Hardware Description Languages

Hardware description languages (HDLs) describe the architecture and behavior of discrete electronic systems. Modern HDLs and their associated simulators are very powerful tools for integrated circuit designers.

Description Styles

A Verilog circuit description can be one of two types: structural or functional.

- A structural description explains the physical makeup of the circuit, detailing gates and the connections between them.
- A functional description, also referred to as an RTL (Register Transfer Level) description, describes what the circuit does.

Modules

The principal design entity in the Verilog language is the module. A module consists of the module name, its input and output description (port definition), a description of the functionality or implementation for the module (module statements and constructs), and named instantiations.

Module Instantiations

A module instantiation consists of the name of the module (module_name) followed by one or more instantiations. An instantiation consists of an instantiation name (instance_name) and a connection list. A connection list is a list of expressions called terminals, separated by commas. These terminals are connected to the ports of the instantiated module.

Module Instantiations

```
module SEQ(BUS0,BUS1,OUT); //description of module SEQ
     input BUS0, BUS1;
     output OUT;
     ...
endmodule

module top( D0, D1, D2, D3, OUT0, OUT1 );
     input D0, D1, D2, D3;
     output OUT0, OUT1;

     SEQ SEQ_1(D0,D1,OUT0),  //instantiations of module SEQ
         SEQ_2(.OUT(OUT1),.BUS1(D3),.BUS0(D2));
endmodule
```

Port Definitions

A port list consists of port expressions that describe the **input** and **output** interfaces for a module. Define the port list in parentheses after the module name, as shown here:

```
module name ( port_list );
```

A port expression in a port list can be any of the following:

- An identifier
- A single bit selected from a bit vector declared within the module
- A group of bits selected from a bit vector declared within the module

An **input port** is a type of wire. Unless otherwise defined by a reg declaration, an **output port** is a type of wire.

Port Names

- It should start with a letter not a number.

Data Types

Wire

A wire data type in a Verilog description represents the physical wires in a circuit. A wire connects gate-level instantiations and module instantiations.

A wire does not store its value. It must be driven in one of two ways:

- By connecting the wire to the output of a gate or module
- By assigning a value to the wire in a **continuous** assignment

wire Declarations

```
wire a;
wire [2:0] b;
```

Register

A reg represents a variable in Verilog. A reg can be a 1-bit quantity or a vector of bits. For a vector of bits, the range indicates the most significant bit and least significant bit of the vector. Both must be nonnegative constants, parameters, or constant-valued expressions.

A register is a simple, 1-bit memory device, either a latch or a flip-flop. A latch is a level-sensitive memory device. A flip-flop is an edge-triggered memory device.

reg Declarations

```
reg x;              //single bit
reg a,b,c;          //3 1-bit quantities
reg [7:0] q;        //an 8-bit vector
```

Operators

Operators identify the operation to be performed on their operands to produce a new value. Most operators are either unary operators, which apply to only one operand, or binary operators, which apply to two operands. Two exceptions are conditional operators, which take three operands, and concatenation operators, which take any number of operands.

Verilog Operators Supported by FPGA Compiler II / FPGA Express

Operator Type	Operator	Description
Arithmetic Operators	+ – * / %	Arithmetic Modules
Relational Operators	> >= < <=	Relational
Equality Operators	== !=	Logical equality Logical inequality
Logical Operators	! && ||	Logical NOT Logical AND Logical OR
Bitwise Operators	~ & | ^ ^~ ~^	Bitwise NOT Bitwise AND Bitwise OR Bitwise XOR Bitwise XNOR
Reduction Operators	& | ~& ~| ^ ~^ ^~	Reduction AND Reduction OR Reduction NAND Reduction NOR Reduction XOR Reduction XNOR

Verilog Operators Supported by FPGA Compiler II / FPGA Express(Continued)

Operator Type	Operator	Description
Shift Operators	<< >>	Shift left Shift right
Conditional Operator	? :	Conditions
Concatenation Operator	{ }	Concatenation

Operator Precedence

The table lists the precedence of all operators, from highest to lowest. All operators at the same level in the table are evaluated from left to right, except the conditional operator (?:), which is evaluated from right to left.

Operator Precedence

Operator	Description
[ ]	Bit-select or part-select
( )	Parentheses
! ~	Logical and bitwise negation
& | ~& ~| ^ ~^ ^~	Reduction operators
+ –	Unary arithmetic
{ }	Concatenation
* / %	Arithmetic
+ -	Arithmetic
<< >>	Shift
> >= < <=	Relational
== !=	Logical equality and inequality
&	Bitwise AND
^ ^~ ~^	Bitwise XOR and XNOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Conditional

Continuous Assignment

If you want to drive a value onto a wire use a continuous assignment to specify an expression for the wire value. You can specify a continuous assignment in two ways:

- Use an explicit continuous assignment statement after the wire declaration.
- Specify the continuous assignment in the same line as the declaration for a wire.

Two Equivalent Continuous Assignments

```
wire a;              //declare
assign a = b & c;    //assign
wire a = b & c;      //declare and assign
```

Procedural Assignment

A procedural assignment updates the value of register data types. It can be a blocking or non-blocking assignment.

- The expression in a blocking procedural assignment is evaluated and assigned when the statement is encountered. In a begin-end sequential statement group, execution of the next statement is blocked until the assignment is complete.

- In a non-blocking procedural assignment, the expression is evaluated when the statement is encountered, and assignment is postponed until the end of the time-step. In a begin-end sequential statement group, execution of the next statement is not blocked and may be evaluated before the assignment is complete.

The following shows the syntax:

```
[ delay ] register_name = [ delay ] expression;     // blocking
[ delay ] register_name <= [ delay ] expression;    // non-blocking
```

Example:

```
begin
  a = 0;
  #10 a = 1;
  #5 a = 2;
end              // time 0: a=0; time 10: a=1; time 15 (#10+#5): a=2;

begin
  a <= 0;
  #10 a <= 1;
  #5 a <= 2;
end             // time 0: a=0; time 5: a=2; time 10: a=1;

begin
  a <= b;
  b <= a;
end             // both assignments are evaluated before a or b changes
```

Always Blocks

An always block can imply latches or flip-flops, or it can specify purely combinational logic. An always block can contain logic triggered in response to a change in a level or the rising or falling edge of a signal. The syntax of an always block is

```
always @ ( event-expression [or event-expression*] )   begin
    ... statements ...
end
```

Always Block Examples

A Simple always Block

```
always @ ( a or b or c ) begin
    f = a & b & c
end
```

```
always @ ( posedge CLOCK or negedge reset ) begin
   if !reset begin
    ... statements ...
   end
   else begin
    ... statements ...
   end
end
```

Structural Description Style Example

Figure 1 depicts a simple module that implements a 2-input NAND gate by instantiating an AND gate and an INV gate. The first line of the module definition gives the name of the module and a list of ports. The second and third lines give the direction for all ports. (Ports are either inputs, outputs, or bi-directionals.)
The fourth line of the description creates a wire variable. The next two lines instantiate the two components, creating copies named instance1 and instance2 of the components AND and INV. These components connect to the ports of the module and are finally connected by use of the variable and_out.

Module Definition

```
module NAND(a,b,z);
     input a,b;      //Inputs to NAND gate
     output z;        //Outputs from NAND gate
     wire and_out; //Output from AND gate

     AND instance1(a,b,and_out);
     INV instance2(and_out, z);
endmodule
```