Introduction to Verilog

The Verilog Language

- Full form of Verilog is Verify Logic
- It is a **Hardware Description Language** used for describing any **digital system**
- It is only for **Digital Circuits** and not for Analog Circuits
- Most common use of a HDL is to provide an alternative to **schematic**.
- Most commonly used languages in digital hardware design(other- VHDL, SystemVerilog)

Program Structure

❖ Every Verilog program starts with the keyword ‘module’ and ends with the keyword ‘endmodule’ as shown below:

```
module <module name> (input, output);
        <define input>
        <define output>
        ….
        <Logic of the program>
        ….
endmodule
```

Lexical Conventions

- Case Sensitivity – Verilog HDL is case sensitive
- White Spaces – Blank spaces, Tabs, New line etc doesn't mean anything
- Identifiers – Must begin with (a-z, A-Z,_)

  May contain(a-z , A-Z, _ , 0-9, $)

  Up to 1024 character long
- Comments – Single Line //

  Multiline /* Text */

Declaration of input and output

- **Example**: input a, b; // two inputs each of one bit
- If ports are more than one bit long then we can define as below:

  input [3:0] a, b; // four bit inputs(a_3-a_0 and b_3-b_0)

  output [2:0] c;

Numbers

Syntax: <size>’<base><number>

Examples:

Decimal: **234** or **3’d234**

Binary: **4’b0010**

Octal: **12’o43xz** which is equivalent to **12’o100011xxxzzz**

Hexadecimal: **8’hax** is equivalent to **8’h1010xxxx**

Data Types

The main data types in Verilog are:

1. Net
2. Reg
3. Vectors
4. Integers
5. Real
6. Parameters
7. String
8. Memory

net Data Type

- Nets represent structural connection between the components.

- Data type declaration:

  wire a, b, y;

- Assignment:

  assign y = a & b;

reg Data Type

- Represent a variable to store data
- Values are retained until updated

DATA TYPE	FUNCTION
reg	Unsigned variable
integer	Signed – 32 bits
real	Double precision floating point variable

Vectors and Memory

Vectors:

- Represent multi-bit buses.

Declaration:

reg [3:0] a;

wire [4:0] b;

Memory:

- Used to model memories like ROM, RAM etc.

Declaration:

reg [7:0] mem_a [0:255]; /* Declares a memory **mem_a** of **8x256** bit size */

mem_a [0] = 7'b10011001; /* Assigns value **10011001** to memory location 0 */

Arrays

- An array is a collection of the same types of variables and accessed using the same name plus one or more indices

Examples:

reg my_rega [0:5]; // Declares an array of 6, 1 bit registers

reg arrayb [7:0] [0:255]; /* Declares a two dimensional array of one bit registers */

wire w_array [7:0] [5:0]; //Declares a 2-D array of wires

Assignment:

my_rega [2] = 1'b1; // Assigns value 1 to the 3rd element of array

arrayb [1][0] = 0; // Assigns value 0 to bit referenced by indices [1][0]

Parameter

- Constant values to be used in the program.

Example:

```
module ram(clk, rst, din, radd, wadd, dout);
        parameter dt_width = 8;
        parameter ad_depth = 256;
        …
        …
        reg [(dt_width-1) : 0] mem [(ad_depth-1) : 0];
        …
endmodule
```

Verilog Operators

- Arithmetic Operator
- Logical Operator
- Relational Operator
- Equality Operator
- Bitwise Operator
- Shift Operator
- Concatenation Operator
- Replication Operator

Arithmetic Operators

Operator Symbol	Operation Performed	Number of Operands
*	multiply	two
/	divide	two
+	add	two
-	subtract	two
%	modulus	two
**	power (exponent)	two

Examples

1) If A = 4’b0011 and B = 4’b0100

Then,

A * B = 4’b0000

A + B = 4’b0111

2) If A = 4 and B = 2

Then,

B = A ** B = 4 ** 2 = 16

A % B = 0

Logical Operators

Operator Symbol	Operation Performed	Number of Operands
!	logical negation	one
&&	logical and	two
||	logical or	two

Examples

1) If a = 3 and b = 0

a && b = 0 (Equivalent to logic 1 && logic 0)

a || b = 1 (Equivalent to logic 1 || logic 0)

!a = 0 (Equivalent to !(logic 1))

!b = 1 (Equivalent to !(logic 0))

2) If a = 3, b = 2

Then, (a == 3 && b == 2) gives 1.

