2 DATA TYPES IN VERILOG

2.1 Overview

Verilog supports only predefined data types. These include bits, bit-vectors, memories, integers, reals, events, and strength types. These define the domain of description in Verilog. Verilog deals mainly in the domain of bits and bytes while describing the circuits. The real type is useful for delays and time and is also useful in higher level modeling such as stochastic analysis and digital signal processing algorithms. The hardware types include net and reg types. This, in general, can be seen as wires and registers. The nets are further declared to be of different types like tri-stated or non-tri-stated and whether the resolution of multiple connections results in anding, oring or uses prior value. The details of these can be seen in the following sections.

2.2 Value Systems

The data types have ramifications on the scope of the description. The value-systems define the kinds of values defined in the language and entail the operations supported on them. These also have corresponding constant (numbers or literals) definitions. The various values in Verilog are:

bits and integers(32 bits), time (64 bits)—bit-vectors and integers can be freely intermixed. Integers are defined to be 32 bits. Time values are 64 bits. The bits actually are of two types as below.

4-state values (0,1,x,z); also known as logic values

128-state types (4 states and 64 strengths (8 '0' and 8 '1' strengths)

floating point types (real numbers)

character strings

delay values – These are single, double, triplet or n-tuple indicating rise, fall any other transition delay

transition values – (01) - change from 0 to 1. These may in user-define primitives or specify blocks

Boolean/conditional values – true /false OR 0/1

units (only for timescale) – femtoseconds (Fs) to seconds (s)

2.3 Data Declarations

2.3.1 Introduction

Different data types in Verilog are declared by data declaration statement. These statements occur in module definitions before usage and some of these can be declared within named sequential blocks. In addition to the value-types that may distinguish different types of data, the hardware characteristics of wires versus registers also are distinguished as net versus reg declarations in Verilog. The term driving is used in hardware descriptions to describe how a value is assigned to a date element. Nets and regs are two main types of data elements in Verilog. Nets are continuously driven from continuous assignments or from structural elements such as module ports, gates, transistors or user defined primitives. Regs are driven strictly from behavioral blocks. Nets are typically implemented as wires in hardware and regs may either be wires, or temporaries or flip-flops (registers).

The different data types in Verilog are declared as one of the following types:

parameter	These are constant valued expressions resolved after compilation and allow modules to be parametrized.
input **output** **inout**	This and the next two types define the direction and size of a port.
net	This is a type of connection or wire in hardware with different resolutions.
reg	This is an abstract type that is like a register and is driven behaviorally.
time	This contains time quantities like delays and simulation time.
integer	This is an integer values type.
real	This is a floating point or real valued type.
event	This indicates a flag that can trigger activity.

These can all be declared at the module level. The other descriptions in Verilog with scope-creation capabilities include tasks, functions and named begin-end blocks.. The non-module scopes are all behavioral scopes. Nets are non-behaviorally driven and thus can not be declared in the other scopes. All other types can be present in tasks and begin-end blocks Each one of these is explained in sections 2.4 through 2.12.

2.3.2 Examples

```
input i1, i2;
        reg [63:0] data;
        time simtime;
```

Example 2-1. Data declarations.

The first line in Example 1-2 is an input declaration line, the second is a 64-bit reg declaration for data. The last line in this example is a time declaration for the variable named simtime.

2.3.3 Syntax

```
data_declarations  ::= parameter_declaration
                   ||= input_declaration
                   ||= output_declaration
                   ||= inout_declaration
                   ||= net_declaration
                   ||= reg_declaration
                   ||= time_declaration
                   ||= integer_declaration
                   ||= real_declaration
                   ||= event_declaration
```

2.4 Reg Declaration

2.4.1 Introduction

Reg declarations are done for all signals that are driven from the behavioral descriptions Regs retain a given value until they are assigned a new value in the sequential description (initial or always blocks). Reg types are more abstract than net types but are closely tied to concepts of registers (with storage) and can be realized in hardware as such. They can also be realized as wires or may be temporaries with no hardware mapping depending on their usage within a behavioral block.

2.4.2 Examples

```
reg rl, r2;
reg [63:0] data_a, data_b, data_c;
```

Example 2-2. Reg declarations.

