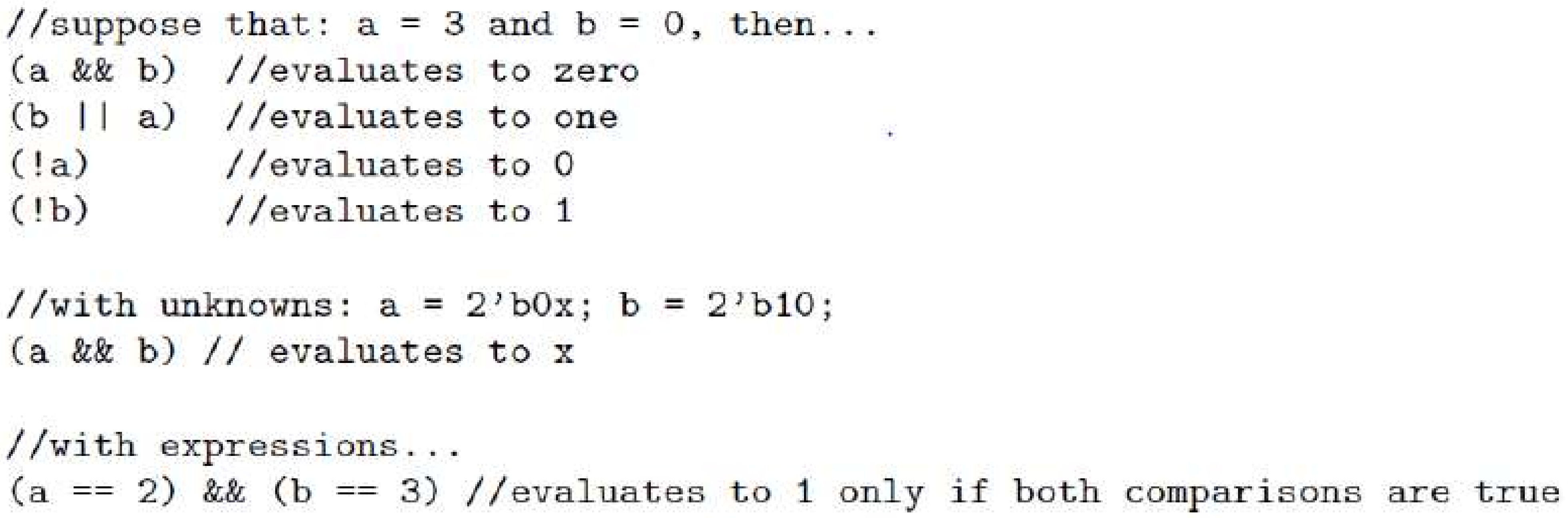
==	equality	two
!=	inequality	two
===	case equality	two
!==	case inequality	two

Logical

!	logical negation	one
&&	logical and	two
||	logical or	two

- ❑ These are usually used within conditional statements
- ❑ Results may not always be boolean, but it will be single-bit

More Operators - Examples

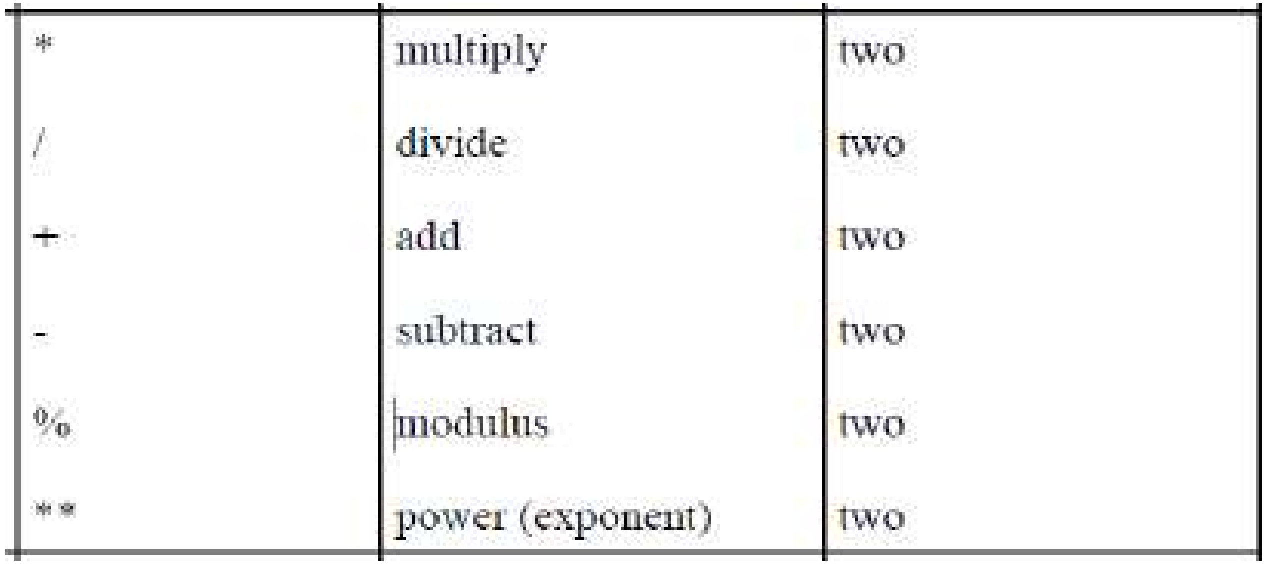


```
//suppose that: a = 3 and b = 0, then...
(a && b)  //evaluates to zero
(b || a)  //evaluates to one
(!a)      //evaluates to 0
(!b)      //evaluates to 1

//with unknowns: a = 2'b0x; b = 2'b10;
(a && b) // evaluates to x

//with expressions...
(a == 2) && (b == 3) //evaluates to 1 only if both comparisons are true
```

