3

Figure 3-0
Example 3-0
Syntax 3-0
Table 3-0

Data Types

The set of Verilog HDL data types is designed to represent the data storage and transmission elements found in digital hardware.

3.1 Value Set

The Verilog HDL value set consists of four basic values:

- `0` - represents a logic zero, or false condition
- `1` - represents a logic one, or true condition
- `x` - represents an unknown logic value
- `z` - represents a high-impedance state

The values `0` and `1` are logical complements of one another.

When the `z` value is present at the input of a gate, or when it is encountered in an expression, the effect is usually the same as an `x` value. Notable exceptions are the MOS primitives, which can pass the `z` value.

Almost all of the data types in the Verilog language store all four basic values. The exceptions are the `event` type, which has no storage, and the `trireg` net data type, which retains its first state when all of its drivers go to the high impedance value, and `z`. All bits of vectors can be independently set to one of the four basic values.

The language includes strength information in addition to the basic value information for scalar net variables. This is described in detail in Chapter 6, *Gate and Switch Level Modeling.*

3.2 Registers and Nets

There are two main groups of data types: the register data types and the net data types. These two groups differ in the way that they are assigned and hold values. They also represent different hardware structures.

3.2.1 Nets

The `net` data types represent physical connections between structural entities, such as gates. A `net` does not store a value (except for the `trireg` `net`, discussed in Section 3.7.3). Instead, it must be driven by a driver, such as a gate or a continuous assignment. See Chapter 6, *Gate and Switch Level Modeling*, and Chapter 5, *Assignments*, for definitions of these constructs. If no driver is connected to a `net`, its value will be high-impedance (`z`)—unless the `net` is a `trireg`.

3.2.2 Registers

A register is an abstraction of a data storage element. The keyword for the register data type is `reg`. A register stores a value from one assignment to the next. An assignment statement in a procedure acts as a trigger that changes the value in the data storage element. The Verilog language has powerful constructs that allow you to control when and if these assignment statements are executed. These control constructs are used to describe hardware trigger conditions, such as the rising edge of a clock, and decision-making logic, such as a multiplexer. Chapter 8, *Behavioral Modeling*, describes these control constructs.

The default initialization value for a `reg` data type is the unknown value, `x`.

CAUTION

Registers can be assigned negative values, but, when a register is an operand in an expression, its value is treated as an unsigned (positive) value. For example, a minus one in a four-bit register functions as the number 15 if the register is an expression operand. See Section 4.1.2 for more information on numeric conventions in expressions.

3.2.3 Declaration Syntax

The syntax for net and register declarations is as follows:

<net_declaration>
::= <NETTYPE> <expandrange>? <delay>? <list_of_variables> ;
||= trireg <charge_strength>? <expandrange>? <delay>? <list_of_variables> ;
||= <NETTYPE> <drive_strength>?
<expandrange>? <delay>? <list_of_assignments> ;

<reg_declaration>
::= reg <range>? <list_of_register_variables> ;

<list_of_variables>
::= <name_of_variable> <,<name_of_variable>>*

<name_of_variable>
::= <IDENTIFIER>

<list_of_register_variables>
::= <register_variable> <,<register_variable>>*

<register_variable>
::= <name_of_register>

<name_of_register>
::= <IDENTIFIER>

<expandrange>
::= <range>
||= scalared <range>
iff [the data type is not a trireg] the following syntax is available:
||= vectored <range>

<range>
::= [<constant_expression> : <constant_expression>]

<list_of_assignments>
::= <assignment> <,<assignment>>*

<charge_strength>
::= (<CAPACITOR_SIZE>)

<drive_strength>
::= (<STRENGTH0> , <STRENGTH1>)
||= (<STRENGTH1> , <STRENGTH0>)

Syntax 3-1: Syntax for <net_declaration>

<NETTYPE> is one of the following keywords:

```
wire tri tri1 supply0
wand triand tri0 supply1
wor trior trireg
```

<IDENTIFIER> is the name of the net that is being declared. See Chapter 2, *Lexical Conventions*, for a discussion of identifiers.