2.4.3 Syntax

```
reg_declaration
    ::= reg [range] list_of_register_ identifiers;
list_of_ register_ identifiers ::= register _ identifier , { register _ identifier }
```

2.5 Net Declaration

2.5.1 Introduction

The net is a set of data types in a Verilog description that represents the physical wires in a circuit. A net connects gate-level instantiations, module instantiations and continuous assignments. The Verilog language allows you to read a value from a net from within behavioral descriptions, but you cannot assign a value to a net within the behavioral descriptions. (An always block is a specific type of begin...end block). A net does not store its value. It must be driven in one of two ways:

- By connecting the net to the output of a gate or module.
- By assigning a value to the net in a continuous assignment.

Different net-types defined in Verilog are described below and the tables in Figure 2-1 summarize their functionality. Resolution is a rule for resolving values with multiple drivers the following is also explained in table on next page.

wire a net with 0,1,x values and resolution based on equality

wand a net with 0,1,x values and resolution of wired and

wor a net with 0,1,x values and resolution of wired or

tri a net with values of 0,1,x,z and resolution of tri-state bus

tri0 a net with values of 0,1,x,z and resolution of tri-state bus and a default value of 0 when no driving value

tri1 a net with values of 0,1,x,z and resolution of tri-state bus and a default value of 1 when no driving value

trior a net with values of 0,1,x,z and resolution of tri-state for z-non-z values using ‘or’ function of non-z values

triand a net with values of 0,1,x,z and resolution of tri-state for z-non-z value using ‘and’ function for non-z values

trireg a net with values of 0,1,z,x and tri-state resolution together with charge storage (previous value used in resolving new value)

supply0, supply1 (gnd and vdd)

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

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

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

tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	1

tri0	0	1	x	z
0	0	0	0	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0

trireg	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	P

Figure 2-1. Tables of Net Types and Resolution Functions

wire

The wire data-type defines a type that is a simple connection between two places where it is used. Multiple drivers resolve using tri-state function during simulation but synthesis creates non-tri-states for wire types. In the Example 2-3, 2-wire declarations are made. The first declares a scalar wire w. The second declares a vector wire w2 with 3 bits. Its most significant bit (msb) has an index of 2 and its least significant bit (Isb) has an index of 0.

```
wire w1;
wire [2:0] w2;
```

Example 2-3. Wire declarations.

wand

The wand (wired-and) data type is a specific type of wire that uses the and function to find the resulting value when multiple drivers are present. In Example 2-4, two variables drive the variable c. The value of out is determined by the logical and of i1 and i2.

```
module wand_test(out, i1, i2);
input i1, i2;
output out;
wand out;
assign out = i1;
assign out = i2;
endmodule
```

Example 2-4. Wand (wired-AND) declarations and usage.

You can assign a delay value in a wand declaration, and you can use the Verilog keywords scalared and vectored for simulation.

wor

The wor (wired-OR) data type is a specific type of wire. In Example 2-5, two variables drive the variable c. The value of out is determined by the logical OR of in1 and in2.

```
module wor_test(il, i2, out);
input il, i2;
output out;
wor out;
assign out  = in1;
assign out = in2;
endmodule
```

Example 2-5. wor (wired-OR) declarations and usage.

tri

The tri (three-state) data type is a specific type of wire where the resolution of multiple drivers is done using the rules of tri-state bus. All variables that drive the tri must have a value of Z (high-impedance), except one. This single variable determines the value of the tri.

In Example 2-6, three variables drive the variable out. They are set in another module such that only one driver is active at a time.

```
module tri_test (out,  select, a, b, c);
      input [1:0] select, a, b, c;
      output out;
      tri out;

      assign out = a;        //make the tri connection
      assign out = b;
      assign out = c;
endmodule

module abc(a, b, c, select)
      output a, b, c;
      input select;
      always @ ( select ) begin
      a = 1'bz;            //set all variables to Z
      b = 1'bz;
      c = 1'bz;
      case ( select )   //set only one variable to non-Z
      2'b00:a=1'b1;
      2'b01:b=l'b0;
      2'b10:c=l'b1;
```

```
        endcase
        end
    endmodule
```

Example 2-6. Tri (Three-State) declarations and usage.