More Operators - Examples

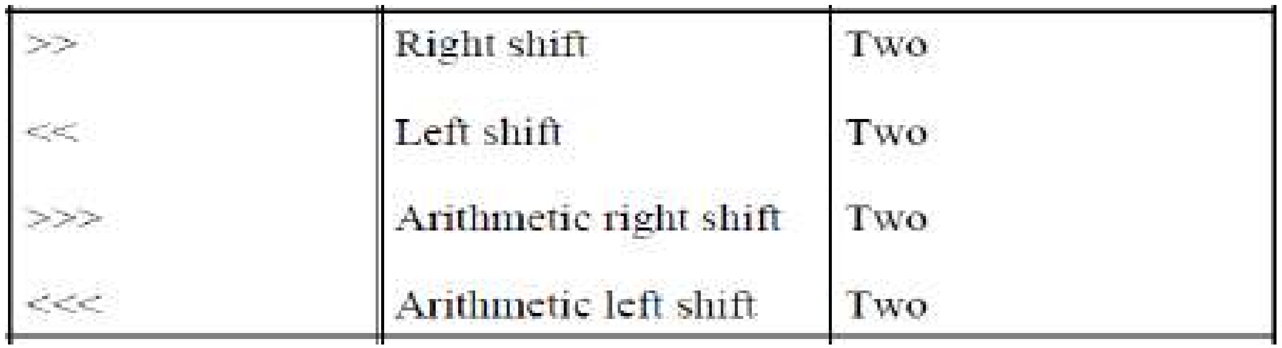
```
//let a = 4, b = 3, and...
//x = 4'b1010, y = 4'b1101,
//z = 4'b1xxz, m = 4'b1xxz, n = 4'b1xxx

a  ==  b  //evaluates to logical 0
x  !=  y  //evaluates to logical 1
x  ==  z  //evaluates to x
z ===  m  //evaluates to logical 1
z ===  n  //evaluates to logical 0
m !==  n  //evaluates to logical 1
```

More Operators – Arithmetic & Shift



Symbol	Meaning	# of Operands
*	multiply	two
/	divide	two
+	add	two
-	subtract	two
%	modulus	two
**	power (exponent)	two

Arithmetic



>>	Right shift	Two
<<	Left shift	Two
>>>	Arithmetic right shift	Two
<<<	Arithmetic left shift	Two

Shifts

- They might be synthesisable or not, depending on the tool

More Operators - Examples

```
//suppose that: a = 4'b0011;
//              b = 4'b0100;
//              d = 6; e = 4; f = 2;
//then,
a + b  //add a and b; evaluates to  4'b0111
b - a  //subtract a from b; evaluates to  4'b0001
a * b  //multiply a and b; evaluates to  4'b1100
d / e  //divide d by e, evaluates to  4'b0001. Truncates fractional part
e ** f //raises e to the power f, evaluates to 4'b1111
       //power operator is most likely not synthesible
```

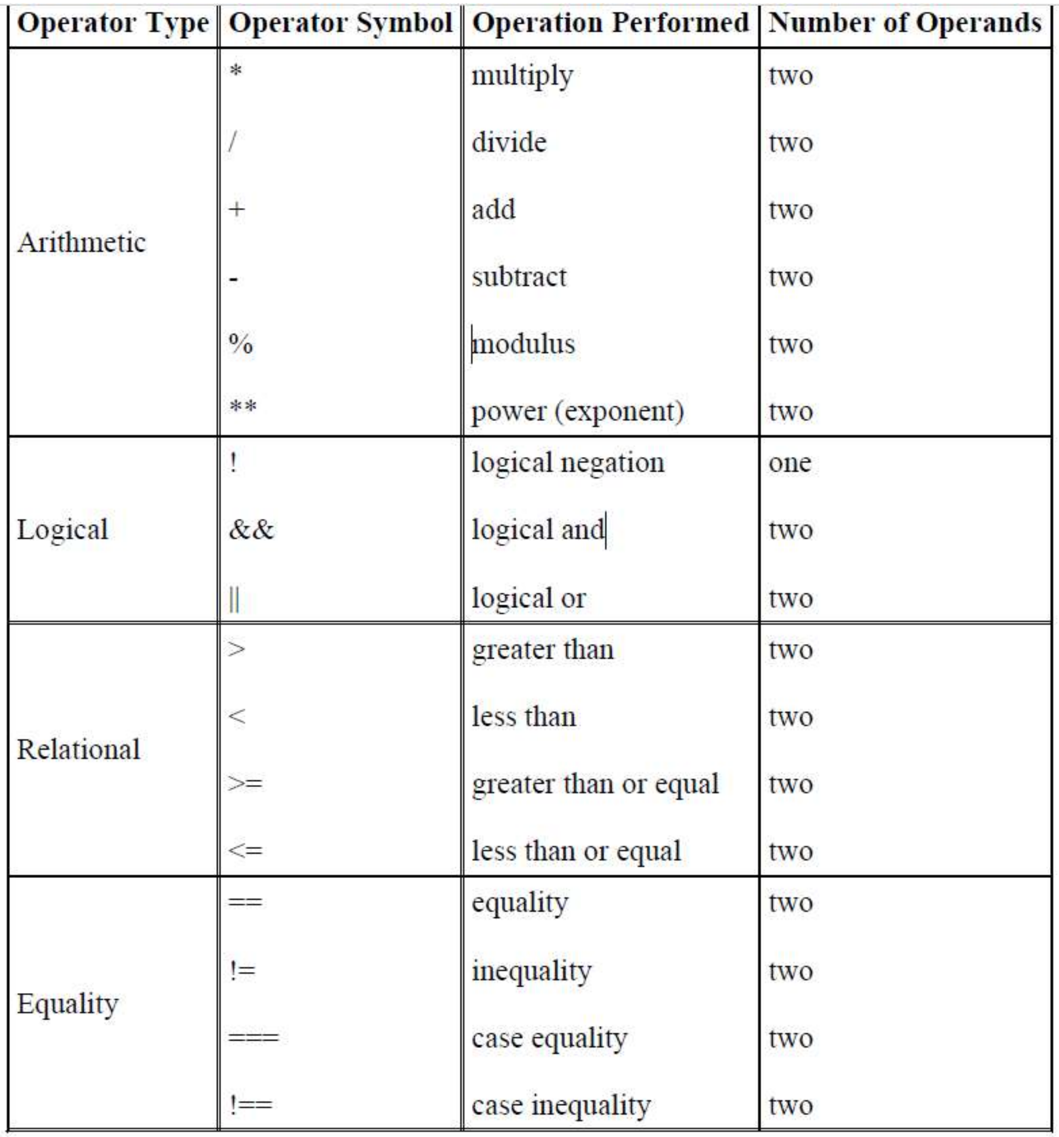
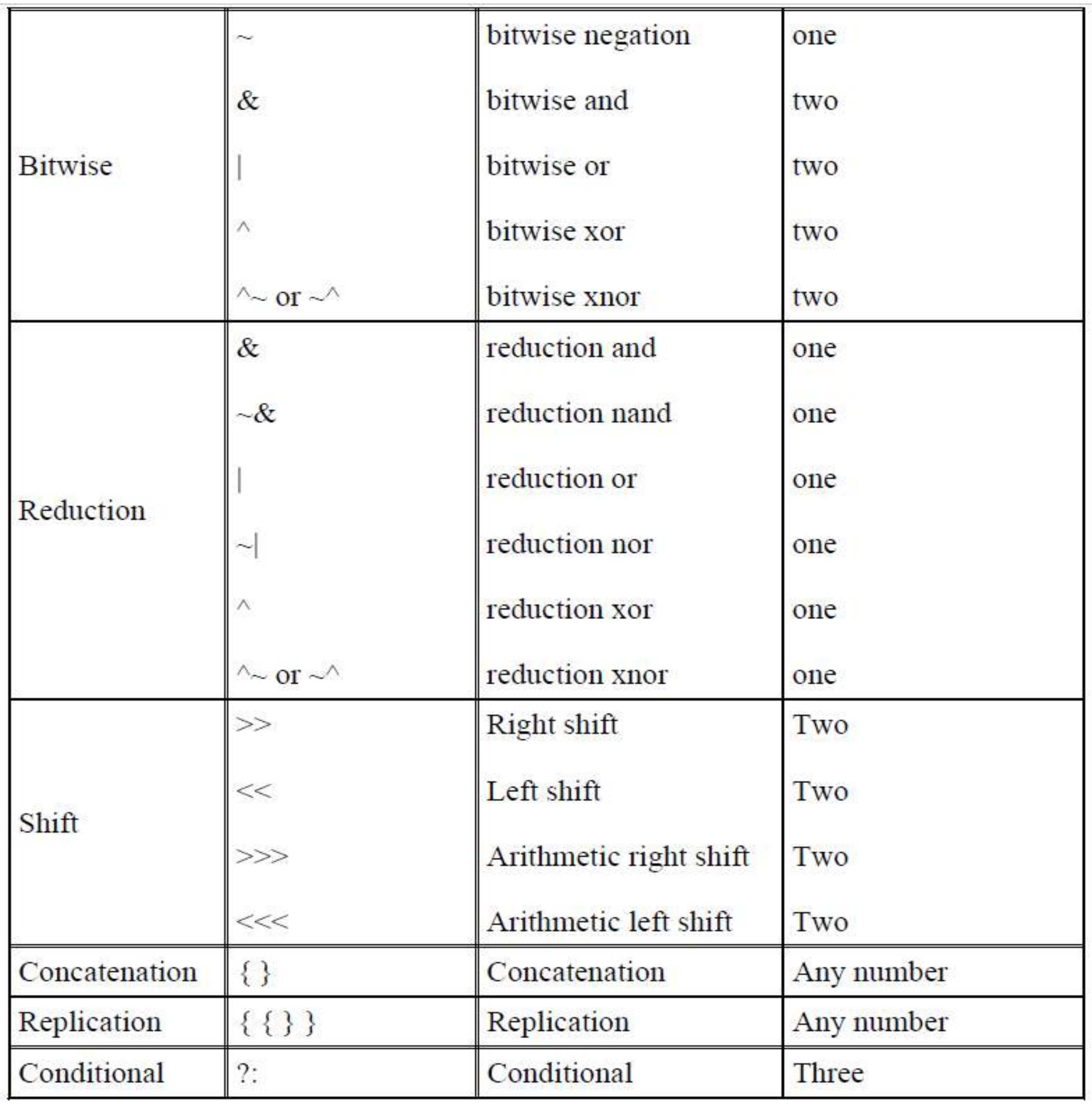
More Operators – Concatenation & Replication

```
//let a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = {b, c} // y is then 4'b0010
y = {a, b, c, d, 3'b001} // y is then 11'b10010110001
y = {a, b[0], c[1]} // y is then 3'b101
```