<delay> specifies the propagation delay of the net (as explained in Chapter 6, *Gate and Switch Level Modeling*), or, when associated with a <list_of_assignments>, it specifies the delay executed before the assignment (as explained in Chapter 5, *Assignments*).

<CAPACITOR_SIZE> is one of the following keywords:

```
small medium large
```

<STRENGTH0> is one of the following keywords:

```
supply0  strong0  pull0  weak0  highz0
```

<STRENGTH1> is one of the following keywords:

```
supply1  strong1  pull1  weak1  highz1
```

Syntax 3-2: Definitions for <net_declaration> syntax

3.2.4 Declaration Examples

The following are examples of register and net declarations:

```
reg a;              // a scalar register
wand w;             // a scalar net of type 'wand'
reg[3:0] v;         // a 4-bit vector register made up of
                    // (from most to least significant)
                    // v[3], v[2], v[1] and v[0]
tri [15:0] busa;    // a tri-state 16-bit bus
reg [1:4] b;        // a 4-bit vector register
trireg (small) storeit;     // a charge storage node
                            // of strength small
```

Example 3-1: Register and net declarations

If a set of nets or registers shares the same characteristics, they can be declared in the same declaration statement. The following is an example:

```
wire w1, w2;         // declares 2 wires
reg [4:0] x, y, z;  // declares 3 5-bit registers
```

3.3 Vectors

A net or reg declaration without a <range> specification is one bit wide; that is, it is scalar. Multiple bit net and reg data types are declared by specifying a <range>, and are known as vectors.

3.3.1 Specifying Vectors

The <range> specification gives addresses to the individual bits in a multi-bit net or register. The most significant bit (msb) is the left-hand value in the <range> and the least significant bit (lsb) is the right-hand value in the <range>.

The range is specified as follows:

```
[ msb_expr : lsb_expr ]
```

Both msb_expr and lsb_expr are non-negative constant expressions. There are no restrictions on the values of the indices. The msb and lsb expressions can be any value, and lsb_expr can be a greater value than msb_expr, if desired.

Vector nets and registers obey laws of arithmetic modulo 2 to the power *n*, where *n* is the number of bits in the vector. Vector nets and registers are treated as unsigned quantities.

3.3.2 Vector Net Accessibility

A vector can be used as a single entity or as a group of n scalars, where n is the number of bits in the vector. The keyword vectored allows you to specify that a vector can be modified *only* as an indivisible entity. The keyword scalared explicitly allows access to bit and parts. This is also the default case. The Verilog-XL process of accessing bits within a vector is known as vector *expansion*.

Only when a net is *not* specified as `vectored` can bit selects and part selects be driven by outputs of gates, primitives, and modules—or be on the left-hand side of continuous assignments. You cannot declare a `trireg` with the `vectored` keyword.

The following are examples of vector net declarations:

```
tri1 scalared [63:0] bus64;//a bus that will be expanded
tri vectored [31:0] data; //a bus that will not be expanded
```

Example 3-2: Vector net declarations

3.4 Strengths

There are two types of strengths that can be specified in a net declaration. They are as follows:

- **charge strength** used when declaring a net of type `trireg`
- **drive strength** used when placing a continuous assignment on a net in the same statement that declares the net

Gate declarations can also specify a drive strength. See Chapter 6, *Gate and Switch Level Modeling*, for more information on gates and for important information on strengths.

3.4.1 Charge Strength

The <charge_strength> specification can be used only with `trireg` nets. A `trireg` net is used to model charge storage; <charge_strength> specifies the relative size of the capacitance. The <CAPACITOR_SIZE> declaration is one of the following keywords:

- `small`
- `medium`
- `large`

When no size is specified in a `trireg` declaration, its size is `medium`.

The following is a syntax example of a strength declaration:

```
trireg (small) st1 ;
```

A `trireg` net can model a charge storage node whose charge decays over time. The simulation time of a charge decay is specified in the `trireg` net's delay specification (see Section 6.16.2).

3.4.2 Drive Strength

The <drive_strength> specification allows a continuous assignment to be placed on a net in the same statement that declares that net. See Chapter 5, *Assignments*, for more details.

Net strength properties are described in detail in Chapter 6, *Gate and Switch Level Modeling*.

