SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – I – DIGITAL DESIGN USING VERILOG HDL– SECA3021

UNIT-I (DIGITAL DESIGN USING VERILOG HDL)

Hardware modeling with the Verilog HDL: Encapsulation, modeling primitives, Types of Modelling. Logic system, Data types and operators. Behavioral descriptions in verilog HDL. Styles for Synthesis of combinational logic and sequential logic. HDL based Synthesis – Technology Independent design

Verilog HDL is a hardware description language that can be used to model a digital system at many levels of abstraction ranging from the algorithmic level to the gate level to the switch level. The complexity of the digital system being modeled could vary from that of a simple gate to a complete electronic digital system, or anything in between. The digital system can be described hierarchically and timing can be explicitly modeled within the same description.

1.1 Typical Design Flow

A typical design flow for designing VLSI IC circuits is shown in Figure 2.1. Un shaded blocks show the level of design representation; shaded blocks show processes in the design flow.

The design flow shown in Figure 1.1 is typically used by designers who use HDLs. In any design, specifications are written first. Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed. At this point, the architects do not need to think about how they will implement this circuit. A behavioral description is then created to analyze the design in terms of functionality, performance, compliance to standards, and other high-level issues. Behavioral descriptions are often written with HDLs

The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of EDA tools.

Logic synthesis tools convert the RTL description to a gate-level netlist. A gatelevel netlist is a description of the circuit in terms of gates and connections between them. Logic synthesis tools ensure that the gate-level netlist meets timing, area, and power specifications. The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout. The layout is verified and then fabricated on a chip.

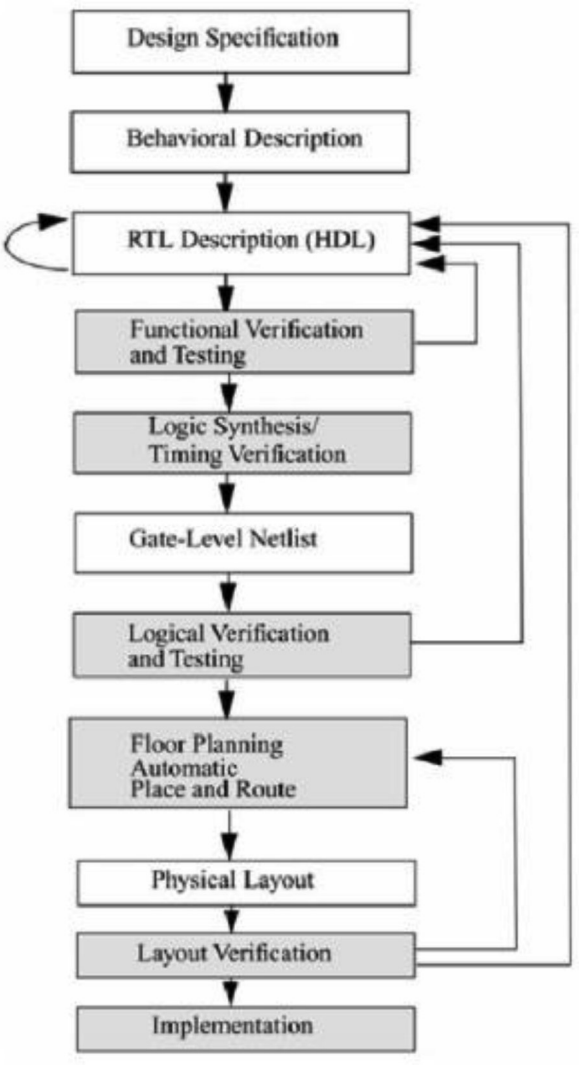

Figure 1.1: Typical Design Flow

1.2 Design Methodologies

There are two basic types of digital design methodologies: a top-down design methodology and a bottom-up design methodology. In a top-down design methodology, we define the top-level block and identify the sub-blocks necessary to build the top-level block. We further subdividethe sub-blocks until we come to leaf cells, which are the cells that cannot further be divided. Figure 1.2 shows the top-down design process.

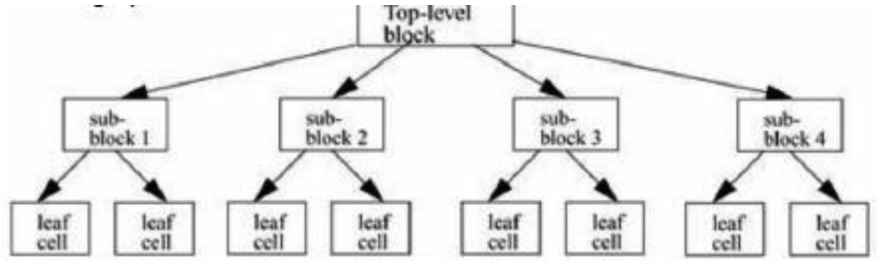

Figure 1.2: Top-down Design Methodology

In a bottom-up design methodology, we first identify the building blocks that are available to us. We build bigger cells, using these building blocks. These cells are then used for higher-level blocks until we build the top-level block in the design. Figure 1.3 shows the bottom-up design process

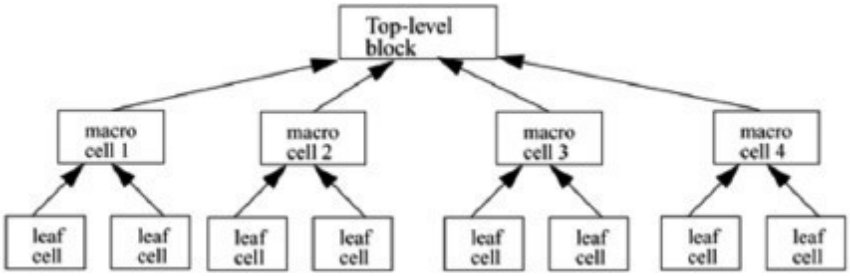

Figure 1.3: Bottom-up Design Methodology

1.2.1 Levels for design description

Verilog supports designing at many different levels of abstraction. Three of them are veryimportant:

- Behavioral level
- Register-Transfer LevelGate Level

Behavioral Level: This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

Register-Transfer Level: Designs using the Register-Transfer Level specify the characteristics of a circuit by operationsand the transfer of data between the registers.An explicit clock is used. RTL design contains exact timing bounds: operations are scheduled to occur at certain times. Modern RTL code definition is "Any code that is synthesizable is called RTL code".

Gate Level: Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.

1.3 Data types

Verilog HDL has two groups of data types a) Net type: A net type represents a physical connection between structural elements. Its value is determined from the value of its drivers such as a continuous assignment or a gate output. If no driver is connected to a net, the net defaults to a value of z. b)Variable type: A variable type represents an abstract data storage element. It is assigned values only within an always statement or an initial statement, and its value is saved from one assignment to the next. A variable type has a default value of x.

Net types

Here are the different kinds of nets that belong to the net data type

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

1.3.1 Variable types:

There are five different kinds of variable types reg, integer, time, real, realtime

Register: Registers represent data storage elements. Registers retain value until another value is placed onto them. Register data types are commonly declared by the keyword reg. The default value for a reg data type is x.

Integer: An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Although it is possible to use reg as a general-purpose variable, it is more convenient to declare an integer variable for purposes such as counting. The default width for an integer is the host-machine word size, which is implementation- specific but is at least 32 bits. Registers declared as data type reg store values as unsigned quantities, whereas integers store values as signed quantities.

Real: Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 106). Real numbers cannot have a range declaration, and their default value is 0. When a realvalue is assigned to an integer, the real number is rounded off to the nearest integer.

Time: Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation specific but is at

least 64 bits. The system function $time is invoked to get the current simulation time.

Arrays: Arrays are allowed in Verilog for reg, integer, time, real, realtime and vector register data types. Multi-dimensional arrays can also be declared with any number of dimensions. Arrays of nets can also be used to connect ports of generated instances. Each element of the array canbe used in the same fashion as a scalar or vector net.

1.4 Styles of Modelling:

1.4.1 Gate Level Modelling: The Built-in Primitive Gates: The following built-in primitive gates are available in Verilog HDL.

i. Multiple-input gates: and, nand, or,nor,xor,xnor
ii. Multiple-output gates: buf, not
iii. Tristate gates: buflfO, bufifl, notifO, notifl
iv. Pull gates: pullup, pulldown Multiple-input
v. MOS switches: cmos, nmos, pmos, rcmos, rnmos, rpmos
vi. Bidirectional switches: tran, tranifO, tranifl, rtran, rtranifO,rtranifl

A gate can be used in a design using a gate instantiation. Here is a simple format of a gate instantiation. gate_type[instance_name] (terml , term2 , . . . , termN);

Example: 4x1 Multiplexer:

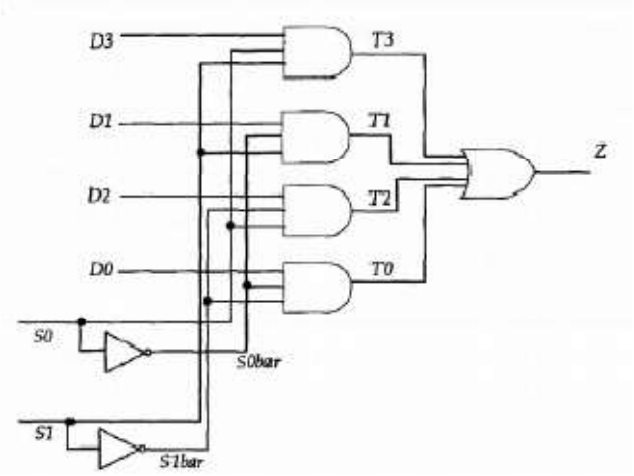

Figure 1.4 Multiplexer

```
moduleMUX4x1 (Z, DO, Dl, D2, D3, SO, Si);
output Z;
10
input DO, Dl, D2, D3, SO, SI;
and (TO, DO, SObar, Slbar),
(Tl, Dl, SObar, S1),
(T2, D2, SO, Slbar),
(T3, D3, SO, S1);
not (SObar,SO),
```

```
(Slbar, S1);
or (Z, TO, Tl, T2, T3);
endmodule
```

1.4.2 Dataflow Modeling

For small circuits, the gate-level modeling approach works very well because the numbers of gates is limited and the designer can instantiate and connect every gate individually. Also, gate-level modeling is very intuitive to a designer with a basic knowledge of digital logic design. However, in complex designs the number of gates is very large. Thus, designers can design more effectively if they concentrate on implementing the function at a level of abstraction higher than gate level. Dataflow modeling provides a powerful way to implement a design. Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data rather than instantiation of individual gates. Later in this chapter, the benefits of dataflow modeling will become more apparent. With gate densities on chips increasing rapidly, dataflow modeling has assumed great importance. No longer can companies devote engineering resources to handcrafting entire designs with gates. Currently, automated tools are used to create a gate-level circuit from a dataflow design description. This process is called logic synthesis. Dataflow modeling has become a popular design approach as logic synthesis tools have become sophisticated. This approach allows the designer to concentrate on optimizing the circuit in terms of data flow. For maximum flexibility in the design process, designers typically use a Verilog description style that combines the concepts of gate-level, data flow, and behavioral design. In the digital design community, the term RTL (Register Transfer Level) design is commonly used for a combination of dataflow modeling and behavioral modeling.

Example:

Master Slave Flip-flop:

```
module MSDFF_DF (D, C, Q, Qbar) ;
input D, C;
output Q, Qbar;
wireNotC, NotD, NotY, Y, Dl, D2, Ybar, Yl, Y2;
assignNotD = ~ D;
assign Note = ~ C;
assign NotY = ~ Y;
assign D1= - (D & C) ;
assign D2 = ~ (C &NotD);
assign Y = ~ (Dl St Ybar);
assign Ybar = ~ (Y & D2);
assign Yl = ~ (y & Note);
assign Y2 = - (NotY&NotC);
assign Q = ~ (Qbar&Yl);
assignQbar = ~ (Y2 & Q);
endmodule
```

8 bit Magnitude Comparator:

```
moduleMagnitudeComparator (A, B, AgtB, AeqB, AltB) ;
parameter BUS= 8;
parameter EQ_DELAY = 5, LT_DELAY = 8, GT_DELAY = 8;
input [1 : BUS]A, B;
outputAgtB, AeqB, AltB;
assign %EQ_DELAY AeqB = A == B; 19
assign $GT_DELAY AgtB = A > B;
assign $LT_DELAY AltB = A < B;
endmodule
```

1.4.3 Behavioral Modeling:

Behavioral modeling is the highest level of abstraction in the Verilog HDL. The other modeling techniques are relatively detailed. They require some knowledge of how hardware or hardware signals work. The abstraction in this modeling is as simple as writing the logic in C language. This is a very powerful abstraction technique. All that a designer need is the algorithm of the design, which is the basic information for any design. Most of the behavioral modeling is done using two important constructs: initial and always. All the other behavioral statements appear only inside these two structured procedure constructs.

Example:

4x1 Multiplexer

```
module mux4( input a, b, c, d
input [1:0] sel,
output out );
always @(a or b or c or d or sel)
begin
if(sel==0)
out = a;
else if (sel==1)
out = b;
else if ( sel == 2 )
out = c ;
else if ( sel == 3 )
out = d;
end
endmodule
```

D flip-flop

```
module RisingEdge_DFlipFlop(D,clk,Q);
input D; // Data input
input clk; // clock input
```

```
output Q; // output Q
always @(posedgeclk)
27
begin
Q <= D;
end
endmodule
```

Shift Register (Serial In Serial Out)

```
module shift (C, SI, SO);
input C,SI;
output SO;
reg [7:0] tmp;
always @(posedge C)
begin
tmp = tmp<< 1;
tmp[0] = SI;
end
assign SO = tmp[7];
endmodule
```

Procedural Assignments:

Procedural assignments are for updating reg, integer, time, and memory variables. There is a significant difference between procedural assignment and continuous assignment as described below –

Continuous assignments drive net variables and are evaluated and updated whenever an input operand changes value.

Procedural assignments update the value of register variables under the control of the procedural flow constructs that surround them.

The right-hand side of a procedural assignment can be any expression that evaluates to a value. However, part-selects on the right-hand side must have constant indices. The lefthand side indicates the variable that receives the assignment from the right-hand side. The left-hand side of a procedural assignment can take one of the following forms

− register, integer, real, or time variable

− An assignment to the name reference of one of these data types. bit-select of a register, integer, real, or time variable

− An assignment to a single bit that leaves the other bits untouched. part-select of a register, integer, real, or time variable

− A part-select of two or more contiguous bits that leaves the rest of the bits untouched. For the part-select form, only constant expressions are legal. memory element − A single word of a memory. Note that bit-selects and part-selects are illegal on memory element references. concatenation of any of the above

− A concatenation of any of the previous four forms can be specified, which effectively

partitions the result of the right-hand side expression and assigns the partition parts, in order, to the various parts of the concatenation.

Nonblocking (RTL) Assignments: The non-blocking procedural assignment allows you to schedule assignments without blocking the procedural flow. You can use the non-blocking procedural statement whenever you want to make several register assignments within the same time step without regard to order or dependance upon each other.

<lvalue> <= <timing_control> <expression>

Where lvalue is a data type that is valid for a procedural assignment statement, <= is the non-blocking assignment operator, and timing control is the optional intra-assignment timing control. The timing control delay can be either a delay control or an event control (for example, @(posedge clk)). The expression is the right-hand side value the simulator assigns to the left-hand side. The non-blocking assignment operator is the same operator the simulator uses for the less-than-orequal relational operator. The simulator interprets the <= operator to be a relational operator when you use it in an expression, and interprets the <= operator to be an assignment operator when you use it in a non-blocking procedural assignment construct. How the simulator evaluates non-blocking procedural assignments When the simulator encounters a non-blocking procedural assignment, the simulator evaluates and executes the non-blocking procedural assignment in two steps as follows –

The simulator evaluates the right-hand side and schedules the assignment of the new value to take place at a time specified by a procedural timing control.

The simulator evaluates the right-hand side and schedules the assignment of the new value to take place at a time specified by a procedural timing control. At the end of the time step, in which the given delay has expired or the appropriate event has taken place, the simulator executes the assignment by assigning the value to the left-hand side.

Case Statement: The case statement is a special multi-way decision statement that tests whether an expression matches one of a number of other expressions, and branches accordingly. The case statement is useful for describing, for example, the decoding of a microprocessor instruction. The case statement has the following syntax –

The case expressions are evaluated and compared in the exact order in which they are given. During the linear search, if one of the case item expressions matches the expression in parentheses, then the statement associated with that case item is executed. If all comparisons fail, and the default item is given, then the default item statement is executed. If the default statement is not given, and all of the comparisons fail, then none of the case item statements is executed.

Apart from syntax, the case statement differs from the multi-way if-else-if construct in two important ways –

The conditional expressions in the if-else-if construct are more general than comparing one expression with several others, as in the case statement.

The case statement provides a definitive result when there are x and z values in an expression.

Looping Statements: There are four types of looping statements. They provide a means of controlling the execution of a statement zero, one, or more times. forever continuously executes a statement. repeat executes a statement a fixed number of times. while executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all. for controls execution of its associated statement(s) by a three-step process, as follows

− Executes an assignment normally used to initialize a variable that controls the number of loops executed Evaluates an expression

- if the result is zero, the for loop exits, and if it is not zero, the for loop executes its associated statement(s) and then performs step 3 Executes an assignment normally used to modify the value of the loopcontrol variable, then repeats step 2.

Procedures: Always and Initial Blocks All procedures in Verilog are specified within one of the following four Blocks. 1) Initial blocks 2) Always blocks 3) Task 4) Function The initial and always statements are enabled at the beginning of simulation. The initial blocks executes only once and its activity dies when the statement has finished. In contrast, the always blocks executes repeatedly. Its activity dies only when the simulation is terminated. There is no limit to the number of initial and always blocks that can be defined in a module. Tasks and functions are procedures that are enabled from one or more places in other procedures.

1.4.4 Structural Modelling:

The structural model of Verilog HDL is described using:

- Gate instantiation
- UDP instantiation
- Module instantiation

Module: A module defines a basic unit in Verilog HDL. It is of the form:

```
modulemodule_name ( port_list );
Declarations_and_Statements
endmodule
```

The port list gives the list of ports through which the module communicates with the external modules.

Ports: A port can be declared as input, output or inout. A port by default is a net. However, it can be explicitly declared as a net. An output or an inout port can optionally be redeclared as a regregister. In either the net declaration or the register declaration the net or register must have the same size as the one specified in the port declaration. Here are some examples of declarations.

```
module Micro {PC, Instr, NextAddr);
// Port declarations:
input [3:1] PC;
output [1:8] Instr;
```

```
inout [16:1]NextAddr;
 // Redeclarations:
 wire [16:1] NextAddr;
 //Optional; but if specified must have same range as in its port declaration.
 reg [1:8] Instr;
 /* Instr has been redeclared as a reg so that it can be assigned a value within an always
 statement or an initial statement. */
 endmodule
```

Module Instantiation: A module can be instantiated in another module, thus creating hierarchy. A module instantiation statement is of the form:

module_name instance_name(port_associations);

Port associations can be by position or by name; however, associations cannot be mixed. A port association is of the form:

port_expr // By position.

.PortName (port_expr)// By name.

Where port_expr can be any of the following:

i. an identifier (a register or a net)

ii. a bit-select

iii. a part-select

iv. a concatenation of the above

v. an expression (only for input ports)

In positional association, the port expressions connect to the ports of the module in the specified order. In association by name, the connection between the module port and the port expression is explicitly specified and thus the order of port associations is not important. Here is an example of a full-adder built using two half-adder modules.