Concatenation

```
//let a = 1'b1, b = 2'b00, c = 2'b10, d = 3'b110
y = { 4{a} }              // y = 4'b1111
y = { 4{a}, 2{b} }        // y = 8'b11110000
y = { 4{a}, 2{b}, c } // y = 8'b1111000010
```

Replication

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
	||	logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two
Bitwise	~	bitwise negation	one
	&	bitwise and	two
	|	bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
	|	reduction or	one
	~|	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one
Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Verilog Basics – reg keyword

- "reg" keyword correspond to a variable that can hold values between assignments
- Because of this property, edge sensitive or level sensitive storage elements can be modelled by regs
- regs are declared inside a module body, but outside of a procedural block
- Does not neccessarily need to represent memory elements. Combinational logic can also be modelled with regs

Currently, we are interested in this…

Regs are more commonly used in modelling of storage elements and sequential circuits. These will be covered in upcoming lectures…

Verilog Basics – Procedural Blocks

- ❑ Assignments to regs can only be done within procedural blocks
- ❑ Procedural blocks can be used within a module body
- ❑ First two types of procedural blocks are "initial" and "always" blocks
- ❑ Procedural block encapsulation can be specified with "begin" and "end" keywords, in a similar manner with C's curly braces { } .
- ❑ If "begin" and "end" is not used, only the first upcoming statement is within that procedural block's scope.

```
module AND( output reg Q, input A ,
input B );

initial
begin
.
<initialization>
.
end

always @( ..sensitivity_list.. )
begin
.
<functionality>
.
end
endmodule
```

Verilog Basics - Procedural Blocks

- Initial block:
  - Non-synthesisable procedural blocks
  - In hardware modelling, it can be used to give initial values to regs, for simulation purposes only.
  - Initial blocks are executed just once, starting at time 0.
  - Mainly used in testbenches

```
module test(
    // ...
);

    reg r1,r2,r3;
    reg r4 = 1;  //Alternative way

    initial
    begin
        r1 = 0;
        r2 = 1;
        r3 = 0;
    end


    // ...... //


endmodule
```

Verilog Basics – Procedural Blocks

- Always block:
  - Similar to initial, but the statements inside it are continuously evaluated, in looping fashion, unless a sensitivity list is used
  - @( ) statement represent a sensitivity list for an always block
  - The block is triggered once a change occurs in at least one of the expressions specified in sensitivity list
  - Assignments done within procedural blocks with "=" operator are evaluated sequentially (blocking assignment)

```
module test(
      // ...
);

    reg r1,r2,r3;

    initial
    begin
        r1 = 0;
        r2 = 1;
        r3 = 0;
    end


    always @(r1,r2)
    begin
        r3 = ~r1 & r2;
    end


endmodule
```

