

Verification using SystemVerilog

-- Preetham Lakshmikanthan

Table of Contents

1. Introduction to SV
2. Commonly Used Terminologies in SV
3. Data Types
4. Object-Oriented Programming (OOP) Concepts
5. SV Stratified Event Queue/Scheduler
6. SV Tasks and Functions

Table of Contents Continued...

- 7. Verification Specific SV Constructs
- 8. Functional Coverage
- 9. Verification Plan and SV Testbench Architecture
- 10. Modelling Testbench Blocks for the RAM Design
- 11. Assertions
- 12. Direct Programming Interface (DPI)

References

1. SystemVerilog for Verification: A Guide to Learning the Testbench Language Features

Author(s): Chris Spear and Greg Tumbush

ISBN-10: 1461407141

ISBN-13: 978-1461407140

Publisher: Springer Publications (3rd Edition 2012)

Weblink: <http://www.chris.spear.net/systemverilog/default.htm>

2. www.testbench.in – SystemVerilog Tutorial

Weblink: http://www.testbench.in/SV_00_INDEX.html

References Continued...

3. “SystemVerilog – UNIFIED HARDWARE DESIGN, SPECIFICATION, AND VERIFICATION LANGUAGE” – The IEEE 1800-2012 LRM

Weblink: <http://standards.ieee.org/getieee/1800/download/1800-2012.pdf>

4. www.systemverilog.in – SystemVerilog Resources

Weblink: http://www.systemverilog.in/systemverilog_introduction.php

5. Writing Testbenches using SystemVerilog

Author(s): Janick Bergeron

ISBN-10: 1846280230

ISBN-13: 978-0387292212

Publisher: Springer Publications (2006 Edition)

References Continued...

6. <http://www.asic-world.com/systemverilog/tutorial.html> – SystemVerilog Tutorial

7. “ECE 745: ASIC VERIFICATION” – Dr. Meeta Yadav, North Carolina State University

Weblink:

engineeringonline.ncsu.edu/onlinecourses/coursehomepages/FALL08/ECE745.html

8. Hardware Verification with SystemVerilog: An Object-Oriented Framework

Author(s): Mike Mintz and Robert Ekendahl

ISBN-10: 0313262322

ISBN-13: 978-0313262326

Publisher: Springer Publications (2007 Edition)

References Continued...

9. SystemVerilog Assertions Handbook, 3rd Edition ... for Dynamic and Formal Verification

Author(s): Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari and Lisa Piper

ASIN: B0096CEVQM

Publisher: VhdlCohen Publishing

10. Verification Suite using SystemVerilog – A Complete Testbench Example

Weblink: <http://wiki.usgroup.eu/wiki/public/tutorials/svverification>

1. Introduction to SystemVerilog (SV)

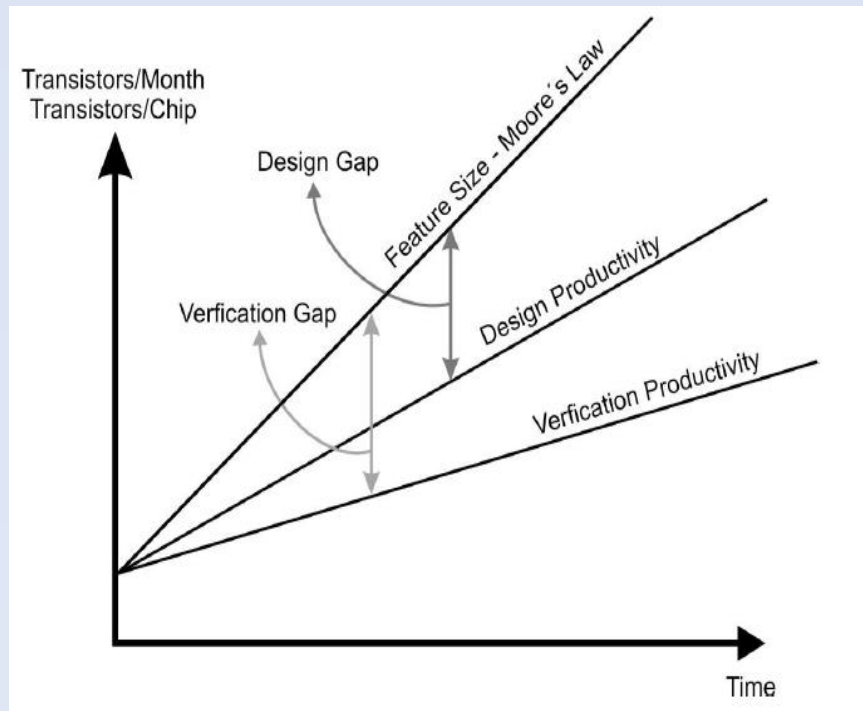
Need for Verification

Today, the verification effort takes almost 70% of the total design cycle time. This stringent verification is needed

- Due to the complex functionality of present-day designs that could lead to more bugs
 - Increase in features lead to an increase in logic functionality – more gates and flip-flops
 - Multiple clocks and deeper pipelines are being introduced to improve performance
 - Numerous power states are being added in designs to save energy
- To avoid bug escapes to silicon, which otherwise requires costly re-spins
 - In 1994, Intel Corporation spent \$475 million to replace flawed Pentium chips that had a floating point division bug
- To verify numerous interfaces and bus transactions present in SoCs, due to the integration of multiple functional components on the chip
- To validate millions of lines of code written using different languages
 - RTL is written in VHDL, testbenches in Verilog and functional models are coded in C

Need for Verification Continued...

Verification Gap - the big challenge confronting the industry



Hardware Verification Languages such as SystemVerilog are used to reduce the verification gap

Diagram Reference:

Warren A. Hunt Jr.: **Introduction: Special Issue on Microprocessor Verifications.** Formal Methods in System Design 20(2): 135-137 (2002)

Deliverables of an RTL Verification Engineer

- Develop a verification plan by reviewing the design specification
 - Done by the verification lead by extracting features from the design specification
- Design and develop the various testbench components in the verification environment
 - These components must be bug free and designed in such a way that they are usable at both the block level as well as the full-chip level
- Understand the working of various IPs, Bus Protocols and Interconnection Architectures to develop concise and complete test-cases to achieve the coverage goals
- Maintain the verification environment and append test-cases to the repository in order to enhance the coverage
- Develop scripts to automate job submission for regressions and analyze reports
- Find bugs in the RTL code and report them to the RTL designer. A key point to note is that it is not the job of a verification engineer to fix the bugs found

Limitations of Verilog for Verification

❖ Weak randomization

- No constructs to easily constrain randomization.

For e.g.: Consider the case of a RAM with 1024 locations. Random numbers in Verilog are generated in the signed range of values from $-(2^{32} - 1)$ to (2^{32}) . Randomly generating addresses for testing of the RAM DUV is of no use unless the address is constrained in the 0 to 1023 range.

❖ Limited constructs for handling large data for complex designs

- Verilog does not allow changing the dimensions of an array once it is declared.

For e.g.: Consider the case of network packets that vary in size from one packet to the next. In Verilog, an array with the maximum packet size is declared. Hence, for smaller packets some elements are unused, which is a waste of memory.

❖ Complicated way of validating the functionality of a design and capturing the designer's intent

- No pre-defined constructs to automatically capture the designer's intent or to validate the functionality of a design.

