Session #2
Writing Verilog Modules

Hardware Description Types

- Verilog lets you create description of digital device using any of the levels of abstraction listed below:
  - **Structural:** description mentions lower level modules and the way they should be connected; functionality may be implied, but not explicitly included
  - **Dataflow:** equations (boolean or arithmetic) are used to describe how data is processed inside
  - **RTL(R**egister **T**ransfer **L**evel**):** the description is divided into parts representing storage in the design (**R**egisters) and combinatorial logic between storage elements (**T**ransfer function)
  - **Behavioral:** algorithm of the behavior of the device is described; internal structure is not described (but may be implied by certain algorithms)

Data Types

- In Verilog, there are four basic types of values used for hardware modeling:

Value	*Representation*
0	Logic Zero/False Condition
1	Logic One/True Condition
X	Unknown Value
Z	High Impedance

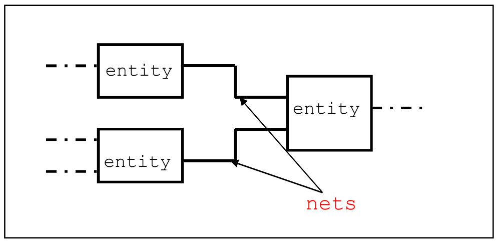
Value identifiers are case insensitive.
You can use both "X" and "x" or "Z" and "z".

Confidential

Data Types

Two classes of data types:

- ***Nets***
  - ***wire***
  - ***wand, wor,***
  - ***tri,***
  - ***triand, trior***
  - ***tri0, tri1, trireg,***



Nets are used for connecting various modules

Registers are used for data storage within entities.

There are some more implications which are discussed further.

- ***Variables or Registers***
  - ***reg***
  - ***integer***
  - ***real***
  - ***time***
  - ***realtime***

"wire" and "reg" are the most commonly used for hardware modeling, so we focus more on these types!!!

Confidential

Net Data Types

- Net data types are used to model physical connections.
- They do not store values (there is only one exception - **trireg**, which stores a previously assigned value).
- The net data types have the value of their drivers. If a net variable has no driver, then it has a high-impedance value (z).
- Nets can be declared in a net declaration statement or in a net declaration assignment.

Examples:

```
wire [7:0] a;
tri tristate_buffer;
```

Confidential

Net Data Types

- ***wire, tri***

  Both wire net and tri net are identical.

  Two different names are used for more readability. Preferably wire nets may be used if a net has only one driver. If a net has more than one driver, then tri net may be used instead.

- ***trireg***

  The trireg nets are used to model net capacity. If the trireg net driver has 0, 1, or x value, then it becomes a trireg net value and its strength can be one of the drive strengths, depending on the driver strength.

  If the driver has a z value, then the trireg net keeps the previously driven value and the strength can be one of the charge strengths, depending on the strength specified during net declaration.

wire tri	0	1	X	Z
0	0	X	X	0
1	X	1	X	1
X	X	X	X	X
Z	0	1	X	Z

Confidential

Register Data Types

- Register provide a means for modeling data storage elements. They store assigned values until a new assignment occurs.

reg Register Type:

The ***reg*** register is a 1-bit wide data type.

If more than one bit is required then range declaration should be used. Negative values assigned to reg data type variables are treated as unsigned values.

Reg data type variables can be declared as memory.

Example:

```
reg scalar;
reg [7:0] vector;
reg [7:0] mem [31:0];
```

Confidential

Register Data Types

integer Register Type:

The ***integer*** register is a 32-bit wide data type.

Integer declarations cannot contain range specification.

Integer variables can be declared as memory.

Integers store signed values.

Example:

```
integer i;
integer i_mem [7:0];
```

Confidential

Register Data Types

time Register Type:

The **time** register is a 64-bit wide data type that is used to store simulation time and to check timing dependence.

Time type registers store values as unsigned numbers.

Time declaration cannot contain range specification.

Time data type variables can be declared as memories.

Example:

```
time t;
time t_mem [3:0];
```

Confidential

Register Data Types

Examples: integer and time

```
integer Number ;
/*declaration of a integer type variable named Number*/
time Delay ;
/*declaration of a time type variable named Delay*/

initial begin
 Delay = 5 ;
 Number = 145 ;
 # Delay $display($time,” Number = %d”, Number);
 Number = - 145 ;
 # Delay $display($time,” Number = %d”, Number);
end
```

Confidential

Register Data Types

real and ***realtime*** Register Type:

The **real** register is a 64-bit wide data type that stores floating-point values.