Half Adder:

```
module HA (A, B, S, C);
input A, B;
output S, C;
parameter AND_DELAY = 1, XOR_DELAY = 2;
assign #XOR_DELAY s=A ^ B;
assign #AND_ DELAY C= A & B;
endmodule
```

Full Adder:

```
module FA (P, Q, Cin, Sum, Cout);
input P, Q, Cin;
output Sum, Cout;
parameter OR_DELAY = 1;
wire SI, CI, C2;
//Two module instantiations:
```

HA h1 (P, Q, S1, C1);
// Associatingby position.
HA h2 (.A(Cin), .S(Sum), .B(S1), .C(C2)); //Associating by name.
// Gate instantiation:
or #OR_DELAY 01 (Cout, CI, C2) ;
endmodule

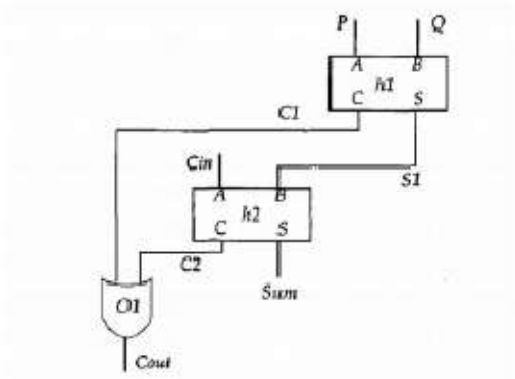

Figure 1.5: Full Adder using Two Half Adders

In the first module instantiation, HA is the name of the module, h1 is the instance name and ports are associated by position, that is, P is connected to module (HA) port A, Q is connected to module port B, S1 to S and C1 to module port C. In the second instantiation, the port association is by name, that is, the connections between the module (HA) ports and the port expressions are specified explicitly.

Example:

Decade Counter:

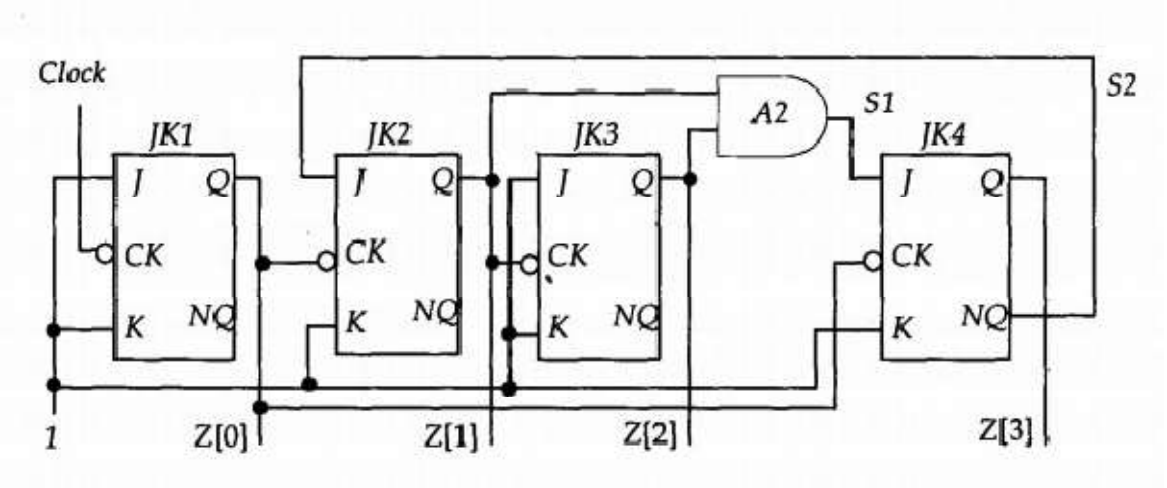

Figure 1.6: Decade Counter

```
module Decade_Ctr(Clock, Z);
input Clock;
output [0:3] Z;
wire SI, S2;
and A1 {SI, Z[2], Z[l]);// Primitive gate instantiation.
// Four module instantiations:
JK_FF JK1 (.J(1'b1), .K(1'b1), .ck(Clock),
        .Q(z[0]), .NQ ( )),
JK2 (.J(1'b1), .K(1'b1), .ck(Z[0]),
        .Q(z[1]), .NQ ()),
JK3 (.J(1'b1), .K(1'b1), .ck(Z[1]),
        .Q(z[2]), .NQ ()),
        JK4 (.J(1'b1), .K(1'b1), .ck(Z[0]),
        .Q(z[1]), .NQ (S2));
endmodule
```

3-bit UP-DOWN counter

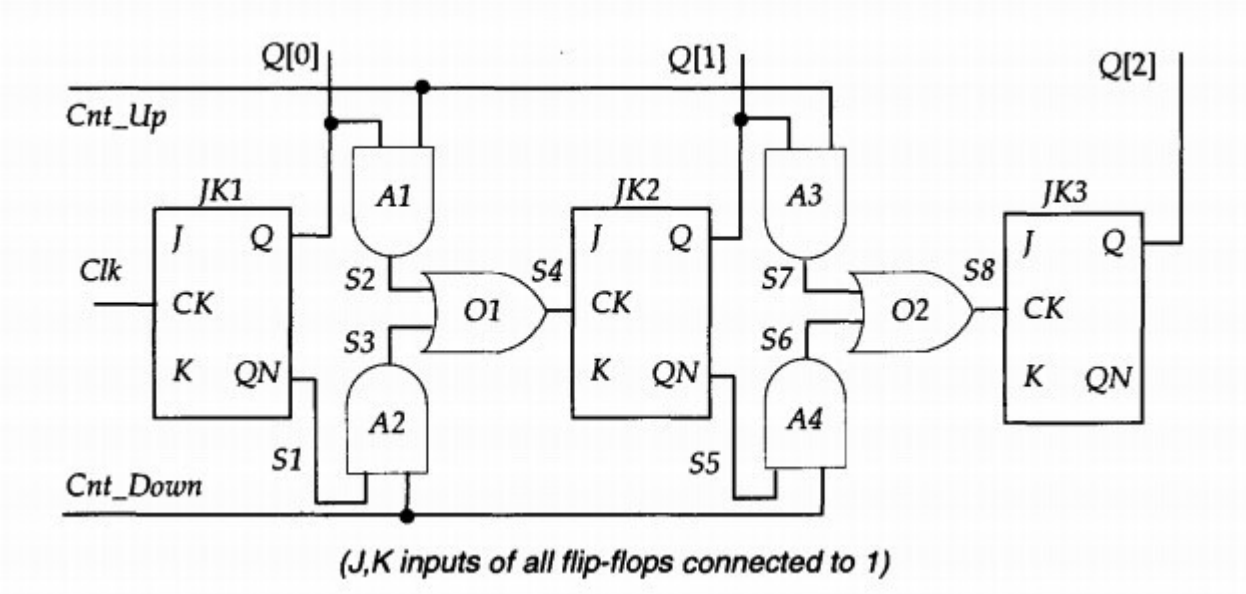

Figure 1.7: 3-bit UP-DOWN counter

```
moduleUp_Down {Clk, Cnt_Up, Cnt_Down, Q);
inputClk, Cnt__Up, Cnt_Down;
output [0:2] Q;
wire S1, S2, S3, S4, S5,S6, S7, S8;
JK_FF JK1 (l'bl, l'bl, Clk, Q[Q], S1),
JK2 (l'bl, l'bl, S4, Q[l], S5),
```

```
JK3 (l'bl, l'bl, S8, Q[2], );
and A1 (S2, Cnt_Up, Q[Q]),
A2 (S3, SI, Cnt_Down),
A3 (S7, Q[l] ,Cnt_Up),
A4 (S6, S5, Cnt_Down);
or 01 (S4, S2, S3),
02 (S8,S7, S6);
endmodule
```

1.5 USER-DEFINED PRIMITIVES (UDP):

The primitives available in Verilog are the entire gate or switch types. Verilog has the provision for the user to define primitives them. The designers occasionally like to use their own custom-built primitives when developing a design. Verilog provides the ability to define User-Defined Primitives (UDP). These primitives are self contained and do not instantiate other modules or primitives. UDPs are instantiated exactly like gate level primitives. UDPs are basically of two types combinational and sequential. A combinational UDP is used to define a combinational scalar

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An inout declaration is not supported by a UDP. The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other module.

```
primitiveudp_and(out, a, b);
output out;
input a, b;
table
// a b: Out;
0 0: 0;
0 1: 0;
1 0: 0
1 1: 1;
endtable
endprimitive
```

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state.

```
primitive latch(q, d, clock, clear); // d-latch
output q; reg q; //q declared as reg to create internal storage
input d, clock, clear;
initial q = 0; //initialize output to value 0
table
//state table
//d clock clear: q : q+ ;
```

```
? ? 1 : ? : 0 ; //clear condition;
  1 1 0 : ? : 1; //latchq =data=1

  0 1 0 : ? : 0; //latchq =data=0

? 0 0 : ? : - ; //retain original state if clock = 0
endtable
  endprimitive
```

Reference Books:

1. Advanced Digital Design With the Verilog HDL, Michael D. Ciletti, 2nd Edition, PHI, ISBN: 978–0– 07–338054–4 2015.
2. Digital Systems Design Using Verilog, Charles Roth, Lizy K. John, Byeong KilLee, Cengage Learning, ISBN-10: 1285051076, 2015.
3. Fundamentals of Digital Logic with Verilog Design, Stephen Brown and Zvonko Vranesic, 6th Edition, McGraw Hill publication, ISBN: 978–0–07–338054–4, 2014.
4. System Verilog for Design - A Guide to Using System Verilog for Hardware Design and Modeling, Stuart Sutherland, Simon David mann and Peter Flake, 2E, Springer Science, ISBN-13: 978-0387-3339-91, 2006.
5. System Verilog for Verification-A Guide to Learning the Testbench Language Features, C Spear, Springer Science, IEEE press, ISBN-13: 978-0387-2703-64,2006.
6. System Verilog golden reference guide-A concise guide to System Verilog Doulos, IEEE Standard-1800- 2009, Version 5.0,ISBN: 0-9547345-9-9, 2012.
7. Step-by-Step Functional Verification with System Verilog and OVM, SasanIman, Hansen Brown Publishing Company,ISBN-13: 978-0-9816-5621-2, 2008.

Questions to Practice:

PART -A

1	Describe in detail about the types of modelling
2	Identify the data types and its functions.
3	Describe in detail about synthesis in Verilog
4	Classify the Data Operators in Verilog
5	Develop a Verilog code for D Flip Flop using Behavioural modelling

PART-B

1	Develop a Verilog code for shift registers using structural modelling
2	Classify how Data Types are used in Verilog HDL
3	Demonstrate in detail about the User defined Data Types
4	Develop a Verilog code for Ripple carry Adder using structural modelling
5	Develop a Verilog code for Up/Down Counter using structural modelling
6	Develop a Verilog code for Full Adder using two half adders using structural modelling

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – II – INTRODUCTION TO SYSTEM VERILOG – SECA3021

UNIT-II (INTRODUCTION TO SYSTEM VERILOG)

System Verilog standards, Key System Verilog enhancements for hardware design. Advantages of System Verilog over Verilog, Data Types: Verilog data types, System Verilog data types, 2 - State Data types, Bit, byte, shortint, int, longint. 4 - State data types. Logic, Enumerated data types, User Defined data types, Struct data types, Strings, Packages, Type Conversion: Dynamic casting, Static Casting, Memories: Arrays, Dynamic Arrays, Multidimensional Arrays, Packed Arrays, Associative Arrays, Queues, Array Methods, Tasks and Functions: Verilog Tasks and Functions

2.1 Introduction :

SystemVerilog is a standard set of extensions to the IEEE 1364- 2005 Verilog Standard (commonly referred to as "Verilog-2005"). The SystemVerilog extensions to the Verilog HDL that are described in this book are targeted at design and writing synthesizable models. These extensions integrate many of the features of the SUPERLOG and C languages. SystemVerilog also contains many extensions for the verification of large designs, integrating features from the SUPERLOG, VERA C, C++, and VHDL languages, along with OVA and PSL assertions. These verification assertions are in a companion book, SystemVerilog for Verification.

This integrated whole created by SystemVerilog greatly exceeds the sum of its individual components, creating a new type of engineering language, a Hardware Description and Verification Language or HDVL. Using a single, unified language enables engineers to model large, complex designs, and verify that these designs are functionally correct.

2.2 Generations of the System Verilog standard:

A major portion of SystemVerilog was released as an Accellera standard in June of 2002 under the title of SystemVerilog 3.0. This initial release of the SystemVerilog standard allowed EDA companies to begin adding the SystemVerilog extensions to existing simulators, synthesis compilers and other engineering tools. The focus of this first release of the SystemVerilog standard was to extend the synthesizable constructs of Verilog, and to enable modeling hardware at a higher level of abstraction.

SystemVerilog began with a version number of 3.0 to show that SystemVerilog is the third major generation of the Verilog language. Verilog-1995 is the first generation, which represents the standardization of the original Verilog language defined by Phil Moorby in the early 1980s. Verilog-2001 is the second major generation of Verilog, and SystemVerilog is the third major generation.

Accellera continued to refine the SystemVerilog 3.1 standard by working closely with major Electronic Design Automation (EDA) companies to ensure that the SystemVerilog specification could be implemented as intended. A few additional modeling and verification constructs were also defined. In May of 2004, a final Accellera SystemVerilog draft was ratified by Accellera, and called System- Verilog 3.1a.

Prior to the donation of SystemVerilog 3.1a to the IEEE, the IEEESA had already begun work on the next revision of the IEEE 1364 Verilog standard. At the encouragement of Accellera, the IEEE-SA organization decided not to immediately add the SystemVerilog extensions to work already in progress for extending Verilog 1364. Instead, it was decided to keep the SystemVerilog extensions as a separate document. To ensure that the reference manual for the base Verilog language and the reference manual for the SystemVerilog extensions to Verilog remained synchronized, the IEEE-SA dissolved the 1364 Working Group and made the 1364 Verilog reference manual part of the responsibility of the 1800 SystemVerilog Working Group. The 1800 Working Group formed a subcommittee to update the 1364 Verilog standard in parallel with the specification of the P1800 SystemVerilog. For the most part, the work done on the 1364 revisions was limited to errata corrections and clarifications. Most extensions to Verilog were specified in the P1800 standard. The 1800 SystemVerilog Working Group released a ballot draft for an updated Verilog P1364 standard at the same time as the ballot draft for the new P1800 SystemVerilog standard. Both standards were approved at the same time.

2.3 Key SystemVerilog enhancements for hardware design

The following list highlights some of the more significant enhancements SystemVerilog adds to the Verilog HDL for the design and verification of hardware: This list is not intended to be all inclusive of every enhancement to Verilog that is in SystemVerilog. This list just highlights a few key features that aid in writing synthesizable hardware models.

- Interfaces to encapsulate communication and protocol checking within a design
- C like data types, such as int
- User-defined types, using typedef
- Enumerated types
- Type casting
- Structures and unions
- Packages for definitions shared by multiple design blocks
- External compilation-unit scope declarations
- ++, --, += and other assignment operators
- Explicit procedural blocks
- Priority and unique decision modifiers
- Programming statement enhancements
- Pass by reference to tasks, functions and modules

2.4 Packages

In Verilog, declarations of variables, nets, tasks and functions must be declared within a module, between the module...endmodule keywords. The objects declared within a module are local to the module. For modeling purposes, these objects should be referenced within the module in which they are declared. Verilog also allows hierarchical references to these objects from other modules for verification purposes, but these cross-module references do not represent

hardware behavior, and are not synthesizable. Verilog also allows local variables to be defined in named blocks (formed with begin...end or fork...join), tasks and functions. These declarations are still defined within a module, however, and, for synthesis purposes, only accessible within the module. Verilog does not have a place to make global declarations, such as global functions.

A declaration that is used in multiple design blocks must be declared in each block. This not only requires redundant declarations, but it can also lead to errors if a declaration, such as a function, is changed in one design block, but not in another design block that is supposed to have the same function. Many designers use include files and other coding tricks to work around this shortcoming, but that, too, can lead to coding errors and design maintenance problems.

SystemVerilog adds user-defined types, using typedef. It is often desirable to use the definition of user-defined types in multiple modules. Using Verilog rules, where declarations are always local to a module, it would be necessary to duplicate a user-defined type definition in each and every module in which the definition is used. Redundant local definitions would not be desirable for user-defined types.

2.4.1 Package definitions

To enable sharing a user-defined type definition across multiple modules, SystemVerilog adds packages to the Verilog language. The concept of packages is leveraged from the VHDL language. SystemVerilog packages are defined between the keywords package and endpackage.

The synthesizable constructs that a packages can contain are:

- parameter and localparam constant definitions
- const variable definitions
- typedef user-defined types
- Fully automatic task and function definitions
- import statements from other packages
- Operator overload definitions

Packages can also contain global variable declarations, static task definitions and static function definitions.

Packages can contain parameter, localparam and const constant declarations. The parameter and localparam constants are Verilog constructs. A const constant is a SystemVerilog constant. In Verilog, a parameter constant can be redefined for each instance of a module, whereas a localparam cannot be directly redefined. In a package, however, a parameter constant cannot be redefined, since it is not part of a module instance. In a package, parameter and localparam are synonymous.

Example:

```
package definitions;

  parameter VERSION = "1.1";

  typedef enum {ADD, SUB, MUL} opcodes_t;

  typedef struct {
    logic [31:0] a, b;
    opcodes_t    opcode;
  } instruction_t;

  function automatic [31:0] multiplier (input [31:0] a, b);
    // code for a custom 32-bit multiplier goes here
    return a * b;  // abstract multiplier (no error detection)
  endfunction
endpackage
```

2.4.2 Referencing package contents

Modules and interfaces can reference the definitions and declarations in a package four ways:

- Direct reference using a scope resolution operator
- Import specific package items into the module or interface
- Wildcard import package items into the module or interface
- Import package items into the $unit declaration space

SystemVerilog allows specific package items to be imported into a module, using an import statement. When a package definition or declaration is imported into a module or interface, that item becomes visible within the module or interface, as if it were a locally defined name within that module or interface. It is no longer necessary to explicitly reference the package name each time that package item is referenced.

2.4.3 Synthesis guidelines

When a module references a task or function that is defined in a package, synthesis will duplicate the task or function functionality and treat it as if it had been defined within the module. To be synthesizable, tasks and functions defined in a package must be declared as automatic, and cannot contain static variables. This is because storage for an automatic task or function is effectively allocated each time it is called. Thus, each module that references an automatic task or function in a package sees a unique copy of the task or function storage that is not shared by any other module. This ensures that the simulation behavior of the pre-synthesis reference to the package task or function will be the same as post-synthesis behavior, where the functionality of the task or function has been implemented within one or more modules.

For similar reasons, synthesis does not support variables declarations in packages. In simulation, a package variable will be shared by all modules that import the variable. One module can write to the variable, and another module will see the new value. This type of inter-module communication without passing values through module ports is not synthesizable.

2.5 $unit compilation-unit declarations

SystemVerilog adds a concept called a compilation unit to Verilog. A compilation unit is all source files that are compiled at the same time. Compilation units provide a means for software tools to separately compile sub-blocks of an overall design. A sub-block might comprise a single module or multiple modules. The modules might be contained in a single file or in multiple files. A sub-block of a design might also contain interface blocks and testbench program blocks.

SystemVerilog extends Verilog's declaration space by allowing declarations to be made outside of package, module, interface and program block boundaries. These external declarations are in a compilation-unit scope, and are visible to all modules that are compiled at the same time.

The compilation-unit scope can contain:

- Time unit and precision declarations
- Variable declarations
- Net declarations
- Constant declarations
- User-defined data types, using typedef, enum or class
- Task and function definitions

The following example illustrates external declarations of a constant, a variable, a user-defined type, and a function.

```
/******************* External declarations ******************/
parameter VERSION = "1.2a";   // external constant

reg resetN = 1;               // external variable (active low)

typedef struct packed {       // external user-defined type
  reg [31:0] address;
  reg [31:0] data;
  reg [ 7:0] opcode;
} instruction_word_t;

function automatic int log2 (input int n);  // external function
  if (n <=1) return(1);
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
  return(log2);
endfunction
```

```
/********************* module definition *********************/
// external declaration is used to define port types
module register (output instruction_word_t  q,
                 input  instruction_word_t  d,


                 input  wire                clock );

  always @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;  // use external reset
    else q <= d;
endmodule
```

A declaration in the compilation-unit scope is not the same as a global declaration. A true global declaration, such as a global variable or function, would be shared by all modules that make up a design, regardless of whether or not source files are compiled separately or at the same time.

SystemVerilog's compilation-scope only exists for source files that are compiled at the same time. Each time source files are compiled, a compilation-unit scope is created that is unique to just that compilation. For example, if module CPU and module controller both reference an externally declared variable called reset, then two possible scenarios exist:

• If the two modules are compiled at the same time, there will be a single compilation-unit scope. The externally declared reset variable will be common to both modules.