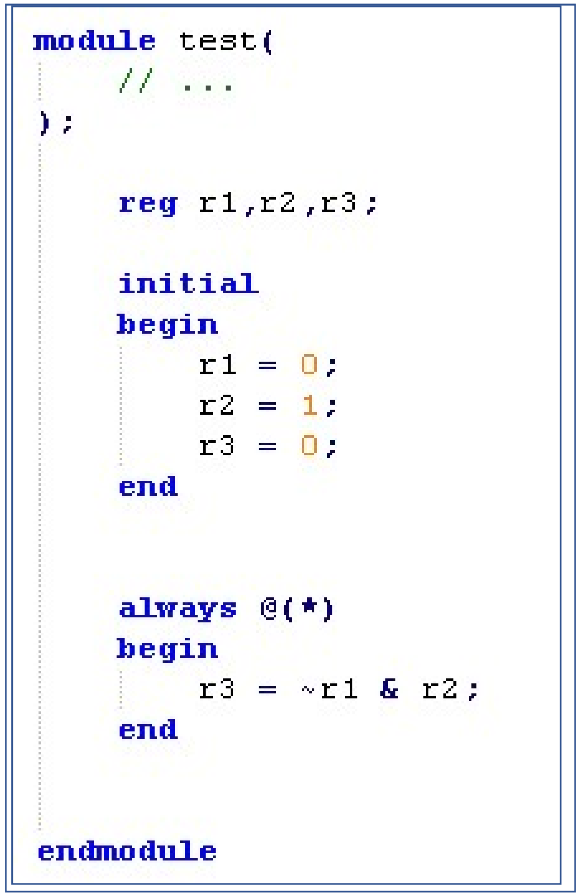
Verilog Basics – Procedural Blocks

- What about this case?

- When modelling combinational logic, sensitivity list must be properly filled. Otherwise, latches will occur.

- @(*) operator is later introduced to cope with this potential oversight. When modelling combinational logic, it auto-detects which signals could trigger the block.

Important: If there are multiple procedural blocks inside a module, they will be evaluated concurrently.



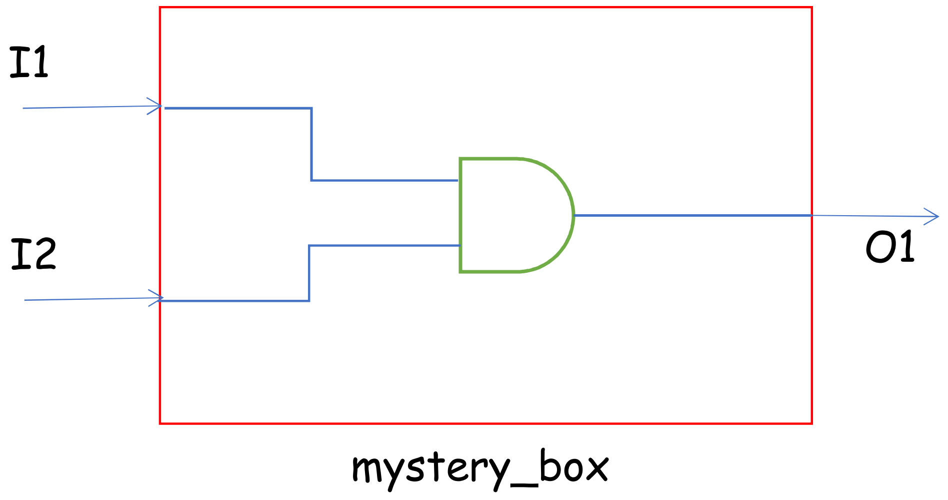
```
module test(
        // ...
);

        reg r1,r2,r3;

        initial
        begin
                r1 = 0;
                r2 = 1;
                r3 = 0;
        end


        always @(*)
        begin
                r3 = ~r1 & r2;
        end


endmodule
```

✓

Combinational circuits with procedural blocks

- Returning back to our mystery box…

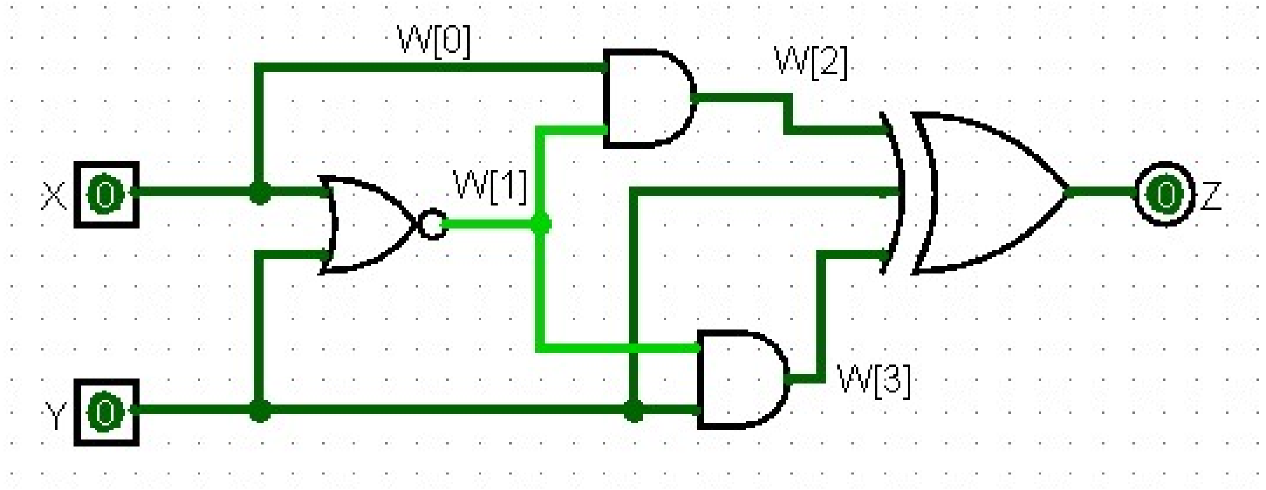
```
module mystery_box(
    input I1,
    input I2,
    output reg O1
);

    initial I1=0;
    initial I2=0;

    always @(I1,I2)
        O1 = I1 & I2;

    // Functionally same as before!

endmodule
```