Real registers **cannot be used with** event control, concatenations ({}), modulus operator (%), case equality (===, !==), bit-wise operators (~, &, |, ^, ^~, ~^), reduction operators (^, ~^, ^~, &, &~, |, |~) and shift operators (<<, >>).

Bit-selects and part-selects on real type variables are not allowed.

The **realtime** registers are treated in the same way as real registers.

Example:

```
real r;
realtime rt1, rt2;
```

Confidential

Register Data Types

Examples: real and realtime

```
module test;
real Number_1, Number _2;
realtime Delay, RealTime;

initial begin
   Delay = 1.4;
   RealTime = -1.4;
   #Delay Number_1 = 14.5;
   # Delay $display($time,"  Number_1 = %e", Number_1);
   Number_2 = -12e6;
   # Delay $display($time,"  Number_2 = %e", Number_2);
   # Delay $display($time,"  RealTime = %e", RealTime);
End

endmodule
```

Confidential

Register Data Types

Some points to remember about "reg"

- These correspond to variables in the C language.
- Register data types always retain their value until another value is placed on them.
- DO NOT confuse with hardware registers built with flip-flops.
- A "reg" type variable is the one that can hold a value.
- Unlike nets, registers do not need any drivers.
- In synthesis, the compiler will generate latches or flip-flops for them. However, if it can be sure their output does not need to be stored it will synthesize them into wires.
- It can be sure they do not have to store if their outputs is based only on their present inputs.

Confidential

Vectors

- Vectors are multiple bit widths ***net*** or ***reg*** data type variables that can be declared by specifying their range.

<reg, wire> [msb #: lsb #] <identifier>

Where: **msb** - most significant bit
lsb - least significant bit

```
// Sample Vectors
reg    [0:7]   Control ;
wire   [-4:3]  Bus ;
reg    [15:0]  Data;
wire   [7:0]   Data_1 ;
```

Vectors can be declared for all types of **net** data types and for **reg** data types.

Specifying vectors for **integer**, **real**, **realtime**, and **time** data types is illegal. They can be declared as arrays.

Vector nets and registers are treated as unsigned values

In Verilog MSB is *always on the left* !

Confidential

Bit Select and Part Select

There are two special operations on vectors available in Verilog:

- **Part select** means selection of a continuous group of bits in a vector.
- **Bit select** means selection of one bit in a vector.

```
// Vectors:
reg    [15:0]  Data;
wire   [7:0]   Data_1 ;

// Part select:
Data [15:12] = Data_1[3:0]; // transfer 4 bits

// Bit select:
Data [12] = Data_1[2] ;      // transfer 1 bit
```

Confidential

Value assignment to vectors

There are a few rules of assigning values to vectors:

If a value assigned to a vector has more bits than the vector then most significant bits are **truncated**.

```
reg [3:0] data;
assign data = x1z10;

// data gets the value 1z10
```

If a value assigned to vector has less bits than the vector then:

if most significant bit of the value is logic “0” or “1” then empty bits in the vector are filled by logic “0”

if most significant bit of the value is “z” then empty bits in the vector are filled by “z”

if most significant bit of the value is “x” then empty bits in the vector are filled by “x”

```
reg [3:0] data;
assign data = 10;
assign data = x10;
// data get the value xx10
assign data = z01;
// data gets the value zz01
```

Confidential

Memory

Verilog memory is an array of vectors.

– Declaration of memory:

cell_type identifier **[left_index : right_index]**
<cell declaration> **<address VALUE range>**
(not address bus range)

– Valid cell types are: ***reg, integer, time***
– Range can be specified only for ***reg*** cell type
– There's no global memory access – only access to memory word:

memory_name[index]

– Bit-select and part-select on memory word is illegal

```
reg [7:0] mem [31:0]; // 32x8 memory declaration
                      // address bus range is [4:0]
```

Confidential

Identifiers

- The identifier is a unique name, which identifies an object, used as an object reference.
- An identifier can contain a sequence of letters, digits, underscores (_) and dollar signs ($).
- The first character of an identifier can only be a letter or an underscore.
- Identifiers are case sensitive.
- Escaped identifiers start with backslash character (\) and end with white space.

```
reg enable;
wire _ready;
integer group_a;
reg and5;
tri clk$1;
```

```
wire \+^_^+*+*<-> ;
reg \!clk ;

reg \rst;
```

Escaped Identifiers, which one is not valid and why?

reg `\rst;` is wrong because it is not followed by a white space charecter and “;” is considered as a part of the identifier. Correct usage is: **reg** `\rst; ;`