• If each module were compiled separately, then there would be two compilation-unit scopes, possibly with two different reset variables.

In the latter scenario, the compilation that included the external declaration of reset would appear to compile OK. The other file, when compiled separately, would have its own, unique $unit compilation space, and would not see the declaration of reset from the previous compilation. Depending on the context of how reset is used, the second compilation might fail, due to an undeclared variable, or it might compile OK, making reset an implicit net. This is a dangerous possibility! If the second compilation succeeds by making reset an implicit net, there will now be two signals called reset, one in each compilation. The two different reset signals would not be connected in any way.

2.5.1 Coding guidelines:

- Do not make any declarations in the $unit space! All declarations should be made in named packages.
- When necessary, packages can be imported into $unit. This is useful when a module or interface contains multiple ports that are of user-defined types, and the type definitions are in a package.

Directly declaring objects in the $unit compilation-unit space can lead to design errors when files are compiled separately. It can also lead to spaghetti code if the declarations are scattered in multiple files that can be difficult to maintain, re-use, or to debug declaration errors.

2.5.2 Variables and nets in the compilation-unit scope

There is an important consideration when using external declarations. Verilog supports implicit type declarations, where, in specific contexts, an undeclared identifier is assumed to be a net type (typically a wire type). Verilog requires the type of identifiers to be explicitly declared before the identifier is referenced when the context will not infer an implicit type, or when a type other than the default net type is desired.

This implicit type declaration rule affects the declaration of variables and nets in the compilation-unit scope. Software tools must encounter the external declaration before an identifier is referenced. If not, the name will be treated as an undeclared identifier, and follow the Verilog rules for implicit types.

The following example illustrates how source code order can affect the usage of a declaration external to the module. This example will not generate any type of compilation or elaboration error. For module parity_gen, software tools will automatically infer parity as an implicit net type local to the module, since the reference to parity comes before the external declaration for the signal. On the other hand, module parity_check comes after the external declaration of parity in the source code order. Therefore, the parity_check module will use the external variable declaration.

```
module parity_gen (input wire [63:0] data );
  assign parity = ^data; // parity is an
endmodule                // implicit local net


reg parity;  // external declaration is not
             // used by module parity_gen
             // because the declaration comes
             // after it has been referenced


module parity_check (input  wire [63:0] data,
                     output logic       err);
  assign err = (^data != parity); // parity is
                                  // the $unit
endmodule                         // variable
```

2.5.3 Synthesis Guidelines

The synthesizable constructs that can be declared within the compilation- unit scope (external to all module and interface definitions) are:

- typedef user-defined type definitions
- Automatic functions
- Automatic tasks
- parameter and localparam constants
- Package imports

While not a recommended style, user-defined types defined in the compilation-unit scope are synthesizable. A better style is to place the definitions of user-defined types in named packages. Using packages reduces the risk of spaghetti code and file order dependencies.

Declarations of tasks and functions in the $unit compilation-unit space is also not a recommended coding style. However, tasks and functions defined in $unit are synthesizable. When a module references a task or function that is defined in the compilation-unit scope, synthesis will duplicate the task or function code and treat it as if it had been defined within the module. To be synthesizable, tasks and functions defined in the compilation-unit scope must be declared as automatic, and cannot contain static variables. This is because storage for an automatic task or function is effectively allocated each time it is called. Thus, each module that references an automatic task or function in the compilation-unit scope sees a unique copy of the task or function storage that is not shared by any other module. This ensures that the simulation behavior of the presynthesis reference to the compilation-unit scope task or function will be the same as post-synthesis behavior, where the functionality of the task or function has been implemented within the module.

A parameter constant defined within the compilation-unit scope cannot be redefined, since it is not part of a module instance. Synthesis treats constants declared in the compilation-unit scope as literal values. Declaring parameters in the $unit space is not a good modeling style, as the constants will not be visible to modules that are compiled separately from the file that contains the constant declarations.

2.6 Declarations in unnamed statement blocks

Verilog allows local variables to be declared in named begin...end or fork...join blocks. A common usage of local variable declarations is to declare a temporary variable for controlling a loop. The local variable prevents the inadvertent access to a variable at the module level of the same name, but with a different usage. The following code fragment has declarations for two variables, both named i. The for loop in the named begin block will use the local variable i that is declared in that named block, and not touch the variable named i declared at the module level.

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin: loop       // named block
      integer i;      // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

A variable declared in a named block can be referenced with a hierarchical path name that includes the name of the block. Typically, only a testbench or other verification routine would reference a variable using a hierarchical path. Hierarchical references are not synthesizable, and do not represent hardware behavior. The hierarchy path to the variable within the named block can also be used by VCD (Value Change Dump) files, proprietary waveform

displays, or other debug tools, in order to reference the locally declared variable. The following testbench fragment uses hierarchy paths to print the value of both the variables named i in the preceding example:

```
module test;
  reg clock;
  chip chip (.clock(clock));

  always #5 clock = ~clock;

  initial begin
    clock = 0;
    repeat (5) @(negedge clock) ;
    $display("chip.i = %0d", chip.i);
    $display("chip.loop.i = %0d", chip.loop.i);


    $finish;
  end
endmodule
```

2.6.1 Local variables in unnamed blocks

SystemVerilog extends Verilog to allow local variables to be declared in unnamed blocks. The syntax is identical to declarations in named blocks, as illustrated below:

```
module chip (input clock);
  integer i;  // declaration at module level

  always @(posedge clock)
    begin              // unnamed block
      integer i;       // local variable
      for (i=0; i<=127; i=i+1) begin
        ...
      end
    end
endmodule
```

Since there is no name to the block, local variables in an unnamed block cannot be referenced hierarchically. A testbench or a VCD file cannot reference the local variable, because there is no hierarchy path to the variable.

Declaring variables in unnamed blocks can serve as a means of protecting the local variables from external, cross-module references. Without a hierarchy path, the local variable cannot be referenced from anywhere outside of the local scope.

This extension of allowing a variable to be declared in an unnamed scope is not unique to SystemVerilog. The Verilog language has a similar situation. User-defined primitives (UDPs) can have a variable declared internally, but the Verilog language does not require that an instance name be assigned to primitive instances. This also creates a variable in an unnamed scope. Software tools will infer an instance name in this situation, in order to allow the variable within the UDP to be referenced in the tool's debug utilities. Software tools may also assign an inferred name to an unnamed block, in order to allow the tool's waveform display or debug utilities to reference the local variables in that unnamed block. The SystemVerilog standard neither requires nor prohibits a tool inferring a scope name for unnamed blocks, just as the

Verilog standard neither requires nor prohibits the inference of instance names for unnamed primitive instances.

2.7 SystemVerilog Literal Values and Built-in Data Types

SystemVerilog extends Verilog's built-in variable types, and enhances how literal values can be specified. This chapter explains these enhancements and offers recommendations on proper usage. A number of small examples illustrate these enhancements in context. Subsequent chapters contain other examples that utilize SystemVerilog's enhanced variable types and literal values.

The enhancements presented in this chapter include:

- Enhanced literal values
- 'define text substitution enhancements
- Time values
- New variable types
- Signed and unsigned types
- Variable initialization
- Static and automatic variables
- Casting
- Constants

2.8 SystemVerilog variables

2.8.1 Object types and data types:

Verilog Data Types: The Verilog language has hardware-centric variable types and net types. These types have special simulation and synthesis semantics to represent the behavior of actual connections in a chip or system.

- The Verilog reg, integer and time variables have 4 logic values for each bit: 0, 1, Z and X.
- The Verilog wire, wor, wand, and other net types have 120 values for each bit (4-state logic plus multiple strength levels) and special wired logic resolution functions.

System Verilog Data Types: Verilog does not clearly distinguish between signal types, and the value set the signals can store or transfer. In Verilog, all nets and variables use 4-state values, so a clear distinction is not necessary. To provide more flexibility in variable and net types and the values that these types can store or transfer, the SystemVerilog standard defines that signals in a design have both a type and a data type.

Type indicates if the signal is a net or variable. SystemVerilog uses all the Verilog variable types, such as reg and integer, plus adds several more variable types, such as byte and int. SystemVerilog does not add any extensions to the Verilog net types.

Data type indicates the value system of the net or variable, which is 0 or 1 for 2-state data types, and 0, 1, Z or X for 4-state data types. The SystemVerilog keyword bit defines that an object is a 2-state data type. The SystemVerilog keyword logic defines that an object is a 4-state data type. In the SystemVerilog-2005 standard, variable types can be either 2-state or 4-state data types, where as net types can only be 4-state data types.

2.8.2 SystemVerilog 4-state and 2-state variables

The 4-state logic type: The Verilog language uses the reg type as a general purpose variable for modeling hardware behavior in initial and always procedural blocks. The keyword reg is a misnomer that is often confusing to new users of the Verilog language. The term "reg" would seem to imply a hardware "register", built with some form of sequential logic flip-flops. In actuality, there is no correlation whatsoever between using a reg variable and the hardware that will be inferred. It is the context in which the reg variable is used that determines if the hardware represented is combinational logic or sequential logic. SystemVerilog uses the more intuitive logic keyword to represent a general purpose, hardware-centric data type.

```
logic resetN; // a 1-bit wide 4-state variable

logic [63:0] data; // a 64-bit wide variable

logic [0:7] array [0:255]; // an array of 8-bit
                           variables
```

The keyword logic is not actually a variable type, it is a data type, indicating the signal can have 4-state values. However, when the logic keyword is used by itself, a variable is implied. A 4-state variable can be explicitly declared using the keyword pair var logic.

For example:
var logic [63:0] addr; // a 64-bit wide variable

A Verilog net type defaults to being a 4-state logic data type. A net can also be explicitly declared as a 4-state data type using the logic keyword.

For example:
wire logic [63:0] data; // a 64-bit wide net

Semantically, a variable of the logic data type is identical to the Verilog reg type. The two keywords are synonyms, and can be used interchangeably (except that the reg keyword cannot be paired with net type keywords, as discussed in section 3.3.4 on page 47). Like the Verilog reg variable type, a variable of the logic data type can store 4-state logic values (0, 1, Z and X), and can be defined as a vector of any width.

Because the keyword logic does not convey a false implication of the type of hardware represented, logic is a more intuitive keyword choice for describing hardware when 4-state logic is required. In the subsequent examples in this book, the logic type is used in place of the Verilog

reg type (except when the example illustrates pure Verilog code, with no SystemVerilog enhancements).

SystemVerilog 2-state variables: SystemVerilog adds several new 2-state types, suitable for modeling at more abstract levels than RTL, such as system level and transaction level. These types include:

- bit — a 1-bit 2-state integer
- byte — an 8-bit 2-state integer, similar to a C char
- shortint — a 16-bit 2-state integer, similar to a C short
- int — a 32-bit 2-state integer, similar to a C int
- longint — a 64-bit 2-state integer, similar to a C longlong

Variables of the reg or logic data types are used for modeling hardware behavior in procedural blocks. These types store 4-state logic values, 0, 1, Z and X. 4-state types are the preferred types for synthesizable RTL hardware models. The Z value is used to represent unconnected or tri-state design logic. The X value helps detect and isolate design errors. At higher levels of modeling, such as the system and transaction levels, logic values of Z and X are seldom required.

SystemVerilog allows variables to be declared as a bit data type. Syntactically, a bit variable can be used any place reg or logic variables can be used. However, the bit data type is semantically different, in that it only stores 2-state values of 0 and 1. The bit data type can be useful for modeling hardware at higher levels of abstraction. Variables of the bit data type can be declared in the same way as reg and logic types. Declarations can be any vector width, from 1-bit wide to the maximum size supported by the software tool (the IEEE 1364 Verilog standard defines that all compliant software tools should support vector widths of at least 216 bits wide).

```
bit resetN; // a 1-bit wide 2-state variable
bit [63:0] data; // a 64-bit 2-state variable
bit [0:7] array [0:255]; // an array of 8-bit
                            2-state variables
```

2.9 Type casting

Verilog is a loosely typed language that allows a value of one type to be assigned to a variable or net of a different type. When the assignment is made, the value is converted to the new type, following rules defined as part of the Verilog standard. SystemVerilog adds the ability to cast a value to a different type. Type casting is different than converting a value during an assignment. With type casting, a value can be converted to a new type within an expression, without any assignment being made. The Verilog 1995 standard did not provide a way to cast a value to a different type. Verilog-2001 added a limited cast capability that can convert signed values to unsigned, and unsigned values to signed. This conversion is done using the system functions $signed and $unsigned.

2.9.1 Static Casting:

SystemVerilog adds a cast operator to the Verilog language. This operator can be used to cast a value from one type to another, similar to the C language. SystemVerilog's cast operator goes beyond C, however, in that a vector can be cast to a different size, and signed values can be cast to unsigned or vice versa. To be compatible with the existing Verilog language, the syntax of SystemVerilog's cast operator is different than C's. <type>'(<expression>) — casts a value to any type, including user-defined types.

For Example:

```
7+ int'(2.0 * 3.0);  // cast result of
                     // (2.0 * 3.0) to int,
                     //   then add to 7
```

size casting **<size>'(<expression>)** — casts a value to any vector size. For example:

```
logic [15:0] a, b, y;
y = a + b**16'(2);  // cast literal value 2
                    //   to be 16 bits wide
```

sign casting **<sign>'(<expression>)** — casts a value to signed or unsigned. For example:

```
shortint a, b;
int      y;
y = y - signed'({a,b}); // cast concatenation
                        //   result to a signed
                        //   value
```

2.9.1.1 Static casting and error checking:

The static cast operation is a compile-time cast. The expression to be cast will always be converted during run time, without any checking that the expression to be cast falls within the legal range of the type to which the value is cast. In the following example, a static cast is used to increment the value of an enumerated variable by 1. The static cast operator does not check that the result of state + 1 is a legal value for the next_state enumerated type. Assigning an out of range value to next_state using a static cast will not result in a compile-time or run-time error. Therefore, care must be taken not to cause an illegal value to be assigned to the next_state variable.

```
typedef enum {S1, S2, S3} states_t;
states_t state, next_state;

always_comb begin
  if (state != S3)
    next_state = states_t'(state + 1);
  else
    next_state = S1;
end
```

2.9.2 Dynamic casting

The static cast operation described above is a compile-time cast. The cast will always be performed, without checking the validity of the result. When stronger checking is desired, SystemVerilog provides a new system function, $cast, that performs dynamic, runtime checking on the value to be cast.

The $cast system function takes two arguments, a destination variable and a source variable.

```
$cast( dest_var, source_exp );
```

For example:

```
int radius, area;

always @(posedge clock)
  $cast(area, 3.154 * radius ** 2);
    // result of cast operation is cast to
    // the type of area
```

$cast attempts to assign the source expression to the destination variable. If the assignment is invalid, a run-time error is reported, and the destination variable is left unchanged. Some examples that would result in an invalid cast are:

• Casting a real to an int, when the value of the real number is too large to be represented as an int (as in the example, above).

• Casting a value to an enumerated type, when the value does not exist in the legal set of values in the enumerated type list

2.10 User-Defined and Enumerated Types

SystemVerilog makes a significant extension to the Verilog language by allowing users to define new net and variable types. User-defined types allow modeling complex designs at a more abstract level that is still accurate and synthesizable. Using System- Verilog's user-defined types, more design functionality can be modeled in fewer lines of code, with the added advantage of making the code more self-documenting and easier to read.

The enhancements presented in this include:

- Using typedef to create user-defined types
- Using enum to create enumerated types
- Working with enumerated values

2.10.1 User Defined Types:

The Verilog language does not provide a mechanism for the user to extend the language net and variable types. While the existing Verilog types are useful for RTL and gate-level modeling, they do not provide C-like variable types that could be used at higher levels of abstraction. SystemVerilog adds a number of new types for modeling at the system and architectural level. In addition, SystemVerilog adds the ability for the user to define new net and variable types.

SystemVerilog user-defined types are created using the typedef keyword, as in C. User-defined types allow new type definitions to be created from existing types. Once a new type has been defined, variables of the new type can be declared.

```
typedef int unsigned uint;
...
uint a, b;  // two variables of type uint
```

Local typedef definitions: User-defined types can be defined locally, in a package, or externally, in the compilation-unit scope. When a user-defined type will only be used within a specific part of the design, the typedef definition can be made within the module or interface representing that portion of the design. Interfaces are presented in Chapter 10. In the code snippet that follows, a user-defined type called nibble is declared, which is used for variable declarations within a module called alu. Since the nibble type is defined locally, only the alu module can see the definition. Other modules or interfaces that make up the overall design are not affected by the local definition, and can use the same nibble identifier for other purposes without being affected by the local typedef definition in module alu.

```
module alu (...);

  typedef logic [3:0] nibble;

  nibble opA, opB;  // variables of the
                    // nibble type

  nibble [7:0] data; // a 32-bit vector made
                     // from 8 nibble types
  ...
endmodule
```

Shared typedef definitions: When a user-defined type is to be used in many different models, the typedef definition can be declared in a package. These definitions can then be referenced directly, or imported into each module, interface or program block that uses the user-defined types.

A typedef definition can also be declared externally, in the compilation- unit scope. External declarations are made by placing the typedef statement outside of any module, interface or program block. For Example

```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

module counter
(output chip_types::dtype_t [15:0] count,
 input  chip_types::dtype_t clock, resetN);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;
    else         count <= count + 1;
endmodule
```

It is also possible to import package definitions into the $unit compilation- unit space. This can be useful when many ports of a module are of user-defined types, and it becomes tedious to directly reference the package name for each port declaration. Example illustrates importing a package definition into the $unit space, for use as a module port type.

```
package chip_types;
  `ifdef TWO_STATE
    typedef bit dtype_t;
  `else
    typedef logic dtype_t;
  `endif
endpackage

import chip_types::dtype_t;  // import definition into $unit

module counter
(output dtype_t [15:0] count,
 input  dtype_t clock, resetN);

  always @(posedge clock, negedge resetN)
    if (!resetN) count <= 0;
    else         count <= count + 1;
endmodule
```

If the package contains many typedefs, instead of importing specific package items into the $unit compilation-unit space, the package can be wildcard imported into $unit.

2.11 Enumerated Types:

Enumerated types provide a means to declare an abstract variable that can have a specific list of valid values. Each value is identified with a user-defined name, or label. In the following example, variable RGB can have the values of red, green and blue:

enum {red,green,blue} RGB;

The Verilog language does not have enumerated types. To create pseudo labels for data values, it is necessary to define a parameter constant to represent each value, and assign a value to that constant. Alternatively, Verilog's 'define text substitution macro can be used to define a set of macro names with specific values for each name. The following example shows a simple state machine sequence modeled using Verilog parameter constants and 'define macro names: The parameters are used to define a set of states for the state machine, and the macro names are used to define a set of instruction words that are decoded by the state machine.

Example:

```
`define FETCH 3'h0
`define WRITE 3'h1
`define ADD   3'h2
`define SUB   3'h3
`define MULT  3'h4
`define DIV   3'h5
`define SHIFT 3'h6
`define NOP   3'h7

module controller (output reg        read, write,
                   input  wire [2:0] instruction,
                   input  wire       clock, resetN);

  parameter WAITE = 0,
            LOAD  = 1,
            STORE = 2;

  reg [1:0] State, NextState;


  always @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always @(State) begin
    case (State)
      WAITE:  NextState = LOAD;
      LOAD:   NextState = STORE;
      STORE:  NextState = WAITE;
    endcase
  end

  always @(State, instruction) begin
    read = 0; write = 0;
    if (State == LOAD && instruction == `FETCH)
      read = 1;
    else if (State == STORE && instruction == `WRITE)
      write = 1;
  end
endmodule
```

The variables that use the constant values—State and NextState in the preceding example—must be declared as standard Verilog variable types. This means a software tool cannot limit the valid values of those signals to just the values of the constants. There is nothing that would limit State or NextState in the example above from having a value of 3, or a value with one or more bits set to X or Z. Therefore, the model itself must add some limit checking on

the values. At a minimum, a synthesis “full case” pragma would be required to specify to synthesis tools that the state variable only uses the values of the constants that are listed in the case items. The use of synthesis pragmas, however, would not affect simulation, which could result in mismatches between simulation behavior and the structural design created by synthesis.

