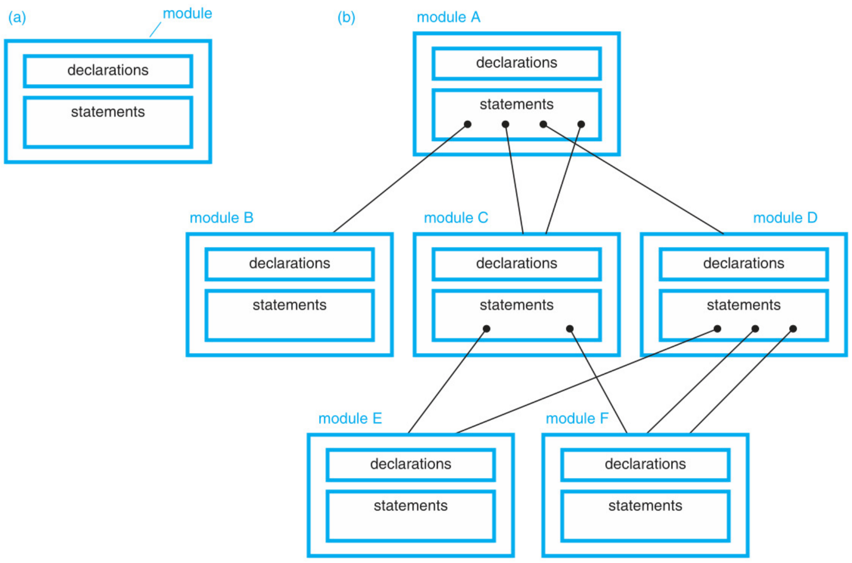
Verilog

VERILOG: it's a programming language based on C that allows us to design, simulate and synthesize just about any digital circuit.

- **Module:** it is the basic unit of design and programming in Verilog that contains declarations and statements. A Verilog module may correspond to a piece of hardware. The declarations describes the names and types of the module's inputs and outputs as well as signals, constants and functions. The statements specify the operation of the module's outputs and internal signals.
  Modules, much like functions, can instantiate each other, or in other words, they can call on each other. Values can be passed to other modules as long as if using declared input and output signals.



  The module below start with a declaration that begins with the keyword "*module*" followed by its name and key words for its inputs and outputs.

Program 5-1 Verilog model for an "inhibit" gate.

```
module VrInhibit( X, Y, Z ); // also known as 'BUT-NOT'
  input X, Y;                // as in 'X but not Y'
  output Z;                  // (see [Klir, 1972])

  assign Z = X & ~Y;
endmodule
```

Input: the signal is an input to the module.

Output: the signal is an output of the module.

Inout: the signal can be used as a model input or output

Multibit vector: signals can be declared using a range specification as in $[\boldsymbol{msb} : \boldsymbol{lsb}]$ in ascending or descending order, such that $[\mathbf{0} : \mathbf{7}]$ represents 8 bits.

Operator	Operation
&	AND
|	OR
^	Exclusive OR (XOR)
~^, ^~	Exclusive NOR (XNOR)
~	NOT

Nets: correspond to a $\boldsymbol{wire}$ in a physical unit. Verilog as many kinds of nets, the default net type is $\boldsymbol{wire}$, unless otherwise declared differently.

Reg and integers: a $\boldsymbol{reg}$ is a single bit or a vector of bits and takes values of $\mathbf{0}, \mathbf{1}, \boldsymbol{x}$ or $\boldsymbol{z}$. An $\boldsymbol{integer}$ is a signed number of 32 bits or more. $\boldsymbol{x}$ is an unknown and $\boldsymbol{z}$ is high-impedence.

Variables: unlike nets, variables can only be changed within a module and not from the outside. Also, inputs and inouts cannot have variables, but output ports can.

Literals: as seen in the table below, the syntax of $\boldsymbol{b}$ stands for binary, $\boldsymbol{o}$ for octal, $\boldsymbol{h}$ for hexadecimal and $\boldsymbol{d}$ for decimal. Unsized literals default to 32 bits. A question mark ? stands for $\boldsymbol{z}$.