Confidential

Number Representation

There are two types of numbers:

sized numbers:	**<size> ' <base> <value>**
unsized number:	**' <base> <value>**

size	is width of numbers in bits and is always decimal
base	is the base format for number
value	is an actual value of number

Base format specifiers:

- **'d or 'D - Decimal**
- **'b or 'B - Binary**
- **'h or 'H - Hexadecimal**
- **'o or 'O - Octal**

Examples:

```
12'h123 Hex 123 using 12 Bits
4'b1010 Binary 1010 using 4 Bits
20'd44 Decimal 44 using 20 Bits
6'o77 Octal 77 using 6 bits
```

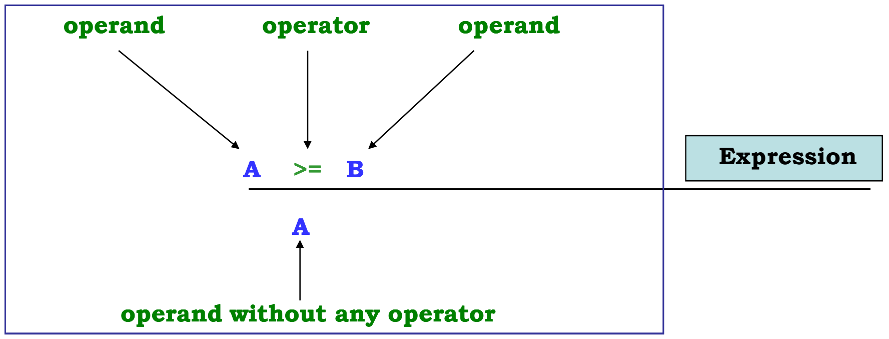
Numbers **without size and base** specified are treated as **signed, decimal integers.**
Numbers **with base specified** are treated as **unsigned integers.**

Number Representation Rules

- Base letters, hex digits, X and Z are not case sensitive in numbers.
- The characters Z and ? are equivalent in numbers.
- Numbers may not contain embedded spaces. However spaces are allowed either side of the base.
- Negative numbers are represented in two's complement.
- An underscore is not allowed as the first character of a number. Otherwise, underscores may be included for readabilityare ignored.
- The size indicates the exact number of bits.
- The size of an unsized number may default to 32 or more bits, depending on the implementation.
- If the size is greater than the number of bits specified, the number is padded on the left with 0s, unless the leftmost bit is X or Z, in which case the X or Z is used to pad to the left.
- If the size is less than the number of bits specified, the number is truncated from the left.

Expressions

- An expression is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operator.



Confidential

Operands

An operand can be one of the following:

- Constant number
- Net
- Register variables (reg, integer, time, real, realtime)
- Bit-select of a net
- Bit-select of reg, integer and time variables
- Part-select of a net
- Part-select of reg, integer and time variables
- Memory word

Confidential

Operands

Examples:

```
wire B; wire [3:0] X, Y;  reg C;  reg [3:0] Z;
integer L;  real A;  reg [1:0] M[0:255];

// Real and constant number.
    A > 1.02        // 1.02 is constant number, A is real
// Net and register.
    C == B          // C is a reg, B is a net
// Net bit-select.
    X[0] !== Y[0]   // A and B are nets bit-select
// Bit-select of type reg and integer.
    Z[0] && L[0]    // Z is bit-select of reg, L bit-select of
                    // integer
// Net part-select.
    X[3:1] > Y[2:0] // A and B are nets part-select
// Part-select of type reg and integer.
    Z[1:0] + L[1:0] // Z is part-select of reg, L
                    // part-select of  integer
// Memory word.
    ~& M[0]         // M is memory word
```

Confidential

Logical Operators

Logical operators ***return single bit*** value:

- 0 (false)
- 1 (true)
- X (unknown).

Operands equal 0 are treated as *false*, operands with numeric value other than 0 are treated as *true*.
Operands without numeric value are treated as *unknown*.

Symbol	Operation	Operands
&&	"Logical" and	2
||	"Logical" or	2
!	negation	1

Confidential

Bit Wise Operators

Bitwise operators are used to perform standard logical operations on signals and corresponding bits within vectors.

Symbol	Operation	Operands
&	"bitwise" and	2
|	"bitwise" or	2
~	negation	1
^ , (~^ / ^~)	"bitwise" xor , xnor	2

Confidential

Reduction Operators

Reduction operators start from the right, a perform recursive logical operations, bit by bit.