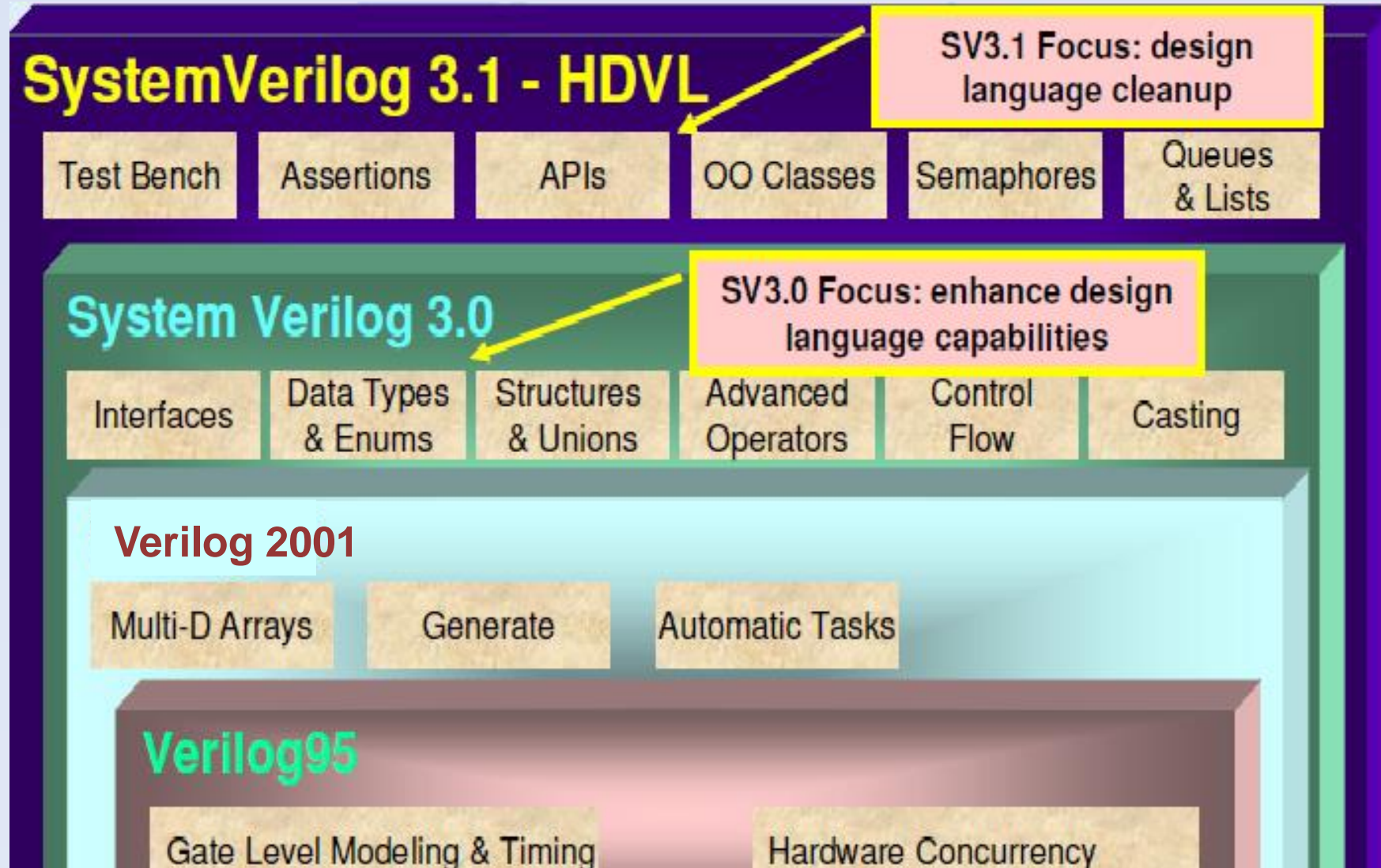
For e.g.: It is extremely difficult to code and automatically validate a common bus request and its corresponding grant functionality. Try coding and validating {if 'request' signal is true in the current clock tick, 'grant' signal must be true anytime within the next 4 clock ticks and then the 'request' signal becomes false immediately following 'grant'} in Verilog.

Limitations of Verilog for Verification Continued...

- ❖ **Lack of standardized interfaces or hooks to easily connect testbenches to the DUV**
 - Ports and clocks are connected via port maps and these occur at all places where instantiations are done. No unique interface is present in Verilog to help standardize this.
- ❖ **Code re-use is very minimal, requiring repeating similar code for different tests**
 - Lack of an Object Oriented Programming approach severely limits the amount of code that can be re-used in Verilog.
- ❖ **No way of the verifying functional coverage of any design as only code coverage is allowed by the IEEE 1364-2001 standard**
 - Although 100% line coverage, path coverage, expression coverage and toggle coverage is achieved for a design, it still does not mean that 100% functionality of the design has been covered.

Features of SV over Verilog

Evolution
of
Language
Features



Features of SV over Verilog Continued...

- With SystemVerilog, a complete verification environment can be built with its new features such as:
 - Dynamic data types and dynamic memory allocation
 - Interfaces
 - Object Oriented Programming (OOP)
 - Constraint Randomization
 - Functional Coverage constructs
 - Assertions

History of SystemVerilog

- **Background:**

- It was developed originally by Accellera (www.accellera.org) to dramatically improve productivity in the design of large gate-count, IP-based, bus-intensive chips
- Originally intended to be the 2005 update to Verilog
- Contains hundreds of enhancements and extensions to Verilog
- It was standardized by IEEE -1800 and published in 2005
- Officially superseded Verilog in 2009
- The current standard is IEEE 1800-2012

- **Purpose:**

- Is a Hardware Design and Verification Language (HDVL)
- Provides system-extensions to Verilog 2001 (IEEE 1364-2001). The IEEE 1364-2001 Verilog standard is a proper subset of SystemVerilog.

- **Backward Compatibility:**

- Backward compatible with Verilog 1995 / Verilog 2001

- **Tool Support:**

- Supported by most Electronic Design Automation (EDA) tool vendors: Mentor Graphics, Cadence Design Systems and Synopsys