Parameters: much like in C, we can declare values to variables the same way in Verilog. Such that $\boldsymbol{a} = \mathbf{8}$ assing the value of $\mathbf{8}$ to the variable $\boldsymbol{a}$.

Compiler directives: a $`include$ filename includes a file as if it is a part of the program. A $`define$ identifier text is used

Vector declarations: just like seen above, we have $\boldsymbol{msb}$ and $\boldsymbol{lsb}$ in the brackets. Logical elements can also be used in vectors as $\{\mathbf{4}'\boldsymbol{b}\mathbf{0011}\ \&\mathbf{4}'\boldsymbol{b}\mathbf{0101}\boldsymbol{s}\}$ is $\mathbf{4}'\boldsymbol{b}\mathbf{0001}$. Vectors of different sizes can be combined and the shorter one will be padded to the left, as in $\mathbf{2}'\boldsymbol{b}\mathbf{11}\ \&\ \mathbf{4}'\boldsymbol{b}\mathbf{1101}$ becomes $\mathbf{4}'\boldsymbol{b}\mathbf{0011}\ \&\ \mathbf{4}'\boldsymbol{b}\mathbf{1101}$ and results in

Literal	Meaning
1'b0	A single 0 bit
1'b1	A single 1 bit
1'bx	A single unknown bit
8'b00000000	An 8-bit vector of all 0 bits
8'h07	An 8-bit vector of five 0 and three 1 bits
8'b111	The same 8-bit vector (0-padded on left)
16'hF00D	A 16-bit vector that makes me hungry
16'd61453	The same 16-bit vector, with less hunger

to define values.

$\boldsymbol{4'b0001}$. As for literas $\boldsymbol{8'bx}$ is an 8-bit vector of all x's, and $\boldsymbol{b'8z00}$ is equivalent to $\boldsymbol{8'zzzzzz00}$.

```
reg [7:0] byte1, byte2, byte3;
reg [15:0] word1, word2;
reg [1:16] Zbus;
```

`2'b1011`	Tricky or an error, leftmost “10” ignored
`4'b1?zz`	A 4-bit vector with three high-Z bits
`8'b01x11xx0`	An 8-bit vector with some unknown bits

Operator	***Operation***
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (remainder)
**	Exponentiation
<<	(Logical) shift left
>>	(Logical) shift right
<<<	Arithmetic shift left
>>>	Arithmetic shift right

Signed operations and shifts: a sum of $\boldsymbol{4'sb1101} + \boldsymbol{1'b1}$ would result in 14 (13 + 1) because $\boldsymbol{1'b1}$ is unsigned. A $\boldsymbol{logical\ shift}$ also does not work with signed values and $\boldsymbol{arithmetic\ shifts}$ need to be used instead.

A $\boldsymbol{shift}$ operation simply shift the bits to the left or right and pads the space with 0's. For instance $\boldsymbol{8'b11010011} \ll \boldsymbol{3}$ becomes $\boldsymbol{8'b10011000}$. $\$\boldsymbol{signed}(\square)$ and $\$\boldsymbol{unsigned}(\square)$ functions can be used to convert between each other.

Arrays: function in the same way as in C where the store a set of variable of one type, except that here we have to specify the range. A $[\boldsymbol{msb} : \boldsymbol{lsb}]\ \boldsymbol{identifier}\ [\boldsymbol{start} : \boldsymbol{end}]$ implies the it is an array that stores $[\boldsymbol{start} : \boldsymbol{end}]$ number of elements of size $[\boldsymbol{msb} : \boldsymbol{lsb}]$.

```
reg identifier [start:end];
reg [msb:lsb] identifier [start:end];
integer identifier [start:end];
wire identifier [start:end];
wire [msb:lsb] identifier [start:end];
```

Logical operations: $\mathbf{0}$ is always false and, unknowns $\boldsymbol{x}, \boldsymbol{z}$ and any other value is considered true.

For unsigned values, if a number is shorter than the other, it is padded to the left with zeroes. For signed values, the left most bit is replicated before the comparison.