Symbol	Operation	Operators
& , ~&	reduction and , nand	1
| , ~|	reduction or , nor	1
^ , (~^ / ^~)	reduction xor , xnor	1

If zeros and ones are to be checked a reduction operator can be used! They produce single bit values !

Confidential

Examples

```
// Values of the operands:
A = 4'b1101 ;     C = 4'b01x1 ;
B = 4'b1100 ;     D = 4'b0000 ;

A && D     //result is 1'b0 (false)
A || D     //result is 1'b1 (true)
!C         //result is 1'bx (unknown)

~A         //result is 4'b0010
A & B      //result is 4'b1100
A | B      //result is 4'b1101
A ^ C      //result is 4'b10x0

&A         // result is 1'b0
|A         // result is 1'b1
^B         // result is 1'b0
^~C        // result is 1'bx
```

Logical Operators

Bitwise Operators

Reduction Operators

Confidential

Equality Operators

Equality operators strictly compare two operands. There are two forms of equality operators in Verilog.

Symbol	Operation	Operators
==	Logical equality	2
!=	Logical inequality	2
===	Case equality	2
!==	Case inequality	2

The logical equality operators will yield an **x** if either operand has **x** or **z** in its bits.
The case equality operators compare both operands bit by bit and compare all bits, including **x** and **z**. Case equality operators never result in **x**.
Return value is a single bit in both cases!

Confidential

Relational Operators

Usually relational operators control conditional or multi-branching operations.

Symbol	Operation	Operands
>	greater than	2
<	less than	2
>=	greater or equal	2
<=	less or equal	2

The return value is always a single bit. The expressions return a logical value of 1 if expression is true and 0 if the expression is false. If the operands have any x or z bits the expression takes value x.

Confidential

Arithmetic operators

These are used to perform arithmetic operations. Typical arithmetic operations are mentioned below:

Symbol	Operation	Operands
+	addition	2
-	subtraction	2
*	multiplication	2
/	division	2
%	modulus	2
-	unary minus	1

Confidential

Operators

```
A = 2'b00;    C = 2'b1x;    D = 2'b1x ;    F = 3'b111;
B = 2'b01;    E = 3'b010;

A == B       // result is 1'b0 (false)
A != B       // result is 1'b1 (true)
C == D       // result is 1'bx (unknown)
C === D      // result is 1'b1 (true)

A > B        // result is 1'b0 (false)
A < B        // result is 1'b1 (true)
B < E        // result is 1'b1 (true); B is zero-extended
             // on the Msb side to match F size

B + B        // result is 2'b10 (2)
F - E        // result is 3'b101 (5)
E * 2        // result is 3'b100 (4)
F / 3        // result is 3'b010 (2)
F % 3        // result is 3'b001 (1)
  - E        // result is 3'b110 (2's complement of 2)
```

Equality

Relational

Arithmetic

Confidential

Concatenation Operator

Concatenation operators allow you to group signals, buses or single bits. The replication operator allows you to repeat concatenations.

Symbol	Operation	Operands
{ , }	concatenation	2 or more
{ n{ , }}	replication	2 or more

Confidential

Shift Operators

Shift operators are used to shift elements of vectors.

Info: Insert images from wiki

Symbol	Operation	Operators
>>	shift right	2
<<	shift left	2

The vacated portion will be filled with zeros.

Verilog-2001 has introduced Arithmetic Shift operators >>> and <<<, in which during a right shift the left most bit (sign bit) is preserved by replicating in the vacated portion.

Confidential

Operators

Examples:

```
A = 2'b01 ;          C = 4'b0010 ;
B = 3'b0x1 ;         D = 8'b1111_1111 ;


{A, B}       // result is 5'b010x1
{B, A, A}    // result is 7'b0x10101
{ 2{A, B}}   // result is 10'b010x1010x1
{ 2{A, B}, 2{B, A}} // result is 20'b010x1010x10x1010x101


C << 2       // result is 4'b1000
C >> 1       // result is 4'b0001
D << 8       // result is 8'b0000_0000
D >> 8       // result is 8'b0000_0000
```

Concatenation Operator

Shift Operators

Confidential

Conditional Operator

Conditional operator is short hand version of case or if..then..else type statements

Symbol	Operation	Operators
? :	conditional	3

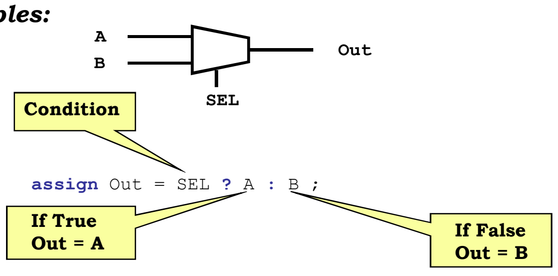
In Verilog, "TRUE" refers to a non-zero value and "FALSE" refers to a zero value.