The drivers may be gate outputs as in the example below.

```
module tri_test (out,  select, a, b, c);
        input [1:0] select, a, b, c;
        output out;
        tri out;

        nand(out,     a1,a2);     //make the tri connectio
        nand(out, b1, b2)
        nand( out, c1, c2, c3);
endmodule
```

Example 2-7. Nets of tri types declared and used for multiple drivers with tri-state resolution.

supply0 / supply1

The supply0 and supply1 data types define wires tied to logic 0 (ground) and logic 1 (power or vss/vdd). Using supply0 and supply1 is the same as declaring a wire and assigning a 0 or a 1 to it. In Example 2-7 power is tied to power supply (always logic 1 – overriding strength) and gnd is tied to ground (always logic 0 – overriding strength).

```
        supply0 gnd;
        supply1 power;
```

Example 2-8. supply0 and supply1 constructs.

trireg

The trireg net is like a tri wire but has a capacitive nature and stores its last value when all drivers are tri-stated. Thus, the trireg net will always have a value of 0 or 1 or x but not z. The capacitance on the trireg is specified by a size that is either large, medium or small with the default value of medium when unspecified. For a Verilog model as follows, we obtain the resulting value on wire trg of trireg type when the transistor driving it is in off state.

```
module m;
        reg c0, c1, i1, i2
        tri d0, d1, d2;
        trireg d;
        and(d0, i1, i2);
        nmos  n1  (d1,  d0,  c0),   n2(d, d1, c1);
```

```
        initial
        begin

                $monitor("time = %d d = %d      c0=%d  c1=%d  d0=%d  d1=%d
                i1=%d i2=%d", $time, d, c0, c1, d0, d1, i1, i2);
                    #1
                i1 = 1;
                i2 = 1;
                c0 = 1;
                c1 = 1;
                #5
                c0 = 0;

            end

    endmodule
```

```
C1>.
time =          0 d = x      c0=x  c1=x  d0=x  d1=x  i1=x  i2=x
time =          1 d = 1      c1=1  c1=1  d0=1  d1=1  i1=1  i2=1
time =          6d = 1       c0=0  c1=1  d0=1  d1=z  i1=1  i2=1
```

Example 2-9. Trireg net and the switch-level modeling example.

In this example, if the trireg is replaced with a tri, then the value of d at time 6 units will be 'z'.

2.5.2 Syntax

net_declaration ::= NETTYPE [expandrange] [delay] list_of_net_ identifiers;
| **trireg** [charge_strength] [expandrange] [delay]
list_of_ net_ identifiers; | NETTYPE [drive_strength] [expandrange] [delay]
list_of_net_decl_assignments ;
list_of_net_ identifiers ::= *net_* identifier, { *net_* identifier }
NETTYPE ::= : **wire** | **tri** | **tri1** | **supply0** | **wand** | **triand** | **tri0** | **supply1** | **wor** | **trior** | **trireg**

expandrange
::= range
| **scalared** range
| **vectored** range

charge_strength
::= (small)
| (medium)
| (large)

In the above syntax definitions expandrange optionally defines vectors and is explained in section 2.7.2, delay expressions are defined in section 2.12,

drive_strength specifications are given while giving transistor-level models and are explained in Chapter 17.

2.5.3 Examples

```
wire w1, w2;

tri [7:0] t1, t2;

trireg large trg1, trg2;

triand [63:0] #(10:5) trnd1;
```

Example 2-10. Net type declarations.

In Example 2-10, the first line with the keyword 'wire' declares w1 and w2 to be single-bit or scalar wires. The second line declares 8-bit-vector-wires of type tri with names of t1 and t2. The trireg (a capacitive net) with the size large are declared on the next line with names of trg1 and trg2. The 64-bit triand type net with minimum and typical delays is declared on the 4^{th} line above.

2.6 Port Types

2.6.1 Introduction

You must explicitly declare the direction (whether input, output, or bidirectional) of each port that appears in the port list of a port definition. Three kinds of ports are defined in Verilog—input, output, inout. The ports must be either nets or regs. The only place where a reg can occur is output port. Constants and expressions are allowed in port declarations.

input

All input ports of a module are declared with an input statement. An input is a type of wire and is governed by the syntax of wire. You can use a range specification to declare an input that is a vector of signals, as for input b in the following example. The input statements can appear in any order in the description but must be declared before they are used. For example:

```
input a;
input [2:0] b;
```

output

All output ports of a module are declared with an output statement. Unless otherwise defined by a reg, wand, wor, or tri declaration, an output is a type of wire and is governed by the syntax of wire. An output statement can appear in any order in the description, but you must declare it before you use it. You can use a range specification to declare an output that is a vector of signals. If you use a reg

declaration for an output, the reg must have the same range as the vector of signals. For example:

```
output a;
output [2:0]b;
reg [2:0] b;
```

inout

You can declare bidirectional ports with the inout statement. An inout is a type of wire and is governed by the syntax of wire. You must declare an inout before you use it. For example:

```
inout a;
inout [2:0]b;
```

2.6.2 Examples

```
module fullAdder(cOut, sum, aIn, bIn, cIn);
        input aIn, bIn, cIn;
        output cOut, sum;
        wire aIn, bIn, cin;
        reg cOut, sum;
         .........
endmodule
```

Example 2-11. Port type declarations.