2. Commonly Used Terminologies in SV

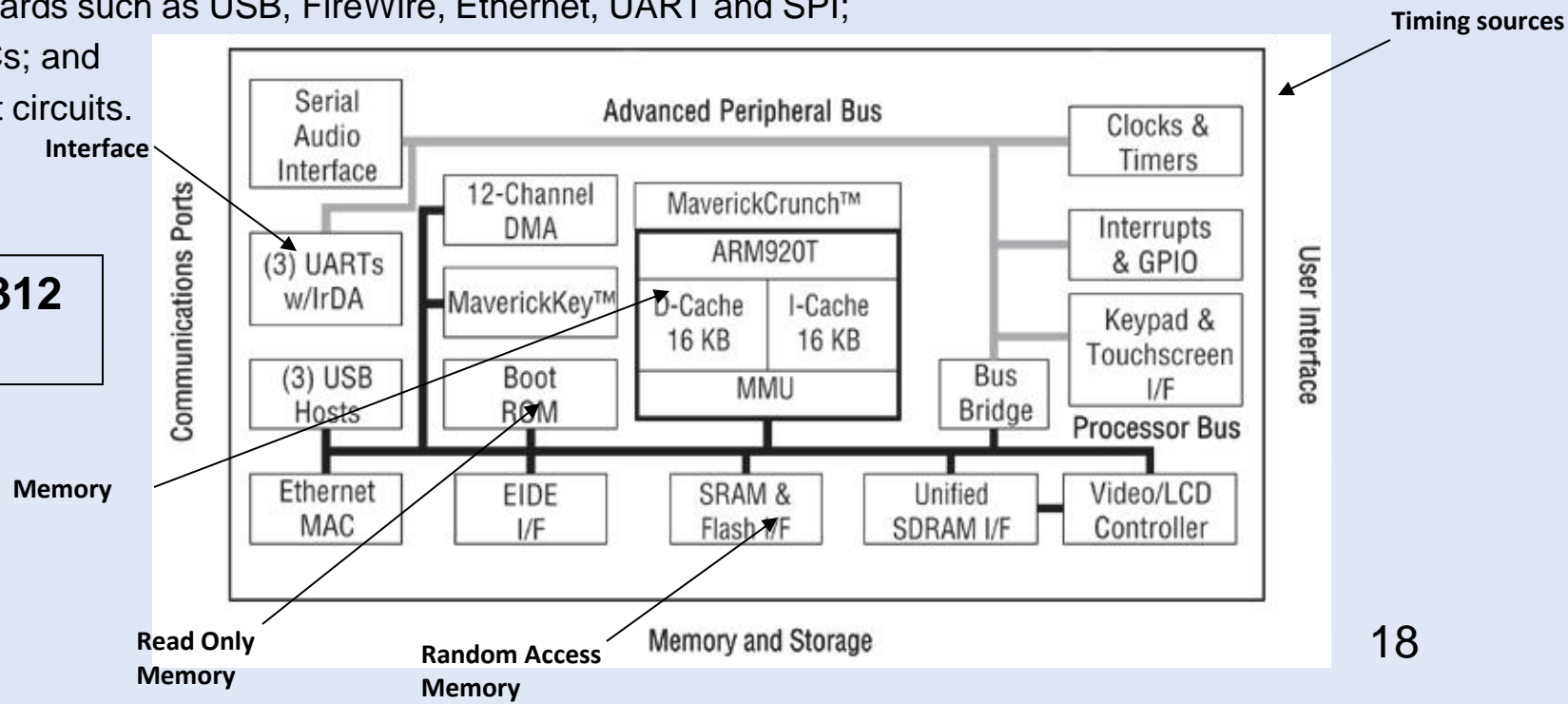
SoC: System-on-Chip

Definition: System-on-Chip or SoC is an integrated circuit which incorporates all necessary components of a computing system

Typical components found on an SoC are:

- A microcontroller, microprocessor or DSP core(s);
- Memory blocks including ROM, RAM, EEPROM and flash memory;
- Interconnection buses;
- Timing sources including oscillators and Phase-Locked Loops (PLLs);
- Peripherals including counter-timers, real-time timers and power-on reset generators;
- External interfaces including industry standards such as USB, FireWire, Ethernet, UART and SPI;
- Analog interfaces including ADCs and DACs; and
- Voltage regulators and power management circuits.

Cirrus Logic EP9312 SoC Architecture



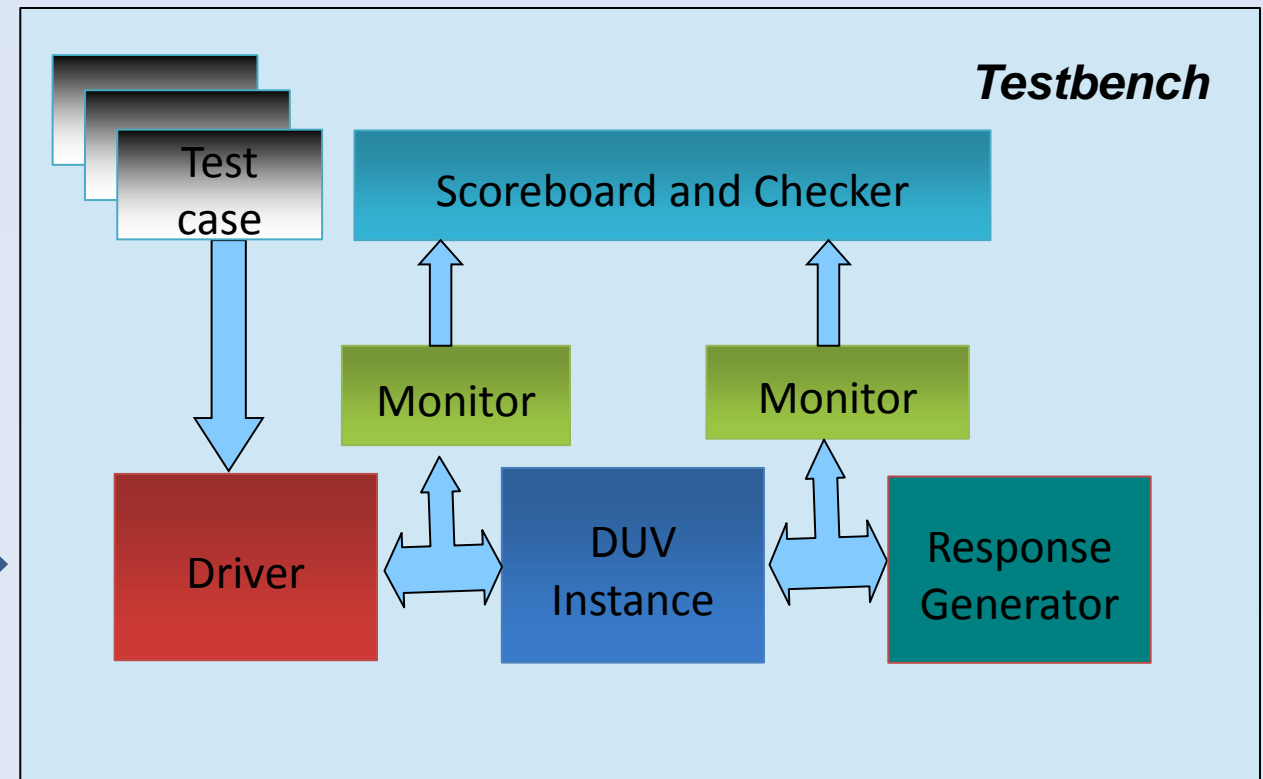
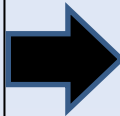
Testbench

Definition: An RTL testbench is the infrastructure needed to run RTL verification

Features:

- A testbench is a separate entity from the RTL design or the Design Under Verification (DUV)
- It is well structured and modular
- It is written in an HDL
- Any testbench should meet the following objectives:
 - Drive stimulus to the inputs of the RTL design
 - Monitor the response at the output of the RTL design
 - Automatically check the response with expected results
 - Maintain results along with the associated coverage metrics
 - Create warning, error and coverage reports

Typical Functional Blocks of the Generic Testbench Architecture



Protocol

Definition: Protocol is a “set of rules” defined that govern the transfer of data as well as control information between devices. For proper communication to happen, the protocol must define the syntax of data and address formats, and the semantics of transfer operations which includes the arbitration behavior when device contention occurs.