Conditional operators provide a concise tri-state output description.

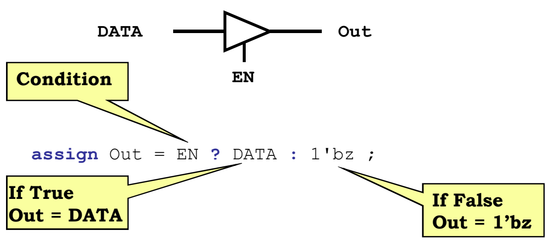
Confidential

Operators

Examples:

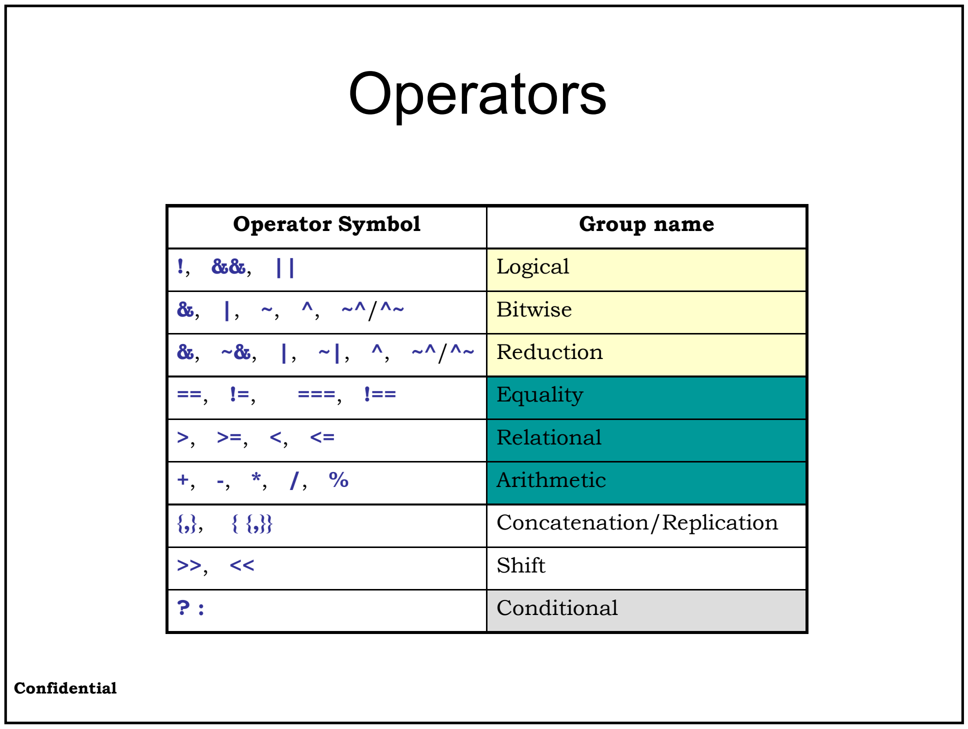


```
assign Out = SEL ? A : B ;
```



```
assign Out = EN ? DATA : 1'bz ;
```