Example: State machine modeled with enumerated types

```
package chip_types;
  typedef enum {FETCH, WRITE, ADD, SUB,
                MULT, DIV, SHIFT, NOP  } instr_t;
endpackage

import chip_types::*;  // import package definitions into $unit

module controller (output logic   read, write,
                   input  instr_t instruction,
                   input  wire    clock, resetN);

  enum {WAITE, LOAD, STORE} State, NextState;

  always_ff @(posedge clock, negedge resetN)
    if (!resetN) State <= WAITE;
    else         State <= NextState;

  always_comb begin
    case (State)
      WAITE: NextState = LOAD;
      LOAD:  NextState = STORE;
      STORE: NextState = WAITE;
    endcase
  end

  always_comb begin
    read = 0; write = 0;
    if (State == LOAD && instruction == FETCH)
      read = 1;
    else if (State == STORE && instruction == WRITE)
      write = 1;
  end
endmodule
```

In this example, the variables State and NextState can only have the valid values of WAITE, LOAD, and STORE. All software tools will interpret the legal value limits for these enumerated type variables in the same way, including simulation, synthesis and formal verification.

2.12 Arrays:

2.12.1 Unpacked arrays:

The basic syntax of a Verilog array declaration is:

<data_type> <vector_size> <array_name> <array_dimensions>

For example: reg [15:0] RAM [0:4095]; // memory array

Verilog-1995 only permitted one-dimensional arrays. A one-dimensional array is often referred to as a memory, since its primary purpose is to model the storage of hardware memory devices such as RAMs and ROMs. Verilog-1995 also limited array declarations to just the variable types reg, integer and time. Verilog-2001 significantly enhanced Verilog-1995 arrays by

allowing any variable or net type except the event type to be declared as an array, and by allowing multi-dimensional arrays. Beginning with Verilog-2001, both variable types and net types can be used in arrays.

```
// a 1-dimensional unpacked array of
// 1024 1-bit nets
wire n [0:1023];

// a 1-dimensional unpacked array of
// 256 8-bit variables
reg [7:0] LUT [0:255];

// a 1-dimensional unpacked array of
// 1024 real variables
real r [0:1023];
```

Verilog restricts the access to arrays to just one element of the array at a time, or a bit-select or part-select of a single element. Any reading or writing to multiple elements of an array is an error.

```
integer i [7:0][3:0][7:0];
integer j;

j = i[3][0][1];  // legal: selects 1 element
j = i[3][0];     // illegal: selects 8 elements
```

SystemVerilog refers to the Verilog style of array declarations as unpacked arrays. With unpacked arrays, each element of the array may be stored independently from other elements, but grouped under a common array name. Verilog does not define how software tools should store the elements in the array. For example, given an array of 8-bit wide elements, a simulator or other software tool might store each 8-bit element in 32-bit words.

SystemVerilog extends unpacked array dimensions to include the Verilog event type, and the SystemVerilog types: logic, bit, byte, int, longint, shortreal, and real. Unpacked arrays of user-defined types defined using typedef can also be declared, including types using struct and enum.

```
bit [63:0] d_array [1:128]; // array of vectors
shortreal cosines [0:89];  // array of floats
typedef enum {Mo, Tu, We, Th, Fr, Sa, Su} Week;
Week Year [1:52];  // array of Week types
```

SystemVerilog also adds to Verilog the ability to reference an entire unpacked array, or a slice of multiple elements within an unpacked array. A slice is one or more contiguously numbered elements within one dimension of an array. These enhancements make it possible to copy the contents of an entire array, or a specific dimension of an array into another array.

In order to directly copy multiple elements into an unpacked array, the layout and element type of the array or array slice on the lefthand side of the assignment must exactly match

the layout and element type of the right-hand side. That is, the element type and size and the number of dimensions copied must be the same.

2.12.2 Packed arrays

The Verilog language allows vectors to be created out of single-bit types, such as reg and wire. The vector range comes before the signal name, whereas an unpacked array range comes after the signal name.

SystemVerilog refers to vector declarations as packed arrays. A Verilog vector is a one-dimensional packed array.

```
wire [3:0] select;  // 4-bit "packed array"
reg [63:0] data;    // 64-bit "packed array"
```

SystemVerilog adds the ability to declare multiple dimensions in a packed array.

```
logic [3:0][7:0] data;  // 2-D packed array
```

SystemVerilog defines how the elements of a packed array are stored. The entire array must be stored as contiguous bits, which is the same as a vector. Each dimension of a packed array is a sub field within the vector. In the packed array declaration above, there is an array of 4 8-bit sub-arrays.

SystemVerilog also adds dynamic array types to Verilog:

- Dynamic arrays
- Associative arrays
- Sparse arrays
- Strings (character arrays)

Dynamically sized arrays are not synthesizable, and are intended for use in verification routines and for modeling at very high levels of abstraction.

2.13 Tasks and Functions:

SystemVerilog makes several enhancements to Verilog tasks and functions. These enhancements make it easier to model large designs in an efficient and intuitive manner.

2.13.1 Implicit task and function statement grouping:

In Verilog, multiple statements within a task or function must be grouped using begin...end. Tasks also allow multiple statements to be grouped using fork...join. SystemVerilog simplifies task and function definitions by not requiring the begin...end grouping for multiple statements. If the grouping is omitted, multiple statements within a task or function are executed sequentially, as if within a begin...end block.

```
function states_t NextState(states_t State);
  NextState = State;   // default next state
  case (State)
    WAITE:  if (start) NextState = LOAD;
    LOAD:   if (done)  NextState = STORE;
    STORE:             NextState = WAITE;
  endcase
endfunction
```

2.13.2 Returning function values

In Verilog, the function name itself is an inferred variable that is the same type as the function. The return value of a function is set by assigning a value to the name of the function. A function exits when the execution flow reaches the end of the function. The last value that was written into the inferred variable of the name of function is the value returned by the function.

```
function [31:0] add_and_inc (input [31:0] a,b);
  begin
    add_and_inc = a + b + 1;
  end
endfunction
```

SystemVerilog adds a **return** statement, which allows functions to return a value using **return**, as in C.

```
function int add_and_inc (input int a, b);
  return a + b + 1;
endfunction
```

To maintain backward compatibility with Verilog, the return value of a function can be specified using either the return statement or by assigning to the function name. The return statement takes precedence. If a return statement is executed, that is the value returned. If the end of the function is reached without executing a return statement, then the last value assigned to the function name is the return value, as it is in Verilog. Even when using the return statement, the name of the function is still an inferred variable, and can be used as temporary storage before executing the return statement. For example:

```
function int add_and_inc (input int a, b);
  add_and_inc = a + b;
  return ++add_and_inc;
endfunction
```

2.13.3 Returning before the end of tasks and functions

In Verilog, a task or function exits when the execution flow reaches the end, which is denoted by endtask or endfunction. In order to exit before the end a task or function is reached using Verilog, conditional statements such as if...else must be used to force the execution flow to jump to the end of the task or function. A task can also be forced to jump to its end using the disable keyword, but this will affect all currently running invocations of a re-entrant task. The following example requires extra coding to prevent executing the function if the input to the function is less than or equal to 1.

```
function automatic int log2 (input int n);
  if (n <=1)
    log2 = 1;
  else begin  // skip this code when n<=1


      log2 = 0;
      while (n > 1) begin
        n = n/2;
        log2 = log2+1;
      end
    end
  endfunction
```

The SystemVerilog return statement can be used to exit a task or function at any time in the execution flow, without having to reach the end of the task or function. Using return, the example above can be simplified as follows:

```
function automatic int log2 (input int n);
  if (n <=1) return 1;  // abort function
  log2 = 0;
  while (n > 1) begin
    n = n/2;
    log2++;
  end
endfunction
```

Using return to exit a task or function before the end is reached can simplify the coding within the task or function, and make the execution flow more intuitive and readable

2.13.4 Named task and function ends

SystemVerilog allows a name to be specified with the endtask or endfunction keyword. The syntax is:

```
endtask : <task_name>
endfunction : <function_name>
```

The white space before and after the colon is optional. The name specified must be the same as the name of the corresponding task or function. For example:

```
function int add_and_inc (int a, b);
  return a + b + 1;
endfunction : add_and_inc


task automatic check_results (
  input packet_t sent,
  ref   packet_t received,
  ref   logic    done );
  static int error_count;
  ...
endtask: check_results
```

Specifying a name with the endtask or endfunction keyword can help make large blocks of code easier to read, thus making the model more maintainable.

2.13.5 Empty tasks and functions

Verilog requires that tasks and functions contain at least one statement (which can be an empty begin...end statement group). SystemVerilog allows tasks and functions to be completely empty, with no statements or statement groups at all. An empty function will return the current value of the implicit variable that represents the name of the function. An empty task or function is a place holder for partially completed code. In a top-down design flow, creating an empty task or function can serve as documentation in a model for the place where more detailed functionality will be filled in later in the design flow.

Reference Books:

1. Advanced Digital Design With the Verilog HDL, Michael D. Ciletti, 2nd Edition, PHI, ISBN: 978–0– 07–338054–4 2015.
2. Digital Systems Design Using Verilog, Charles Roth, Lizy K. John, Byeong KilLee, Cengage Learning, ISBN-10: 1285051076, 2015.
3. Fundamentals of Digital Logic with Verilog Design, Stephen Brown and Zvonko Vranesic, 6th Edition, McGraw Hill publication, ISBN: 978–0–07–338054–4, 2014.
4. System Verilog for Design - A Guide to Using System Verilog for Hardware Design and Modeling, Stuart Sutherland, Simon David mann and Peter Flake, 2E, Springer Science, ISBN-13: 978-0387-3339-91, 2006.
5. System Verilog for Verification-A Guide to Learning the Testbench Language Features, C Spear, Springer Science, IEEE press, ISBN-13: 978-0387-2703-64,2006.
6. System Verilog golden reference guide-A concise guide to System Verilog Doulos, IEEE Standard-1800- 2009, Version 5.0,ISBN: 0-9547345-9-9, 2012.
7. Step-by-Step Functional Verification with System Verilog and OVM, SasanIman, Hansen Brown Publishing Company,ISBN-13: 978-0-9816-5621-2, 2008.

Questions to Practice:

PART -A

1	Identify how System Verilog is going to be a trendsetter in VLSI Industry
2	Identify the data types and its functions.
3	Describe in detail about static casting
4	Classify the use cases of arrays in System Verilog
5	Explain in detail about Empty Tasks and Functions

PART-B

1	With the advent of VLSI Industry, Demonstrate how industry experts define the support of System Verilog
2	Classify how Data Types are used in System Verilog
3	Demonstrate in detail about the User defined Data Types
4	Demonstrate in detail about the impact of tasks and Functions in System Verilog
5	Enumerate Packed and Unpacked Arrays in detail.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – III – CONNECTING THE TESTBENCH AND DESIGN– SECA3021

UNIT-III (CONNECTING THE TESTBENCH AND DESIGN)

Verilog interface signals - Limitations of Verilog interface signals, SystemVerilog interfaces, SystemVerilog port connections, Interface instantiation. Interfaces Arguments, Interface Modports, Interface References, Tasks and functions in interface, Verilog Event Scheduler, SystemVerilog Event Scheduler, Clocking Block, Input and Output Skews, Typical Testbench Environment, Verification plan

SystemVerilog extends the Verilog language with a powerful interface construct. Interfaces offer a new paradigm for modeling abstraction. The use of interfaces can simplify the task of modeling and verifying large, complex designs. This chapter contains a number of small examples, each one showing specific features of interfaces. These examples have been purposely kept relatively small and simple, in order to focus on specific features of interfaces.

3.1 Interface concepts:

The Verilog language connects modules together through module ports. This is a detailed method of representing the connections between blocks of a design that maps directly to the physical connections that will be in the actual hardware. For large designs, however, using module ports to connect blocks of a design together can become tedious and redundant. Consider the following example that connects five blocks of a design together using a rudimentary bus architecture called main_bus, plus some additional connections between some of the design blocks. Figure shows the block diagram for this simple design, and example lists the Verilog source code for the module declarations involved.

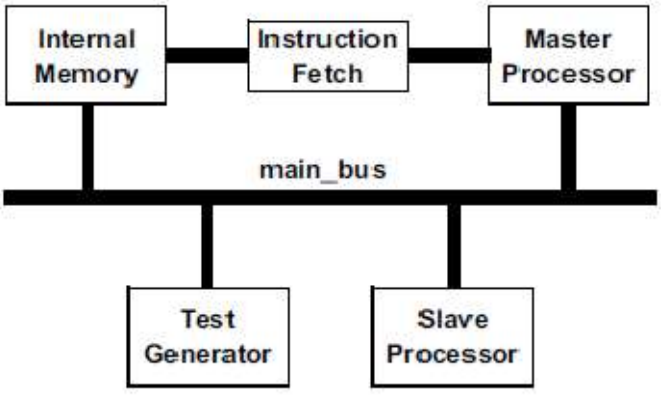

Figure 3.1: Block diagram of a simple design

Example: Verilog module interconnections for a simple design

```
/********************* Top-level Netlist ******************/
module top (input wire clock, resetN, test_mode);
  wire [15:0] data, address, program_address, jump_address;
  wire [ 7:0] instruction, next_instruction;
  wire [ 3:0] slave_instruction;
  wire        slave_request, slave_ready;
  wire        bus_request, bus_grant;
  wire        mem_read, mem_write;
  wire        data_ready;

 processor proc1 (
   // main_bus ports
   .data(data),
   .address(address),
   .slave_instruction(slave_instruction),
   .slave_request(slave_request),
   .bus_grant(bus_grant),
   .mem_read(mem_read),
   .mem_write(mem_write),
   .bus_request(bus_request),
   .slave_ready(slave_ready),
   // other ports
   .jump_address(jump_address),
   .instruction(instruction),
   .clock(clock),
   .resetN(resetN),
   .test_mode(test_mode)
 );

 slave slave1 (
   // main_bus ports
   .data(data),
   .address(address),
   .bus_request(bus_request),
   .slave_ready(slave_ready),
   .mem_read(mem_read),
   .mem_write(mem_write),
   .slave_instruction(slave_instruction),
   .slave_request(slave_request),
   .bus_grant(bus_grant),
   .data_ready(data_ready),
   // other ports
   .clock(clock),
   .resetN(resetN)
 );

dual_port_ram ram (
  // main_bus ports
  .data(data),
  .data_ready(data_ready),
  .address(address),
  .mem_read(mem_read),
  .mem_write(mem_write),
  // other ports
  .program_address(program_address),
  .data_b(next_instruction)
);
```

signals for main_bus must be individually connected to each module instance

main_bus connections

main_bus connections

```
    test_generator test_gen(
    // main_bus ports
    .data(data),
    .address(address),
    .mem_read(mem_read),
    .mem_write(mem_write),
    // other ports
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );

  instruction_reg ir (
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .resetN(resetN)
  );
endmodule
```

main_bus connections

```
/********************* Module Definitions ********************/
module processor (
    // main_bus ports
    inout  wire [15:0] data,
    output reg  [15:0] address,
    output reg  [ 3:0] slave_instruction,
    output reg         slave_request,
    output reg         bus_grant,
    output wire        mem_read,
    output wire        mem_write,
    input  wire        bus_request,
    input  wire        slave_ready,
    // other ports
    output reg  [15:0] jump_address,
    input  wire [ 7:0] instruction,
    input  wire        clock,
    input  wire        resetN,
    input  wire        test_mode
  );
  ...  // module functionality code
endmodule
```

ports for main_bus must be individually declared in each module definition

```
module slave (
    // main_bus ports
    inout  wire [15:0] data,
    inout  wire [15:0] address,
    output reg         bus_request,
    output reg         slave_ready,
    output wire        mem_read,
    output wire        mem_write,
    input  wire [ 3:0] slave_instruction,
    input  wire        slave_request,
    input  wire        bus_grant,
    input  wire        data_ready,
    // other ports
    input  wire        clock,
    input  wire        resetN
  );
  ...  // module functionality code
endmodule
```

main_bus port declarations

```
module dual_port_ram (
    // main_bus ports
    inout  wire [15:0] data,
    output wire        data_ready,
    input  wire [15:0] address,
    input  tri0        mem_read,
    input  tri0        mem_write,
    // other ports
    input  wire [15:0] program_address,
    output reg  [ 7:0] data_b
  );
  ...  // module functionality code
endmodule
```

main_bus port declarations

```
module test_generator (
    // main_bus ports
    output wire [15:0] data,
    output reg  [15:0] address,
    output reg         mem_read,
    output reg         mem_write,
    // other ports
    input  wire        clock,
    input  wire        resetN,
    input  wire        test_mode
  );
  ...  // module functionality code
endmodule
```

main_bus port declarations

```
module instruction_reg (
    output reg  [15:0] program_address,
    output reg  [ 7:0] instruction,
    input  wire [15:0] jump_address,
    input  wire [ 7:0] next_instruction,
    input  wire        clock,
    input  wire        resetN
  );
  ...  // module functionality code
endmodule
```

3.2 Disadvantages of Verilog's module ports

Verilog's module ports provide a simple and intuitive way of describing the interconnections between the blocks of a design. In large, complex designs, however, Verilog's module ports have several shortcomings. Some of these are:

- Declarations must be duplicated in multiple modules.
- Communication protocols must be duplicated in several modules.
- There is a risk of mismatched declarations in different modules.
- A change in the design specification can require modifications in multiple modules.

One disadvantage of using Verilog's module ports to connect major blocks of a design together is readily apparent in the example code above. The signals that make up main_bus in the preceding example must be declared in each module that uses the bus, as well as in the top-level netlist that connects the design together. In this simple example, there are only a handful of signals in main_bus, so the redundant declarations are mostly just an inconvenience. In a large, complex design, however, this redundancy becomes much more than an inconvenience. A large design could have dozens of modules connected to the same bus, with dozens of duplicated declarations in each module. If the ports of one module should inadvertently be declared differently than the rest of the design, a functional error can occur that may be difficult to find.

The replicated port declarations also mean that, should the specification of the bus change during the design process, or in a next generation of the design, then each and every module that shares the bus must be changed. All netlists used to connect the modules using the bus must also be changed. This wide spread effect of a change is counter to good coding styles. One goal of coding is to structure the code in such a way that a small change in one place should not require changing other areas of the code. A weakness in the Verilog language is that a change to the ports in one module will usually require changes in other modules.

Another disadvantage of Verilog's module ports is that communication protocols must be duplicated in each module that utilize the interconnecting signals between modules. If, for example, three modules read and write from a shared memory device, then the read and write control logic must be duplicated in each of these modules. Yet another disadvantage of using module ports to connect the blocks of a design together is that detailed interconnections for the design must be determined very early in the design cycle. This is counter to the top-down design paradigm, where models are first written at an abstract level without extensive design detail. At an abstract level, an interconnecting bus should not require defining each and every signal that makes up the bus. Indeed, very early in the design specification, all that might be known is that the blocks of the design will share certain information. In the block diagram shown in Figure 10-1 on page 264, the main_bus is represented as a single connection. Using Verilog's module ports to connect the design blocks together, however, does not allow modeling at that same level of abstraction. Before any block of the design can be modeled, the bus must first be broken down to individual signals.

3.3 Advantages of SystemVerilog interfaces

SystemVerilog adds a powerful new port type to Verilog, called an interface. An interface allows a number of signals to be grouped together and represented as a single port. The declarations of the signals that make up the interface are contained in a single location. Each module that uses these signals then has a single port of the interface type, instead of many ports with the discrete signals. Example shows how SystemVerilog's interfaces can reduce the amount of code required to model the simple design shown in Figure. By encapsulating the signals that make up main_bus as an interface, the redundant declarations for these signals within each module are eliminated.