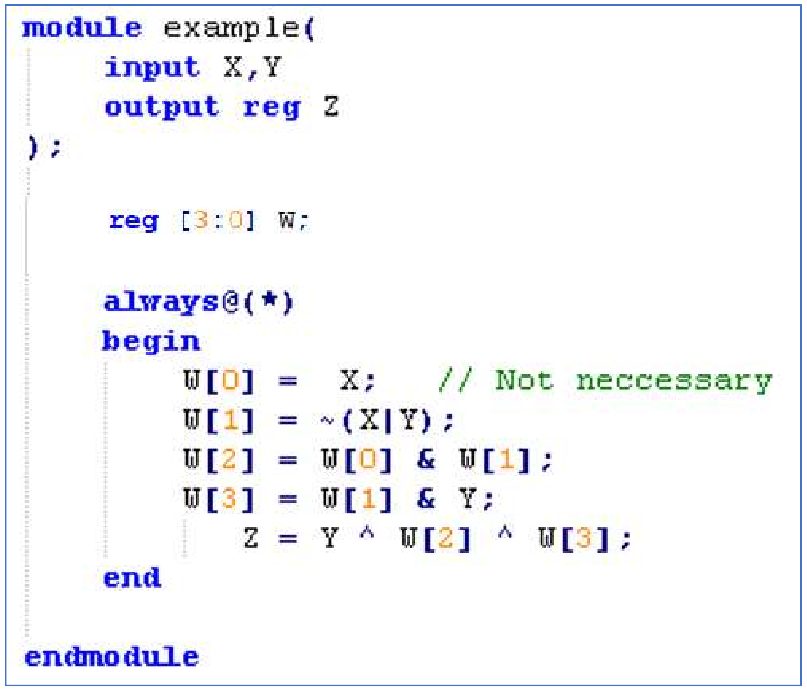


- <u>**We modeled the same circuit, but in a different way**</u>

Verilog Basics – Procedural Blocks

- How would you express the circuit below using procedural assignments only?





```
module example(
    input X,Y
    output reg Z
);

    reg [3:0] W;

    always@(*)
    begin
        W[0] =  X;   // Not neccessary
        W[1] = ~(X|Y);
        W[2] = W[0] & W[1];
        W[3] = W[1] & Y;
          Z = Y ^ W[2] ^ W[3];
    end

endmodule
```

Behavioural Modeling

- ❑ Why bother modelling combinational circuits like this?
  - ❑ What we've seen so far was dataflow modeling…
  - ❑ Using procedural blocks allows the usage of procedural statements (or behavioral statements), this approach is called behavioural modelling.

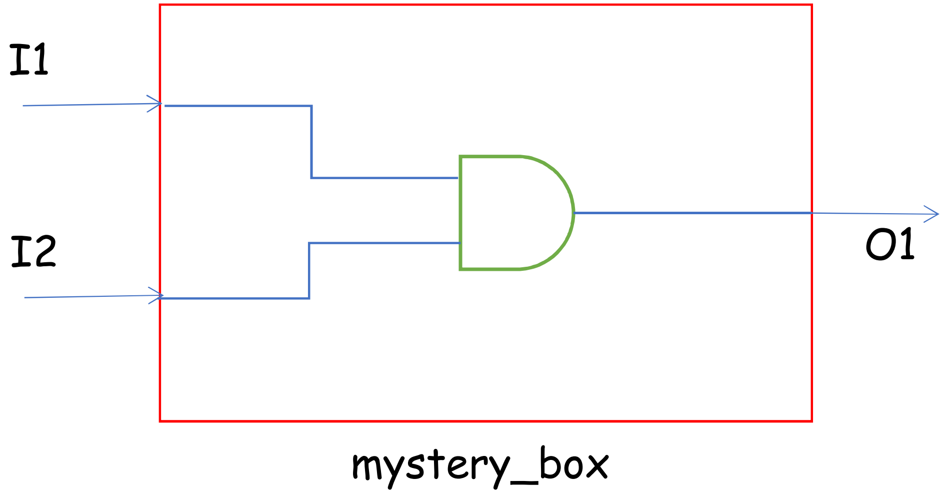
  These are;
    - If-else-else if blocks
    - Case statement
    - Loops (not all are sytnhesizable)

      (more on these in upcoming lectures…)

Combinational circuits with procedural blocks

- A small example for behavioural modelling…

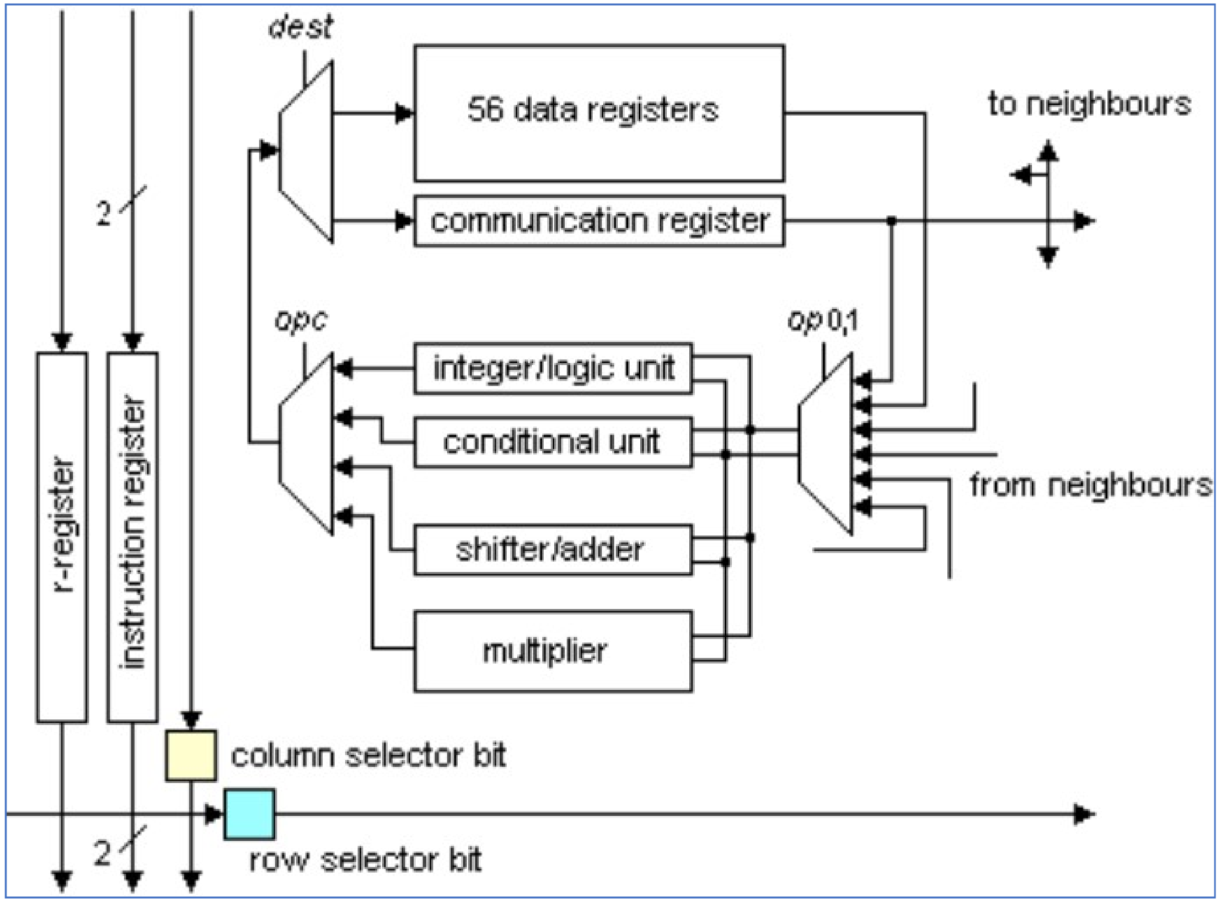
```
module mystery_box(
    input I1,
    input I2,
    output reg O1
);

    initial I1=0;
    initial I2=0;

    always @(*)
    begin
        if(I1==1 && I2==1)
            O1 = 1;
        else
            O1 = 0;
    end

    // Functionally same as before!

endmodule
```