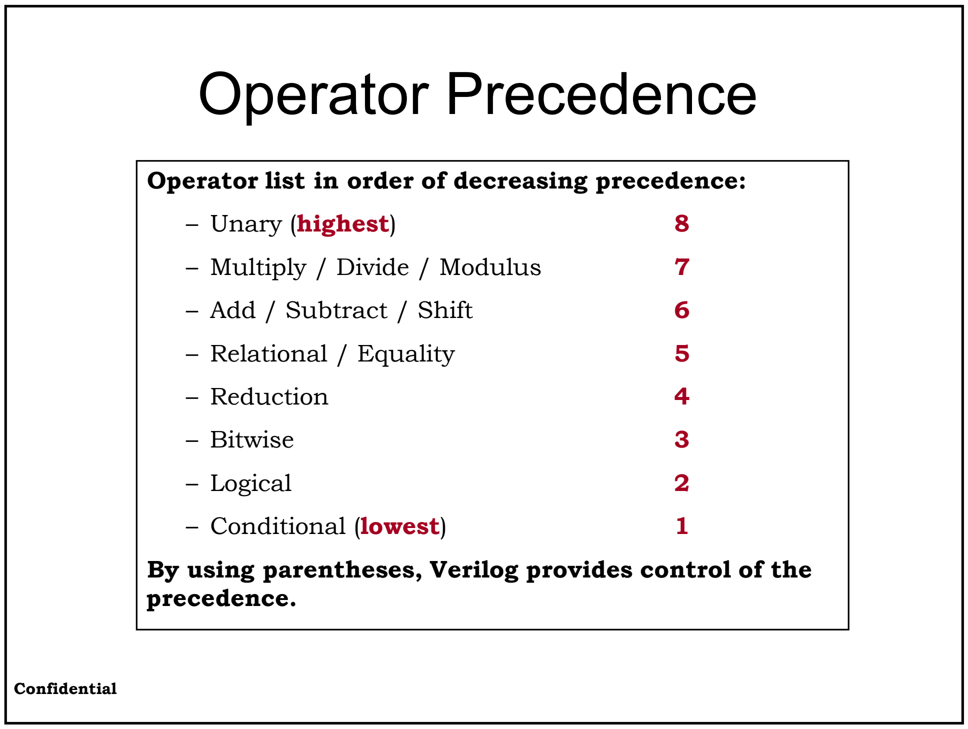
Confidential



Operators

Operator Symbol	Group name
!, &&, ||	Logical
&, |, ~, ^, ~^/^~	Bitwise
&, ~&, |, ~|, ^, ~^/^~	Reduction
==, !=, ===, !==	Equality
>, >=, <, <=	Relational
+, -, *, /, %	Arithmetic
{,}, { {,}}	Concatenation/Replication
>>, <<	Shift
? :	Conditional

Confidential



Operator Precedence

Operator list in order of decreasing precedence:

– Unary (**highest**)	**8**
– Multiply / Divide / Modulus	**7**
– Add / Subtract / Shift	**6**
– Relational / Equality	**5**
– Reduction	**4**
– Bitwise	**3**
– Logical	**2**
– Conditional (**lowest**)	**1**

By using parentheses, Verilog provides control of the precedence.

Confidential

Continuous Assignment

```
assign target = expression ;
```

This should be of a "net" data type only.

This can be of "reg" or "net" type.

"assign" is the keyword, followed by an equation.

Used for data flow modeling. Allows the use of boolean expressions.

Suitable for modeling combinational logic.

Used for updating nets.

Multiple "assign" statements are monitored in parallel.

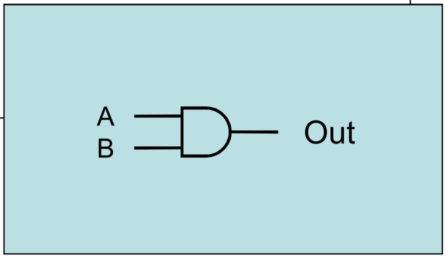
Target and expressions can be bit select, part select or the whole element of relevent data types for LHS and RHS.

