* Lecture 3 :Verilog Hardware Description Language, Syntax Lexical Conventions, Data Types and Memories

Engr. Carla May C. Ceribo

CPE09 – Introduction to HDL

The language is made up of statements, groups of statements, and keywords to identify the different types of statement groups. These are properties of Verilog statements:

* Statements are composed of tokens.
* Statements can be continued across line boundaries, but individual tokens cannot.
* Statements within a given group are usually separated by ";", but statement groups are usually not separated by ";".
* In general, the ";" is a statement separator, not a terminator.
* Verilog syntax appears to be something of a cross between C and Pascal.

*Verilog Syntax

*Tokens

Verilog is a token-based language. The source stream which a Verilog processor sees is a sequence of tokens. These are the types of Verilog tokens:

White Space

- White space is any sequence of space, tab, newline, or form feed. White space separates tokens and may be arbitrarily long between tokens.

* Comments Verilog has two kinds of comments:

  // single line comment

  /* block comment */

* Operator

  Verilog uses the same symbols as C does for operators, with a few extras. Operators are single, double, or triple character combinations. There are unary and binary operators, as well as one ternary operator (?:). The +, -, &, |, ^, and ~^ operators can be either unary or binary, depending on context.

*Tokens

* Constants

Verilog uses three types of constants: Integer Constants Real Constants and String

* Identifiers

Identifiers name objects. Objects which can be named are modules, instances, nets, registers, parameters, tasks, functions, blocks, and source macro. Identifiers may be made up of any number of letters, digits, and {_$}:

* simple_identifier ::= [a-zA-Z_]{[a-zA-Z0-9_$]}

*Tokens

* Keywords

* Verilog contains keywords, which are predefined, non-escaped identifiers. An escaped identifer is not treated as a keyword. For example, \begin is not a keyword.

*Tokens

* System Tasks and Functions

System tasks and functions are predefined tasks and functions which provide common operations. Identifiers which start with "$" are considered to be system tasks or system functions. These are used to provide common facilities to the model. This includes output capabilities and information facilities. Like user-defined tasks, system tasks must be invoked as a procedural statement.

*Tokens

List of Operators

<table>
<tr><th>{}, {{}}</th><th>concatenation, replication</th></tr>
<tr><td>+ - * /</td><td>arithmetic</td></tr>
<tr><td>%</td><td>modulus</td></tr>
<tr><td>> >= < <=</td><td>relational</td></tr>
<tr><td>!</td><td>logical negation</td></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>||</td><td>logical or</td></tr>
<tr><td>==</td><td>logical equality</td></tr>
<tr><td>!=</td><td>logical inequality</td></tr>
<tr><td>===</td><td>case equality</td></tr>
<tr><td>!==</td><td>case inequality</td></tr>
<tr><td>~</td><td>bit-wise negation</td></tr>
<tr><td>&</td><td>bit-wise and</td></tr>
<tr><td>|</td><td>bit-wise inclusive or</td></tr>
<tr><td>^</td><td>bit-wise exclusive or</td></tr>
<tr><td>^~ ~^</td><td>bit-wise equivalence</td></tr>
<tr><td>&</td><td>reduction and</td></tr>
<tr><td>~&</td><td>reduction nand</td></tr>
<tr><td>|</td><td>reduction or</td></tr>
<tr><td>~|</td><td>reduction nor</td></tr>
<tr><td>^</td><td>reduction xor</td></tr>
<tr><td>~^ ^~</td><td>reduction xnor</td></tr>
<tr><td><<</td><td>left shift</td></tr>
<tr><td>>></td><td>right shift</td></tr>
<tr><td>?:</td><td>conditional</td></tr>
</table>

These are examples of integer constants:

* 123
* 1'b1
* 8'h81
* 'o7773
* 12'bx
* 16'd5
* 3'b1xz
* 32'h8f_32_ab_f7

Sized vs. Unsized

A sized constant has the size specifier present, while an unsized constant does not. Unsized numbers have a default size of 32 bits.

Signed vs. Unsigned

If both the size and radix are omitted, the constant is a signed number, represented in 2's complement. Otherwise, the constant is an unsigned, positive number. This is only visible if the constant is preceded by a unary minus sign.

Radix Specifiers

The radix specifier indicates that the digits following are decimal ('d), hexidecimal ('h), octal ('o), or binary ('b). Each digit represents the appropriate number of bits for the radix.