Example: SystemVerilog module interconnections using interfaces

```
/******************* Interface Definitions *******************/
interface main_bus;
  wire  [15:0] data;
  wire  [15:0] address;
  logic [ 7:0] slave_instruction;
  logic        slave_request;
  logic        bus_grant;
  logic        bus_request;
  logic        slave_ready;
  logic        data_ready;
  logic        mem_read;
  logic        mem_write;
endinterface

/********************** Top-level Netlist ********************/
module top (input logic clock, resetN, test_mode);
  logic [15:0] program_address, jump_address;
  logic [ 7:0] instruction, next_instruction;

  main_bus bus ( );  // instance of an interface
                     // (instance name is bus)

  processor proc1 (
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
    .jump_address(jump_address),
    .instruction(instruction),
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );

slave slave1 (
  // main_bus ports
  .bus(bus),  // interface connection
  // other ports
  .clock(clock),
  .resetN(resetN)
);
```

signals for main_bus are defined in just one place

each module instance has a single connection for main_bus

main_bus connections

```
dual_port_ram ram (
  // main_bus ports
  .bus(bus),  // interface connection
  // other ports
```

main_bus connections

```
/********************* Module Definitions ********************/
module processor (
    // main_bus interface port
    main_bus bus,  // interface port
    // other ports
    output logic [15:0] jump_address,
    input  logic [ 7:0] instruction,
    input  logic        clock,
    input  logic        resetN,
    input  logic        test_mode
  );
  ...  // module functionality code
endmodule
```

each module definition has a single port for main_bus

```
module slave (
    // main_bus interface port
    main_bus bus,  // interface port
    // other ports
    input  logic        clock,
    input  logic        resetN
  );
  ...  // module functionality code
endmodule
```

main_bus port declaration

```
    .program_address(program_address),
    .data_b(next_instruction)
  );

    test_generator test_gen(
    // main_bus ports
    .bus(bus),  // interface connection
    // other ports
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );

  instruction_reg ir (
    .program_address(program_address),
    .instruction(instruction),
    .jump_address(jump_address),
    .next_instruction(next_instruction),
    .clock(clock),
    .resetN(resetN)
  );
endmodule
```

main_bus connections

```
module dual_port_ram (
    // main_bus interface port
    main_bus bus,  // interface port
    // other ports
    input  logic [15:0] program_address,
    output logic [ 7:0] data_b
  );
  ...  // module functionality code
endmodule

module test_generator (
    // main_bus interface port
    main_bus bus,  // interface port
    // other ports
    input  logic        clock,
    input  logic        resetN,
    input  logic        test_mode
  );
  ...  // module functionality code
endmodule

module instruction_reg (
    output logic [15:0] program_address,
    output logic [ 7:0] instruction,
    input  logic [15:0] jump_address,
    input  logic [ 7:0] next_instruction,
    input  logic        clock,
    input  logic        resetN
  );
  ...  // module functionality code
endmodule
```

main_bus port declaration

main_bus port declaration

In example, above, all the signals that are in common between the major blocks of the design have been encapsulated into a single location—the interface declaration called main_bus. The top-level module and all modules that make up these blocks do not repetitively declare these common signals. Instead, these modules simply use the interface as the connection between them. Encapsulating common signals into a single location eliminates the redundant declarations of Verilog modules. Indeed, in the preceding example, since clock and resetN are also common to all modules, these signals could have also been brought into the interface.

3.4 SystemVerilog interface contents

SystemVerilog interfaces are far more than just a bundle of wires. Interfaces can encapsulate the full details of the communication between the blocks of a design. Using interfaces:

• The discrete signal and ports for communication can be defined in one location, the interface.

• Communication protocols can be defined in the interface.

• Protocol checking and other verification routines can be built directly into the interface.

With Verilog, the communication details must be duplicated in each module that shares a bus or other communication architecture. SystemVerilog allows all the information about a

communication architecture and the usage of the architecture to be defined in a single, common location. An interface can contain type declarations, tasks, functions, procedural blocks, program blocks, and assertions. SystemVerilog interfaces also allow multiple views of the interface to be defined. For example, for each module connected to the interface, the data_bus signal can be defined to be an input, output or bidirectional port.

3.5 Differences between modules and interfaces

There are three fundamental differences that make an interface differ from a module. First, an interface cannot contain design hierarchy. Unlike a module, an interface cannot contain instances of modules or primitives that would create a new level of implementation hierarchy. Second, an interface can be used as a module port, which is what allows interfaces to represent communication channels between modules. It is illegal to use a module in a port list. Third, an interface can contain modports, which allow each module connected to the interface to see the interface differently.

3.6 Interface declarations

Syntactically, the definition of an interface is very similar to the definition of a module. An interface can have ports, just as a module does. This allows signals that are external to the interface, such as a clock or reset line, to be brought into the interface and become part of the bundle of signals represented by the interface. Interfaces can also contain declarations of any Verilog or SystemVerilog type, including all variable types, all net types and user-defined types. Example 10-3 shows a definition for an interface called main_bus, with three external signals coming into the interface: clock, resetN and test_mode. These external signals can now be connected to each module through the interface, without having to explicitly connect the signals to each module. Notice in this example how the instance of interface main_bus has the clock, resetN and test_mode signals connected to it, using the same syntax as connecting signals to an instance of a module.

Example: The interface definition for main_bus, with external inputs

```
/****************** Interface Definitions ******************/
interface main_bus (input logic clock, resetN, test_mode);
  wire  [15:0] data;
  wire  [15:0] address;
  logic [ 7:0] slave_instruction;
  logic        slave_request;
  logic        bus_grant;
  logic        bus_request;
  logic        slave_ready;
  logic        data_ready;
  logic        mem_read;
  logic        mem_write;
endinterface
```

discrete signals are inputs to the interface

```
/********************** Top-level Netlist ******************/
module top (input logic clock, resetN, test_mode);
  logic [15:0] program_address, jump_address;
  logic [ 7:0] instruction, next_instruction;

  main_bus bus (    // instance of an interface
    .clock(clock),
    .resetN(resetN),
    .test_mode(test_mode)
  );

processor proc1 (
  // main_bus ports
  .bus(bus),  // interface connection
  // other ports
  .jump_address(jump_address),
  .instruction(instruction)
);

  ...

/*** remainder of netlist and module definitions are not  ***/
/*** listed - they are similar to example 10-2, but       ***/
/*** clock and resetN do not need to be passed to each    ***/
/*** module instance as discrete ports.                   ***/
```

discrete signals are connected to the interface instance

discrete signals do not need to be connected to each design block instance

3.7 Using interfaces as module ports

With SystemVerilog, a port of a module can be declared as an interface type, instead of the Verilog input, output or inout port directions.

A module port can be explicitly declared as a specific type of interface. This is done by using the name of an interface as the port type. The syntax is:

```
module <module_name> (<interface_name> <port_name>);
```

For example:

```
interface chip_bus;
  ...
endinterface


module CACHE (chip_bus pins,  // interface port
              input    clock);
  ...
endmodule
```

An explicitly named interface port can only be connected to an interface of the same name. An error will occur if any other interface definition is connected to the port. Explicitly named interface ports ensure that a wrong interface can never be inadvertently connected to the port. Explicitly naming the interface type that can be connected to the port also serves to document directly within the port declaration exactly how the port is intended to be used.

3.8 Instantiating and connecting interfaces

An instance of an interface is connected to a port of a module instance using a port connection, just as a discrete net would be connected to a port of a module instance. This

requires that both the interface and the modules to which it is connected be instantiated. The syntax for an interface instance is the same as for a module instance. If the definition of the interface has ports, then signals can be connected to the interface instance, using either the port order connection style or the named port connection style, just as with a module instance.

A module input, output or inout port can be left unconnected on a module instance. This is not the case for an interface port. A port that is declared as an interface, whether generic or explicit, must be connected to an interface instance or another interface port. An error will occur if an interface port is left unconnected. On a module instance, a port that has been declared as an interface type must be connected to an interface instance, or another interface port that is higher up in the hierarchy. If a port declaration has an explicitly named interface type, then it must be connected to an interface instance of the identical type. If a port declaration has a generic interface type, then it can be connected to an interface instance of any type.

3.9 Interface modports

Interfaces provide a practical and straightforward way to simplify connections between modules. However, each module connected to an interface may need to see a slightly different view of the connections within the interface. For example, to a slave on a bus, an interrupt_request signal might be an output from the slave, whereas to a processor on the same bus, interrupt_request would be an input.

SystemVerilog interfaces provide a means to define different views of the interface signals that each module sees on its interface port. The definition is made within the interface, using the modport keyword. Modport is an abbreviation for module port. A modport definition describes the module ports that are represented by the interface. An interface can have any number of modport definitions, each describing how one or more other modules view the signals within the interface.

A modport defines the port direction that the module sees for the signals in the interface. Examples of two modport declarations are:

```
interface chip_bus (input logic clock, resetN);
  logic interrupt_request, grant, ready;
  logic [31:0] address;
  wire  [63:0] data;

  modport master (input  interrupt_request,
                  input  address,
                  output grant, ready,
                  inout  data,
                  input  clock, resetN);

   modport slave  (output interrupt_request,
                   output address,
                   input  grant, ready,
                   inout  data,
                   input  clock, resetN);
endinterface
```

The modport definitions do not contain vector sizes or types. This information is defined as part of the signal type declarations in the interface. The modport declaration only defines whether the connecting module sees a signal as an input, output, bidirectional inout, or ref port.

3.10 Using tasks and functions in interfaces:

Interfaces can encapsulate the full details of the communication protocol between modules. For instance, the main_bus protocol in the previous example includes handshaking signals between the master processor and the slave processor. In regular Verilog, the master processor module would need to contain the procedural code to assert and de-assert its handshake signals at the appropriate time, and to monitor the slave handshake inputs. Conversely, the slave processor would need to contain the procedural code to assert and de-assert its handshake signals, and to monitor the handshake inputs coming from the master processor or the RAM.

Describing the bus protocol within each module that uses a bus leads to duplicated code. If any change needs to be made to the bus protocol, the code for the protocol must be changed in each and every module that shares the bus.

3.10.1 Interface methods:

SystemVerilog allows tasks and functions to be declared within an interface. These tasks and functions are referred to as interface methods. A task or function that is defined within an interface is written using the same syntax as if it had been within a module, and can contain the same types of statements as within a module. These interface methods can operate on any signals within the interface. Values can be passed in to interface methods from outside the interface as input arguments. Values can be written back from interface methods as output arguments or function returns.

Interface methods offer several advantages for modeling large designs. Using interface methods, the details of communication from one module to another can be moved to the interface. The code for communicating between modules does not need to be replicated in each module. Instead, the code is only written once, as interface methods, and shared by each module connected using the interface. Within each module, the interface methods are called, instead of implementing the communication protocol functionality within the module. Thus, an interface can be used not only to encapsulate the data connecting modules, but also the communication protocols between the modules.

3.10.2 Importing interface methods

If the interface is connected via a modport, the method must be specified using the import keyword. The import definition is specified within the interface, as part of a modport definition. Modports specify interface information from the perspective of the module. Hence, an import declaration within a modport indicates that the module is importing the task or function.

The import declaration can be used in two ways:

• Import using just the task or function name

• Import using a full prototype of the task or function

Import using a task or function name: The simplest form of importing a task or function is to simply specify the name of the task or function. The basic syntax is:

```
modport ( import <task_function_name> );
```

An example of using this style is:

```
modport in  (import Read,
             import parity_gen,
             input  clock, resetN );
```

The second style of an import declaration is to specify a full prototype of the task or function arguments. This style requires that the keyword task or function follow the import keyword. It also requires that the task or function name be followed by a set of parentheses, which contain the formal arguments of the task or funciton. The basic syntax of this style of import declarations is:

```
modport (import task <task_name>(<task_formal_arguments) );

modport (import function <function_name> (<formal_args>) );
```

For example:

```
modport in (import task Read
                      (input  [63:0] data,
                       output [31:0] address),
            import function parity_gen
                      (input [63:0] data),
            input  clock, resetN);
```

A full prototype can serve to document the arguments of the task or function directly as part of the modport declaration. This additional code documentation can be convenient if the actual task or function is defined in a package, and therefore the definition is not in the package source code for easy visual reference.

3.11 Exporting tasks and functions

SystemVerilog interfaces and modports provide a mechanism to define a task or function in one module, and then export the task or function through an interface to other modules.

Exporting tasks or functions into an interface is not synthesizable. This modeling style should be reserved for abstract models that are not intended to be synthesized. An export declaration in an interface modport does not require a full prototype of the task or function arguments. Only the task or function name needs to be listed in the modport declaration.

If an exported task or function has default values for any of its formal arguments, then each import declaration of the task or function must have a complete prototype of the task/function arguments. A full prototype for the import declaration is also required if the task or function call uses named argument passing instead of passing by position.

The code fragments in example 10-10 show a function called check that is declared in module CPU. The function is exported from the CPU through the master modport of the chip_bus interface. The same function is imported into any modules that use the slave modport of the interface. To any module connected to the slave modport, the check function appears to be part of the interface, just like any other function imported from an interface. Modules using the slave modport do not need to know the actual location of the check function definition.

```
interface chip_bus (input logic clock, resetN);
  logic        request, grant, ready;
  logic [63:0] address, data;

  modport master (output request, ...
                  export check );

  modport slave  (input  request, ...
                  import check );
endinterface

module CPU (chip_bus.master io);

  function check (input parity, input [63:0] data);
    ...
  endfunction
   ...
endmodule
```

Exporting a task or function to the entire interface: The export declaration allows a module to export a task or function to an interface through a specific modport of the interface. A task or function can also be exported to an interface without using a modport. This is done by declaring an extern prototype of the task or function within the interface.

Example: Exporting a function from a module into an interface

```
interface chip_bus (input logic clock, resetN);
  logic        request, grant, ready;
  logic [63:0] address, data;

  extern function check(input logic        parity,
                        input logic [63:0] data);

  modport master (output request, ...);

  modport slave  (input  request, ...
                  import function check
                                 (input logic        parity,
```

```
                                    input logic [63:0] data) );
endinterface

module CPU (chip_bus.master io);

  function check (input logic parity, input logic [63:0] data);
    ...
  endfunction
   ...
endmodule
```

Restrictions on exporting tasks and functions: SystemVerilog places a restriction on exporting functions through interfaces. It is illegal to export the same function name from two different modules, or two instances of the same module, into the same interface. For example, module A and module B cannot both export a function called check into the same interface.

SystemVerilog places a restriction on exporting tasks through interfaces. It is illegal to export the same task name from two different modules, or two instances of the same module, into the same interface, unless an extern forkjoin declaration is used. The multiple export of a task corresponds to a multiple response to a broadcast. Tasks can execute concurrently, each taking a different amount of time to execute statements, and each call returning different values through its outputs. The concurrent response of modules A and B containing a call to a task called task1 is conceptually modeled by

```
fork
  <hierarchical_name_of_module_A>.task1(q, r);
  <hierarchical_name_of_module_B>.task1(q, r);
join
```

Because an interface should not contain the hierarchical names of the modules to which it is connected, the task is declared as extern forkjoin, which infers the behavior of the fork...join block above. If the task contains outputs, it is the last instance of the task to finish that determines the final output value.

This construct can be useful for abstract, non-synthesizable transaction level models of busses that have slaves, where each slave determines its own response to broadcast signals. The **extern forkjoin** can also be used for configuration purposes, such as counting the number of modules connected to an interface. Each module would export the same task, name which increments a counter in the interface.

3.12 System Verilog Event Scheduling:

This section gives an overview of the interactions and behavior of SystemVerilog elements, especially with respect to the scheduling and execution of events. Updates to IEEE STD 1800-20051 divide the SystemVerilog time slot into 17 ordered regions, nine ordered regions for the execution of SystemVerilog statements and eight ordered regions for the execution of PLI code. The purpose of dividing a time slot into these ordered regions is to provide predictable interactions between the design and testbench code.

Every change in the state of a net or variable in the system description being simulated is considered an update event. When an update event is executed, all the processes that are sensitive to those events are considered for evaluation known as an evaluation event. Examples of processes include, initial, always, always_comb, always_latch, and always_ff procedural blocks, continuous assignments, asynchronous tasks, and procedural assignment statements. A single time slot is divided into multiple regions where events can be scheduled. This event scheduling supports obtaining clear and predictable interactions that provide for the ordering of particular types of execution. This allows properties and checkers to sample data when the design under test is in a stable state. Property expressions can be safely evaluated, and testbenches can react to both properties and checkers with zero delay, all in a predictable manner. This same mechanism also allows for non-zero delays in the design, clock propagation, and/or stimulus and response code to be mixed freely and consistently with cycle-accurate descriptions.

The term simulation time is used to refer to the time value maintained by the simulator to model the actual time it would take for the system description being simulated. A time slot includes all simulation activity that is processed in the event regions for each simulation time SystemVerilog event Regions The new SystemVerilog event regions are developed to support new SystemVerilog constructs and also to prevent race conditions being created between the RTL design and the new verification constructs.

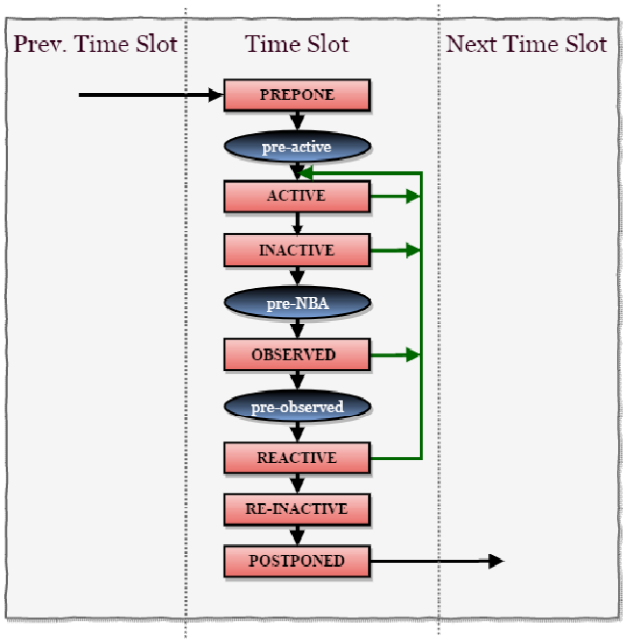

Figure 3.2 System Verilog Event Scheduling

These new regions guarantee predictability and consistency between design, testbenches, and assertions

Preponed region: The values of variables that are used in concurrent assertions are sampled in the Preponed region. (Evaluation is done at observed region). Preponed region is executed only once in each time slot, immediately after advancing simulation time.

Pre-active region: The Pre-active region is specifically for a PLI callback control point that allows for user code to read and write values and create events before events in the Active region are evaluated

Active region: The Active region holds current events being evaluated and can be processed in any order.

- Execute all module blocking assignments.
- Evaluate the Right-Hand-Side (RHS) of all nonblocking assignments and schedule updates into the NBA region.
- Execute all module continuous assignments
- Evaluate inputs and update outputs of Verilog primitives.
- Execute the $display and $finish commands

Inactive region: The Inactive region holds the events to be evaluated after all the active events are processed. In this region #0 blocking assignments are scheduled.

Pre-NBA region: The Pre-NBA region is specifically for a PLI callback control point that allows for user code to read and write values and create events before the events in the NBA region are evaluated

Non-blocking Assignment Events region (NBA): The principal function of this region is to execute the updates to the Left-Hand-Side (LHS) variables that were scheduled in the Active region for all currently executing nonblocking assignments.

Post-NBA region: The Post-NBA region is specifically for a PLI callback control point that allows for user code to read and write values and create events after the events in the NBA region are evaluated

Observed region: The principal function of this region is to evaluate the concurrent assertions using the values sampled in the Preponed region. A criterion behind this decision is that the property evaluations must only occur once in any clock triggering time slot. During the property evaluation, the pass/fail code shall be scheduled in the Reactive region of the current time slot.

Post-observed region: The Post-observed region is specifically for a PLI callback control point that allows for user code to read values after properties are evaluated (in Observed or earlier region).

Reactive region: The code specified in the program block, and pass/fail code from property expressions, are scheduled in the Reactive region. The principal function of this region is to evaluate and execute all current program activity in any order

- Execute all program blocking assignments.
- Execute the pass/fail code from concurrent assertions.
- Evaluate the Right-Hand-Side (RHS) of all program nonblocking assignments and schedule

- Execute all program continuous assignments
- Execute the $exit and implicit $exit commands

Re-Inactive Events region: In this region #0 blocking assignments in a program process are scheduled.