3.5 Implicit Declarations

The syntax shown in Section 3.2.3, *Declaration Syntax*, is used to explicitly declare variables. In the absence of an explicit declaration of a variable, statements for gate, user-defined primitive, and module instantiations assume an *implicit* variable declaration. This happens if you do the following: in the terminal list of an instance of a gate, a user-defined primitive, or a module, specify a variable that has not been explicitly declared previously in one of the declaration statements of the instantiating module.

These implicitly declared variables are scalar nets of type wire.

3.6 Net Initialization

The default initialization value for a net is the value `z`. Nets with drivers assume the output value of their drivers, which defaults to `x`. The `trireg` net is an exception to these statements. The `trireg` defaults to the value `x`, with the strength specified in the net declaration (`small`, `medium`, or `large`).

3.7 Net Types

There are several distinct types of nets. Each is described in the sections that follow.

3.7.1 `wire` and `tri` Nets

The `wire` and `tri` nets connect elements. The net types `wire` and `tri` are identical in their syntax and functions; two names are provided so that the name of a net can indicate the purpose of the net in that model. A `wire` net is typically used for nets that are driven by a single gate or continuous assignment. The `tri` net type might be used where multiple drivers drive a net.

Logical conflicts from multiple sources on a `wire` or a `tri` net result in unknown values unless the net is controlled by logic strength.

Table 3-1 is a truth table for `wire` and `tri` nets. Note that it assumes equal strengths for both drivers. Please refer to Section 6.10 for a discussion of *logic strength modeling*.

wire/ tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Table 3-1: Truth table for `wire` and `tri` nets

3.7.2 Wired Nets

Wired nets are of type `wor`, `wand`, `trior`, and `triand`, and are used to model wired logic configurations. Wired nets resolve the conflicts that result when multiple drivers drive the same net. The `wor` and `trior` nets create wired `or` configurations, such that when any of the drivers is `1`, the net is `1`. The `wand` and `triand` nets create wired `and` configurations, such that if any driver is `0`, the net is `0`.

The net types `wor` and `trior` are identical in their syntax and functionality—as are the `wand` and `triand`. Table 3-2 gives the truth tables for wired nets. Note that it assumes equal strengths for both drivers. Please refer to Section 6.10 for a discussion of logic strength modeling.

wand/ triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

wor/ trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Table 3-2: Truth tables for `wand/triand` and `wor/trior` nets

3.7.3 `trireg` Net

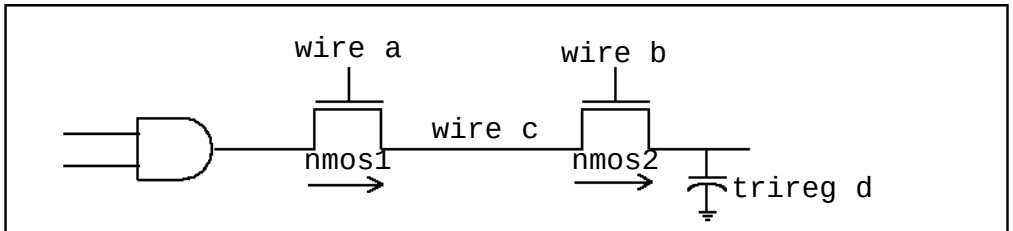
The `trireg` net stores a value and is used to model charge storage nodes. A `trireg` can be one of two states:

The Driven State — When at least one driver of a `trireg` has a value of `1`, `0`, or `x`, that value propagates into the `trireg` and is the `trireg`'s driven value.

Capacitive State — When all the drivers of a `trireg` net are at the high impedance value (`z`), the `trireg` net retains its last driven value; the high impedance value does not propagate from the driver to the `trireg`.

The strength of the value on the `trireg` net in the capacitive state is `small`, `medium`, or `large`, depending on the size specified in the declaration of the `trireg`. The strength of a `trireg` in the driven state is `strong`, `pull`, or `weak` depending on the strength of the driver. You cannot declare a `trireg` with the `vectored` keyword.