Features:

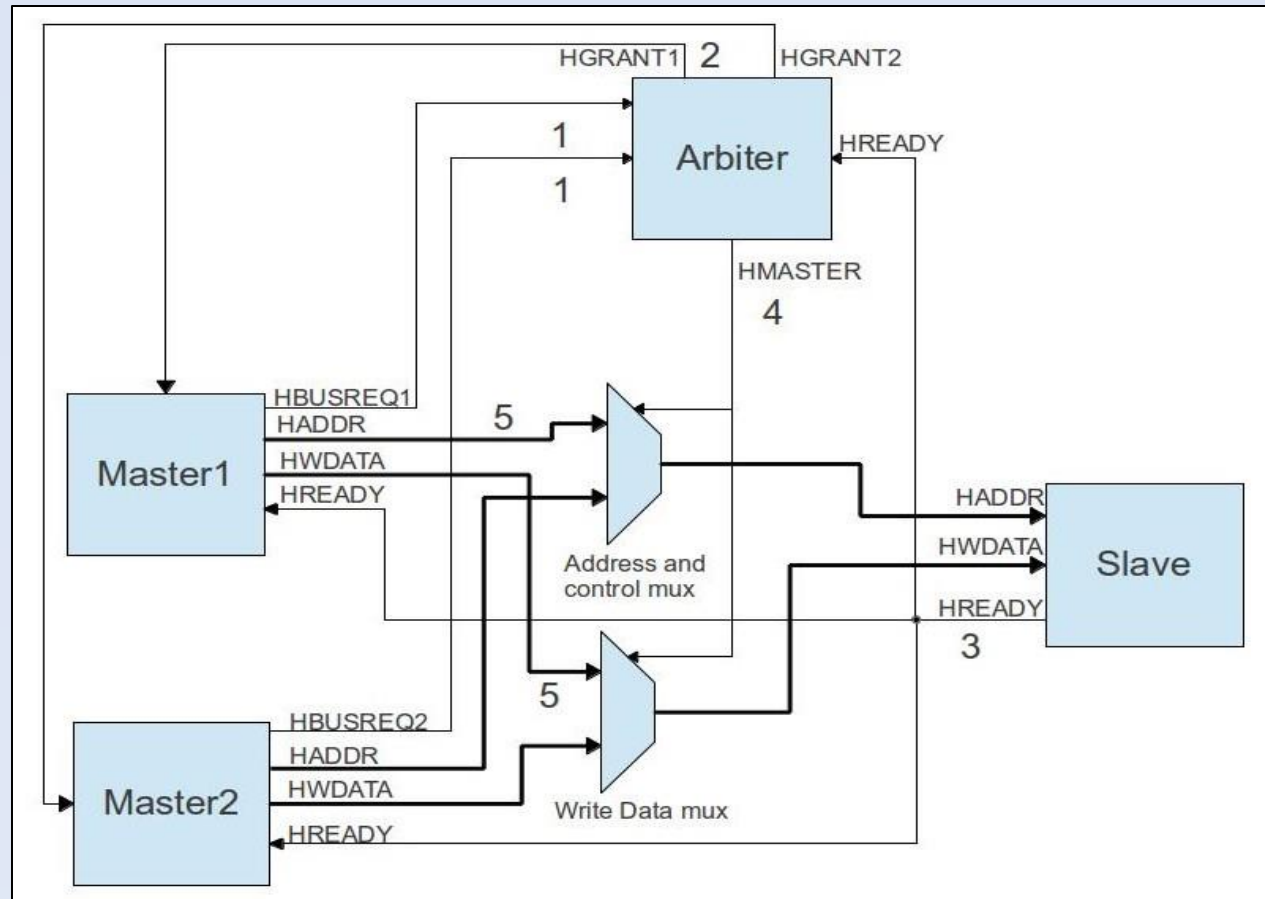
- Protocols help eliminate bus contention issues and also help avoid deadlock between devices.
- The transfers or transactions associated with a protocol can be represented with a behavioral model that accepts signal inputs and produces outputs.
- This behavioral model helps in analyzing the system prior to building and testing the actual hardware. It is commonly referred to as the Bus Functional Model (BFM) or Transaction Verification Model (TVM).

Consider the various components of a design communicating with each other through a common bus. In such a case, to ensure that proper communication happens, a bus protocol such as AHB is used. Based on the design's bandwidth and other constraints, an appropriate protocol can be chosen. Features of the AHB protocol are listed here:

- AHB stands for Advanced High-performance Bus.
- It is a revised version of the AMBA (Advanced Micro-controller Bus Architecture) protocol from ARM (www.arm.com).
- Supports several bus masters, burst transfers and pipelined operations.
- When multiple components communicate with each other on a common bus in a master-slave configuration, AHB uses an arbitration scheme to avoid bus contention issues.
- For more information, please refer http://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture and also go through the references and links given on that page.

Protocol Continued...

An example of the resolution of a bus contention issue in an AMBA AHB based system is explained. The main components of the system, as seen from the block diagram, are two masters (Master1 and Master2), a slave and an arbiter. The data transfer between a master and the slave is through a common bus.



BFM-Bus Functional Model

Definition: The Bus Functional Model or BFM is a behavioral description of the functionality of a given bus protocol

Features:

- The purpose of these models is to simulate system bus transactions prior to building and testing the actual hardware
- Bus functional models are simplified simulation models that accurately reflect the I/O level behavior of a device without modeling its internal computational abilities
- A BFM has two interfaces. On one side is a functional interface that accepts transactions and on the other side is a pin interface that operates the requisite bus protocol. The functionality of the BFM is to bridge those two interfaces
- BFMs are pre-verified and commonly used in testbenches as drivers and monitors
- Use of BFMs speeds up the process of verification
- For many standard protocols, BFMs are available as off-the-shelf Intellectual Properties (IPs)
- They are non-synthesizable

Examples:

A microprocessor BFM, a RAM BFM, a BFM for the AHB protocol explained on the previous slide. The bus functional model of a microprocessor would be able to generate Peripheral Component Interconnect (PCI) read and write transactions to a PCI device model to initialize and test its functionality

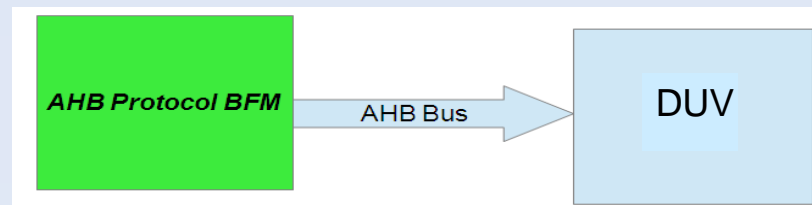
VIP-Verification Intellectual Property

Definition: VIP is an acronym for Verification Intellectual Property. VIPs are a set of completely verified, fully functional, re-usable and ready to use verification modules that typically consist of BFMs, traffic drivers, protocol monitors, and functional coverage blocks that easily plug into testbenches.

