Verilog/SystemVerilog

Ryan Hou

2025-03-12

Table of contents

Preface

My notes on ???.

Resources

Some relevant resources:

- Resource Name

Textbooks:

- Book 1

1 Introduction

1.1 Perspective

> Note 1: Definition - Some definition
>
> **Term** is defined as blah blah blah...

This note does ...

1.2 High Level Ideas

2 Verilog - Data Types

2.1 Data Types

Almost all data types can only have one of four different values (0, 1, X, Z) except for `real` and `event` types.

(Note difference between logic and bit in SystemVerilog, where bit is a two-state variable)

`wire`

`reg`

Non-synthesizable types:

`integer` - general purpose 32bit int. Not synthesizable, good for loop counters, simulation tasks, etc.

`time` - unsigned 64b for storing time quantities

`realtime` - stores time as floating point

`real` - float

Strings are stored in `reg`, using 1 byte per char

2.2 Scalar/Vector

Note that part selection is inclusive from [high:low]

2.3 Arrays

```
reg        y1 [11:0];        // y is an scalar reg array of depth=12, each 1-bit wide
wire [0:7] y2 [3:0]          // y is an 8-bit vector net with a depth of 4
reg  [7:0] y3 [0:1][0:3];    // y is a 2D array rows=2,cols=4 each 8-bit wide
```

2.4 Net Types & Strength

2.5 References

- ChipVerify - Verilog

3 Verilog - Building Blocks

3.1 Modules

3.2 Assign

3.3 Operators

```
// Arithmetic Operators
a + b
a - b
a * b
a / b
a % b
a ** b

// Relational Operators

// Equality Operators
a === b // a equal to b, including x and z
a !== b // a not equal to b, including x and z
a == b  // a equal to b, result can be unknown
a != b  // a not equal to b, result can be unknown

// Logical Operators


// Bitwise


// Shift
```

3.4 Concatenation

Concatenation

Replication

Sign extension

3.5 Always Block

3.6 Initial Block

There are mainly two types of procedural blocks in Verilog - initial and always

initial block is not synthesizable

`$finish`

3.7 Generate

4 Verilog - Behavioral Modeling

4.1 Blocking / Non-Blocking

4.2 Functions & Tasks

4.3 Delays

Inter-assignment Delays:

```
// Delay is specified on the left side
#<delay> <LHS> = <RHS>
```

Intra-assignment Delays:

```
// Delay is specified on the right side
<LHS> = #<delay> <RHS>
```

```
    // Inter-assignment delay: Wait for #5 time units
    // and then assign a and c to 1. Note that 'a' and 'c'
    // gets updated at the end of current timestep
    #5  a <= 1;
        c <= 1;

    // Intra-assignment delay: First execute the statement
    // then wait for 5 time units and then assign the evaluated
    // value to q
    q <= #5 a & b | c;
```

5

6

7 SystemVerilog - Data Types

logic

bit

byte

int

string

7.0.1 Enumerations

```
enum          {RED, YELLOW, GREEN}         light_1;         // int type; RED = 0, YELLOW = 1
enum bit[1:0] {RED, YELLOW, GREEN}         light_2;         // bit type; RED = 0, YELLOW = 1

// A custom data-type can be created so that the same data-type may be used to declare other

typedef enum {TRUE, FALSE} e_true_false;
e_true_false  answer;
answer = TRUE;
```

7.1 Arrays

7.1.1 Static Array

7.1.2 Dynamic Array

```
[data_type] [identifier_name]  [];

bit [7:0]   stack [];       // A dynamic array of 8-bit vector
string      names [];       // A dynamic array that can contain strings
```

```
int     array [];

initial
array = new [3];
// This creates one more slot in the array, while keeping old contents
array = new [array.size() + 1] (array);
```

Methods:

size()

delete()

7.1.3 Associative Array

```
int         m_data [int];           // Key is of type int, and data is also of type int
int         m_name [string];        // Key is of type string, and data is of type int

m_name ["Rachel"] = 30;
m_name ["Orange"] = 2;

m_data [32'h123] = 3333;
```

7.1.4 Queue

```
int     m_queue [$];        // Unbound queue, no size


m_queue.push_back(23);      // Push into the queue

int data = m_queue.pop_front(); // Pop from the queue
```

7.1.5 Packed Array

There are two types of arrays: packed and unpacked

A packed array is guaranteed to be represented as a contiguous set of bits.

```
bit [3:0]   data;           // Packed array or vector
logic       queue [9:0];    // Unpacked array
```

Example of multi-dimensional packed array:

```
bit [2:0][3:0][7:0]     m_data;     // An MDA, 12 bytes

initial begin
    // 1. Assign a value to the MDA
    m_data[0] = 32'hface_cafe;
    m_data[1] = 32'h1234_5678;
    m_data[2] = 32'hc0de_fade;
end
```