- Again, we modeled the same circuit in a different way

Structural Modeling

- ❑Modules are instantiated as submodules inside a top level design…
- ❑Design Hierarchy
- ❑Divide & Conquer
- ❑Describe different blocks as separate modules
- ❑More readable, modular, neat looking designs
- ❑Easier to debug/track errors



Structural Modeling - Instantiation

- ❑ Instantiation means creating an instance of a module within another module
- ❑ A module can be formed from various lower-level modules connected together

- ❑ Created submodules are called instances

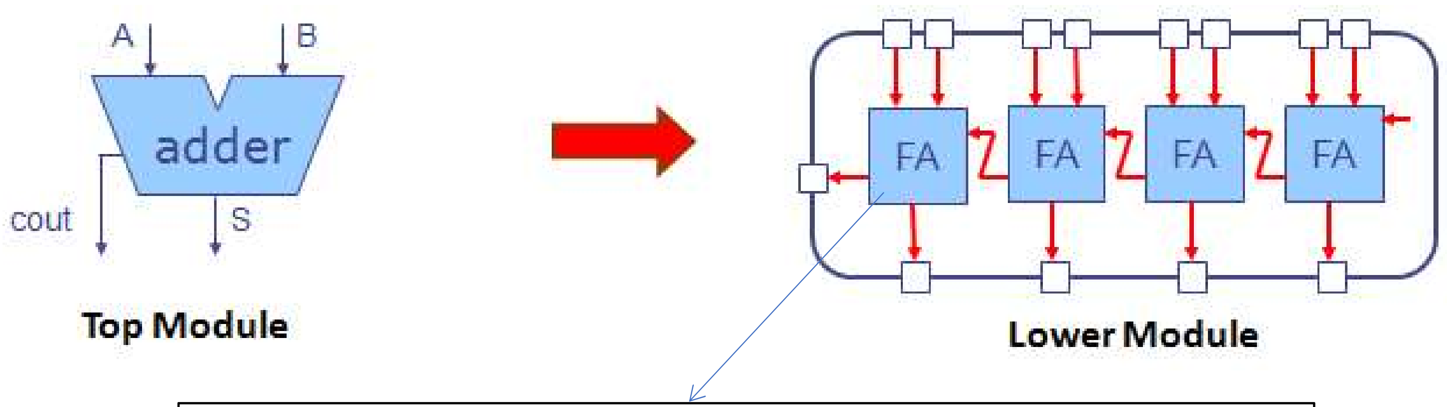
```
module <upper_module_name> (<module_terminals>)
.
.
<module_name> <instance_name> (<terminals>)
.
.
<module_internals>
.
.
endmodule
```

Instantiation

- The order of the statements (incuding instantiations) are not important.
- Remember that the code is not executed in sequence!

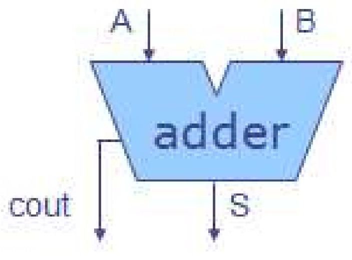
Structural Modeling - Instantiation

- Instantiation means creating an instance of a module within another module
- A module can be formed from various lower-level modules connected together

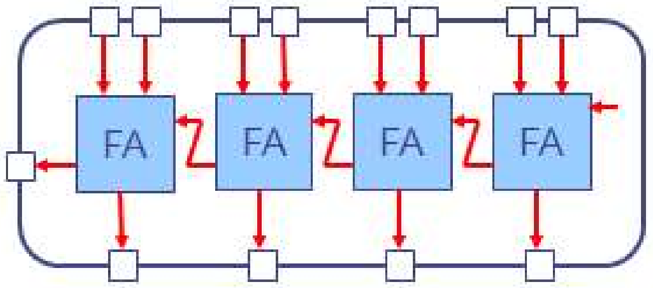


```
module FA( input a, b, c_in,
           output c_out, sum );

assign sum = a ^ b ^ c_in;
assign c_out = ( c_in & ( a ^ b ) ) | ( a & b );

endmodule;
```

Structural Modeling - Instantiation



Top Module



Lower Module

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

Vectors (data type)

```
wire c0 = 0;
wire c1, c2, c3;

FA fa0( A[0], B[0], c0, c1, S[0] );
FA fa1( A[1], B[1], c1, c2, S[1] );
FA fa2( A[2], B[2], c2, c3, S[2] );
FA fa3( A[3], B[3], c3, cout, S[3] );

endmodule
```

```
module FA( input a, b, c_in,
           output c_out, sum );

assign sum = a ^ b ^ c_in;
assign c_out = ( c_in & ( a ^ b ) ) | ( a &
b );

endmodule;
```

Instantiations

- Full adder is an another module and must be defined in the **same project**.

Structural Modeling - Instantiation

```verilog
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );
wire c0 = 0;
wire c1, c2, c3;

FA fa0( A[0], B[0], c0, c1, S[0] );
FA fa1( A[1], B[1], c1, c2, S[1] );
FA fa2( A[2], B[2], c2, c3, S[2] );
FA fa3( A[3], B[3], c3, cout, S[3] );

endmodule
```

Instantiation by ordered list

Same

```verilog
FA fa0
(
      .a(A[0]),
      .b(B[0]),
      .c_in(c0),
      .c_out(c1),
      .sum(S[0])
);
```

Instantiation by mapping

- ❑Verilog has two ways of instantiating a module
- ❑In ordered list style, as the name implies, port order matters. In mapping style, it does not.