Characteristics:

- VIPs are standards-compliant, plug and play modules that help in reducing the overall verification time
- They enable verification engineers to completely focus their efforts in verifying designs rather than setting up complex verification environments
- VIP blocks integrate seamlessly into advanced verification environments, where testbenches are built using a mix of languages and methodologies like VHDL, Verilog, SystemC and UVM
- In order to support a mix of languages and methodologies, translation wrappers are written resulting in a multiple layered implementation. This layered approach limits simulation speed
- VIPs are analogous to design IPs or Implementation IPs (IIPs)

The AHB protocol BFM is an example of a VIP which can drive transactions on the AHB bus

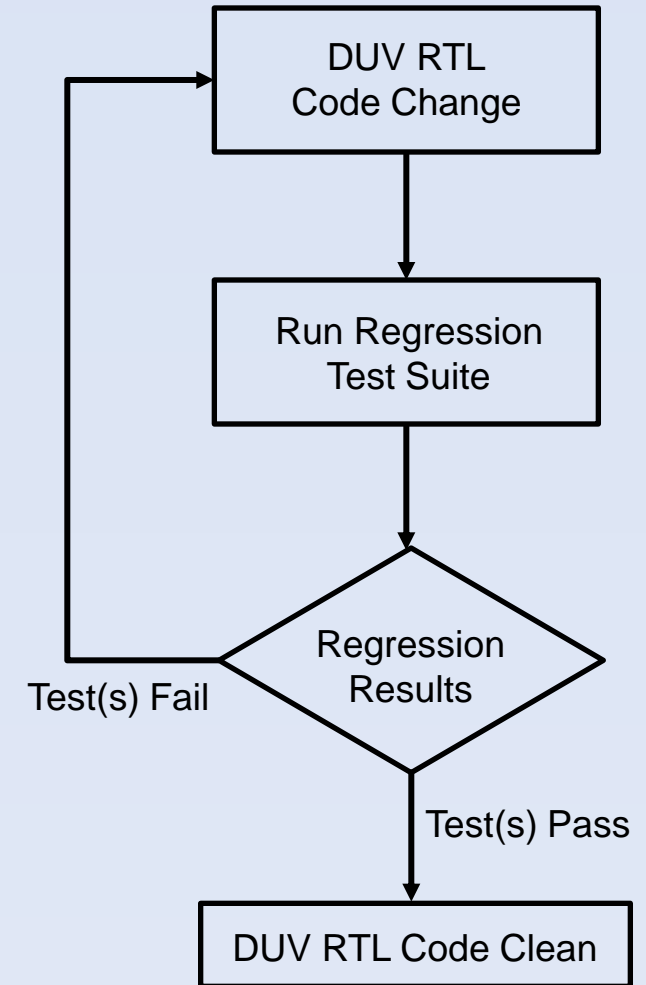


Regression

Definition: Regression is the process of re-running verification of the DUV with a known set of test cases. This is done in order to ensure that the functionality of the DUV is correct, and that new code changes have not introduced any new bugs in the design.

Features:

- Regressions are run every time the RTL design changes. These changes could be due to:
 - New additions and/or modifications that are made to the RTL design
 - A bug being fixed in the RTL design
- Ideally, the regression test suite is made up of many small tests, instead of a few large tests. This helps isolate errors faster.
- They are regularly run at different phases of the design and verification cycle.
- Regressions should be run before releasing the design for synthesis or before generating coverage reports to ensure that the design is clean.



Randomization

Definition: Randomization is the process of creating random stimuli.

Consider that all positive 32-bit integers need to be verified for some purpose. The range for this is from 0 to $(2^{32})-1$. In exhaustive testing, all the 2^{32} integer values need to be verified. However, in random testing only a few random values from the range are verified, as long as they are valid and legal values and satisfy the coverage metrics. For the above example, the coverage metrics state that 1000 unique representative positive integers are good enough to verify all the positive 32-bit integers. Then random testing would involve randomly generating 1000 unique values within the 0 to $(2^{32})-1$ range for verification.

Exhaustive testing, which, in many cases is impossible to do, or takes humongous amounts of time, could thus be replaced with random testing.

In Verilog, the **\$random()** function is used to generate random numbers. The function returns a new 32-bit random number each time it is called. This random number is a signed integer - it can be positive or negative.

Randomization can be applied on data and control signals and is extensively used during verification.

Directed testing checks specific features of a design and can only detect expected bugs. Random testing detects bugs that one did not anticipate.

Constrained Random Verification

Definition: Verification using automatically generated constrained random stimuli is called as Constrained Random Verification. The constraints or rules determine the legal values that can be assigned to those random variables.

Consider a machine whose instructions have an opcode and two 32-bit operands, `op_a` and `op_b`. The constraint here is that it is necessary for '`op_a`' to only take values less than `h'ff`, while '`op_b`' has to only take values greater than `h'ff`.

- Generating random numbers for the above example and then constraining them in a Verilog testbench is tedious involving many lines of code
- Imagine specifying the constraint `((op_a < h'ff) && (op_b > h'ff))` such that the generated random numbers automatically adhere to those constrained values. Much easier, isn't it ?

Constrained random verification is a key feature and is used extensively in testbenches

- Helps to achieve better functional coverage and in testing corner cases
- Using this feature drastically reduces simulation time, thereby contributing to the reduction in the verification gap

Assertion Based Verification (ABV)

Definition: An assertion is a statement that specifies behavior of a design. Verification of a design using assertions is called as Assertion Based Verification.

Features of Assertions:

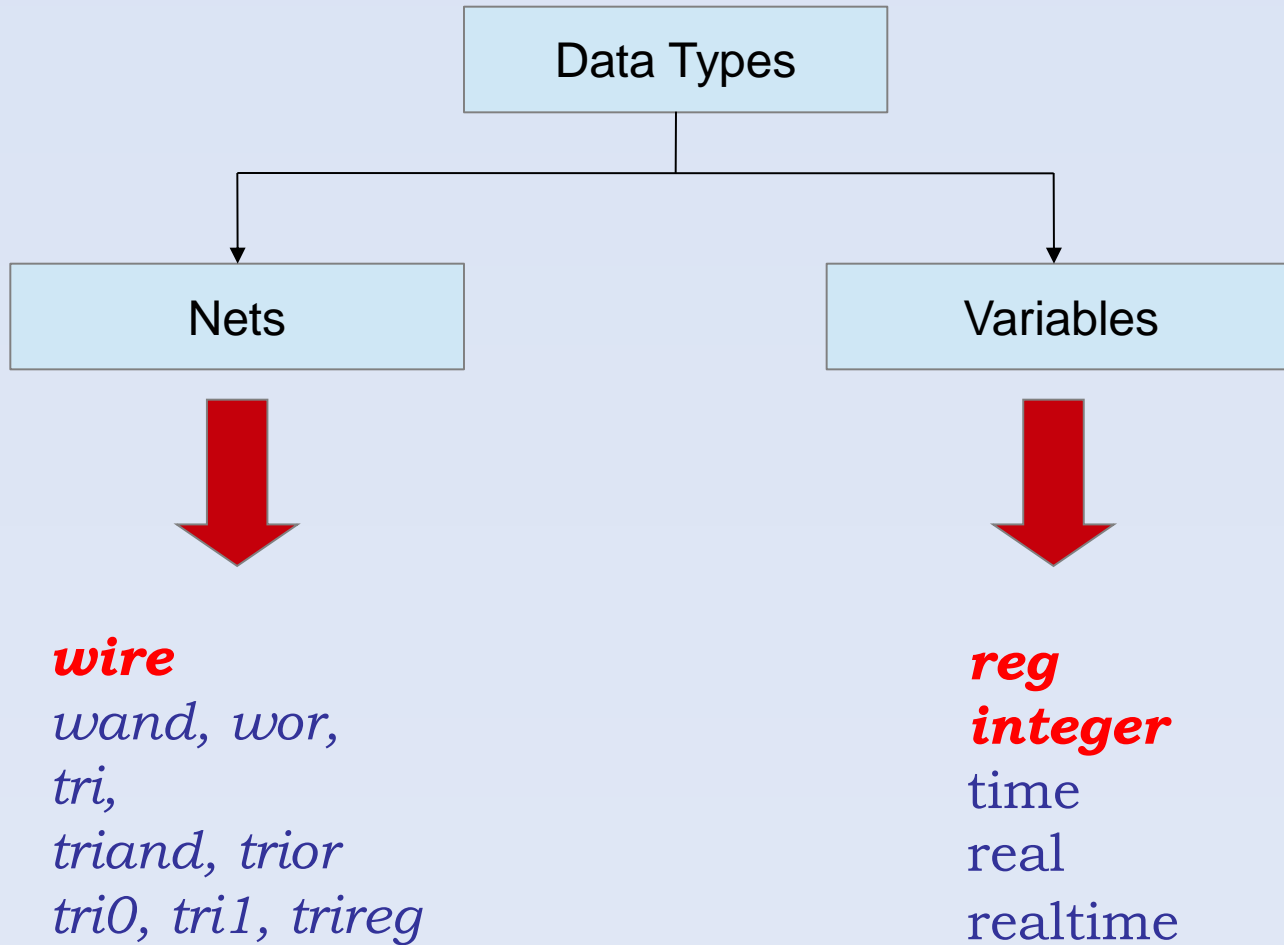
- Analogous to assertions used in programming languages like C, C++ or Java
- They are used to:
 - Document the functionality of the design
 - Check whether the intent of the design is met over simulation time
- They can be specified by:
 - The design engineer as part of the RTL design
 - The verification engineer as part of the testbench
- An assertion is a behavioral statement that is required to be true, and a directive to verification tools to verify that this statement holds true