7.1.6 Unpacked Array

7.2 Structs

```
// Structures -> a collection of variables of different data types
struct {
    byte    val1;
    int     val2;
    string  val3;
} struct_name;

// use typedef to actually make it a type

typedef struct {
    string fruit;
    int    count;
    byte     expiry;
} st_fruit;
```

Packed vs unpacked struct. Struct is unpacked by default

A packed structure is a mechanism for subdividing a vector into fields that can be accessed as members and are packed together in memory without gaps.

```
typedef struct packed {
  bit [3:0] mode;
  bit [2:0] cfg;
  bit       en;
} st_ctrl;
```

7.3 User-Defined Types

```
// Declare an alias for this long definition
typedef unsigned shortint           u_shorti;
typedef enum {RED, YELLOW, GREEN}   e_light;
typedef bit [7:0]                   ubyte;
```

7.3.1 Alias

alias keyword

8 SystemVerilog - Control Flow

8.1 Functions

8.2 Tasks

9 SystemVerilog - Processes

Example of forking processes

```
module fork_join;

  initial begin
    fork
      //Process-1
      begin
        $display($time,"\tProcess-1 Started");
        #5;
        $display($time,"\tProcess-1 Finished");
      end

      //Process-2
      begin
        $display($time,"\tProcess-2 Started");
        #20;
        $display($time,"\tProcess-2 Finished");
      end

      //Process-3
      begin
        $display("this is process 3");
      end
    join

    $finish;
  end
endmodule
```

Types of fork joins:

- fork join - Finishes when all child threads are over
- fork join_any - Finishes when any child thread gets over
- fork join_none - Finishes soon after child threads are spawned

10 SystemVerilog - Interface

Example

```
interface apb_if (input pclk);
    logic [31:0]    paddr;
    logic [31:0]    pwdata;
    logic [31:0]    prdata;
    logic           penable;
    logic           pwrite;
    logic           psel;
endinterface
```

Direction doesn't need to be explicitly stated, but `modport` helps specify directionality.

Port directions:

```
interface myBus (input clk);
  logic [7:0]  data;
  logic      enable;

  // From TestBench perspective, 'data' is input and 'write' is output
  modport TB  (input data, clk, output enable);

  // From DUT perspective, 'data' is output and 'enable' is input
  modport DUT (output data, input enable, clk);
endinterface
```

Example usage:

```
module dut (myBus busIf);
  always @ (posedge busIf.clk)
    if (busIf.enable)
      busIf.data <= busIf.data+1;
    else
      busIf.data <= 0;
endmodule
```

```
// Filename : tb_top.sv
module tb_top;
  bit clk;

  // Create a clock
  always #10 clk = ~clk;

  // Create an interface object
  myBus busIf (clk);

  // Instantiate the DUT; pass modport DUT of busIf
  dut dut0 (busIf.DUT);

  // Testbench code : let's wiggle enable
  initial begin
    busIf.enable  <= 0;
    #10 busIf.enable <= 1;
    #40 busIf.enable <= 0;
    #20 busIf.enable <= 1;
    #100 $finish;
  end
endmodule
```

10.1 Virtual Interface

SystemVerilog interface is static in nature, whereas classes are dynamic in nature. because of this reason, it is not allowed to declare the interface within classes, but it is allowed to refer to or point to the interface. A virtual interface is a variable of an interface type that is used in classes to provide access to the interface signals.

Syntax:

```
virtual interface_name instance_name;
```

Example:

```
//virtual interface
virtual intf vif;
```

```
//constructor
function new(virtual intf vif);
  //get the interface from test
  this.vif = vif;
endfunction
```

10.2 Clocking Block

```
interface my_int (input bit clk);
    // Rest of interface code

    clocking cb_clk @(posedge clk);
        default input #3ns output #2ns;
        input enable;
        output data;
    endclocking
endinterface
```

Example Usage:

```
// To wait for posedge of clock
@busIf.cb_clk;

// To use clocking block signals
busIf.cb_clk.enable = 1;
```

11 SystemVerilog - Object Oriented Programming

11.1 Class

To declare a class:

```
class myClass;
    bit [2:0] my_bits;
    logic [7:0] my_logic;

    // Constructor
    function new (bit [2:0] bits = 3'b101, logic [7:0] in_byte = 0);
        this.my_bits = bits;
        this.my_logic = in_byte;
    endfunction

    // Class Method
    function display();
        $display("myClass - my_bits:%0b, my_logic:%0b",
                            this.my_bits, this.my_logic);
    endfunction

endclass
```

To use a class:

```
module tb_top;

    myClass myobj1;

    myClass myobjs [5]; // an array of 5 objects


    initial begin
```

```
        // Create a single new object
        myobj1 = new(3'b111, 2);
        myobj1.display();

        // Creating an array of objects of the class
        for (int i = 0; i < $size(myobjs); i++) begin
            myobjs[i] = new();
            myobjs[i].display();
        end

    end

endmodule
```

Note that instantiating `myClass` gives you a **class handle** (essentially a pointer), which is `null` when unassigned. An actual class object is created when using `new()`.