Figure 3-1 shows a schematic that includes the following items: a `trireg` net whose size is `medium`, its driver, and the simulation results.



simulation time	wire a	wire b	wire c	trireg d
0	1	1	strong 1	strong 1
10	0	1	HiZ	medium 1

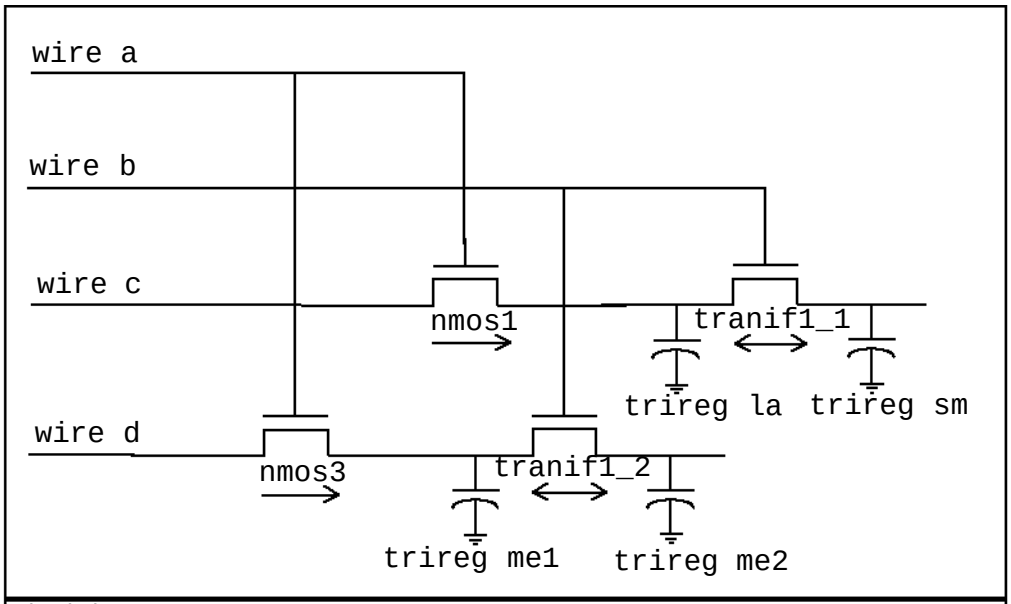
Figure 3-1: Simulation values of a `trireg` *and its driver*

Simulation of the design in Figure 3-1 reports the following results:

1. At simulation time 0, `wire a` and `wire b` have a value of `1`. A value of `1` with a `strong` strength propagates from the AND gate through the NMOS switches connected to each other by `wire c`, into `trireg d`.
2. At simulation time 10, `wire a` changes value to `0`, disconnecting `wire c` from the AND gate. When `wire c` is no longer connected to the AND gate, its value changes to `HiZ`. The `wire b`'s value remains `1` so `wire c` remains connected to `trireg d` through the NMOS2 switch. The `HiZ` value does not propagate from `wire c` into `trireg d`. Instead, `trireg d` enters the capacitive state, storing its last driven value of `1`. It stores the `1` with a `medium` strength.

Capacitive networks

A capacitive network is a connection between two or more `trireg`s. In a capacitive network whose `trireg`'s are in the capacitive state, logic and strength values can propagate between `trireg`s. Figure 3-2 shows a capacitive network in which the logic value of some `trireg`s change the logic value of other `trireg`s of equal or smaller size.



simulation time	wire a	wire b	wire c	wire d	trireg la	trireg sm	trireg me1	trireg me2
0	1	1	1	1	1	1	1	1
10	1	[0]	1	1	1	1	1	1
20	1	0	[0]	1	[0]	1	1	1
30	1	0	0	[0]	0	1	[0]	1
40	[0]	0	0	0	0	1	0	1
50	0	[1]	0	0	0	[0]	[X]	[X]