Examples:

- 1) Assert the bus grant happens not more than 6 clock cycles after the bus request arrives
- 2) Assert that the divide by zero condition never occurs during the division operation

3. Data Types

Overview of Verilog data types



- A **wire** is a net type used to model physical connections - it does not store values, but takes the value of its driver
- A **reg** variable is used to model storage elements such as latches, flip-flops and memories - it stores a value
- An **integer** is a variable of 32-bit width which stores signed values

Overview of Verilog data types Continued...

- The value set for variables and nets contains 4-values: 0, 1, x, and z
 - 0 — represents a logic zero value or a false condition
 - 1 — represents a logic one value or a true condition
 - x — represents an unknown logic value
 - z — represents a high-impedance or open circuit value
- A net or reg declaration without a range specification is considered 1-bit wide and is known as a scalar. Multiple bit net and reg data types are declared by specifying a range and are known as vectors
- Verilog data types lead to increased simulation run times and more memory utilization because they are static in nature - all variables are alive for the entire simulation time

New data types in SystemVerilog

The following SV data types were introduced to help in efficient coding of testbenches, in reducing simulation run times and in reducing memory usage.

- Logic
- Two-state
 - int, shortint, longint, bit, byte
- Packed arrays, Unpacked arrays, Mixed Multi-dimensional arrays, Dynamic arrays, Associative arrays and Queues
- Structures and Unions (Aggregate types)
- Strings (Enhanced from Verilog)
- Events (Enhanced from Verilog)
- User-defined data type
- Enumerated data type
- Classes (explained in detail in the next Chapter)

Logic

Need for the Introduction of the Logic data type in SV

Verilog 2001 has wire and reg data types and users are confused with the rules on their usage. SystemVerilog has introduced the logic data type which can be used in place of both the wire and the reg data types, with the caveat that the wire does not have multiple drivers.

Verilog 2001:
`module pl_soc (input wire ip1, input wire ip2,
output reg op1, output wire op2);`

confusing?  simpler!

SystemVerilog:
`module pl_soc (input Logic ip1, input Logic ip2,
output Logic op1, output Logic op2);`

Verilog 2001:
`wire w;
assign w = memdata;

reg r;
always @*
r = memdata;`

SystemVerilog:
`// Easier defining everything as Logic type
Logic w;
assign w = memdata;

Logic r;
always @*
r = memdata;`

Definition: The logic data type is a 4-state type that can take values 0, 1, x and z. It can inter-changeably be used in place of the wire and reg data types as long as the wire data type does not have multiple drivers.

Syntax : logic variable_name;

Examples:

```
logic rst_n, clk;           // 1-bit wide 4-state variables – 'rst_n' and 'clk'  
logic [31:0] word32;        // A 32-bit wide 4-state variable – 'word32'
```

Characteristics of the Logic data type:

- By default this is an unsigned data type and its initial value is x
- Cannot be driven by multiple drivers. The net data type 'wire' is to be used in case of multiple drivers.
- In addition to being a variable, it can be driven by continuous assignments and gates
- If all signals are declared using the logic type, SystemVerilog infers a variable or net based on context

Applications of the Logic data type:

- The logic data type can have only one driver and hence can be used to find netlist bugs. If all the signals in a design are declared as logic rather than wires and regs, then a compilation error will occur for cases where multiple drivers for a signal are present. It makes the code simpler and readable in most cases.

Two-State (int, shortint, longint, bit, byte)

Need for the Introduction of Two-State Logic data types in SV

SystemVerilog has introduced several 2-state (0 and 1 value) data types to improve simulator speed and reduce its memory usage. In comparison, the use of Verilog 2001 4-state (0, 1, x and z value) data types require additional bits to encode the x and z values, thereby slowing down simulation and increasing memory usage

Two-State Type	Description
int	32-bit size, signed
shortint	16-bit size, signed
longint	64-bit size, signed
bit	User-defined size, unsigned
byte	8-bit size, signed

Definition: A two-state data type is something that can only take values 0 and 1. int, shortint, longint, bit and byte are the new 2-state data types introduced in SV. By default int, shortint, longint and byte are signed data types, while bit is an unsigned data type.

Syntax : 2_state_data_type variable_name;

Examples:

```
int a;           // 'a' is a 2-state, 32-bit signed integer
shortint b;      // 'b' is a 2-state, 16-bit signed integer
longint c;       // 'c' is a 2-state, 64-bit signed integer
bit d;           // 'd' is a 2-state, single bit value
byte e;          // 'e' is a 2-state, 8-bit signed integer
bit [7:0] f;     // 'f' is a 2-state, 8-bit unsigned integer
```

Can count upto 127 as this is a signed data type

Can count upto 255 as this is an unsigned data type

Characteristics of Two-State data types:

- Their initial value is 0 and they cannot be used to represent the 'x' un-initialized state
- Can assign 4-state values to a 2-state variable - x and z values are automatically converted to a 0 value

Applications of Two-State data types:

- These data types are mostly used only in testbenches. Care must be taken to connect 2-state variables in a testbench with 4-state variables in the DUV. 'x' and 'z' values received from the DUV might automatically be converted to 0's in the testbench leading to undesirable behavior

The \$isunknown() operator is used to check whether any bit of an expression or a variable is 'x' or 'z'. It returns a value 1, if 'x' or 'z' is found

```
Example:
if(!$isunknown(somevar)==1))
    $display (" 2-state variable detected");
else
    $display (" 4-state variable detected");
```

