Data Types and Operators

Verilog uses variables to represent values of wires in the physical circuit.

There are two kinds of variables.

- *nets*: Used to represent structural connectivity
- *registers*: Used as storage elements to store information during simulation.

  It is an abstraction of a hardware storage element but need not correspond directly to physical storage in the circuit.

Registers can be one of the following data types.

- reg
- integer
- real
- realtime
- time

Verilog has a 4-valued logic set:

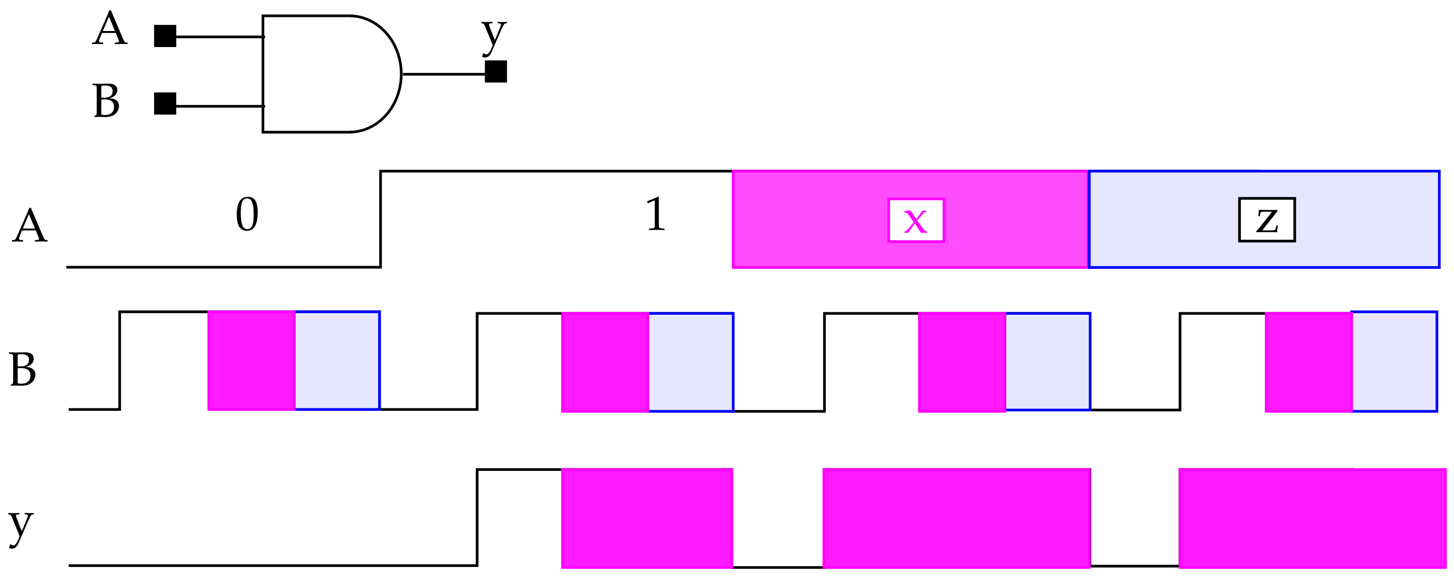
Logic 0 and *1*, *X* to represent an **unknown** logic value and *z* to represent a high impedance condition.

Data Types and Operators

The value *x* is used by the simulator to denote ambiguity -- cannot determine whether the value is *0* or *1*.

The value *z* is used when the driver of a net is disabled or disconnected.

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



Data Types and Operators

Input values in Verilog can also have a strength associated with them.

This allows pMOS and nMOS transistor (*switch-level*) modeling.

For example, pseudo-xMOS includes "always-on" xMOS transistors that act as a **weak** pullup or pulldown resistor.

These strengths allow the simulator to *resolve* the final logic value when contention exists.

Data Types

Data types are designed to allow structural connectivity and storage (*reg*) to be defined, and allow for procedural computation, (*integer*, etc).