2.6.3 Syntax

```
list_of_ports
    ::= ( port {,port })

port
    ::= [port_expression]
    | . port_identifier ( [port_expression] )

port_expression
    ::= port_reference
    | { port_reference ,port_reference }

port_reference
    ::= port_identifier
    | port_identifier[ constant_expression ]
    | port_identifier [ msb_constant_expression :lsb_constant_expression ]
```

2.7 Aggregates – 1 and 2 Dimensional Arrays (Vectors and Memories)

2.7.1 Introduction

Only 1- and 2-dimensional arrays are supported in Verilog. One-dimensional arrays are called bit-vectors and may be nets or regs. The 2-dimensional aggregates are called memories and are reg kind. You can define an optional range for all the data types presented in this chapter. The range provides a means for creating a bit-vector. The syntax for a range specification is [msb : lsb]. Expressions for msb (most significant bit) and lsb (least significant bit) must be non-negative constant-valued expressions. Constant-valued expressions are composed only of constants, Verilog parameters, and operators. There is no maximum size for the length of a bit-vector and this is limited only by a particular implementation of simulation, synthesis, or other tools used with the Verilog source.

2.7.2 Examples

```
reg [31:0] data;
reg [23:0] addr;
wire [63:0] system_bus;
```

Example 2-12. Aggregate declarations.

In the first line above, data is a bit-vector reg of 32 bits width, declared with a range specification of 31:0 with the most significant bit taking the index of 31 and the least significant bit that of 0. The third line is a wire declaration of 64 bits width and the index of bits ranging from 63 to 0. Some examples are:

```
wire vectored [31:0] bus1;
tri scalared [63:0] bus2;
```

Example 2-13. Vectored and scalared bit-vector net declarations

In the above example, the first line indicates that the vector bus1 is used as a vector in connecting ports and can be maintained as a vector while on the second line the term scalared indicates usage of the vector bit-selects bus2[n] in connecting on port boundaries and the word ‘scalared’ indicates to the tools of such usage. These directives of vectored and scalared are only for efficient simulation or synthesis and do not change the functionality of the design.

```
reg [31:0] memory_bank1 [0: (1024*64)-1];
```

Example 2-14. Memory declarations.

In the above example, memory_bank is declared to be a memory of 64K locations of 32 bits each. Elements of memory are reg and are declared and used as such and the memory is addressable in terms of the entire 32 bits (in terms of the word-size).

2.7.3 Syntax

Vectored Net and Reg Declarations

This syntax is covered in net declaration in section 2.5.2.

Memory Declarations

memory_variable
::= memory_identifier [constant_expression : constant_expression]

Special Facility for Vectored Nets

expandrange
::= range
| **scalared** range
| **vectored** range

2.8 Delays on Nets

2.8.1 Introduction

Nets may have delays associated with them as declared in the declarations. These delays are the delays from the time a driver on a net changes to the time of actual change on the net. Delays are given by the number or the expression following the '#' symbol. These can be constants, parameters, expressions of these or even dynamic expressions using other variables. Delays can be rise, fall, or hold(change to z) delays and each of these delays in turn may have three values—minimum, typical and maximum. The rise, fall and hold delay specifications are separated by commas and the min-typ-max specifications are separated by colons. The rise delay includes delays when values change from 0 to 1, 0 to x and x to 1. Fall delay applies to changes from 1 to 0, 1 to x and x to 0. The hold delay values apply for changes from 0 to z, 1 to z and x to z changes. The same concepts of delays are useful for gates, transistors, user-defined primitive instances and behavioral descriptions.

2.8.2 Examples

```
tri #5 t1, t2;
wire #(10,9,8) w1, w2;
wand #(10:8:6, 9:8:6) w3;
```

Example 2-15. Net declarations with delay specifications.

In the first example above, t1 and t2 have rise, fall and hold delays of five time units. In the second line, wires w1 and w2 have three different values for the three changes—ten for rise, nine for fall, and eight for hold. The rise delay includes delays when values change from 0 to 1,0 to x and x to 1. Fall delay applies to changes from 1 to 0, 1 to x and x to 0. The hold delay values apply for changes from 0 to z, 1 to z and x to z changes.