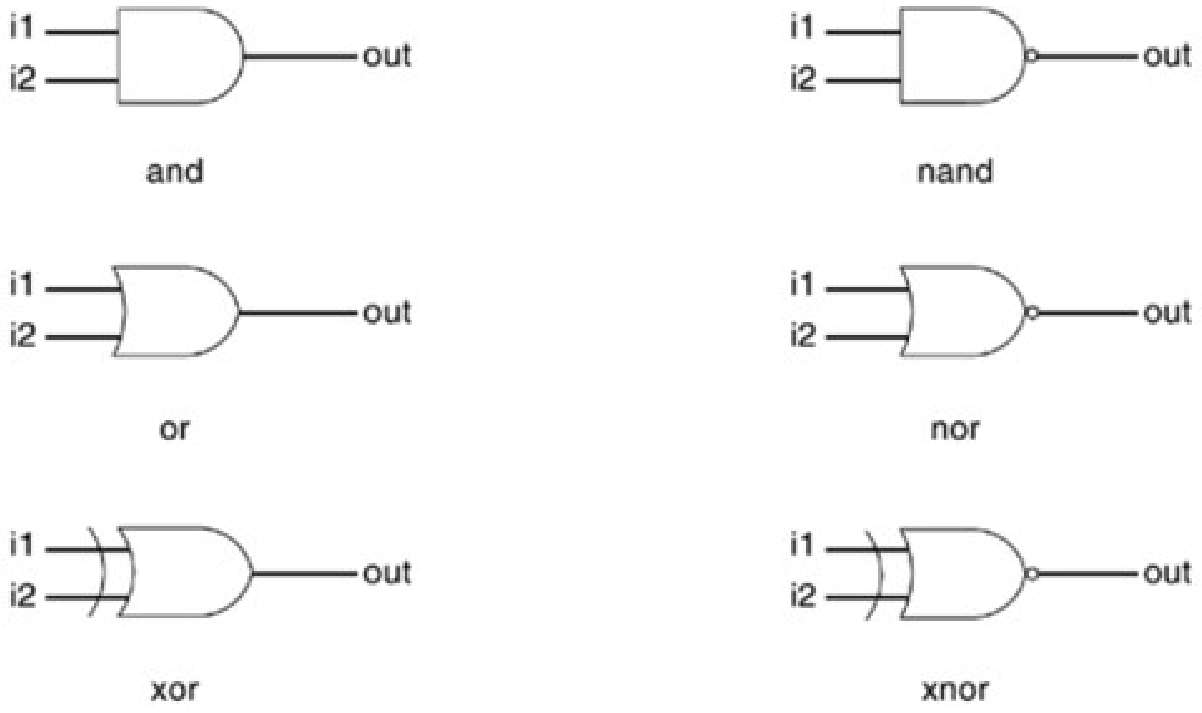
Structural Modeling - Instantiation

❑If we use mapping style instantiation, the same adder code would look like this:

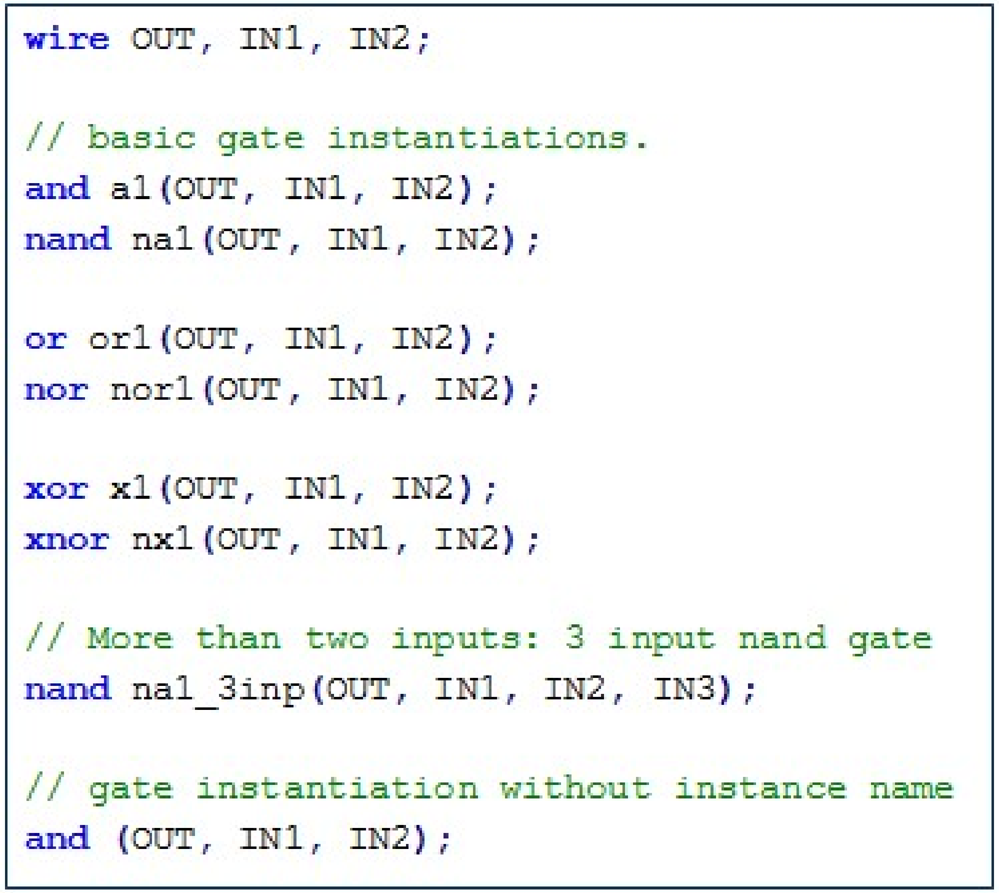
```
module adder( input [3:0] A, B,
              output cout,
              output [3:0] S );

wire c0 = 0;
wire c1, c2, c3;

FA fa0( .c_out(c1), .sum(S[0]), .a(A[0]), .b(B[0]), .c_in(c0) );
FA fa1( .c_out(c2), .sum(S[1]), .a(A[1]), .b(B[1]), .c_in(c1) );
FA fa2( .c_out(c3), .sum(S[2]), .a(A[2]), .b(B[2]), .c_in(c2) );
FA fa3( .c_out(cout), .sum(S[3]), .a(A[3]), .b(B[3]), .c_in(c3) );

endmodule
```

Structural Modeling - Primitives

- Another concept of structural modeling is the language-defined primitives
- Verilog provides logic gates as primitives



Structural Modeling - Primitives

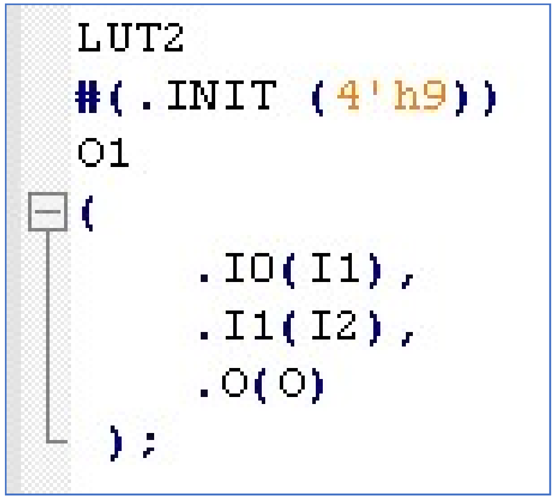


```
wire OUT, IN1, IN2;

// basic gate instantiations.
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);

or or1(OUT, IN1, IN2);
nor nor1(OUT, IN1, IN2);

xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);

// More than two inputs: 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);

// gate instantiation without instance name
and (OUT, IN1, IN2);
```