Padding and Truncation

The number of bits represented by the digits in the value part of the constant may be more or less than the given size. If there are more, then the high order bits are truncated. For example,

7'h8f is equivalent to 7'h0f

If the size is greater than the number of bits in the value part (which is a much more common case), then the number is padded on the left (high order part) with 0. However, if the left-most digit in the value part is x or z, then the number is padded with x or z.

12'h3 is equivalent to 12'b000000000011

12'h3x is equivalent to 12'b00000011xxxx

12'bx is equivalent to 12'bxxxxxxxxxxxx

12'oz37 is equivalent to 12'bzzzzzz011111

* A <u>string constant</u> is a sequence of characters enclosed in " (double quotes). String constants may be used where ever a vector is allowed, and, in general, is equivalent to a vector whose width is 8 times the number of characters in the string. The value of the vector is the same as if the ASCII values of each character were concatenated.

For example

* "Hello World"   is represented as 88'h48656c6c6f_20_576f726c64

*Identifiers

shiftreg_a
busa_index
error_condition
merge12
_bus23
n$657

Escaped Identifiers:

\wire*
\busa+index
\-clock
error-condition
\net1/\net2

Hierarchical identifier

abc.def
top.foo.bar.xyz
system.board.chip.wire123

1. Module	Net names, top-level register names, task names, function names, module and primitive instance names, and port names exist in the module's scope. That is, two different modules can each have a net named "net1 or a module instance named "foo_inst". However, there can be only one "net1" or "foo_inst" in a single module.
2. Task, Function, Block	Tasks, functions, and named blocks allow registers, parameters, and named blocks to be defined within them. These names exist in hierarchical name spaces. That is, an identifier in an outer scope (module, task, function, or block) may be redefined in an inner scope (task, function, or block).
3. Global	There is a single scope which contains all module types (i.e. the name used in the module definition). Thus, there can be only one module of type "DF99".
4. Macros	Source macros have a single global scope that crosses module boundaries.

*Namespaces

always	inout	rtranif0	and	input	rtranif1
assign	integer	scalared	begin	join	small
buf	large	specify	bufif0	macromodule	specparam
bufif1	medium	strength	case	module	strong0
casex	nand	strong1	casez	negedge	supply0
cmos	nmos	supply1	deassign	nor	table
default	not	task	defparam	notif0	time
disable	notif1	tran	edge	or	tranif0
else	output	tranif1	end	parameter	tri
endcase	pmos	tri0	endmodule	posedge	tri1
endfunction	primitive	triand	endprimitive	pull0	trior
endspecify	pull1	trireg	endtable	pulldown	vectored
endtask	pullup	wait	event	rcmos	wand
for	real	weak0	force	realtime	weak1
fork	reg	while	function	release	wire
highz0	repeat	wor	highz1	rnmos	xnor
if	rpmos	xor	initial	rtran	

*Keywords

```
// Data Types - Example
module sum_product();

  integer a, b;
  integer sum_int;

  real x, y;
  real prod_real;

  initial begin
    a = 3;
    b = 9;
    sum_int = a + b;
    $display("\n\t a = %0d, b = %0d, sum = %0d", a, b, sum_int);

    x = 99.67;
    y = -33.41;
    prod_real = x * y;
    $display("\n\t x = %0.2f, y = %0.2f, prod_real = %0.2f", x, y, prod_real);
  end

endmodule
```

```
module literal_values();

  reg [7:0] my_var;

  // All the assignments are grouped in an 'initial' procedure
  initial begin
    my_var = 8'd137;              // 137 in decimal
    $display("my_var = %d", my_var);

    #2 my_var = 8'h89;            // 137 in hexadecimal
    $display("my_var = %x", my_var);

    #2 my_var = 8'b1000_1001;     // 137 in binary
    $display("my_var = %b", my_var);

    #2 my_var = 8'o211;           // 137 in octal
    $display("my_var = %o", my_var);

    #2 my_var = 8'hZ1;            // zzzz_0001
    $display("my_var = %b", my_var);

    #2 my_var = 1'b1;             // 8bit variable gets 1 bit value
    $display("my_var = %b", my_var);

    #2 my_var = 12'b1111_1111_0000; // 8 bit variable gets 12bit value
    $display("my_var = %b", my_var);
  end

endmodule
```