11.2 Inheritance

Child classes can inherit methods/properties of a parent class. The `extend` keyword denotes a parent class, and the `super` keyword gives access to the parent class functions/properties.

```
class myChildClass extends myClass;
    bit [1:0] childProp;

    // Child Class Constructor
    function new ();
        super.new();
        this.childProp = 2'b11;
    endfunction

    function display();
        super.display();
        $display(" Child - childProp = %0b", this.childProp);
    endfunction

endclass
```

If a function exists for both parent and child, invocation of function depends on the type of the handle used. E.g., if both parent and child have function `display()`, where the function

is not virtual in the parent, the parent's function is run if the handle is of the parent class, and the child's function is run if the handle is of the child class. Functions/properties are called based on the type of the handle, rather than the type of the object the handle points to.

11.3 Abstract/Virtual Class

Objects of an abstract class cannot be created. A class must extend it to be used.

```
virtual class BaseClass;
    bit [7:0] data;

endclass

class ChildClass extends BaseClass;
    // Define the child class here
endclass
```

11.4 Polymorphism

Polymorphism allows base class handles to hold subclass objects and reference subclass properties/methods. Also allows child class method to have different definition from its parent class method when parent class method is `virtual`. We can only set a base class handle to a child class object, but NOT a child class handle to a base class object!

```
module tb;
    Packet      bc;     // bc stands for BaseClass
    ExtPacket   sc;     // sc stands for SubClass

    initial begin
        sc = new (32'hfeed_feed, 32'h1234_5678);

        // Assign sub-class to base-class handle
        bc = sc;

        bc.display ();
        sc.display ();
    end
endmodule
```

Use `$cast()` to assign parent class handle to a variable of child class. However, they need to be compatible or else a runtime error occurs.

It is legal to assign a super-class handle to a subclass variable if the super-class handle refers to an object of the given subclass.

```
parent_class = child_class ;
child_class  = parent_class; //allowed because parent_class is pointing to child_class.
// however this results in a compilation error. This is where $cast can be used instead:
$cast(child_class,parent_class);
```

Syntax:

```
$cast(<target_object_handle>,<original_object_handle>);
```

Example:

```
module
    initial begin
        bc = new (32'hface_cafe);

        // Dynamic cast base class object to sub-class type
        $cast (sc, bc);

        bc.display ();
    end
endmodule
```

Example 2:

```
class parent;
    virtual task print();
        $display("calling from parent class");
    endtask
endclass

class child extends parent;
    task print();
        $display("calling from child class");
    endtask
endclass
```

```
module casting();
    initial
    begin
        int x;
        parent p_obj;
        child c_obj1, c_obj2;

        x = 45.23; // implict casting
        $display("x = %0d", x);

        x = int'("s"); //explicit casting
        $display("x = %0d", x);

        p_obj = new();
        c_obj1 = new();

        // legal assignment (upcasting)
        p_obj = c_obj1;
        p_obj.print();

        // explicit dynamic casting (downcasting)
        if($cast(c_obj2,p_obj))
        begin
            $display("casting was successfull!!");
            c_obj2.print();
        end
        else
            $error("cast unsuccessfull!");
    end
endmodule

// Outputs:
// x = 45
// x = 115
// calling from child class
// casting was successfull!!
// calling from child class
```

11.4.1 Virtual Methods

For a base class handle to use the method of a subclass (while the handle points to a subclass object), the function (of the same name in the base class, if there is one) needs to be declared `virtual`. Otherwise, the method of the base class will be called instead.

```
class Packet;
   int addr;

   function new (int addr);
      this.addr = addr;
   endfunction

   // Here the function is declared as "virtual"
   // so that a different definition can be given
   // in any child class
   virtual function void display ();
        $display ("[Base] addr=0x%0h", addr);
    endfunction
endclass

class ExtPacket extends Packet;

    // This is a new variable only available in child class
    int data;

   function new (int addr, data);
      super.new (addr);     // Calls 'new' method of parent class
      this.data = data;
   endfunction

    function display ();
        $display ("[Child] addr=0x%0h data=0x%0h", addr, data);
    endfunction
endclass

module tb;
   Packet bc;
   ExtPacket sc;

    initial begin
        sc = new (32'hfeed_feed, 32'h1234_5678);
```

```
        bc = sc;
        bc.display ();
    end
endmodule

// Output:
// [Child] addr=0xfeedfeed data=0x12345678
// If it was NOT virtual:
// Output:
// [Base] addr=0xfeedfeed
```

11.5 Typedef

use `typedef` if compiler errors on a class being used before the declaration of the class itself.

12 SystemVerilog Constraints

13 SystemVerilog Assertion

14 Synthesis

14.1 Verilog

Verilog constructs that are not synthesizable:

- Initial blocks
- Delay constructs
- Real Data Types
- Forj/Join Constructs
- Random Functions
- X and Z States
- Primitives
- Force and Release

15 Summary

In summary...

References