Postponed Region: The principal function of this region is to execute the $strobe and $monitor commands that will show the final updated values for the current time slot. This region is also used to collect functional coverage for items that use strobe sampling.

3.13 Verification with interfaces

Using only Verilog-style module ports, without interfaces, a typical design and verification paradigm is to develop and test each module of a design, independent of other modules in the design. After each module is independently verified, the modules are connected together to test the communication between modules. If there is a problem with the communication protocols, it may be necessary to make design changes to multiple modules.

Interfaces enable a different paradigm for verification. With interfaces, the communication channels can be developed as interfaces independently from other modules. Since an interface can contain methods for the communication protocols, the interface can be tested and verified independent of the rest of the design. Modules that use the interface can be written knowing that the communication between modules has already been verified.

Reference Books:

1. Advanced Digital Design With the Verilog HDL, Michael D. Ciletti, 2nd Edition, PHI, ISBN: 978–0– 07–338054–4 2015.
2. Digital Systems Design Using Verilog, Charles Roth, Lizy K. John, Byeong KilLee, Cengage Learning, ISBN-10: 1285051076, 2015.
3. Fundamentals of Digital Logic with Verilog Design, Stephen Brown and Zvonko Vranesic, 6th Edition, McGraw Hill publication, ISBN: 978–0–07–338054–4, 2014.
4. System Verilog for Design - A Guide to Using System Verilog for Hardware Design and Modeling, Stuart Sutherland, Simon David mann and Peter Flake, 2E, Springer Science, ISBN-13: 978-0387-3339-91, 2006.
5. System Verilog for Verification-A Guide to Learning the Testbench Language Features, C Spear, Springer Science, IEEE press, ISBN-13: 978-0387-2703-64,2006.
6. System Verilog golden reference guide-A concise guide to System Verilog Doulos, IEEE Standard-1800- 2009, Version 5.0,ISBN: 0-9547345-9-9, 2012.
7. Step-by-Step Functional Verification with System Verilog and OVM, SasanIman, Hansen Brown Publishing Company,ISBN-13: 978-0-9816-5621-2, 2008.

Questions to Practice:

PART -A

1. Identify how System Verilog is is used in Interface signals
2. Identify the Limitations of Verilog interface signals.
3. Describe in detail about Interface Instantiation
4. Classify the use of tasks and functions in interface
5. Explain in detail about Modports in System Verilog

PART-B

1. Demonstrate in detail about Interface and modules in System Verilog
2. Classify Interface types that are used in System Verilog
3. Demonstrate in detail about the System Verilog Event Scheduler
4. Demonstrate in detail about the impact of tasks and Functions in Interface
5. Enumerate Interface methods and Exporting Tasks & Functions in detail.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – IV – CONSTRAINED RANDOMIZATION– SECA3021

UNIT-IV (CONSTRAINED RANDOMIZATION)

Random Variables - rand and randc, Randomize( ) Method - Pre/Post Randomize( ) methods, Constraints in the class, Rand_mode and constraint_mode, Constraint and Inheritance, Constraint Overriding, Set Membership, Distribution Constraints, Conditional Constraints - .implication (->), if/else, Inline Constraints

As designs grow larger, it becomes more difficult to create a complete set of stimuli needed to check their functionality. You can write a directed test case to check a certain set of features, but you cannot write enough directed test cases when the number of features keeps doubling on each project. Worse yet, the interactions between all these features are the source for the most devious bugs and are the least likely to be caught by going through a laundry list of features.

The solution is to create test cases automatically using constrained-random tests (CRT). A directed test finds the bugs you think are there, but a CRT finds bugs you never thought about, by using random stimulus. You restrict the test scenarios to those that are both valid and of interest by using constraints. Creating a CRT environment takes more work than creating one for directed tests. A simple directed test just applies stimulus, and then you manually check the result. These results are captured as a golden log file and compared with future simulations to see whether the test passes or fails.

A CRT environment needs not only to create the stimulus but also to predict the result, using a reference model, transfer function, or other techniques. However, once this environment is in place, you can run hundreds of tests without having to hand-check the results, thereby improving your productivity. This trade-off of test-authoring time (your work) for CPU time (machine work) is what makes CRT so valuable.

4.1 Randomize

When you think of randomizing the stimulus to a design, the first thing you may think of are the data fields. These are the easiest to create – just call $random. The problem is that this approach has a very low payback in terms of bugs found: you only find data-path bugs, perhaps with bit-level mistakes. The test is still inherently directed. The challenging bugs are in the control logic. As a result, you need to randomize all decision points in your DUT. Wherever control paths diverge, randomization increases the probability that you'll take a different path in each test case.

You need to think broadly about all design input such as the following:

- Device configuration
- Environment configuration
- Primary input data
- Encapsulated input data

- Protocol exceptions
- Delays
- Transaction status
- Errors and violations

4.1.1 Device configuration:

Over time, in a real world environment, the DUT's configuration becomes more and more random. For example, a verification engineer had to verify a time-division multiplexor switch that had 600 input channels and 12 output channels. When the device was installed in the end-customer's system, channels would be allocated and deallocated over and over. At any point in time, there would be little correlation between adjacent channels. In other words, the configuration would seem random. To test this device, the verification engineer had to write several dozen lines of Tcl code to configure each channel. As a result, she was never able to try configurations with more than a handful of channels enabled. Using a CRT methodology, she wrote a testbench that randomized the parameters for a single channel, and then put this in a loop to configure the whole device. Now she had confidence that her tests would uncover bugs that previously would have been missed.

4.1.2 Environment configuration

The device that you are designing operates in an environment containing other devices. When you are verifying the DUT, it is connected to a testbench that mimics this environment. You should randomize the entire environment, including the number of objects and how they are configured. Another company was creating an I/O switch chip that connected multiple PCI buses to an internal memory bus. At the start of simulation the customer used randomization to choose the number of PCI buses (1–4), the number of devices on each bus (1–8), and the parameters for each device (master or slave, CSR addresses, etc.). Even though there were many possible combinations, this company knew all had been covered.

4.1.3 Primary input data

This is what you probably thought of first when you read about random stimulus: take a transaction such as a bus write or ATM cell and fill it with some random values. How hard can that be? Actually it is fairly straightforward as long as you carefully prepare your transaction classes. You should anticipate any layered protocols and error injection.

4.1.4 Encapsulated Input Data

Many devices process multiple layers of stimulus. For example, a device may create TCP traffic that is then encoded in the IP protocol, and finally sent out inside Ethernet packets. Each level has its own control fields that can be randomized to try new combinations. So you are randomizing the data and the layers that surround it. You need to write constraints that create valid control fields but that also allow injecting errors.

4.1.5 Protocol Exceptions, Errors, and Violations

Anything that can go wrong, will, eventually. The most challenging part of design and verification is how to handle errors in the system. You need to anticipate all the cases where things can go wrong, inject them into the system, and make sure the design handles them gracefully, without locking up or going into an illegal state. A good verification engineer tests the behavior of the design to the edge of the functional specification and sometimes even beyond. When two devices communicate, what happens if the transfer stops partway through? Can your testbench simulate these breaks? If there are error detection and correction fields, you must make sure all combinations are tried. The random component of these errors is that your testbench should be able to send functionally correct stimuli and then, with the flip of a configuration bit, start injecting random types of errors at random intervals.

4.1.6 Delays:

Many communication protocols specify ranges of delays. The bus grant comes one to three cycles after request. Data from the memory is valid in the fourth to tenth bus cycle. However, many directed tests, optimized for the fastest simulation, use the shortest latency, except for that one test that only tries various delays. Your testbench should always use random, legal delays during every test to try to find that (hopefully) one combination that exposes a design bug. Below the cycle level, some designs are sensitive to clock jitter. By sliding the clock edges back and forth by small amounts, you can make sure your design is not overly sensitive to small changes in the clock cycle. The clock generator should be in a module outside the testbench so that it creates events in the Active region along with other design events. However, the generator should have parameters such as frequency and offset that can be set by the testbench during the configuration phase. (Note that you are looking for functional errors, not timing errors. Your testbench should not try to violate setup and hold requirements. These are better validated using timing analysis tools.)

4.2 Randomization in SystemVerilog

The random stimulus generation in SystemVerilog is most useful when used with OOP. You first create a class to hold a group of related random variables, and then have the random-solver fill them with random values. You can create constraints to limit the random values to legal values, or to test-specific features. Note that you can randomize individual variables, but this case is the least interesting. True constrained-random stimuli is created at the transaction level, not one value at a time.

4.2.1 Simple Class with Random Variables

Example 6.1 shows a packet class with random variables and constraints, plus testbench code that constructs and randomizes a packet.

```
class Packet;
  // The random variables
  rand bit [31:0] src, dst, data[8];
  randc bit [7:0] kind;
  // Limit the values for src
  constraint c {src > 10;
                src < 15;}
endclass

Packet p;
initial begin
  p = new();// Create a packet
  assert (p.randomize())
  else $fatal(0, "Packet::randomize failed");
  transmit(p);
end
```

This class has four random variables. The first three use the rand modifier, so that every time you randomize the class, the variables are assigned a value. Think of rolling dice: each roll could be a new value or repeat the current one. The kind variable is randc, which means random cyclic, so that the random solver does not repeat a random value until every possible value has been assigned. Think of dealing cards from a deck: you deal out every card in the deck in random order, then shuffle the deck, and deal out the cards in a different order. Note that the cyclic pattern is for a single variable.

A randc array with eight elements has eight different patterns. A constraint is just a set of relational expressions that must be true for the chosen value of the variables. In this example, the src variable must be greater than 10 and less than 15. Note that the constraint expression is grouped using curly braces: {}. This is because this code is declarative, not procedural, which uses begin...end.

The randomize() function returns 0 if a problem is found with the constraints. The procedural assertion is used to check the result. This example uses a $fatal to stop simulation, but the rest of the book leaves out this extra code. You need to find the tool-specific switches to force the assertion to terminate simulation. This book uses assert to test the result from randomize(), but you may want to test the result, call your special routine that prints any useful information and then gracefully shut down the simulation.

You should not randomize an object in the class constructor. Your test may need to turn constraints on or off, change weights, or even add new constraints before randomization. The constructor is for initializing the object's variables, and if you called randomize() at this early stage, you might end up throwing away the results.

All variables in your classes should be random and public. This gives your test the maximum control over the DUT's stimulus and control. You can always turn off a random variable. If you forget to make a variable random, you must edit the environment, which you want to avoid.

4.2.2 Checking the Result from Randomization

The randomize() function assigns random values to any variable in the class that has been labeled as rand or randc, and also makes sure that all active constraints are obeyed. Randomization can fail if your code has conflicting constraints (see next section), and so you should always check the status. If you don't check, the variables may get unexpected values, causing your simulation to fail.

Example checks the status from randomize() by using a procedural assertion. If randomization succeeds, the function returns 1. If it fails, randomize() returns 0. The assertion checks the result and prints an error if there was a failure. You should set your simulator's switches to terminate when an error is found. Alternatively, you might want to call a special routine to end simulation, after doing some housekeeping chores like printing a summary report.

4.3 Constraint:

Useful stimulus is more than just random values – there are relationships between the variables. Otherwise, it may take too long to generate interesting stimulus values, or the stimulus might contain illegal values. You define these interactions in SystemVerilog using constraint blocks that contain one or more constraint expressions. SystemVerilog chooses random values so that the expressions are true.

At least one variable in each expression should be random, either rand or randc. The following class fails when randomized, unless age happens to be in the right range. The solution is to add the modifier rand or randc before age.

```
class Child;
  bit [31:0] age;  // Error Ð should be rand or randc
  constraint c_teenager {age > 12;
                         age < 20;}
endclass
```

The randomize() function tries to assign new values to random variables and to make sure all constraints are satisfied. In above example, since there are no random variables, randomize() just checks the value of son to see if it is in the bounds specified by the constraint c_teenager. Unless the variable happens to fall in the range of 13:19, randomize() fails. While you can use a constraint to check that a nonrandom variable has a valid value, use an assert or if-statement instead. It is much easier to debug your procedural checker code than read through an error message from the random solver.

The below Example shows a simple class with random variables and constraints

```
class Stim;
  const bit [31:0] CONGEST_ADDR = 42;
  typedef enum {READ, WRITE, CONTROL} stim_e;
  randc stim_e kind;   // Enumerated var
  rand bit [31:0] len, src, dst;
  bit congestion_test;

  constraint c_stim {
    len < 1000;
    len > 0;
    if (congestion_test) {
      dst inside {[CONGEST_ADDR-100:CONGEST_ADDR+100]};
      src == CONGEST_ADDR;
    }
    else
      src inside {0, [2:10], [100:107]};
  }
endclass
```

4.3.1 Simple Expressions

Example showed a constraint block with several expressions. The first two control the values for the len variable. As you can see, a variable can be used in multiple expressions. There can be a maximum of only one relational operator (<, <=, ==, >=, or >) in an expression. Below Sample incorrectly tries to generate three variables in a fixed order.

```
class order;
  rand bit [7:0] lo, med, hi;
  constraint bad  {lo < med < hi;} // Gotcha!
endclass
```

Result:

```
lo =  20, med = 224, hi = 164
lo = 114, med =  39, hi = 189
lo = 186, med = 148, hi = 161
lo = 214, med = 223, hi = 201
```

Example shows the results, which are not what was intended. The constraint bad in Sample is broken down into multiple binary relational expressions, going from left to right: ((lo < med) < hi). First, the expression (lo < med) is evaluated, which gives 0 or 1. Then hi is constrained to be greater than the result. The variables lo and med are randomized but not constrained.

```
class order;
  rand bit [15:0] lo, med, hi;
  constraint good {lo < med;   // Only use binary constraints
                   med < hi;}
endclass
```

4.3.2 Equivalence Expressions

The most common mistake with constraints is trying to make an assignment in a constraint block, but it can only contain expressions. Instead, use the equivalence operator to set a random variable to a value, e.g., len==42. You can build complex relationships between one or more random variables, such as len == header.addr_mode * 4 + payload.size().

4.3.3 Weighted Distributions

The dist operator allows you to create weighted distributions so that some values are chosen more often than others. The dist operator takes a list of values and weights, separated by the := or the :/ operator. The values and weights can be constants or variables. The values can be a single value or a range such as [lo:hi]. The weights are not percentages and do not have to add up to 100. The := operator specifies that the weight is the same for every specified value in the range, whereas the :/ operator specifies that the weight is to be equally divided between all the values.

```
rand int src, dst;
constraint c_dist {
   src dist {0:=40, [1:3]:=60};
   // src = 0, weight = 40/220
   // src = 1, weight = 60/220
   // src = 2, weight = 60/220
   // src = 3, weight = 60/220

   dst dist {0:/40, [1:3]:/60};
   // dst = 0, weight = 40/100
   // dst = 1, weight = 20/100
   // dst = 2, weight = 20/100
   // dst = 3, weight = 20/100
}
```

In example, src gets the value 0, 1, 2, or 3. The weight of 0 is 40, whereas, 1, 2, and 3 each have the weight of 60, for a total of 220. The probability of choosing 0 is 40/220, and the probability of choosing 1, 2, or 3 is 60/220 each. Next, dst gets the value 0, 1, 2, or 3. The weight of 0 is 40, whereas 1, 2, and 3 share a total weight of 60, for a total of 100. The probability of choosing 0 is 40/100, and the probability of choosing 1, 2, or 3 is only 20/100 each. Once again, the values and weights can be constants or variables. You can use variable weights to change distributions on the fly or even to eliminate choices by setting the weight to zero, as shown in Sample.

In Sample, the len enumerated variable has three values. With the default weighting values, longword lengths are chosen more often, as w_lwrd has the largest value.

```
// Bus operation, byte, word, or longword
class BusOp;
  // Operand length
  typedef enum {BYTE, WORD, LWRD } length_e;
  rand length_e len;

  // Weights for dist constraint
  bit [31:0] w_byte=1, w_word=3, w_lwrd=5;

  constraint c_len {
    len dist {BYTE := w_byte,      // Choose a random
              WORD := w_word,      // length using
              LWRD := w_lwrd};     // variable weights
  }
endclass
```

4.3.4 Set Membership and the Inside Operator

You can create sets of values with the inside operator. The SystemVerilog solver chooses between the values in the set with equal probability, unless you have other constraints on the variable. As always, you can use variables in the sets.

```
rand int c;             // Random variable
int lo, hi;             // Non-random variables used as limits
constraint c_range {
  c inside {[lo:hi]};   // lo <= c && c <= hi
}
```

In Sample, SystemVerilog uses the values for lo and hi to determine the range of possible values. You can use this to parameterize your constraints so that the testbench can alter the behavior of the stimulus generator without rewriting the constraints. Note that if lo > hi, an empty set is formed, and the constraint fails. You can use $ as a shortcut for the minimum and maximum values for a range, as shown in Sample. This is helpful when you are building constraints for variables with different ranges.

```
rand bit [6:0] b;           // 0 <= b <= 127
rand bit [5:0] e;           // 0 <= e <= 63
constraint c_range {
  b inside {[$:4], [20:$]}; // 0 <= b <= 4 || 20 <= b <= 127
  e inside {[$:4], [20:$]}; // 0 <= e <= 4 || 20 <= e <= 63
}
```

4.3.5 Using an Array in a Set

You can choose from a set of values by storing them in an array.

```
rand int f;
int fib[5] = õ{1,2,3,5,8};
constraint c_fibonacci {
  f inside fib;
}
```

This is expanded into the following set of constraints:

Sample 6.13 Equivalent set of constraints

```
constraint c_fibonacci {
  (f == fib[0]) ||    // f==1
  (f == fib[1]) ||    // f==2
  (f == fib[2]) ||    // f==3
  (f == fib[3]) ||    // f==5
  (f == fib[4]);      // f==8
}
```

All values in the set are chosen equally, even if they appear multiple times. You can also think of the inside constraint as being turned into a foreach constraint, as explained.

Example: Repeated values in inside constraint

```
class Weighted;
  rand int val;
  int array[] = '{1,1,2,3,5,8,8,8,8,8};
  constraint c {val inside array;}
endclass

Weighted w;
initial begin
  int count[9], maxx[$];
  w = new();

  repeat (2000) begin
    assert(w.randomize());
    count[w.val]++;          // Count the number of hits
   end

  maxx = count.max();      // Get largest value in count

  // Print histogram of count
  foreach(count[i])
  if (count[i]) begin
    $write("count[%0d]=%5d ", i, count[i]);
    repeat (count[i]*40/maxx[0]) $write("*");
    $display;
  end
end
```

Output:

```
count[1]= 3941 ****************************************
count[2]= 4038 ****************************************
count[3]= 3978 ****************************************
count[5]= 4027 ****************************************
count[8]= 4016 ****************************************
```

4.3.6 Conditional Constraints

Normally, all constraint expressions are active in a block. For example, a bus supports byte, word, and longword reads, but only longword writes. SystemVerilog supports two implication operators, -> and if-else. When you are choosing from a list of expressions, such as an enumerated type, the implication operator, ->, lets you create a case-like block. The parentheses around the expression are not required, but do make the code easier to read.

```
class BusOp;
  ...
  constraint c_io {
    (io_space_mode) ->
      addr[31] == 1'b1;
  }
```

If you have a true-false expression, the `if-else` operator may be better.

Sample 6.20 Constraint block with `if-else` operator

```
class BusOp;
  ...
  constraint c_len_rw {
    if (op == READ)
      len inside {[BYTE:LWRD]};
    else
      len == LWRD;
  }
```

In constraint blocks, you use curly braces, { }, to group multiple expressions. The begin...end keywords are for procedural code.

4.37 Bidirectional Constraints

By now you may have realized that constraint blocks are not procedural code, executing from top to bottom. They are declarative code, all active at the same time. If you constrain a variable with the inside operator with the set [10:50] and have another expression that constrains the variable to be greater than 20, SystemVerilog solves both constraints simultaneously and only chooses values between 21 and 50. SystemVerilog constraints are bidirectional, which means that the constraints on all random variables are solved concurrently. Adding or removing a constraint on any one variable affects the value chosen for all variables that are related directly or indirectly. Consider the constraint in Sample.

```
rand logic [15:0] r, s, t;
constraint c_bidir {
  r < t;
  s == r;
  t < 30;
  s > 25;
}
```

The SystemVerilog solver looks at all four constraints simultaneously. The variable r has to be less than t, which has to be less than 30. However, r is also constrained to be equal to s, which is greater than 25. Even though there is no direct constraint on the lower value of t, the constraint on s restricts the choices. Table 6-1 shows the possible values for these three variables.