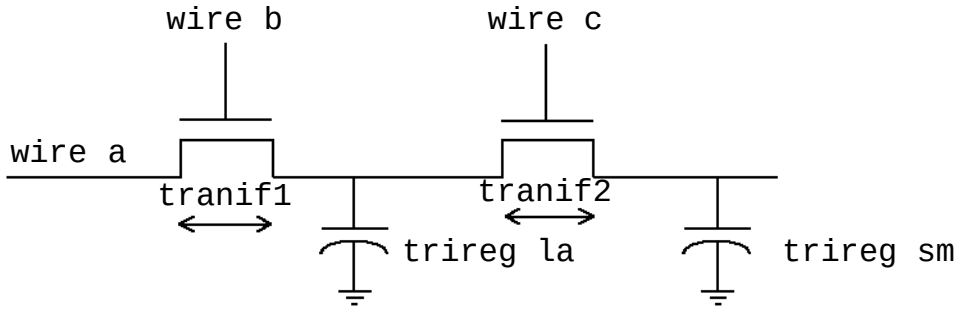
Figure 3-2: Simulation results of a capacitive network

In Figure 3-2, `trireg la`'s size is `large`, `triregs me1` and `me2` are size `medium`, and `trireg sm`'s size is `small`. Simulation reports the following sequence of events:

1. At simulation time 0, `wire a` and `wire b` have a value of `1`. The `wire c` drives a value of `1` into `triregs la` and `sm`, `wire d` drives a value of 1 into `triregs me1` and `me2`.
2. At simulation time 10, `wire b`'s value changes to `0`, disconnecting `trireg sm` and `me2` from their drivers. These `triregs` enter the capacitive state and store the value `1`, their last driven value.
3. At simulation time 20, `wire c` drives a value of `0` into `trireg la`.
4. At simulation time 30, `wire d` drives a value of `0` into `trireg me1`.
5. At simulation time 40, `wire a`'s value changes to `0`, disconnecting `trireg la` and `me1` from their drivers. These `triregs` enter the capacitive state and store the value `0`.

6. At simulation time 50, the `wire b`'s value changes to `1`. This change of value in `wire b` connects `trireg sm` to `trireg la`; these `trireg`s have different sizes and stored different values. This connection causes the smaller `trireg` to store the larger `trireg`'s value and `trireg sm` now stores a value of `0`.This change of value in `wire b` also connects `trireg me1` to `trireg me2`; these `trireg`s have the same size and stored different values. The connection causes both `trireg me1` and `me2` to change value to `x`.

In a capacitive network, charge strengths propagate from a larger `trireg` to a smaller `trireg`. Figure 3-3 shows a capacitive network and its simulation results.



simulation time	wire a	wire b	wire c	trireg la	trireg sm
0	strong 1	1	1	strong 1	strong 1
10	strong 1	0	1	large 1	large 1
20	strong 1	0	0	large 1	small 1
30	strong 1	0	1	large 1	large 1
40	strong 1	0	0	large 1	small 1

Figure 3-3: Simulation results of charge sharing

In Figure 3-3, `trireg la`'s size is `large` and `trireg sm`'s size is `small`. Simulation reports the following results:

1. At simulation time 0, the value of `wire a`, `b`, and `c` is `1` and `wire a` drives a `strong 1` into `trireg la` and `sm`.
2. At simulation time 10, `wire b`'s value changes to `0`, disconnecting `trireg la` and `sm` from `wire a`. The `trireg`s `la` and `sm` enter the capacitive state. Both `trireg`s share the `large` charge of `trireg la` because they remain connected through `tranif2`.
3. At simulation time 20, `wire c`'s value changes to `0`, disconnecting `trireg sm` from `trireg la`. The `trireg sm` no longer shares `trireg la`'s large charge and now stores a `small` charge.
4. At simulation time 30, `wire c`'s value changes to `1`, connecting the two `trireg`s. These `trireg`s now share the same charge.
5. At simulation time 40, `wire c`'s value changes again to `0`, disconnecting `trireg sm` from `trireg la`. Once again, `trireg sm` no longer shares `trireg la`'s large charge and now stores a `small` charge.

Ideal capacitive state and charge decay

A `trireg` net can retain its value indefinitely or its charge can decay over time. The simulation time of charge decay is specified in the `trireg` net's delay specification.

3.7.4 `tri0` and `tri1` Nets

The `tri0` and `tri1` nets model nets with resistive `pulldown` and resistive `pullup` devices on them. When no driver drives a `tri0` net, its value is `0`. When no driver drives a `tri1` net, its value is `1`. The strength of this value is `pull`. See Chapter 6, *Gate and Switch Level Modeling*, for a description of strength modeling.