Review of the Array data type from Verilog 2001

An array is a collection of elements, all of the same type, and accessed using its name and one or more indices. The array data type for a net or variable is an aggregate declaration that was either a scalar or a vector

Examples:

```
reg a[7:0];           // scalar reg 'a'
wire [0:7] b[7:0];    // Eight-bit wide vector wire 'b' indexed from 0 to 7
reg [7:0] memory[0:255]; // Declaration of a 'memory' of 256 8-bit registers. The indices are 0 to 255

reg [1:16] rega;      // A 16-bit register is not the same
reg mema [1:16];     // as a memory of 16 1-bit registers
```

Verilog 2001 required that the low and high array limits must be part of the array declaration. SystemVerilog has introduced the **compact array declaration style**, where just giving the array size along with the array name declaration is enough

Examples:

- Single-dimension array declaration

```
int sdarr[0:15]; // Verilog 2001 verbose declaration style – 0 and 15 are the lower and higher array limits
int sdarr[16];   // Same example as above, but using SystemVerilog's compact declaration style – only the array size (16) is given
```
- Multi-dimension array declaration

```
int mdarr[0:7][0:7]; // Verilog 2001 verbose declaration style
int mdarr[8][8];     // Same example as above, SystemVerilog's compact declaration style
```

Packed Arrays

Need for the Introduction of Packed Arrays in SV

SystemVerilog has introduced packed arrays as a mechanism for sub-dividing a vector into sub-fields, which are accessed as array elements. The vector can be accessed as a whole (single value) as well as individual array elements. A packed 64-bit register can be accessed as a single entity or as eight 8-bit values or as sixteen 4-bit values.

Characteristics of a Packed Array:

- It is stored as a contiguous set of bits
- It can be made of only single bit data types (bit, logic, reg), enumerated types (explained later), and recursively other packed arrays
- If a packed array is declared as signed, then the array viewed as a single vector shall be signed. The individual elements of the array are unsigned unless they are of a named type declared as signed
- Packed arrays allow arbitrary length integer types; therefore, a 48-bit integer can be made up of 48 bits
- The maximum size of a packed array can be limited, but shall be at least 65536 (2^{16}) bits
- The following operations can be performed on packed arrays. Assume the declaration of an 8-bit packed array 'MPA' for the examples shown.
 - Assignment of an integer value. Example: `MPA = 8'b10101010;`
 - Use of the packed array as an integer in an expression. Example: `if ((MPA + 4) > 16)`

Applications of Packed Arrays:

- They are used to store packet structures on which bit-select and part-select operations could be performed

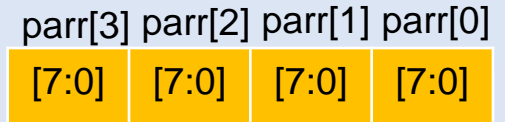
Definition: A packed array is one where the dimensions of the array is declared before the array object name

Syntax : `<data_type> <range (optional)>* <array_name>;`

`<range (optional)>*`
implies 0 or more
range declarations

Examples:

```
reg single;           // A packed 1-bit register
bit [31:0] word_arr;  // Packed array of 32-bit vectors
reg [0:15] myreg;     // A packed 16-bit register
logic [3:0][7:0] parr = 32'h0; // Initializing a packed 32-bit vector with
                               // four 8-bit sub-fields
```



Unpacked Arrays

Need for the Introduction of Unpacked Arrays in SV

SystemVerilog has introduced unpacked arrays so that one can assign to or manipulate entire arrays at once

Definition: An unpacked array is one where the dimensions of the array is declared after the array object name

Syntax : <data_type> <array_name> <range>⁺;

<range>⁺ implies 1 or more range declarations

Examples:

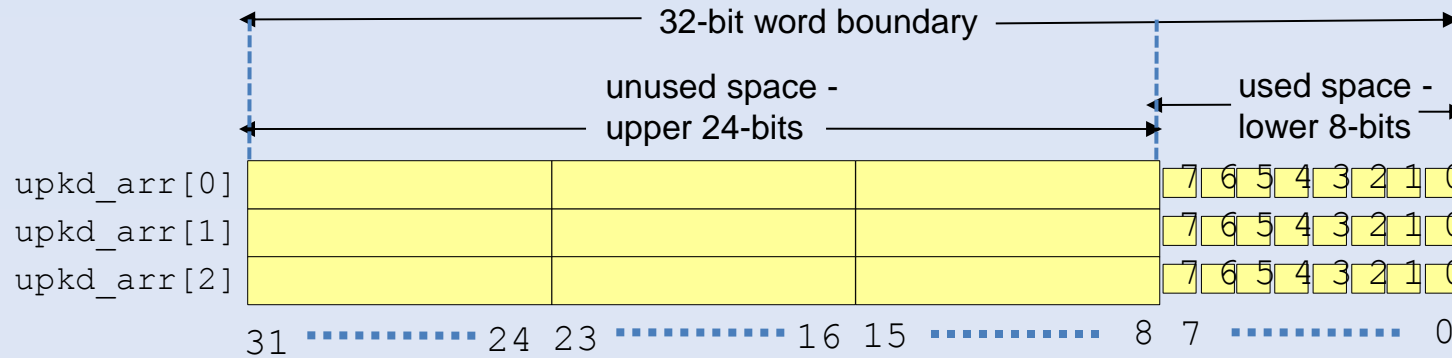
```
int uparr[0:15][0:31];    // Unpacked array declaration using ranges
int uparr[16][32];        // Same as the above declaration, but using array sizes
real float_point[0:1023]; // Unpacked array of real type
int iuarr[4]='{4,3,2,1};  // Initializing an unpacked array of 4 elements
byte banner[0:12] = "hello asics\n"; // String literal assigned to an unpacked array of bytes
```

Characteristics of an Unpacked Array:

- The unpacked dimensions can be arranged in memory in any way that the simulator chooses – does not have to be contiguous
- They can be of any data type
- If an unpacked array is declared as signed, then this applies to the individual elements of the array, since the whole array cannot be viewed as a single vector
- The expressions that specify an address range shall be constant integer expressions
- Implementations may limit the maximum size of an unpacked array, but they shall allow least 16777216 (2^{24}) elements
- They could be fixed-size arrays, dynamic arrays, associative arrays or queues (explained in later slides)
- SystemVerilog simulators store each element on a 32-bit word boundary. An unpacked array, such as the one shown in following example, stores the values in the lower 8-bits of the 32-bit word boundary, while the upper 24 bits are unused:

Unpacked Arrays Continued...

Example:
`bit upkd_arr[0:2][0:7];`



Applications of Unpacked Arrays:

- They are used to model Random Access Memories (RAMs), Read Only Memories (ROMs) and register files where one element is accessed at a time

Operations on Arrays

SystemVerilog allows the following operations to be performed on all arrays, packed or unpacked. Assume that arrays X and Y in the examples below are of the same type and size.

- Reading and writing the entire array. E.g., $X = Y$
- Reading and writing a slice of the array. E.g., $X[0:7] = Y[8:15]$
- Reading and writing an element of the array. E.g., $X[10] = Y[20]$
- Equality operations on the array or slice of the array. E.g., (1) $\text{if } (X==Y) \dots$
(2) $\text{while } (X[0:7] != Y[8:15]) \dots$

These are the most commonly used operations on arrays. A few other operations are present, the details of which the learner can obtain from page 110 of reference [3] - The SystemVerilog IEEE 1800-2012 LRM.