2.8.3 Syntax

delay ::= delay2 | delay3
delay3 ::= #delay_value | #(delay_value [,delay_value [,delay_value]])
delay2 ::= #delay_value | #(delay_value [,delay_value])
delay_value
::= unsigned_number
| parameter_identifier
| (mintypmax_expression [,mintypmax_expression] [,mintypmax_expression])

2.9 Integer and Time

2.9.1 Introduction

The type time is 64-bit wide and contains result of system task $time or computation on other time variables. The type integer is 32 bit and can be assigned and used freely as integer or 32-bit (signed) register in expressions,

2.9.2 Examples

```
time t1, t2;
integer i1, i2;
```

Example 2-16. Integer and time declarations.

2.9.3 Syntax

time_declaration
::= time list_of_register_ identifiers;
integer_declaration
::= integer list_of_register_ identifiers;

2.10 Real Declaration

2.10.1 Introduction

Real numbers are used in Verilog to perform delay descriptions and in abstract algorithms. Their bit representation is done using IEEE floating point standard.

2.10.2 Example

```
real r1, r2;
```

Example 2-17. Real declarations.

2.10.3 Syntax

real_declaration
::= real list_of_ real_identifiers;

list_of_ real_identifiers
::= real_ identifier, {real_ identifier}

2.11 Event Declaration

Event types are special flags that can trigger activity into a process waiting for an event to occur.

Example and syntax are :

event e1;
event_declaration ::= **event** list_of_event_variables;

2.12 Parameter Declarations

You can use a parameter wherever a number is allowed, and you can define a parameter anywhere within a module definition. However, the Verilog language requires that you define the parameter before you use it. Example 2-17 shows two parameter declarations. Parameters TRUE and FALSE are unsized, and have values of 1 and 0, respectively. Parameters SO, S1, S2, and S3 have values 3, 1, 0, and 2, respectively, and are stored as 2-bit quantities. Parameters are a way of defining modules which can be configured at the instance time. The module instantitation statement lets one change the values of parameters in a module to new values for each instance. Separately, the ‘defparam’ statement also allows one to change the parameters from outside. This facility is extensively used for operations such as delay back-annotations from a post-layout circuit into a pre-layout or rtl design.

The parameter data type is automatically deduced by Veriolog compiler using the expression on the right-hand side of '=' in parameter declarations. These expressions may be of any valid data-type, like bits, vectors, integer, or reals in Verilog.

2.13 Examples

Following are scalar and vector parameter declarations:

parameter TRUE=1, FALSE=0;
parameter [1:0] S0=3, S1=1, S2=0, S3=2;

Example 2-18. Parameter declaration examples.

2.14 Syntax

parameter_declaration
::= parameter list_of_param_assignments;

```
list_of_param_assignments
::=param_assignment {,param_assignment}

param_assignment
::=identifier = constant_expression
```

2.15 Hierarchical Names

2.15.1 Introduction

Verilog supports names imported from other modules or other scopes in all places a simple name can occur. These are full hierarchical names as follows: Typically these are used in back-annotations or forward-annotations from outside the current design description. SDF files described in chapter 18, may, for example, be used to generate delay files using full-hierarchical names. These are also very useful for debugging as seen in Chapter 10. The usage of these for circuit description should be done with discretion since one is creating connections across module boundaries without putting the connecting wires or regs on the port list.

2.15.2 Examples

```
testbench.top_system.cpu.alu.fulladder.cin
testbench.top_system.cpu.reg11
```

Example 2-19. Hierarchical names.

The name begins with a top-level module-name testbench and then is followed by instance names until the level at which the name is defined is reached. In the first line of Example 2-19, the testbench instantiates the system with instance name of top_system which in turn instantiates cpu with instance name cpu and then alu followed by instance fulladder of an adder and the cin net.

2.15.3 Syntax

```
Hier_name::= name_of_module *.name_of_item
```

2.16 Exercises

1. Write a declaration for a wire and bus of 64 bits wide and a rise delay of 10 and fall delay of 8 time units.
2. Write memory declarations for a 64K x 8-bit memory.
3. Check the correctness of the following declarations:

```
net n1, n2;
reg[63:0] r1,r2,r3;
reg [0:-5] r;
events e1, e2;
```

4. Write a declaration of a tri-state wire with charge storage of medium capacitance.

5. In Example 2-7, change the net-type to triand and trior and trireg from tri and then compute the expected results. Simulate and verify the results [Use $showvars or driver value display to check component and computed values].