Even the conditional constraints such as -> and if...else, which can look like a procedural if-else statement, are bidirectional. For example, the constraint {(a==1) - > (b==0)} is equivalent to {!(a == 1) || b == 0;}. The solver picks values for the variables that meet this constraint, and does not first check if a==1, then force b==0. In fact, if you add the additional constraint {b==1;}, the solver will set a to 0.

4.3.8 Implication and Bidirectional Constraints

Note that the implication operator says that when x==0, y is forced to 0, but when y==0, there is no constraint on x. However, implication is bidirectional in that if y were forced to a nonzero value, x would have to be 1. Sample 6.26 has the constraint y>0, and so x can never be 0.

```
class Imp2;
  rand bit x;          // 0 or 1
  rand bit [1:0] y;    // 0, 1, 2, or 3
  constraint c_xy {
     y > 0;
    (x==0) -> y==0;
  }
endclass
```

4.4 Controlling Multiple Constraint Blocks

A class can contain multiple constraint blocks. One might make sure you have a valid transaction, but you might need to disable this when testing the DUT's error handling. Or you might want to have a separate constraint for each test. Perhaps one constraint would restrict the data length to create small transactions (great for testing congestion), whereas another would make long transactions. At run-time, you can use the built-in constraint_mode() routine to turn constraints on and off. You can control a single constraint with handle.constraint.constraint_mode(). To control all constraints in an object, use handle.constraint_mode(), as shown in Sample.

```
class Packet;
  rand int length;
  constraint c_short {length inside {[1:32]}; }
  constraint c_long  {length inside {[1000:1023]}; }
endclass

Packet p;
initial begin
  p = new();

  // Create a long packet by disabling short constraint
  p.c_short.constraint_mode(0);
  assert (p.randomize());

  transmit(p);

  // Create a short packet by disabling all constraints
  // then enabling only the short constraint
  p.constraint_mode(0);
  p.c_short.constraint_mode(1);
  assert (p.randomize());
  transmit(p);
end
```

4.5 Valid Constraints

A good randomization technique is to create several constraints to ensure the correctness of your random stimulus, known as "valid constraints." For example, a bus read– modify–write command might only be allowed for a longword data length.

```
class Transaction;
  rand enum {BYTE, WORD, LWRD, QWRD} length;
  rand enum {READ, WRITE, RMW, INTR} opc;

  constraint valid_RMW_LWRD {
    (opc == RMW) -> length == LWRD;
  }
endclass
```

Now you know the bus transaction obeys the rule. Later, if you want to violate the rule, use constraint_mode to turn off this one constraint. You should have a naming convention to make these constraints stand out, such as using the prefix valid as shown above.

4.6 In-Line Constraints:

As you write more tests, you can end up with many constraints. They can interact with each other in unexpected ways, and the extra code to enable and disable them adds to the test complexity. Additionally, constantly adding and editing constraints to a class could cause problems in a team environment. Many tests only randomize objects at one place in the code. SystemVerilog allows you to add an extra constraint using randomize() with. This is equivalent

to adding an extra constraint to any existing ones in effect. Sample shows a base class with constraints, then two randomize() with statements.

```
class Transaction;
  rand bit [31:0] addr, data;
  constraint c1 {addr inside{[0:100],[1000:2000]};}
endclass

Transaction t;

initial begin
  t = new();

  // addr is 50-100, 1000-1500, data < 10
  assert(t.randomize() with {addr >= 50; addr <= 1500;
                             data < 10;});

  driveBus(t);

  // force addr to a specific value, data > 10
  assert(t.randomize() with {addr == 2000; data > 10;});

  driveBus(t);
  end
```

The extra constraints are added to the existing ones in effect. Use constraint_mode if you need to disable a conflicting constraint. Note that inside the with{} statement, SystemVerilog uses the scope of the class. That is why Sample 6.30 used just addr, not t.addr.

A common mistake is to surround your in-line constraints with parenthesis instead of curly braces {}. Just remember that constraint blocks use curly braces, and so your in-line constraint must use them too. Braces are for declarative code.

4.7 Randomizing Individual Variables

Suppose you want to randomize a few variables inside a class. You can call random ize() with the subset of variables. Only those variables passed in the argument list will be randomized; the rest will be treated as state variables and not randomized. All constraints remain in effect. In Sample 6.36, the first call to randomize() only changes the values of two rand variables med and hi. The second call only changes the value of med, whereas hi retains its previous value. Surprisingly, you can pass a nonrandom variable, as shown in the last call, and low is given a random value, as long as it obeys the constraint.

This trick of only randomizing a subset of the variables is not commonly used in real testbenches as you are restricting the randomness of your stimulus. You want your testbench to explore the full range of legal values, not just a few corners.

```
class Rising;
  byte low;            // Not random
  rand byte med, hi;   // Random variable
  constraint up
    { low < med; med < hi; } // See Section 6.4.2
endclass

initial begin
  Rising r;
  r = new();
  r.randomize();      // Randomize med, hi; low untouched
  r.randomize(med);   // Randomize only med
  r.randomize(low);   // Randomize only low
end
```

4.8 Pseudorandom Number Generators

Verilog uses a simple PRNG that you could access with the $random function. The generator has an internal state that you can set by providing a seed to $random. All IEEE-1364-compliant Verilog simulators use the same algorithm to calculate values. Sample shows a simple PRNG, not the one used by SystemVerilog. The PRNG has a 32-bit state. To calculate the next random value, square the state to produce a 64-bit value, take the middle 32 bits, then add the original value.

```
reg [31:0] state = 32'h12345678;
function logic [31:0] my_random;
  logic [63:0] s64;
  s64 = state * state;
  state = (s64 >> 16) + state;
  my_random = state;
endfunction
```

You can see how this simple code produces a stream of values that seem random, but can be repeated by using the same seed value. SystemVerilog calls its PRNG to generate a new value for randomize() and randcase.

Reference Books:

1. Advanced Digital Design With the Verilog HDL, Michael D. Ciletti, 2nd Edition, PHI, ISBN: 978–0– 07–338054–4 2015.
2. Digital Systems Design Using Verilog, Charles Roth, Lizy K. John, Byeong KilLee, Cengage Learning, ISBN-10: 1285051076, 2015.
3. Fundamentals of Digital Logic with Verilog Design, Stephen Brown and Zvonko Vranesic, 6th Edition, McGraw Hill publication, ISBN: 978–0–07–338054–4, 2014.
4. System Verilog for Design - A Guide to Using System Verilog for Hardware Design and Modeling, Stuart Sutherland, Simon David mann and Peter Flake, 2E, Springer Science, ISBN-13: 978-0387-3339-91, 2006.

5. System Verilog for Verification-A Guide to Learning the Testbench Language Features, C Spear, Springer Science, IEEE press, ISBN-13: 978-0387-2703-64,2006.
6. System Verilog golden reference guide-A concise guide to System Verilog Doulos, IEEE Standard-1800- 2009, Version 5.0,ISBN: 0-9547345-9-9, 2012.
7. Step-by-Step Functional Verification with System Verilog and OVM, SasanIman, Hansen Brown Publishing Company,ISBN-13: 978-0-9816-5621-2, 2008.

Questions to Practice:

PART -A

1	Identify how System Verilog is is used in rand and randc
2	Identify the Limitations of Constraints.
3	Describe in detail about Rand_mode
4	Classify the use of Set Membership in constraint
5	Explain in detail about Conditional Constraints in System Verilog

PART-B

1	Demonstrate in detail about Randomization in System Verilog
2	Classify Constraint types that are used in System Verilog
3	Demonstrate in detail about the System Verilog Device and Environment Configuration
4	Demonstrate in detail about the impact of Equivalence and Weighted Constraints
5	Enumerate Randomizing individual variables and PRNG in detail.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF ELECTRICAL AND ELECTRONICS

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

UNIT – V – FUNCTIONAL COVERAGE AND ASSERTION BASED VERIFICATION – SECA3021

UNIT-V (FUNCTIONAL COVERAGE AND ASSERTION BASED VERIFICATION)

Coverage Definition, Code Coverage, Functional Coverage: Cover Group, Creating Cover Group Instances, Coverpoints, Bins - implicit bins, . Explicit bins, Bin creation, Vector and Scalar bins, Cross products, Intersect, Select Expressions, Conditional Expression (iff), Illegal bins, Ignore bins, Coverage Analysis, Covergroup Built-in Methods - .Sample(), . get_coverage(), .get_instance_coverage(), .set_instance_name (string), .start(), . stop()

5.1 Functional Coverage:

As designs become more complex, the only effective way to verify them thoroughly is with constrained-random testing (CRT). This approach elevates you above the tedium of writing individual directed tests, one for each feature in the design. However, if your testbench is taking a random walk through the space of all design states.

Whether you are using random or directed stimulus, you can gauge progress using coverage. Functional coverage is a measure of which design features have been exercised by the tests. Start with the design specification and create a verification plan with a detailed list of what to test and how. For example, if your design connects to a bus, your tests need to exercise all the possible interactions between the design and bus, including relevant design states, delays, and error modes.

Use a feedback loop to analyze the coverage results and decide on which actions to take in order to converge on 100% coverage (Figure 5.1). Your first choice is to run existing tests with more seeds; the second is to build new constraints. Resort to creating directed tests only if absolutely necessary.

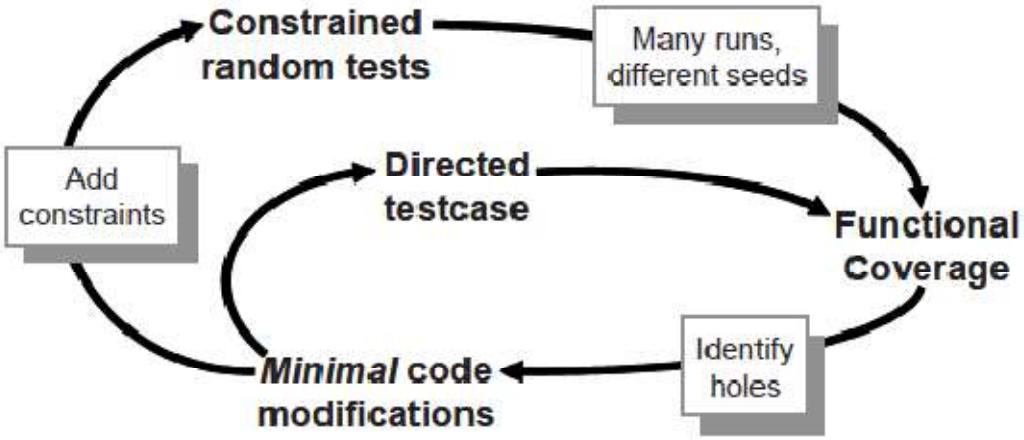

Figure 5.1 Coverage Convergence

Back when you exclusively wrote directed tests, the verification planning was limited. If the design specification listed 100 features, all you had to do was write 100 tests. Coverage was implicit in the tests – the "register move" test moved all combinations of registers back and forth. Measuring progress was easy: if you had completed 50 tests, you were halfway done. This chapter uses "explicit" and "implicit" to describe how coverage is specified. Explicit coverage is described directly in the test environment using SystemVerilog features.

Implicit coverage is implied by a test – when the "register move" directed test passes, you have hopefully covered all register transactions. With CRT, you are freed from hand crafting every line of input stimulus, but now you need to write code that tracks the effectiveness of the test with respect to the verification plan. You are still more productive, as you are working at a higher level of abstraction. You have moved from tweaking individual bits to describing the interesting design states. Reaching for 100% functional coverage forces you to think more about what you want to observe and how you can direct the design into those states.

Gathering Coverage Data: You can run the same random testbench over and over, simply by changing the random seed, to generate new stimulus. Each individual simulation generates a database of functional coverage information, the trail of footprints from the random walk. You can then merge all this information together to measure your overall progress using functional coverage as shown in Figure 5.2.

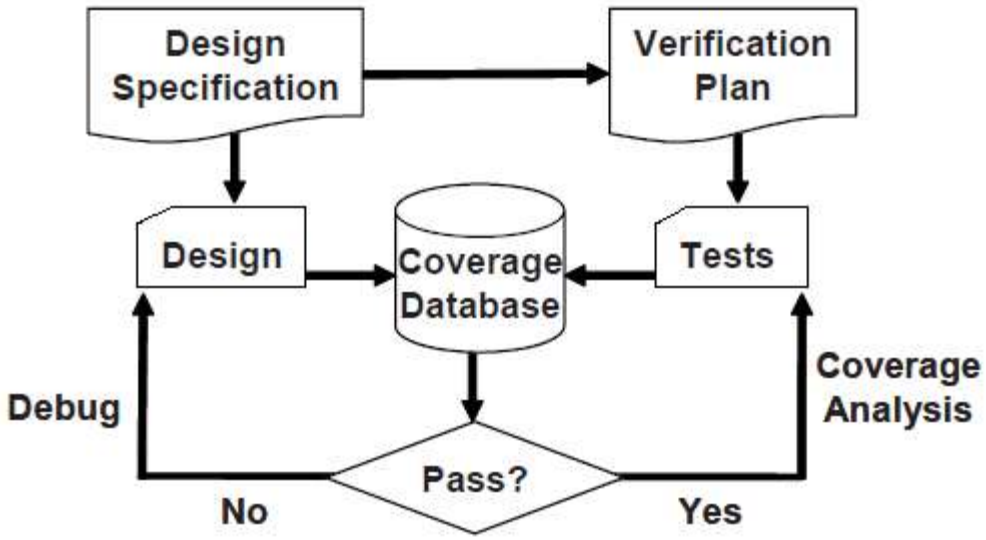

Figure 5.2: Coverage Flow

You then analyze the coverage data to decide how to modify your tests. If the coverage levels are steadily growing, you may just need to run existing tests with new random seeds, or even just run longer tests. If the coverage growth has started to slow, you can add additional constraints to generate more "interesting" stimuli. When you reach a plateau, some parts of the design are not being exercised, and so you need to create more tests.

Lastly, when your functional coverage values near 100%, check the bug rate. If bugs are still being found, you may not be measuring true coverage for some areas of your design. Don't

be in too big of a rush to reach 100% coverage, which just shows that you looked for bugs in all the usual places. While you are trying to verify your design, take many random walks through the stimulus space; this can create many unanticipated combinations.

Each simulation vendor has its own format for storing coverage data and as well as its own analysis tools. You need to perform the following actions with those tools.

 Run a test with multiple seeds. For a given set of constraints (and coverage groups), compile the testbench and design into a single executeable. Now you need to run this constraint set over and over with different random seeds. You can use the Unix system clock as a seed, but be careful, as your batch system may start multiple jobs simultaneously. These jobs may run on different servers or may start on a single server with multiple processors.

 Check for pass/fail. Functional coverage information is only valid for a successful simulation. When a simulation fails because there is a design bug, the coverage information must be discarded. The coverage data measures how many items in the verification plan are complete, and this plan is based on the design specification. If the design does not match the specification, the coverage values are useless. Some verification teams periodically measure all functional coverage from scratch so that it reflects the current state of the design.

 Analyze coverage across multiple runs. You need to measure how successful each constraint set is, over time. If you are not yet getting 100% coverage for the areas that are targeted by the constraints, but the amount is still growing, run more seeds. If the coverage level has plateaued, with no recent progress, it is time to modify the constraints. Only if you think that reaching the last few test cases for one particular section may take too long for constrainedrandom simulation should you consider writing a directed test. Even then, continue to use random stimulus for the other sections of the design, in case this “background noise” finds a bug.

5.2 Coverage Types:

Coverage is a generic term for measuring progress to complete design verification. Your simulations slowly paint the canvas of the design, as you try to cover all of the legal combinations. The coverage tools gather information during a simulation and then postprocess it to produce a coverage report. You can use this report to look for coverage holes and then modify existing tests or create new ones to fill the holes. This iterative process continues until you are satisfied with the coverage level.

5.2.1 Code Coverage

The easiest way to measure verification progress is with code coverage. Here you are measuring how many lines of code have been executed (line coverage), which paths through the code and expressions have been executed (path coverage), which singlebit variables have had the values 0 or 1 (toggle coverage), and which states and transitions in a state machine have been visited (FSM coverage). You don’t have to write any extra HDL code. The tool instruments your design automatically by analyzing the source code and adding hidden code to gather statistics.

You then run all your tests, and the code coverage tool creates a database. Many simulators include a code coverage tool. A postprocessing tool converts the database into a readable form.

The end result is a measure of how much your tests exercise the design code. Note that you are primarily concerned with analyzing the design code, not the testbench. Untested design code could conceal a hardware bug, or may be just redundant code. Code coverage measures how thoroughly your tests exercised the "implementation" of the design specification, and not the verification plan. Just because your tests have reached 100% code coverage, your job is not done.

Example: Incomplete D-flip flop model missing a path

```
module dff(output logic q, q_l,
           input  logic clk, d, reset_l);

  always @(posedge clk or negedge reset_l) begin
    q <= d;
    q_l <= !d;
  end
endmodule
```

The reset logic was accidently left out. A code coverage tool would report that every line had been exercised, yet the model was not implemented correctly.

5.2.2 Functional Coverage

The goal of verification is to ensure that a design behaves correctly in its real environment, be that an MP3 player, network router, or cell phone. The design specification details how the device should operate, whereas the verification plan lists how that functionality is to be stimulated, verified, and measured. When you gather measurements on what functions were covered, you are performing "design" coverage.

For example, the verification plan for a D-flip flop would mention not only its data storage but also how it resets to a known state. Until your test checks both these design features, you will not have 100% functional coverage. Functional coverage is tied to the design intent and is sometimes called "specification coverage," while code coverage measures the design implementation. Consider what happens if a block of code is missing from the design. Code coverage cannot catch this mistake, but functional coverage can.

5.2.3 Bug Rate

An indirect way to measure coverage is to look at the rate at which fresh bugs are found. You should keep track of how many bugs you found each week, over the life of a project. At the start, you may find many bugs through inspection as you create the testbench. As you read the design spec, you may find inconsistencies, which hopefully are fixed before the RTL is written. Once the testbench is up and running, a torrent of bugs is found as you check each module in the system. The bug rate drops, hopefully to zero, as the design nears tape-out. However, you are not yet done. Every time the rate sags, it is time to find different ways to create corner cases.

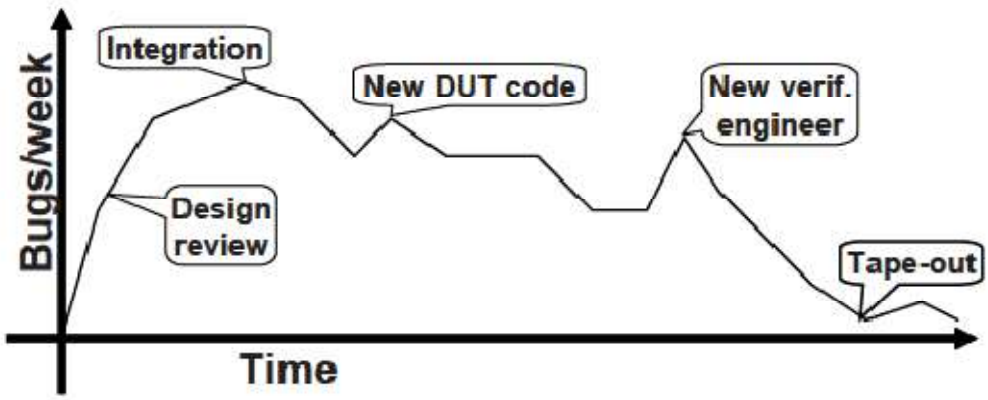

Figure 5.3 Bug Rate during a Project

The bug rate can vary per week based on many factors such as project phases, recent design changes, blocks being integrated, personnel changes, and even vacation schedules. Unexpected changes in the rate could signal a potential problem. As shown in Figure 5.3, it is not uncommon to keep finding bugs even after tape-out, and even after the design ships to customers.