===	case equality
!==	case inequality

Operator	***Operation***
&&	logical AND
||	logical OR
!	logical NOT
==	logical equality
!=	logical inequality
>	greater than
>=	greater than or equal
<	less than
<=	less than or equal

?: conditional operator: as seen below, we have a logical expression, if it is true the first value is selected, if not, the second value is selected.

```
logical-expression ? true-expression : false-expression
X ? Y : Z
(A>B) ? A : B;
(sel==1) ? op1 : (
  (sel==2) ? op2 : (
    (sel==3) ? op3 : (
      (sel==4) ? op4 : 8'bx )))
```

Gate types: $\boldsymbol{and}$, $\boldsymbol{or}$, $\boldsymbol{xor}$ and their complements can have any number of inputs, $\boldsymbol{not}$ is an inverter and $\boldsymbol{buf}$ is a 1-input non-inverting buffer. The other 4 gates are also 1-input buffers and inverters that output its value if the inputed value is the same, otherwise they output a $\boldsymbol{z}$.

```
and     xor     bufif0
nand    xnor    bufif1
or      buf     notif0
nor     not     notif1
```

Procedural statements and always blocks: procedural statements means that in Verilog all statements run concurrently like in real hardware. With an $\boldsymbol{always}$ block, whatever is within the block will run sequentially like in other programming languages. It is important to note, however, that an $\boldsymbol{always}$ block will run concurrently with other statements. The $(*)$ notation is shorthand for "every signal that may change as a result" and is up to the compiler to decide what signlas sohuld be in there.

The last form of the statement seen to the right loops forever and is useful for test benches where we can generate a repetitive signal, like a clock.

An always block will only execute when one or more of the values on its sensitivity list changes.

A $\boldsymbol{reg}$ variable may only be used inside an $\boldsymbol{always\ block}$.

Always blocks naming:

```
always @ (signal-name, signal-name, ... , signal-name)
  procedural-statement
always @ ( * ) procedural-statement

always @ (posedge signal-name) procedural-statement
always @ (negedge signal-name) procedural-statement

always procedural-statement
```

Assign: can be used to assign values to a variable, or for instance, use it to write a

Blocking and nonblocking: blocking (=) assigns a value to the left side variable

Begin-end blocks: are other kinds of blocks where the code runs sequentially logic equation directly instead of the gates and their connections.

```
assign net-name = expression;
assign net-name[bit-index] = expression;
assign net-name[msb:lsb] = expression;
assign net-concatenation = expression;
```

immediately. On the other hand, nonblocking $(<=)$will assign the value in an infinitesimal amount of time after its execution, and for instance, will allow a an always block to run its course before assigning the value to the variable.

and where we can define our own variables for instance.

```
begin
   procedural-statement
   . . .
   procedural-statement
end

begin : block-name
   variable declarations
   parameter declarations

   procedural-statement
   . . .
   procedural-statement
end
```

```
repeat ( integer-expression )
   procedural-statement

while ( logical-expression )
   procedural-statement

forever
   procedural-statement
```

Functions and tasks: they contain only one procedural statement and work similarly to functions in C. The difference between functions and tasks is that tasks do not return a value.

```
function result-type function-name ;
   input declarations
   variable declarations
   parameter declarations

   procedural-statement
endfunction
```

$display and $write: they are like the $\boldsymbol{printf}$ function in C, the difference between $\$\boldsymbol{display}$ and $\$\boldsymbol{write}$ is that $\$\boldsymbol{write}$ does not append a new line at the end.

$monitor: it simply monitors a variable throughout the code, but only one $\$\boldsymbol{monitor}$ can be active at the same time. It can be turned on and off with $\$\boldsymbol{monitoron}$ and $\$\boldsymbol{monitoroff}$

$fflush: show the last of the results before the simulation terminates. Useful in test benches.

$time: returns the simulated time of a module.

$stop: suspends the simulation. If $\$\boldsymbol{stop}(\mathbf{1})$ is used, it returns the simulated time as well.

Test benches: the entity being tested is normally called the $\boldsymbol{UUT}$.

Cases: are another way to write code where we list all the cases for which the function is true. It is easier to uses cases than many $\boldsymbol{if}$ statements.