Confidential

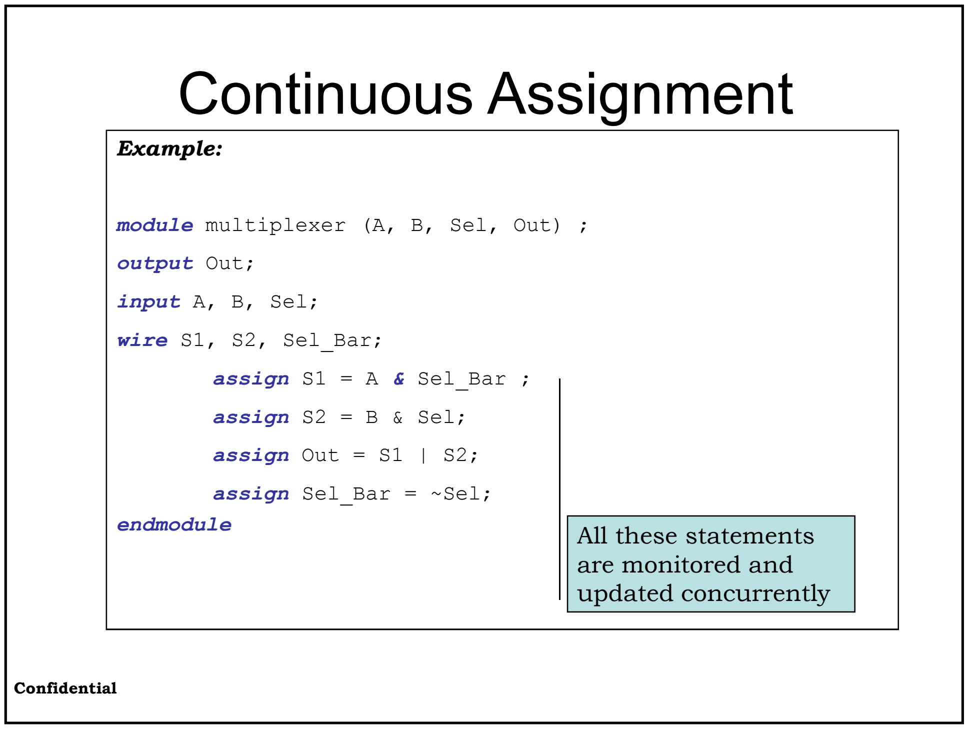
Continuous Assignment

Example:

```
module and_gate (Out, A, B) ;
output Out;         // Output port is wire by default.
input A, B;
        assign Out = A & B ;
endmodule
```



Confidential



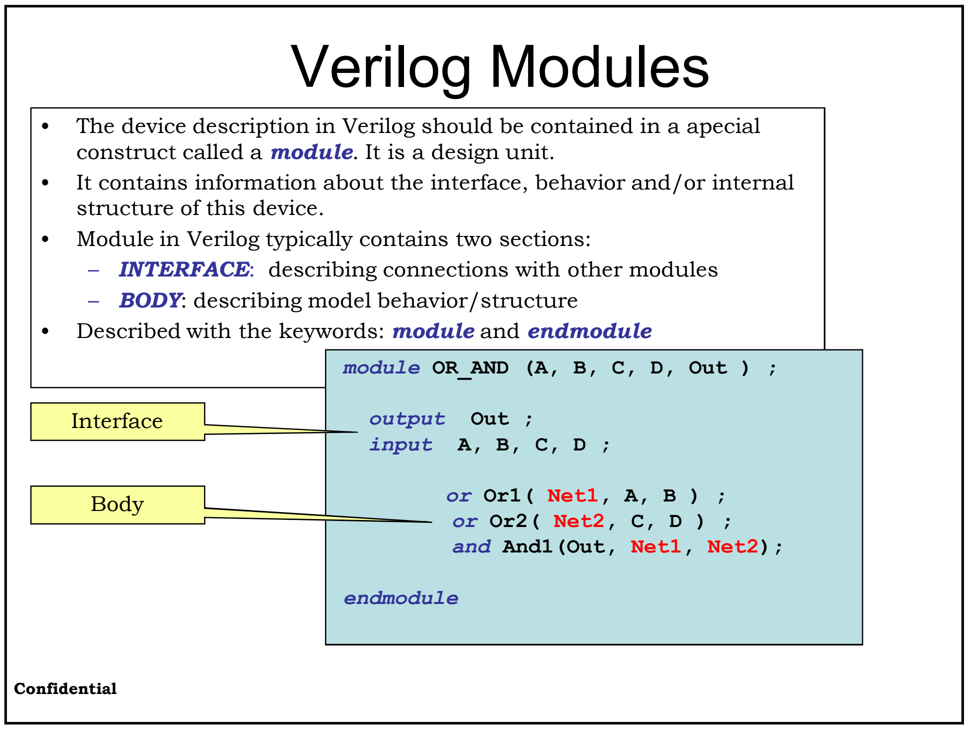
Continuous Assignment

Example:

```
module multiplexer (A, B, Sel, Out) ;
output Out;
input A, B, Sel;
wire S1, S2, Sel_Bar;
        assign S1 = A & Sel_Bar ;
        assign S2 = B & Sel;
        assign Out = S1 | S2;
        assign Sel_Bar = ~Sel;
endmodule
```

All these statements are monitored and updated concurrently

Confidential



Verilog Modules

- The device description in Verilog should be contained in a apecial construct called a ***module***. It is a design unit.
- It contains information about the interface, behavior and/or internal structure of this device.
- Module in Verilog typically contains two sections:
  - ***INTERFACE***: describing connections with other modules
  - ***BODY***: describing model behavior/structure
- Described with the keywords: ***module*** and ***endmodule***

Interface

Body

```
module OR_AND (A, B, C, D, Out ) ;

   output  Out ;
   input  A, B, C, D ;

         or Or1( Net1, A, B ) ;
         or Or2( Net2, C, D ) ;
         and And1(Out, Net1, Net2);

endmodule
```

Confidential

Verilog Modules

- A module is the basic building block in Verilog.
- Elements are grouped into modules to provide the common functionality that is used at many places in the design.
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs).
- Modules CANNOT be nested.Rather, one module can instantiate another module.
- Each instance of module has all the properties of that module.