5.2.4 Assertion Coverage

Assertions are pieces of declarative code that check the relationships between design signals, either once or over a period of time. These can be simulated along with the design and testbench, or proven by formal tools. Sometimes you can write the equivalent check using SystemVerilog procedural code, but many assertions are more easily expressed using SystemVerilog Assertions (SVA). Assertions can have local variables and perform simple data checking.

If you need to check a more complex protocol, such as determining whether a packet successfully went through a router, procedural code is often better suited for the job. There is a large overlap between sequences that are coded procedurally or using SVA. The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant. These error checks should stop the simulation as soon as they detect a problem. Assertions can also check arbitration algorithms, FIFOs, and other hardware. These are coded with the assert property statement.

Some assertions might look for interesting signal values or design states, such as a successful bus transaction. These are coded with the cover property statement. You can measure how often these assertions are triggered during a test by using assertion coverage. A cover property observes sequences of signals, whereas a cover group samples data values and transactions during the simulation. These two constructs overlap in that a cover group can trigger when a sequence completes. Additionally, a sequence can collect information that can be used by a cover group.

5.3 Cover Group

A cover group is similar to a class – you define it once and then instantiate it one or more times. It contains cover points, options, formal arguments, and an optional trigger. A cover group encompasses one or more data points, all of which are sampled at the same time.

You should create very clear cover group names that explicitly indicate what you are measuring and, if possible, reference to the verification plan. The name Parity_Errors_In_Hexaword_Cache_Fills may seem verbose, but when you are trying to read a coverage report that has dozens of cover groups, you will appreciate the extra detail. You can also use the comment option for additional descriptive information.

A cover group can be defined in a class or at the program or module level. It can sample any visible variable such as program/module variables, signals from an interface, or any signal in the design (using a hierarchical reference). A cover group inside a class can sample variables in that class, as well as data values from embedded classes.

Don't define the cover group in a data class, such as a transaction, as doing so can cause additional overhead when gathering coverage data. Imagine you are trying to track how many beers were consumed by patrons in a pub. Would you try to follow every bottle as it flowed from the loading dock, over the bar, and into each person? No, instead you could just have each patron check off the type and number of beers consumed. In SystemVerilog, you should define cover groups at the appropriate level of abstraction. This level can be at the boundary between your testbench and the design, in the transactors that read and write data, in the environment configuration class, or wherever is needed. The sampling of any transaction must wait until it is actually received by the DUT. If you inject an error in the middle of a transaction, causing it to be aborted in transmission, you need to change how you treat it for functional coverage. You need to use a different cover point that has been created just for error handling. A class can contain multiple cover groups. This approach allows you to have separate groups that can be enabled and disabled as needed. Additionally, each group may have a separate trigger, allowing you to gather data from many sources. A cover group must be instantiated for it to collect data. If you forget, no error message about null handles is printed at run-time, but the coverage report will not contain any mention of the cover group. This rule applies for cover groups defined either inside or outside of classes.

5.3.1 Defining a Cover Group in a Class

A cover group can be defined in a program, module, or class. In all cases, you must explicitly instantiate it to start sampling. If the cover group is defined in a class, you do not make a separate name when you instance it; you just use the original cover group name.

Below Example is very similar to the first example of this chapter except that it embeds a cover group in a transactor class, and thus does not need a separate instance name.

```
class Transactor;
  Transaction tr;
  mailbox mbx_in;
  covergroup CovPort;
    coverpoint tr.port;
  endgroup

  function new(mailbox mbx_in);
    CovPort = new();          // Instantiate covergroup
    this.mbx_in = mbx_in;
  endfunction

  task main;
    forever begin
      tr = mbx_in.get;        // Get next transaction
      ifc.cb.port <= tr.port; // Send into DUT
      ifc.cb.data <= tr.data;
      CovPort.sample();       // Gather coverage
    end
  endtask

endclass
```

5.4 Triggering a Cover Group

The two major parts of functional coverage are the sampled data values and the time when they are sampled. When new values are ready (such as when a transaction has completed), your testbench triggers the cover group. This can be done directly with the sample function, as shown in above example, or by using a blocking expression in the covergroup definition. The blocking expression can use a wait or @ to block on signals or events. Use sample if you want to explicitly trigger coverage from procedural code, if there is no existing signal or event that tells when to sample, or if there are multiple instances of a cover group that trigger separately. Use the blocking statement in the covergroup declaration if you want to tap into existing events or signals to trigger coverage.

5.4.1 Sampling Using a Callback

One of the better ways to integrate functional coverage into your testbench is to use callbacks. This technique allows you to build a flexible testbench without restricting when coverage is collected. You can decide for every point in the verification plan where and when values are sampled. And if you need an extra “hook” in the environment for a callback, you can always add one in an unobtrusive manner, as a callback only “fires” when the test registers a callback object. You can create many separate callbacks for each cover group, with little overhead. As explained in Section 8.7.4, callbacks are superior to using a mailbox to connect the testbench to the coverage objects. You might need multiple mailboxes to collect transactions from different points in your testbench. A mailbox requires a transactor to receive transactions, and multiple mailboxes cause you to juggle multiple threads. Instead of an active transactor, use a passive callback.

Push an instance of the coverage callback class into the driver's callback queue, and your coverage code triggers the cover group at the right time. The following two examples define and use the callback Driver_cbs_coverage.

```
program automatic test;
  Environment env;

  initial begin
    Driver_cbs_coverage dcc;

    env = new();
    env.gen_cfg();
    env.build();

    // Create and register the coverage callback
    dcc = new();
    env.drv.cbs.push_back(dcc); // Put into driverÕs Q

    env.run();
    env.wrap_up();
  end

endprogram
```

5.4.2 Cover Group With an Event Trigger

```
event trans_ready;
covergroup CovPort @(trans_ready);
    coverpoint ifc.cb.port;    // Measure coverage
endgroup
```

The advantage of using an event over calling the sample method directly is that you may be able to use an existing event such as one triggered by an assertion, as shown in above example.

5.4.3 Triggering on a SystemVerilog Assertion

If you already have an SVA that looks for useful events like a complete transaction, you can add an event trigger to wake up the cover group.

```
module mem(simple_bus sb);
  bit [7:0] data, addr;
  event write_event;

  cover property
   (@(posedge sb.clock) sb.write_ena==1)
    -> write_event;
endmodule
```

5.5 Data Sampling

How is coverage information gathered? When you specify a variable or expression in a cover point, SystemVerilog creates a number of "bins" to record how many times each value has been seen. These bins are the basic units of measurement for functional coverage. If you sample a one-bit variable, a maximum of two bins are created. You can imagine that SystemVerilog drops a token in one or the other bin every time the cover group is triggered. At the end of each simulation, a database is created with all bins that have a token in them. You then run an analysis tool that reads all databases and generates a report with the coverage for each part of the design and for the total coverage.

5.5.1 Individual Bins and Total Coverage

To calculate the coverage for a point, you first have to determine the total number of possible values, also known as the domain. There may be one value per bin or multiple values. Coverage is the number of sampled values divided by the number of bins in the domain. A cover point that is a 3-bit variable has the domain 0:7 and is normally divided into eight bins. If, during simulation, values belonging to seven bins are sampled, the report will show 7/8 or 87.5% coverage for this point. All these points are combined to show the coverage for the entire group, and then all the groups are combined to give a coverage percentage for all the simulation databases.

This is the status for a single simulation. You need to track coverage over time. Look for trends so that you can see where to run more simulations or add new constraints or tests. Now you can better predict when verification of the design will be completed.

5.5.2 Creating Bins Automatically

SystemVerilog automatically creates bins for cover points. It looks at the domain of the sampled expression to determine the range of possible values. For an expression that is N bits wide, there are 2N possible values. For the 3-bit variable port, there are 8 possible values. The range of an enumerated type is shown in Section 9.6.8. The domain for enumerated data types is the number of named values.

5.5.3 Limiting the Number of Automatic Bins Created

The cover group option auto_bin_max specifies the maximum number of bins to automatically create, with a default of 64 bins. If the domain of values in the cover point variable or expression is greater than this option, SystemVerilog divides the range into auto_bin_max bins. For example, a 16-bit variable has 65,536 possible values, and so each of the 64 bins covers 1,024 values. In reality, you may find this approach impractical, as it is very difficult to find the needle of missing coverage in a haystack of auto-generated bins.

Lowering this limit to 8 or 16, or better yet, explicitly define the bins. The following code takes the chapter's first example and adds a cover point option that sets auto_bin_max to two bins. The sampled variable is still port, which is three bits wide, for a domain of eight possible

values. The first bin holds the lower half of the range, 0–3, and the other hold the upper values, 4–7.

```
covergroup CovPort;
  coverpoint tr.port
    { options.auto_bin_max = 2; }  // Divide into 2 bins
endgroup
```

The coverage report from VCS shows the two bins. This simulation achieved 100% coverage because the eight port values were mapped to two bins. Since both bins have sampled values, your coverage is 100%.

5.5.4 Conditional Coverage

You can use the iff keyword to add a condition to a cover point. The most common reason for doing so is to turn off coverage during reset so that stray triggers are ignored. Below Example gathers only values of port when reset is 0, where reset is active-high.

```
covergroup CoverPort;
  // DonÕt gather coverage when reset==1
  coverpoint port iff (!bus_if.reset);
endgroup
```

Alternately, you can use the start and stop functions to control individual instances of cover groups.

```
initial begin
  CovPort ck = new();       // Instantiate cover group

  // Reset sequence stops collection of coverage data
  #1ns ck.stop();
  bus_if.reset = 1;

  #100ns bus_if.reset = 0; // End of reset
  ck.start();
  ...
end
```

5.5.5 Transition Coverage

You can specify state transitions for a cover point. In this way, you can tell not only what interesting values were seen but also the sequences. For example, you can check if port ever went from 0 to 1, 2, or 3.

```
covergroup CoverPort;
  coverpoint port {
    bins t1 = (0 => 1), (0 => 2), (0 => 3);
  }
endgroup
```

You can quickly specify multiple transitions using ranges. The expression (1,2 => 3,4) creates the four transitions (1=>3), (1=>4), (2=>3), and (2=>4).

You can specify transitions of any length. Note that you have to sample once for each state in the transition. So (0 => 1 => 2) is different from (0 => 1 => 1 => 2) or (0 => 1 => 1 => 1 => 2). If you need to repeat values, as in the last sequence, you can use the shorthand form:(0 => 1[*3] => 2). To repeat the value 1 for 3, 4, or 5 times, use 1[*3:5].

5.5.6 Illegal Bins

Some sampled values not only should be ignored but also should cause an error if they are seen. This is best done in the testbench's monitor code, but can also be done by labeling a bin with illegal_bins. Use illegal_bins to catch states that were missed by the test's error checking. This also double-checks the accuracy of your bin creation: if an illegal value is found by the cover group, it is a problem either with the testbench or with your bin definitions.

```
bit [2:0] low_ports_0_5;      // Only uses values 0-5
covergroup CoverPort;
  coverpoint low_ports_0_5 {
    illegal_bins hi = {[6,7]}; // Give error if seen
  }
endgroup
```

5.6 Coverage Options

You can specify additional information in the cover group using options. There are two flavors of options: instance options that apply to a specific cover group instance and type options that apply to all instances of the cover group, and are analogous to static data members of classes. Options can be placed in the cover group so that they apply to all cover points in the group, or they can be put inside a single cover point for finer control. You have already seen the auto_bin_max and weight options. Here are several more.

5.6.1 Per-Instance Coverage

If your testbench instantiates a coverage group multiple times, by default SystemVerilog groups together all the coverage data from all the instances. However, if you have several generators, each creating very different streams of transactions, you will need to see separate reports. For example, one generator may be creating long transactions while another makes short ones. The cover group in example can be instantiated in each separate generator. It keeps track of coverage for each instance, and has a unique comment string with the hierarchical path to the cover group instance.

```
covergroup CoverLength;
  coverpoint tr.length;
  option.per_instance = 1;
  // Use hierarchical path in comment
  option.comment = $psprintf("%m");
endgroup
```

The per-instance option can only be given in the cover group, not in the cover point or cross point.

5.6.2 Cover Group Comment

You can add a comment into coverage reports to make them easier to analyze. A comment could be as simple as the section number from the verification plan to tags used by a report parser to automatically extract relevant information from the sea of data. If you have a cover group that is only instantiated once, use the type option as shown in below example.

```
covergroup CoverPort;
  type_option.comment = "Section 3.2.14 Port numbers";
  coverpoint port;
endgroup
```

However, if you have multiple instances, you can give each a separate comment, as long as you also use the per-instance option.

```
covergroup CoverPort(int lo,hi, string comment);
  option.comment = comment;
  option.per_instance = 1;
  coverpoint port
    {bins range = {[lo:hi]};
     }
endgroup
...
CoverPort cp_lo = new(0,3, "Low port numbers");
CoverPort cp_hi = new(4,7, "High port numbers");
```

5.6.3 Coverage Goal

The goal for a cover group or point is the level at which the group or point is considered fully covered. The default is 100% coverage. If you set this level below 100%, you are requesting less than complete coverage, which is probably not desirable. This option affects only the coverage report.

```
covergroup CoverPort;
  coverpoint port;
  option.goal = 90;  // Settle for partial coverage
endgroup
```

5.7 Analyzing Coverage Data

In general, assume you need more seeds and fewer constraints. After all, it is easier to run more tests than to construct new constraints. If you are not careful, new constraints can easily restrict the search space. If your cover point has only zero or one sample, your constraints are probably not targeting these areas at all. You need to add constraints that “pull” the solver into new areas. In given example, the transaction length had an uneven distribution. This situation is similar to the distribution seen when you roll two dice and look at the total value.

```
class Transaction;
  rand bit [2:0] hdr_len;
  rand bit [3:0] payload_len;
  rand bit [4:0] len;
  constraint length {len == hdr_len + payload_len; }
endclass
```

The problem with this class is that len is not evenly weighted as shown in Figure 5.4.

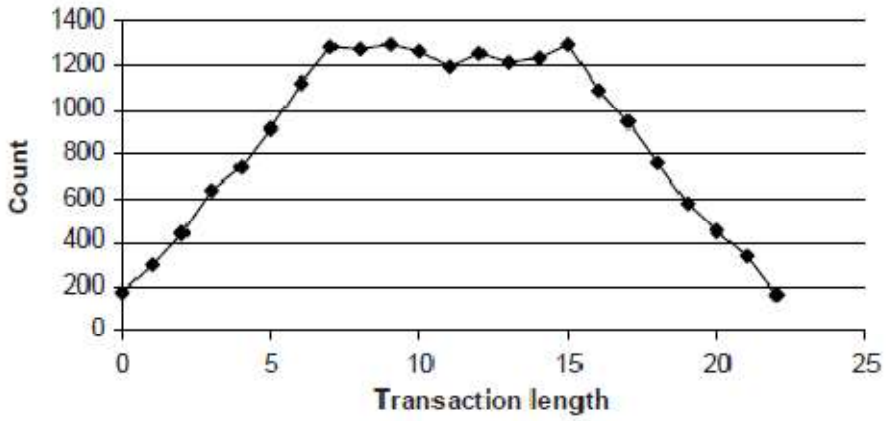

Figure 5.4 Uneven probability for transaction length

If you want to make the total length be evenly distributed, use a solve...before constraint as shown in Figure 5.4.

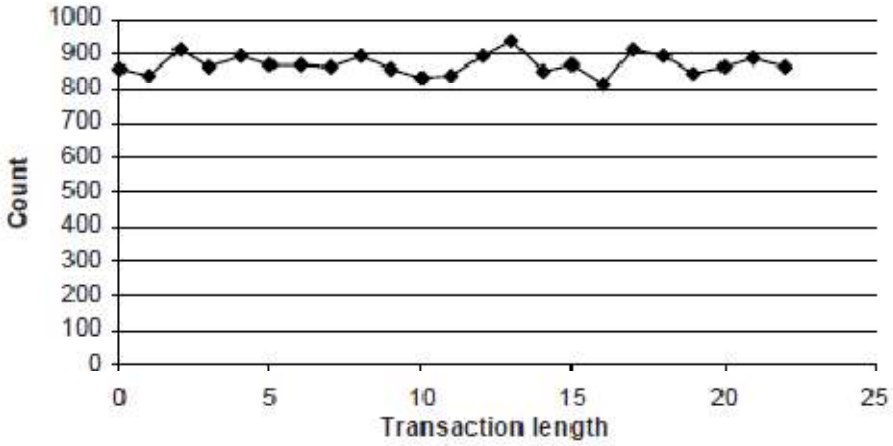

Figure 5.5 Even probability for transaction length with solve...before

The normal alternative to solve...before is the dist constraint. However, this does not work, as len is also being constrained by the sum of the two lengths.

5.8 Measuring Coverage Statistics During Simulation

You can query the level of functional coverage on the fly during simulation. This allows you to check whether you have reached your coverage goals, and possibly to control a random test. At the global level, you can get the total coverage of all cover groups with $get_coverage,

which returns a real number between 0 and 100. This system task looks across all cover groups. You can narrow down your measurements with the get_coverage() and get_inst_coverage() methods. The first function works with both cover group names and instances to give coverage across all instances of a cover group, for example, CoverGroup::get_coverage() or cgInst.get_coverage(). The second function returns coverage for a specific cover group instance, for example cgInst.get_inst_coverage().

You need to specify option.per_instance=1 if you want to gather perinstance coverage. The most practical use for these functions is to monitor coverage over a long test. If the coverage level does not advance after a given number of transactions or cycles, the test should stop. Hopefully, another seed or test will increase the coverage. While it would be nice to have a test that can perform some sophisticated actions based on functional coverage results, it is very hard to write this sort of test. Each test + random seed pair may uncover new functionality, but it may take many runs to reach a goal. If a test finds that it has not reached 100% coverage, what should it do? Run for more cycles? How many more? Should it change the stimulus being generated? How can you correlate a change in the input with the level of functional coverage? The one reliable thing to change is the random seed, which you should only do once per simulation.

Otherwise, how can you reproduce a design bug if the stimulus depends on multiple random seeds? You can query the functional coverage statistics if you want to create your own coverage database. Verification teams have built their own SQL databases that are fed functional coverage data from simulation. This setup allows them greater control over the data, but requires a lot of work outside of creating tests. Some formal verification tools can extract the state of a design and then create input stimulus to reach all possible states.

Reference Books:

1. Advanced Digital Design With the Verilog HDL, Michael D. Ciletti, 2nd Edition, PHI, ISBN: 978–0– 07–338054–4 2015.
2. Digital Systems Design Using Verilog, Charles Roth, Lizy K. John, Byeong KilLee, Cengage Learning, ISBN-10: 1285051076, 2015.
3. Fundamentals of Digital Logic with Verilog Design, Stephen Brown and Zvonko Vranesic, 6th Edition, McGraw Hill publication, ISBN: 978–0–07–338054–4, 2014.
4. System Verilog for Design - A Guide to Using System Verilog for Hardware Design and Modeling, Stuart Sutherland, Simon David mann and Peter Flake, 2E, Springer Science, ISBN-13: 978-0387-3339-91, 2006.
5. System Verilog for Verification-A Guide to Learning the Testbench Language Features, C Spear, Springer Science, IEEE press, ISBN-13: 978-0387-2703-64,2006.
6. System Verilog golden reference guide-A concise guide to System Verilog Doulos, IEEE Standard-1800- 2009, Version 5.0,ISBN: 0-9547345-9-9, 2012.

7. Step-by-Step Functional Verification with System Verilog and OVM, SasanIman, Hansen Brown Publishing Company,ISBN-13: 978-0-9816-5621-2, 2008.

Questions to Practice:

PART -A

1	Identify how System Verilog is is used in implicit bins
2	Identify the advantages of Fucntional Coverage.
3	Describe in detail about Cover Group
4	Classify the use of Cover points
5	Explain in detail about Coverage Analysis

PART-B

1	Demonstrate in detail about Functional Coverage in System Verilog
2	Classify Coverage types that are used in System Verilog
3	Demonstrate in detail about the System Verilog Cover group Built in Methods
4	Demonstrate in detail about the impact of Implicit and Explicit Bins
5	Enumerate Vector and Scalar bins of Functional Coverage in detail.