3.7.5 `supply` Nets

The `supply0` and `supply1` nets model the power supplies in a circuit. The `supply0` nets are used to model Vss (ground) and `supply1` nets are used to model Vdd or Vcc (power). These nets should never be connected to the output of a gate or continuous assignment, because the strength they possess will override the driver. They have `supply0` or `supply1` strengths.

3.8 Memories

The Verilog HDL models memories as an array of register variables. These arrays can be used to model read-only memories (ROMs), random access memories (RAMs), and register files. Each register in the array is known as an *element* or *word* and is addressed by a single array index. There are no multiple dimension arrays in the Verilog Language.

Memories are declared in register declaration statements by specifying the element address range after the declared identifier. Syntax 3-3 gives the syntax for a register declaration statement. Note that this syntax extends the <register_variable> definition given in Section 3.2.3, *Declaration Syntax*.

<register_variable>
::= <name_of_register>
| |= <name_of_memory> [<constant_expression> : <constant_expression>]

<constant_expression>
::=<expression>

<name_of_memory>
::= <IDENTIFIER>

Syntax 3-3: Syntax for <register_variable>

The following example illustrates a memory declaration:

```
reg[7:0] mema[0:255];
```

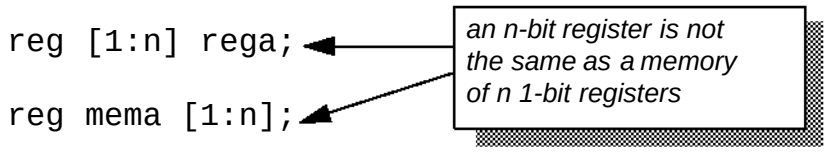
This example declares a memory called `mema` consisting of 256 eight-bit registers. The indices are `0` through `255`. The expressions that specify the indices of the array must be constant expressions.

Note that within the same declaration statement both registers and memories can be declared. This makes it convenient to declare both a memory and some registers that will hold data to be read from and written to the memory in the same declaration statement, as in Example 3-3.

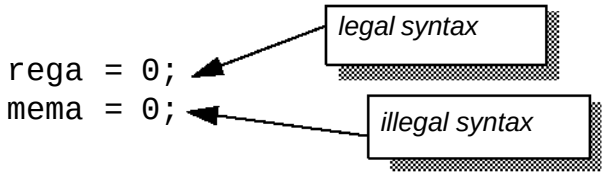
```
parameter                // parameters are run-time
                         // constants - see Section 3.11
wordsize = 16,
memsize = 256;

// Declare 256 words of 16-bit memory plus two registers
reg [wordsize-1:0]       // equivalent to [15:0]
    mem [memsize-1:0],   // equivalent to [255:0]
    writereg,
    readreg;
```

Example 3-3: Declaring memory

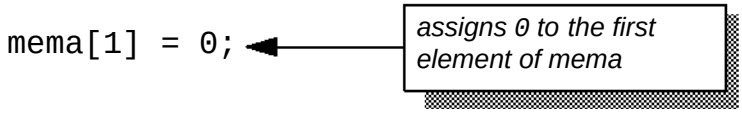
Note that a memory of *n* 1-bit registers is different from an *n*-bit vector register, as shown in the following example:



An *n*-bit register can be assigned a value in a single assignment, but a complete memory cannot; thus the following assignment to `rega` is legal and the succeeding assignment that attempts to clear all of the memory `mema` is illegal, as shown in the following example:



To assign a value to a memory element, an index must be specified, as shown in the following example:



The index can be an expression. This option allows you to reference different memory elements, depending on the value of other registers and nets in the circuit. For example, a program counter register could be used to index into a RAM.

3.9 Integers and Times

In addition to modeling hardware, there are other uses for variables in an HDL model. Although you can use the `reg` variables for general purposes such as counting the number of times a particular net changes value, the `integer` and `time` register data types are provided for convenience and to make the description more self-documenting.

The syntax for declaring `integer` and `time` variables is as follows:

<time_declaration>
::= time <list_of_register_variables> ;

<integer_declaration>
::= integer <list_of_register_variables> ;

Syntax 3-4: Syntax for time and integer declarations

The <list_of_register_variables> item is defined in Section 3.2.3, *Declaration Syntax*.