We listed the data types of registers earlier as *reg*, *integer*, *real*, *time*, *real-time.*

All *register variables* are *static* variables, i.e., they maintain their assigned values during the entire simulation.

Data Types and Operators

A *net* may be assigned value only by a *continuous assignment* stmt or a **force ... release** procedural assignment.

All structural connections are made with nets, and the different ***net types*** allow the code to accurately model the hardware.

- *wire* defines connectivity with no logical behavior or functionality implied.
- *tri* - same as wire except it will be tri-stated in hardware.
- *wand*, *wor* -- a net with multiple drivers, which models hardware as a "wired and" (open collector) or "wired or" (emitter coupled), respectively.
- *triand*, *trior* -- same except the net can be tri-stated.
- *supply0*, *supply1* -- nets connected to the supply rails -- fixed value during the entire simulation.
- *tri0*, *tri1* -- a net connected to ground/power by a resistive pulldown/pullup connection and assumes a 0 or 1, respectively, if the driver is weaker.
- *trireg* - a net that models the charge storage on a net. The amount of charge stored determines the strength. Charge strengths include **small, medium** and **large.**

Data Types and Operators

net_declaration ::=
net_type [**vectored** | **scalared**] [range][delay3] list_of_net_ids; |
net_type [**vectored** | **scalared**] [drive_strength] [range][delay3] list_of_net_ids;
trireg [**vectored** | **scalared**] [charge_strength] [range][delay3] list_of_net_ids;

Each variable in *list_of_net_ids* can be a **scalar** (default) or an array.

delay3 allows the **transport** delay of the wire to be modeled.

vectored can be used to indicate that the net is to be treated as a *single object*, i.e., the individual bits canNOT be referenced or assigned to.

scalared is the default, which allows individual bits to be referenced.

Values are assigned to nets through primitive outputs, continuous assignment stmts or through **input**/**inout** ports.

Multiple drivers having the same strength cause the simulator to assign *x*.

Initial value assignments are *x* (driven or *reg* types) and *z* for undriven nets.

Data Types and Operators

Meaning of the *register* family of variables

- *reg* - stores a logic value.
- *integer* - supports computation
- *time* - stores time as a 64-bit unsigned quantity
- *real* - stores values (e.g., delay) as real numbers
- *realtime* - stores time values as real numbers

reg (an abstraction of hardware storage element) can be assigned only within a procedural stmt, a user-defined sequential primitive, task or function.

It **cannot** be the output of a primitive gate or target of a continuous assignment.

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

Variable type	input	output	inout
net	YES	YES	YES
register	NO	YES	NO

Note that registers cannot be declared to be **input** or **inout** port.

Data Types and Operators

Also, a register variable *cannot* be placed in an **output** port of a primitive gate, which is an implicit assignment.

It *cannot* be the target (LHS) of a continuous assignment stmt, which is an explicit assignment.

These rules result from the fact that a register variable may only be changed within a behavior, task or function within a module.

Memory Declaration (two-dimensional arrays)

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

reg_declaration ::= **reg** [range] register_name {, register_name};
register_name ::= register_identifier | memory_identifier [upper_limit : lower_limit]

For example, for 1024, 32-bit words:

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 supports hierarchical de-referencing, which allows the use of a *path* name (separated by '.'s) to access the variable.

This is useful in test benches to access internal signals.

Strings

Verilog does not have a distinct data type for strings.

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

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.

Operators

Constants

Declared using the keyword **parameter**.

parameter byte_size = 8 // integer

Operators (Arithmetic)

Standard operators here include +, -, *, /,% (modulus)

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 compliment 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.

The value is correct (-3), it's just interpreted as a positive number.

Operators

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.

Reduction Operators

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

Symbol	Operator
&, ~&	reduction and, nand
|, ~|	reduction or, nor
^, ~^, ^~	reduction xor, xnor

For example, if y is **1011_0001**, the *reduction and* operations produces *&y = 0*.

Operators

Logical Operators

Symbol	Operator
!	Logical negation
&&, ||	Logical and, or
==, !=	Logical equality, inequality
===, !==	Case equality, inequality

Operate on Boolean operands. Operands may be a net, register or expression.