Confidential

Ports in a digital circuit

- Ports are used to communicate with the external world.
- There are **three types** of **ports** in Verilog:
  - input
  - output
  - inout
- Input ports are those from which the device accepts the data and they are declared with the keyword ***input.***
- Output ports are those through which the device sends out the data or drives other devices and they are declared with the key word ***output***.
- Inout port are those ports which can either accept data or send data out based on the operating condition and they are declared with the key word ***inout***.

All ports should be listed and declared at the start of the module

Confidential

Putting all together

Start

Module name

```
module dff (DATA, CLK, ENABLE, Q);
```

List of ports

```
input DATA, CLK, ENABLE;
output Q;
```

Declaring the port directions

Storage variable

```
reg Q;
```

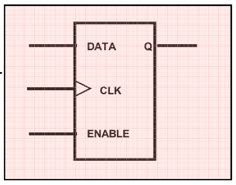
```
always @(posedge CLK)
  if (ENABLE)
    Q = DATA;
```

Functionality, describes how inputs are converted to outputs

```
endmodule
```

End

Described Circuit



A D-Flip Flop

HDL description should exactly mirror the digital circuit in concern.

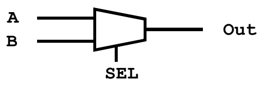
Confidential

Data Flow Modeling

Consider this example:

```
module multiplexer (A, B, Sel, Out) ;
output Out;
input A, B, Sel;
wire S1, S2, Sel_Bar;
       assign S1 = A & Sel_Bar ;
       assign S2 = B & Sel;
       assign Out = S1 | S2;
       assign Sel_Bar = ~Sel;
endmodule
```

This method of expressing the digital logic circuits in terms of equations is called data flow modeling.



Other Options:

```
1. assign Out = (A & Sel) | (B & ~Sel) ;
2. assign Out = Sel ? A : B ;
```

Confidential

Data Flow Modeling

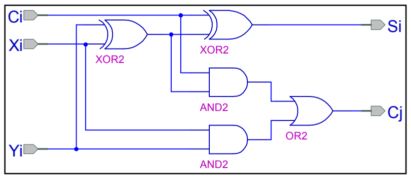
- Dataflow design uses a series of continuous assignment statements to express logic. The statements describe the flow of the data which is fed to the design from the inputs to the outputs.
- The order of statements is not important, they are executed concurrently.

```
module logic(Ci, Xi, Yi, Si, Cj);

input Ci, Xi, Yi;
output Si, Cj;
wire i1,i2,i3;

  assign i1 = Xi ^ Yi;
  assign Si = i1 ^ Ci;
  assign i2 = i1 & Ci;
  assign i3 = Xi & Yi;
  assign Cj = i2 | i3;

endmodule
```

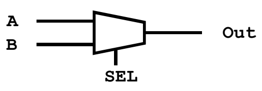


What circuit is this?

Confidential

Testing the HDL models

- ***Why should we test the HDL models?***
  - To verify if the model behaves exactly as per the specifications
  - To paraphrase, to verify the correctness of the implemented circuit in terms of its functionality, structure and other parameters.



A	*B*	*Sel*	*Out*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

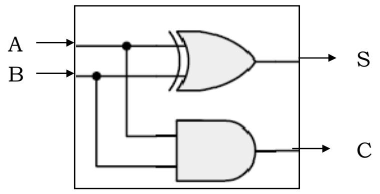
A general process of testing a digital circuit is to force on its inputs the values (Logic HIGH or Logic Zero) in various combinations and observe the output.

The circuit is said to be completely verified if all of the possible input combinations (as given in the truth table) are applied to the circuit and the output of the circuit is matching the respective expected output as per the truth table.

Confidential

Case Study: Half Adder

- Modeling a half adder in Verilog



A	*B*	*Sel*	*Out*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1

```
module ha (A, B, S, C) ;
output Out;
input A, B, Sel;


        assign S = A ^ B;
        assign C = A & B;


endmodule
```

After modeling any circuit verify its functionality in a simulator by applying stimulus at the module's input as per the truth table and observe its output

Confidential

Exercise 1

- Design and model a full adder circuit in Verilog.
- Verify the functionality in the simulator of your choice

30 Minutes

Confidential

Exercise 2

- Design a binary to grey code converter
- Verify its functionality in the simulator of your choice

20 Minutes