A `time` variable is used for storing and manipulating simulation time quantities in situations where timing checks are required and for diagnostics and debugging purposes. This data type is typically used in conjunction with the `$time` system function. The size of a `time` variable is 64 bits.

An `integer` is a general purpose variable used for manipulating quantities that are not regarded as hardware registers. The size of an `integer` variable is 32 bits.

Arrays of `integer` and `time` variables are allowed. They are declared in the same manner as arrays of `reg` variables, as in the following example:

```
integer a[1:64]; // an array of 64 integers
time change_history[1:1000]; // an array of 1000 times
```

The `integer` and `time` variables are assigned values in the same manner as `reg` variables. Procedural assignments are used to trigger their value changes.

`Time` variables behave the same as 64 bit `reg` variables. They are unsigned quantities, and unsigned arithmetic is performed on them. In contrast, `integer` variables are signed quantities. Arithmetic operations performed on `integer` variables produce 2's complement results.

3.10 Real Numbers

The Verilog HDL supports real number constants and variables in addition to integers and time variables. The syntax for real numbers is the same as the syntax for register types, and is described in Section 3.10.1. Except for the following restrictions, real number variables can be used in the same places that integers and time variables are used.

- Not all Verilog HDL operators can be used with real number values. See the tables in Section 4.1 for lists of valid and invalid operators for real numbers.
- Ranges are not allowed on real number variable declarations.
- Real number variables default to an initial value of zero.

3.10.1 Declaration Syntax for Real Numbers

The syntax for declaring real number variables is as follows:

```
<real_declaration>
        ::=real<list_of_variables>;
```

Syntax 3-5: Syntax for real number variable declarations

The <list_of_variables> item is defined in Section 3.2.3, *Declaration Syntax*.

3.10.2 Specifying Real Numbers

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the 8th power). Real numbers expressed with a decimal point must have at least one digit on each side of the decimal point.

The following are some examples of valid real numbers in the Verilog language:

```
1.2
0.1
2394.26331
1.2E12 (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12  (underscores are ignored)
```

The following are invalid real numbers in the Verilog HDL because they do not have a digit to the left of the decimal point:

```
.12
.3E3
.2e-7
```

3.10.3 Operators and Real Numbers

The result of using logical or relational operators on real numbers is a single-bit scalar value. Not all Verilog operators can be used with real number expressions. Table 4-2 in Section 4.1 lists the valid operators for use with real numbers. Real number constants and real number variables are also prohibited in the following contexts:

- edge descriptors (posedge, negedge) applied to real number variables
- bit-select or part-select references of variables declared as `real`
- real number index expressions of bit-select or part-select references of vectors
- real number memories (arrays of real numbers)

3.10.4 Conversion

The Verilog language converts real numbers to integers by rounding a real number to the nearest integer, rather than by truncating it. For example, the real numbers 35.7 and 35.5 both become 36 when converted to an integer, and 35.2 becomes 35. Implicit conversion takes place when you assign a real to an integer.

3.11 Parameters

Verilog parameters do not belong to either the register or the net group. Parameters are not variables, they are constants. The syntax for parameter declarations is as follows:

<parameter_declaration>
::= parameter <list_of_assignments> ;

Syntax 3-6: Syntax for <parameter_declaration>

The <list_of_assignments> is a comma-separated list of assignments, where the right-hand side of the assignment must be a constant expression, that is, an expression containing only constant numbers and previously defined parameters. Example 3-4 shows examples of parameter declarations:

```
parameter msb = 7; // defines msb as a constant value 7

parameter e = 25, f = 9;// defines two constant numbers

parameter average_delay = (r + f) / 2;

parameter byte_size = 8, byte_mask = byte_size - 1;

parameter  r = 5.7;     //declares r as a 'real'
                        // parameter
```

Example 3-4: Parameter declarations

Even though they represent constants, Verilog parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows you to customize module instances. You can modify the parameter with the `defparam` statement, or you can modify the parameter in the module instance statement. Typical uses of parameters are to specify delays and width of variables. See Chapter 12, *Hierarchical Structures*, for complete details on parameter value assignment.