Commonly used with the *conditional assignment operator* and in conditional, *if* stmts within behaviors, functions or tasks.

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"

Operators

Logical Operators

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 when the test is *ambiguous*.

If any bit is unknown, the relation is unknown and the result is ambiguous ("x" value).

Comparisons are made bit by bit and zeros are filled in as necessary.

If the operands are nets or registers, their values are treated as *unsigned words*.

If the operands are integers or reals, they may be signed but they are compared as if they are unsigned.

Since Verilog is loosely typed, care should be taken here.

Operators

Logical Operators

If *A* and *B* are scalers (1-bit variables), *A* && *B* and *A* & *B* return the same result.

If they are vectors, *A* && *B* return true if both words are positive integers.

On the other hand, *A* & *B* returns true **if the word formed** from the bit-wise operation is a positive integer.

For example, suppose *A* = 3'b110 and *B* = 3'b11x, then *A* && *B* is 0 (false) because *B* is false. However, *A* & *B* returns 110, which has a boolean value of true

Relational Operators

The operators compare operands and produce a Boolean result (*true* or *false*).

If the operands are nets or registers, their values are treated as unsigned.

Operators

Relational Operators

If any bit is *unknown*, the relation is unknown and the result returned is ambiguous, "x" value.

Standard operators include <, <=, > and >=.

Shift Operators

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

For example, if *A* = 1011_0011 then *A* << produces 0110_0110 and *A* << *3* produces 1001_1000.

```
module shift;
  reg [1:0] start, result;
    initial
    begin
      start = 1;
      result = (start << 1);
    end
endmodule
```

Result is 10_2

Operators

Conditional Operator

conditional_expression ::= expression ? true_expression: false_expression

Same as C, if *expression* evaluates to Boolean true, then *true_expression* is evaluated, otherwise *false_expression*.

For example, *Y* = (*A* == *B*) ? *A* : *B*; // *Y* gets *A* if *A*==*B* is true.

wire [15:0] bus_a = drive_bus_a ? data ? 16′bz;

Here, *bus_a* gets *data* if *drive_bus_a* is 1, all *"z"* if *drive_bus_a* is 0 and "x" if *drive_bus_a* is "x".

Multiplexers can be easily implemented using nesting

```
wire [1:0] select;
wire [15:0] driver_1, driver_2, driver_3, driver_4;
wire [15:0] bus_a = (select == 2′b00) ? driver_1 :
                    (select == 2′b01) ? driver_2 :
                    (select == 2′b10) ? driver_3 :
                    (select == 2′b11) ? driver_4 :
                    16′bx;
```

Operators

Conditional Operator

Additional rules:

- Logic value "z" is **not** allowed in the *expression*.
- Zeros are automatically filled if operands lengths are different.
- If *expression* is ambiguous, then both *true_expression* and *false_expression* are evaluated and the result is computed on a bitwise basis using:

? :	**0**	**1**	**x**
0	0	x	x
1	x	1	x
x	x	x	x

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

Concatenation Operator

No operand may be an unsigned constant, otherwise the compiler would not be able to size the result.

Expressions and Operands

Operands may be any of these alone or in combination

- nets
- registers
- constants
- numbers
- bit-select of a net
- bit-select of a register
- part-select of a net
- part-select of a register
- memory element
- a function call

The result may be used in an assignment to a net or register or to effect a choice among alternatives.

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.

Precedence	Operator	Symbol
highest	Unary	+ - ! ~
	Mult, Div, Modulus	* / %
	Add, Sub	+ -
	Shift	<< >>
	Relational	< <= > >=
		== != === !==
	Reduction/Logical	& ~&, ^ ^~, | ~|
		&& ||
lowest	Conditional	? :

Use parentheses when precedence needs to be overridden.

Operators

Operator Precedence

Do these produce different results?

A < *SIZE* - 1 && *B* != *C* && *INDEX* != *LASTONE*

(*A* < *SIZE* - 1) && (*B* != *C*) && (*INDEX* != *LASTONE*)