- Instance name is optional for primitives
- More than two inputs can be specified for these gates

Structural Modeling - Primitives

- Vendor specific or technology specific primitives may also be available
- For example, Xilinx library contains LUT primitives that can be instantiated directly



```
LUT2
#(.INIT (4'h9))
O1
(
    .I0(I1),
    .I1(I2),
    .O(O)
);
```

A Xilinx 2-input LUT primitive instantiation

- The part shown with #( ) is called module parameter list
(more on that later)

Structural Modeling - Primitives

- Summary for designing purely combinational circuits with Verilog

Our arsenal:

- **Dataflow modeling:**
  - Continuous assignment
  - Operators
  - Procedural blocks (initial,always) with blocking assignments
- **Structural modeling:**
  - Module instances (submodules)
  - Primitives
- **Behavioral modeling:**
  - Procedural blocks (initial,always) with blocking assignments
  - Procedural statements (if,else,case…)

Misc. Data Types

String:

- A string is a sequence of characters that are enclosed by double quotes.

```
"Hello Verilog World"    // is a string
"a / b"                  // is a string
```

Integer:

- An integer is a 32-bits signed number. Registers store values as unsigned quantities, whereas integers store values as **signed** quantities.

```
integer counter;  // general purpose variable used as a counter.
initial
    counter = -1; // A negative one is stored in the counter
```

Misc. Data Types

Real:

- They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 10^6).
- When a real value is assigned to an integer, the real number is rounded off to the nearest integer.

```
// real
real delta; // Define a real variable called delta
initial
begin
    delta = 4e10; // delta is assigned in scientific
                  // notation
    delta = 2.13; // delta is assigned a value 2.13
end

integer i; // Define an integer i
initial
    i = delta; // i gets the value 2 (rounded value
of 2.13)
end
```

Misc. Data Types

Arrays and Vectors:

- **Wires** or **reg** data types can be declared as vectors (multiple bit widths).
- If bit width is not specified, the default is a scalar (1-bit).

```
// DECLARATION:
wire a;                         // scalar net variable, default
wire [7:0] bus;                 // 8-bit bus
wire [31:0] busA,busB,busC;     // 3 buses of 32-bit width.

reg clock;                      // scalar register, default
reg [0:40] virtual_addr;        // vector register, 41 bits wide

// ACCESSING:
busA[7]                 // bit # 7 of vector busA
bus[2:0]                // Three least significant bits of vector bus,
virtual_addr[0:1]       // Two most significant bits of vector virtual_addr
```

Misc. Data Types

- Multi-dimensional **arrays** can also be declared with any number of dimensions.
- Do NOT confuse arrays with vectors!
- A vector is a single element that is n-bits wide. On the other hand, arrays are multiple elements that are 1-bit or n-bits wide.
- Ports can not be declared as arrays.

$$\underbrace{\texttt{<data\_type>}}_{\text{wire, register, integer etc.}}\ \underbrace{\texttt{<[vector]>}}_{\text{vectoral size, scalar default}}\ \texttt{<name>}\ \underbrace{\texttt{<[array]> <[array]> <[array]>}}_{\text{array size of the dimensions}}\ \texttt{...}$$

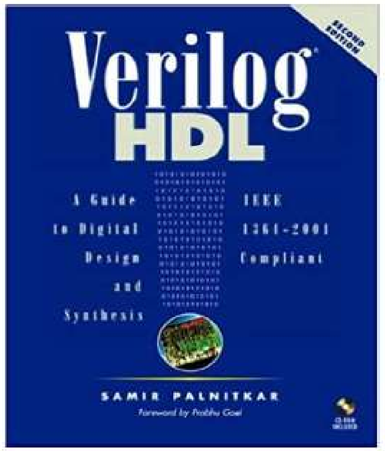
Misc. Data Types

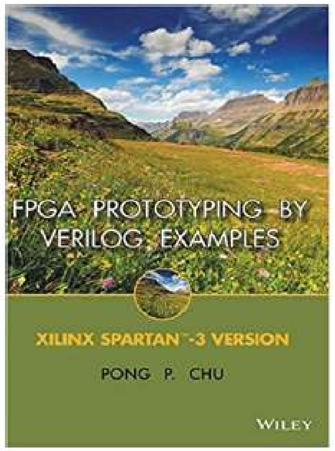
```
// DECLARATION:
integer count[0:7];       // An array of 8 count variables
reg bool[31:0];           // Array of 32 one-bit boolean register variables

reg  [4:0] port_id[0:7];        // Array of 8 port_ids; each port_id is 5 bits wide
wire [7:0] w_array2 [5:0];      // Declare an array of 8 bit vector wire

integer matrix[4:0][0:255];     // Two dimensional array of integers
wire w_array1[7:0][5:0];        // Declare an array of single bit wires

reg [63:0] array_4d [15:0][7:0][7:0][255:0];     //Four dimensional array
```

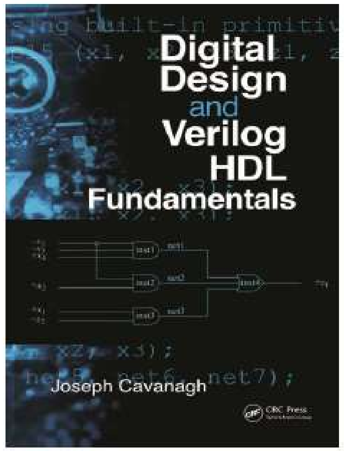
```
// ACCESSING:
count[5] = 0;             // Reset 5th element of array of count variables
chk_point[100] = 0;       // Reset 100th time check point value
port_id[3] = 0;           // Reset 3rd element (a 5-bit value) of port_id array.
matrix[1][0] = 33559;     // Set value of element indexed by [1][0] to

array_4d[0][0][0][0][15:0] = 0;     // Clear bits 15:0 of the register
                                    // Accessed by indices [0][0][0][0]
port_id = 0;          // Illegal syntax - Attempt to write the entire array
matrix [1] = 0;       // Illegal syntax - Attempt to write [1][0]..[1][255]
```

References

- *Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition, Samir Palnitkar*
- *FPGA Prototyping by Verilog Examples: Xilinx Spartan-3 Version,Pong P.Chu*
- *Digital Design and Verilog HDL Fundamentals,Joseph Cavanagh*







Thank You!