Types, operators, hierarchy in SystemVerilog
### integer data types

<table>
<thead>
<tr>
<th>Name</th>
<th>Size and values</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit</td>
<td>single bit, 2 values: 0,1</td>
<td>bit [3:0] HighNibble;</td>
</tr>
<tr>
<td>byte</td>
<td>8 bits, signed</td>
<td>byte a, b;</td>
</tr>
<tr>
<td>shortint</td>
<td>16 bits, signed</td>
<td>shortint c, d;</td>
</tr>
<tr>
<td>int</td>
<td>32 bits, signed</td>
<td>int i,j;</td>
</tr>
<tr>
<td>longint</td>
<td>64 bits, signed</td>
<td>longint lword;</td>
</tr>
</tbody>
</table>

| logic   | single bit, 4 values: 0,1,X,Z          | **NB use mainly logic for synthesis** |
| reg     | identical to logic                     |                                 |

| integer | 32 bits, 4 values: 0,1,X,Z             |                                 |
| wire    | single bit, 4 values 0,1,X,Z           | wire z1,z2;                     |
|         | and 8 strength levels                  |                                 |

(nb. there are more net data types like wire inherited from Verilog)
Unsigned data types

integral data types can be declared as signed or unsigned.

byte, shortint, int, integer, and longint default to signed.
bit, reg, and logic default to unsigned, as do arrays of these types.

<table>
<thead>
<tr>
<th>Name</th>
<th>Size and values</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte signed</td>
<td>8 bits, signed</td>
<td>-128.. +127</td>
</tr>
<tr>
<td>byte unsigned</td>
<td>8 bits, unsigned</td>
<td>0..255</td>
</tr>
<tr>
<td>shortint</td>
<td>16 bits, signed</td>
<td>-32768 ... 32767</td>
</tr>
<tr>
<td>shortint</td>
<td>16 bits, unsigned</td>
<td>0 ... 65536;</td>
</tr>
<tr>
<td>longint</td>
<td>64 bits, signed</td>
<td>-2^63 ... 2^63-1</td>
</tr>
<tr>
<td>longint</td>
<td>64 bits, unsigned</td>
<td>0 ... 2^64-1</td>
</tr>
</tbody>
</table>
Arrays

Examples:

logic [7:0] v; // an 8-element packed logic vector
bit [3:0][7:0] memory; // a 2-dimensional bit array
int x[63:0]; // if dimensions follow the name, the array is ‘unpacked’
logic [7:0] register [0:15]; // this array has packed and unpacked dimensions

Packed dimensions:
are guaranteed to be laid out contiguously in memory
can be copied onto any other packed object of same size
can be sliced, e.g. v[4:3], see 1st example
are restricted to bit types: bit, logic, int

Unpacked dimensions:
can be arranged in memory as simulator chooses
only unpacked arrays of the same type can be copied
can be used with all data types
Enumerated and user defined data types

Examples:

```c
enum  {Monday, Tuesday, Wednesday, Thursday, Friday} WeekDay;

typedef enum {idle, cycle1, cycle2 } state; // state is a user-defined data type
state present,next;

enum {a=0, b=7, c=5, d=8} myEnum; // user can assign values to enum constants
```

The named values of an enumeration type act as int constants. Enumerations are strongly typed:
- `present = 2;` // ERROR, can’t assign a numerical constant to an enum object
- `present = state'(2);` // type casting must be used to convert an int constant

`typedef` can be used to declare user defined data types in the same way as in C
literals

General syntax: \[ size['base'] \] value

Examples:
- 1'b0 - binary 0
- 4'hF - hex base, binary equivalent: 1111
- 10 - decimal base (default)
- 'o6 - octal base, binary equivalent: 110

\( size \) is the number of bits, default is 32 bits

\( 'base \) represents the radix, default is decimal

<table>
<thead>
<tr>
<th>Base</th>
<th>Symbol</th>
<th>Legal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary</td>
<td>b,B</td>
<td>0,1,x,X,z,Z,?,_</td>
</tr>
<tr>
<td>octal</td>
<td>o,O</td>
<td>0-7, x,X,z,Z,?,_</td>
</tr>
<tr>
<td>decimal</td>
<td>d,D</td>
<td>0-9, x,X,z,Z,?,_</td>
</tr>
<tr>
<td>hexadecimal</td>
<td>h,H</td>
<td>0-9,a-f,A-F,x,X,z,Z,?,_</td>
</tr>
</tbody>
</table>

? is the same as z or Z
_ (underscore) is ignored, it is used to enhance legibility of long literals
System Verilog operators

- Operator groups
  - arithmetic
  - logical
  - relational
  - equality (sometimes included in relational)
  - reduction (bitwise logic)
  - shift
  - concatenation and replication
  - conditional
# Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>%</td>
<td>Modulus (remainder)</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
</tr>
</tbody>
</table>

Arithmetic operators will be synthesised to *combinational* logic.
Logical operators
(not to be confused with bitwise/reduction operators)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Uses are similar to those in C. Treat non-zero values (e.g. bits or integers) as logical 1. Zero is false.
Relational and equality operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal</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>Wildcard equality</td>
</tr>
<tr>
<td>!=?</td>
<td>Wildcard inequality</td>
</tr>
</tbody>
</table>

Equality operators:

1. `==` and `!=` result in X if any of their operands contains an X or Z, e.g. `X == 1 -> X`
2. `===` and `!==` check the 4-state logic explicitly therefore, X and Z values shall either match or mismatch, never resulting in X, e.g. `X === 1 -> 0 (a mismatch)`
3. `==?` and `!=?` treat X or Z as wildcard cards that match any value, thus, they too never result in X, e.g. `X ==? 1 -> 1 (a match)`
## Bitwise and Reduction operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise negation (binary complement)</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>~&amp;</td>
<td>bitwise NAND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td>~^</td>
<td>bitwise XNOR</td>
</tr>
</tbody>
</table>

Bitwise means apply to each bit (of a vector or word) individually. Reduction operator can be applied to a vector, e.g. to AND together all the bits of the vector to give a single bit result.
# Shift operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;&gt;</code></td>
<td>logical right shift</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>logical left shift</td>
</tr>
<tr>
<td><code>&gt;&gt;&gt;</code></td>
<td>arithmetic right shift</td>
</tr>
<tr>
<td><code>&lt;&lt;&lt;</code></td>
<td>arithmetic left shift</td>
</tr>
</tbody>
</table>

Uses are similar to those in C which has the logical shifts; distinction between logical and arithmetic shifts was first introduced in Verilog-2001.

- Logical right shift puts 0 at left end.
- Arithmetic right shift replicates leftmost bit (i.e. preserves the sign).
- Logical and Arithmetic left shift both put 0 at right end.
Other operators

- Concatenation
  \{\}  e.g. \{4'b1001, 2'b11\} = 6'b100111

- Replication
  \{n\{m\}\}  - replicate \(m\) for \(n\) times, e.g. \{8\{1'b0\}\} = 8'b00000000

- Conditional
  ? : e.g. (c==1'b1 ? a : 1'bZ)

- Assignment
  =, +=, -=, *=, /=, %=, &=, ^=, <<=, >>=, <<<<, >>>>>

  Notice that these are all blocking assignments

- Increment/decrement
  ++,--  use with caution, \(x++\) is equivalent to: \(x = x+1\)  not: \(x <= x+1\)

  i.e. blocking assignments; in sequential logic counters should be implemented using nonblocking : \(cnt <= cnt+1\)
Concatenation and replication examples

```vhdl
logic a,b;
{a,b} = 2'b11;
{a,b} = {1'b0, 1'b1};  // same as {a,b} = 2'b10

string sv = "Hello";
string s;
s = {sv, " ", "world"};
$display("%s\n", s);  // displays 'Hello world'

logic [7:0] number8 = 'h80;
logic [15:0] number16;

assign number16 = { {8{number8[7]}}, number8[7]};  // manual sign extension from 8 to 16 bits
```
Design hierarchy

2-input mux built of primitive gates

module INV (input logic a, output logic b);
  always_comb
    a = ~b;
endmodule

module ANDOR (input logic a, b, c, d, output logic y);
  always_comb
    y = ((a & b) | (c & d));
endmodule

// 2-to-1 multiplexer
module MUX2 (input logic Sel, A, B, output logic Y);
  logic nSel; // local net
  // Instances of primitive modules
  INV gate1 (.a(Sel), .b(nSel)); // named port mapping
  ANDOR gate2 (.a(A), .b(nSel), .c(Sel), .d(B), .y(Y));
endmodule
Parameters

Example of module declaration with a parameter

module nbitreg
  #(parameter n= 8)
  ( input logic clk,nreset,npreset,  input logic [n-1:0]D, output logic [n-1:0]Q);

always_ff @(posedge clk or negedge nreset or negedge npreset)
  if(~nreset) // both nreset and npreset are async, nreset takes priority
    Q <= {n{1'b0}}; // n 0s
  else if (~npreset)
    Q <= {n{1'b1}}; // n 1s
  else
    Q <= D;
endmodule;

//instantiation example:
nbitreg #(.n(16)) reg16 (.*); // create a 16-bit instance of nbitreg
// note implicit port mapping .*
Port mapping

- named port mapping
  - use where implicit mapping cannot be applied
- implicit port mapping
  - 'magic' SystemVerilog shortcut: .*
- positional (ordered) port mapping
  - inherited from Verilog; avoid
named port mapping

flipflops dff8(.clk(clk),.reset(reset),.D(Data[15:8]),.Q(Qout[7:0]));

// ports are named and explicitly mapped to local signals
// positions are arbitrary, i.e. order in which ports are listed
// does not matter

Named mapping **may** always be used in preference to other mapping styles, but it **must** be used in the following cases:

(1) the port and connecting net sizes do not match, e.g. (...,
    .A(A[15:8]), ...)
(2) the port is unconnected, e.g.: ( ..., .bus(), ... )
positional port mapping – inherited from Verilog

```verilog
dff8(clk,reset,D,Q);
```

// Connecting nets clk, reset, D, Q must be listed in the same
// order as corresponding port declarations

// Local nets and corresponding ports may have different names but
// must have the same types and sizes

Ideally, avoid positional port mapping and use named or implicit mapping instead. Named mapping is less prone to errors and adds legibility to the code.
Implicit port mapping

flipflops dff8(.*);

// local logic signal declarations and corresponding ports must
// have same names, types and sizes

There are six important rules for implicit port connections:

(1) named and implicit ports (.*) may not be mixed in the same instantiation,
(2) positional and implicit ports (.*) may not be mixed in the same instantiation,
(3) implicit mapping may not be used if the port and connecting net sizes do not match,
(4) implicit mapping may not be used if the port and connecting net have different names,
(5) implicit mapping may not be used if the port is unconnected,
(6) all nets or variables connected to the implicit ports must be declared in the instantiating module, either as explicit local declarations or as explicit port declarations.