Data Types and Operators

There are two kinds of variables in Verilog.

- **nets**: used to represent structural connectivity
- **registers**: used as abstract storage elements.

All structural connections are made with nets.

Verilog provides a variety of net types to enable the code to accurately model the hardware.

*wire* is by far the most popular net type.

It defines connectivity with no logical behavior or functionality implied.

Other types include *tri, wand, wor, triand, trior, supply0, supply1, tri0, tri1* and *trireg* to enable more advanced modeling of hardware.

A net may be assigned a value **explicitly** only by a continuous assignment stmt or **implicitly** as an output of a primitive or module.
Data Types and Operators

Register variables can be assigned value only within a behavior, task or function.

*reg* and *integer* are the most common *register* types.

Other types include *time*, *real* and *realtime*.

The rules for using nets and registers in ports of modules and primitives:

<table>
<thead>
<tr>
<th>Variable type</th>
<th>input</th>
<th>output</th>
<th>inout</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>register</td>
<td>NO</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

Memory Declaration (two-dimensional arrays)

Verilog provides an extension to the register variable declaration for this.

For example, for 1024, 32-bit words:

```verilog
reg [31:0] cache_memory [0:1023];
```

Note that *bit-select* and *part-select* are not valid with memories, only entire *words* can be addressed.

To access bits, assign a *word* to a 32-bit register.
**Data Types and Operators**

Verilog has a 4-valued logic set:

- *Logic 0, 1, x* to represent an **unknown** logic value and *z* to represent a high impedance (floating) condition.

A truth table is used to define the mapping of inputs to outputs for each of these values.
Data Types and Operators

Strings
Verilog does not have a distinct data type for strings.

Instead, a string must be stored within a properly sized register by a procedural assignment stmt.

```verilog
reg [8*num_char-1 : 0] string_holder;
```

Don’t forget, each character requires 8 bits.

Assignments of shorter strings, e.g., "Hello World" to a 12 character array, will cause zeros to be assigned starting at the MSB.

Constants
Declared using the keyword `parameter`.

```verilog
parameter byte_size = 8 ; // integer
```
Operators

Operators (Arithmetic)

Standard operators here include +, -, *, / and %

When arithmetic operations are performed on vectors, the result is determined by modulo $2^n$ arithmetic.

Note that a negative value is stored in 2’s complement format, but is interpreted as an unsigned value when used in an expression.

For example, if $A = 5$ and $B = 2$, the result $B-A$ yields 29 (11101) in a 5-bit register when printed as an unsigned value.

The value interpretation is -3.

Bitwise Operators

Standard operations include ~, &, |, ^, ~^ (bitwise exclusive nor)

If the operands do not have the same size (in the case of a binary operation), the shorter word is extended with 0 padding.
Operators

Reduction Operators

These are unary operators which create a single bit value for a data word of multiple bits.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reduction and, nand</td>
</tr>
<tr>
<td>~&amp;</td>
<td>reduction nand</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^, <del>^, ^</del></td>
<td>reduction xor, xnor</td>
</tr>
</tbody>
</table>

For example, if $y$ is $1011_0001$, the reduction and, $\&$, operation produces $y = 0$.

Logical Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and, or</td>
</tr>
<tr>
<td>==, !=</td>
<td>Logical equality, inequality</td>
</tr>
<tr>
<td>===, !==</td>
<td>Case equality, inequality</td>
</tr>
</tbody>
</table>
Operators

Logical Operators (cont):
Operate on Boolean operands. Operands may be a net, register or expression and are treated as unsigned.

For example, if ((a < size -1) && (b != c) && (index != last_one)) ...

The case equality operators (===) determine whether 2 words match identically on a bit-by-bit basis, including bits that have values "x" and "z"

The logical equality operators (==) are less restrictive.
They are used in expressions to test whether 2 words are identical but it produces an "x" result if any bits are "x".

Relational Operators
The operators <, <=, > and >= compare operands and produce a Boolean result (true or false).
If the operands are nets or registers, their values are treated as unsigned.
Operators

Shift Operators

Shift operators, >> and <<, are unary operators which perform left or right shifts, zero filling vacated positions.

Conditional Operator:

conditional_expression ::= expression ? true_expression: false_expression

If expression evaluates to Boolean true, then true_expression is evaluated, otherwise false_expression.

Y = (A == B) ? A : B;     // Y gets A if A==B is true.

Concatenation Operator

Forms a single operand from two or more operands, and is useful for forming logical buses.

If $A = 1011$ and $B = 0001$, then $\{A, B\} = 1011_0001$.
$\{4\{a\}\} = \{a, a, a, a\}$
Operators

Operator Precedence

Verilog evaluates expression left-to-right.

Verilog also uses *short-circuit* evaluation of Boolean expression. Evaluation is terminated as soon as it is clear what the result will be.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>highest</td>
<td>Unary</td>
<td>+, -, !, ~</td>
</tr>
<tr>
<td></td>
<td>Mult, Div, Modulus</td>
<td>* / %</td>
</tr>
<tr>
<td></td>
<td>Add, Sub</td>
<td>+, -</td>
</tr>
<tr>
<td></td>
<td>Shift</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td></td>
<td>Relational</td>
<td>&lt;=, &gt;=</td>
</tr>
<tr>
<td></td>
<td>Reduction/Logical</td>
<td>&amp;=, ^~,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&amp;&amp;,</td>
</tr>
<tr>
<td>lowest</td>
<td>Conditional</td>
<td>?:</td>
</tr>
</tbody>
</table>

Use parentheses when precedence needs to be overridden.