Else, the above expression gives 0

Relational Operators

Operator Symbol	Operation Performed	Number of Operands
>	greater than	two
<	less than	two
>=	greater than or equal	two
<=	less than or equal	two

Examples

If a = 4’b1100 and b = 4’b0110

a < b gives 0

a > b gives 1

a <= b gives 0

a >= b gives 1

Equality Operators

Operator Symbol	Operation Performed	Number of Operands
==	equality	two
!=	Inequality	Two
===	case equality	two
!==	case inequality	two

Example

== logical equality (test for 1,0) all other will result in x

=== logical equality (test for 1,0,x,z)

If a = 4’b1010, b = 4’b1101, c = 4’b1010, d = 4’b1xxz, e = 4’b1xx0,
f = 4’b1xxz

Then,

a == b gives 0

a != b gives 1

d == e gives x

a === c gives 1

a !== c gives 0

a === e gives 0

Bitwise Operators

Operator Symbol	Operation Performed	Number of Operands
~	bitwise negation	one
&	bitwise and	two
|	bitwise or	two
^	bitwise xor	two
^~ or ~^	bitwise xnor	two

Example

If a = 4’b1010 and b = 4’b1101

Then,

~a = 4’b0101

a & b = 4’b1000

a | b = 4’b1111

a ^ b = 4’b0111

a ^~ b = 4’b1000

Shift Operators

Operator Symbol	Operation Performed	Number of Operands
>>	Right shift	two
<<	Left shift	two
>>>	Arithmetic right shift	two
<<<	Arithmetic left shift	two

Example

If a = 4’b1100

Then,

a >> 1 = 4’b0110 (Right shift by 1 bit)

a << 1 = 4’b1000 (Left shift by 1 bit)

Note: Operator >>> or <<< works same as >> or << if working on unsigned numbers but in case of signed numbers

>> : preserve the sign bit

>>> : does not preserve the sign bit

Concatenation and Replication Operator

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number

Examples

If a = 1’b1 and b = 2’b00

Then,

y = {a, b} gives 3’b100 // Concatenation operation

y = { 4 { a } } gives 4’b1111 // Replication operation

y = { 4 { a }, 2 { b } } gives 8’b11110000 //Replication cum Concatenation operation

Conditional Operator

Operator Symbol	Operation Performed	Number of Operands
? :	Conditional	Three

Syntax and Example

Syntax:

condition ? < True Expression > : < False Expression >

Example:

If x = 20

Then,

assign out = (x == 20) ? a : b (Where, a and b are inputs)

gives out = a

assign out = (x == 3) ? a : b

gives out = b

Internal Variable Monitoring System Tasks

$display and $write

- **$display** and **$write** displays the specified variables once when the command is executed.
- Only difference is that **$display** automatically adds a newline character to the end of its output which **$write** does not.

Example:

```
module disp;
        reg [15:0] mreg;
        initial
            begin
                mreg = 101;
                $display (“mreg = %h hex, %d decimal”, mreg, mreg);
                $write (“mreg = %o octal mreg =%b bin”, mreg, mreg);
                $write (“simulation time is %t”, $time);
             end
```

Simulation Results

mreg = 0065 hex, 101 decimal

mreg = 000145 octal mreg = 0000000001100101 bin simulation time is 0

$strobe

- $strobe displays simulation data at the end of the current simulation time.

Example:

```
module disp_check();
        reg a;
        initial
            begin
                a = 1'b0;
                $strobe ("Strobe : Value of a is %b", a);
                $display ("Display : Value of a is %b", a);
                a = 1'b1;
             end
endmodule
```

Simulation Results

Display: Value of a is 0

Strobe: Value of a is 1

$monitor

- Each time a variable or an expression in the argument changes value, the entire argument list is displayed at the end of the time step.

Example:

```
module monitor();
    reg [1:0] a;
            initial
                begin
                        a = 2'b00;
                        #5 a = 2'b01;
                        #5 a = 2'b10;
                        #5 a = 2'b11;
                end
            $monitor("Display : Value of a is %b", a);
endmodule
```

Simulation Results

At 5 units time→ Display: Value of a is 00

At 10 units time→ Display: Value of a is 01

At 15 units time→ Display: Value of a is 10

At 20 units time→ Display: Value of a is 11

$stop and $finish

- **$stop** suspends the simulation

Example:

```
#100 $stop // Suspends the simulation at 100 time units
```

- **$finish** makes the simulator exit and pass the control to the operating system

Example:

```
#500 $finish // Terminates the simulation at 500 time units
```

$time

- Returns time value which is a 64-bit integer, scaled to the timescale unit of the current module

Example:

```
`timescale 10ns/ 10ps
module test();
        reg set;
        parameter p = 2;
        initial
            begin
                $monitor($time, " set = %d", set);
                #p set = 0;
                #p set = 1;
            end
endmodule
```

Simulation Results

At 2 units time→ set = x

At 4 units time→ set = 0

At 6 units time→ set = 1

`timescale

- The **`timescale** compiler directive specifies the time unit and precision for the modules that follow it.

Syntax:

`timescale <time unit> / <time precision>

- The ***time unit*** is the unit of measurement of delays and simulation time
- ***time precision*** specifies how delay values are rounded-off before being used in the simulation

Example:

```
`timescale 10ns/ 10ps
module tb_top();
        reg clk;
        …..
        initial
            begin
                clk = 1'b0;
                forever #10 clk = ~clk;  // 10 time units = 100 ns
            end
        …..
endmodule
```

Verilog code for clock signal