Mixed Multi-Dimensional Arrays

SystemVerilog allows the mixed use of packed and unpacked dimensions in an array definition leading to a mixed multi-dimensional array

Example:

```
logic [2:0][9:0] mymix_array [0:3] [0:5] [0:7]; // Is a mixed 5-dimensional array with 2 packed
// dimensions - [2:0] and [9:0], and 3 unpacked
// dimensions - [0:3], [0:5] and [0:7]
```

Multi-dimensional array access via indexing must adhere to the following rules:

- All unpacked dimensions are first referenced from the left-most to the right-most dimension in that order
- All packed dimensions are then referenced from the left-most to the right-most dimension in that order

Example:

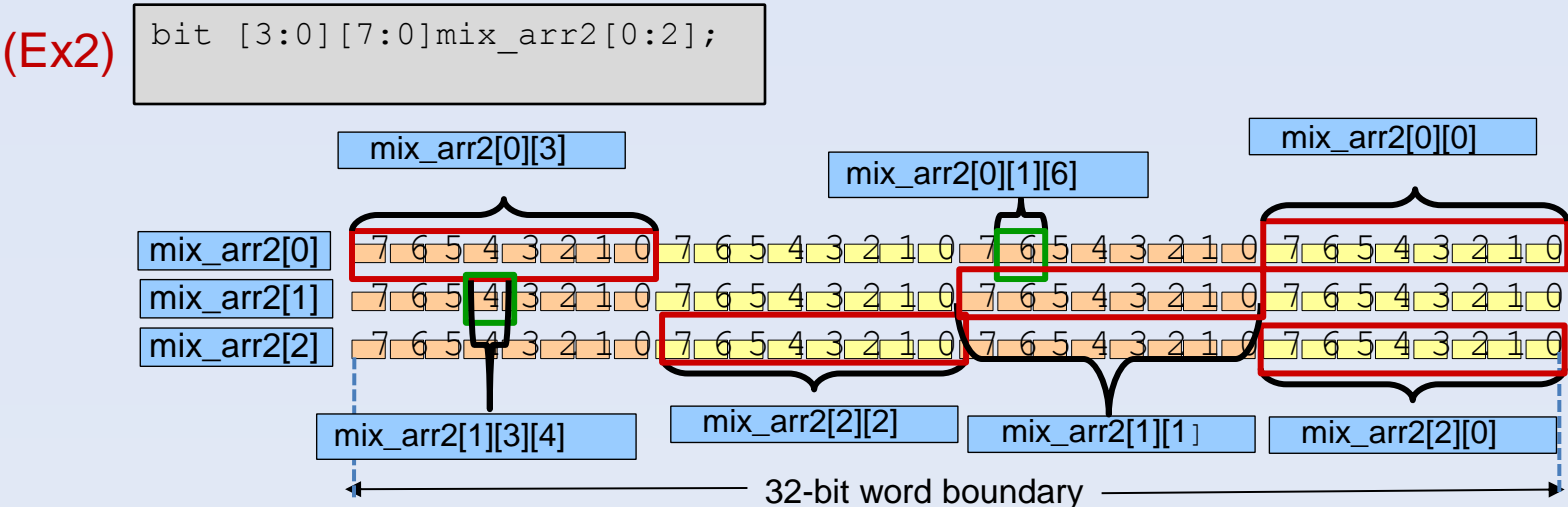
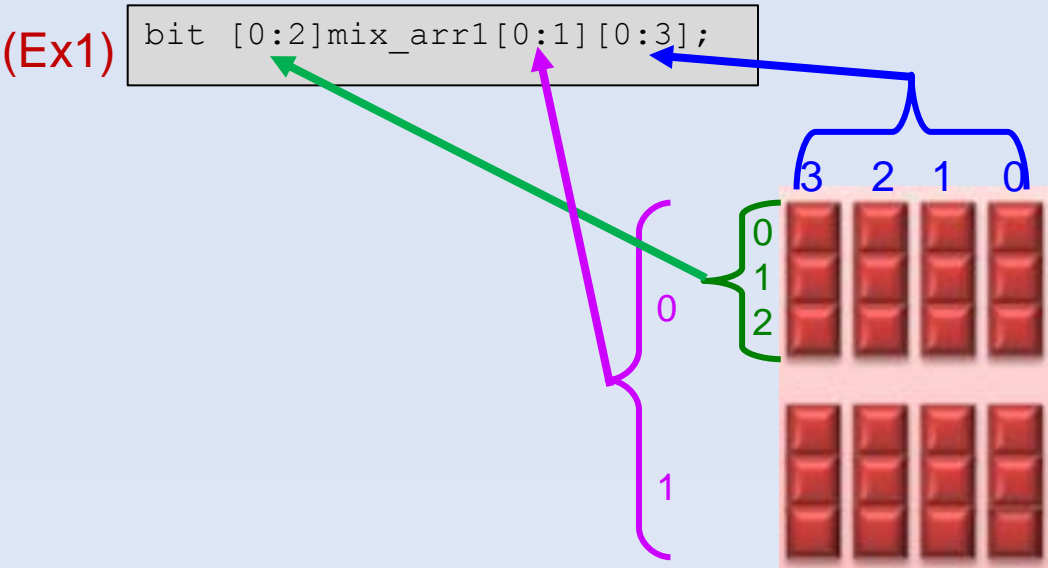
Left Right Left Right
logic [2:0][9:0] mymix_array [0:3] [0:5] [0:7];

mymix_array [0] [4] [6] [2] [7] = 1'b1;

The 0 index of the array is from the 1st unpacked dimension [0:3]
The 4 index of the array is from the 2nd unpacked dimension [0:5]
The 6 index of the array is from the 3rd unpacked dimension [0:7]
The 2 index of the array is from the 1st packed dimension [2:0]
The 7 index of the array is from the 2nd packed dimension [9:0]

Mixed Multi-Dimensional Arrays Continued...

Examples to see how different dimensions of mixed arrays are laid out in memory to understand how the array elements are accessed



Dynamic Arrays

Need for the Introduction of Dynamic Arrays in SV

The Verilog 2001 array type is a fixed-size array, which has already been set at compile time. For the case where the size of the array is not known until run-time, SystemVerilog has introduced dynamic arrays that can be allocated and re-sized during simulation so that a minimal amount of memory is used.

Definition: A dynamic array is an unpacked array whose size can be set or changed at run time. The space for a dynamic array does not exist until the array is explicitly created at run-time.

Syntax : <data_type> array_name [];

Examples:

```
integer i_dynarr[];           // Dynamic array of integers
bit [31:0] word_dynarr[];     // Dynamic array of 32-bit vectors
integer dd_dynarr[8][];       // Fixed-size unpacked array composed of 8 dynamic sub-arrays of integers
integer md_dynarr[][];        // A dynamic array of dynamic arrays of integers
```

Characteristics of a Dynamic Array:

- It is declared with empty square brackets []. This means that the array size is not specified at compile time, rather it will be given later at run time
- The default size of an un-initialized dynamic array is 0
- Dynamic arrays support all variable data types as element types, including arrays
- An out-of-bound access in a dynamic array leads to a run-time error

Applications of Dynamic Arrays:

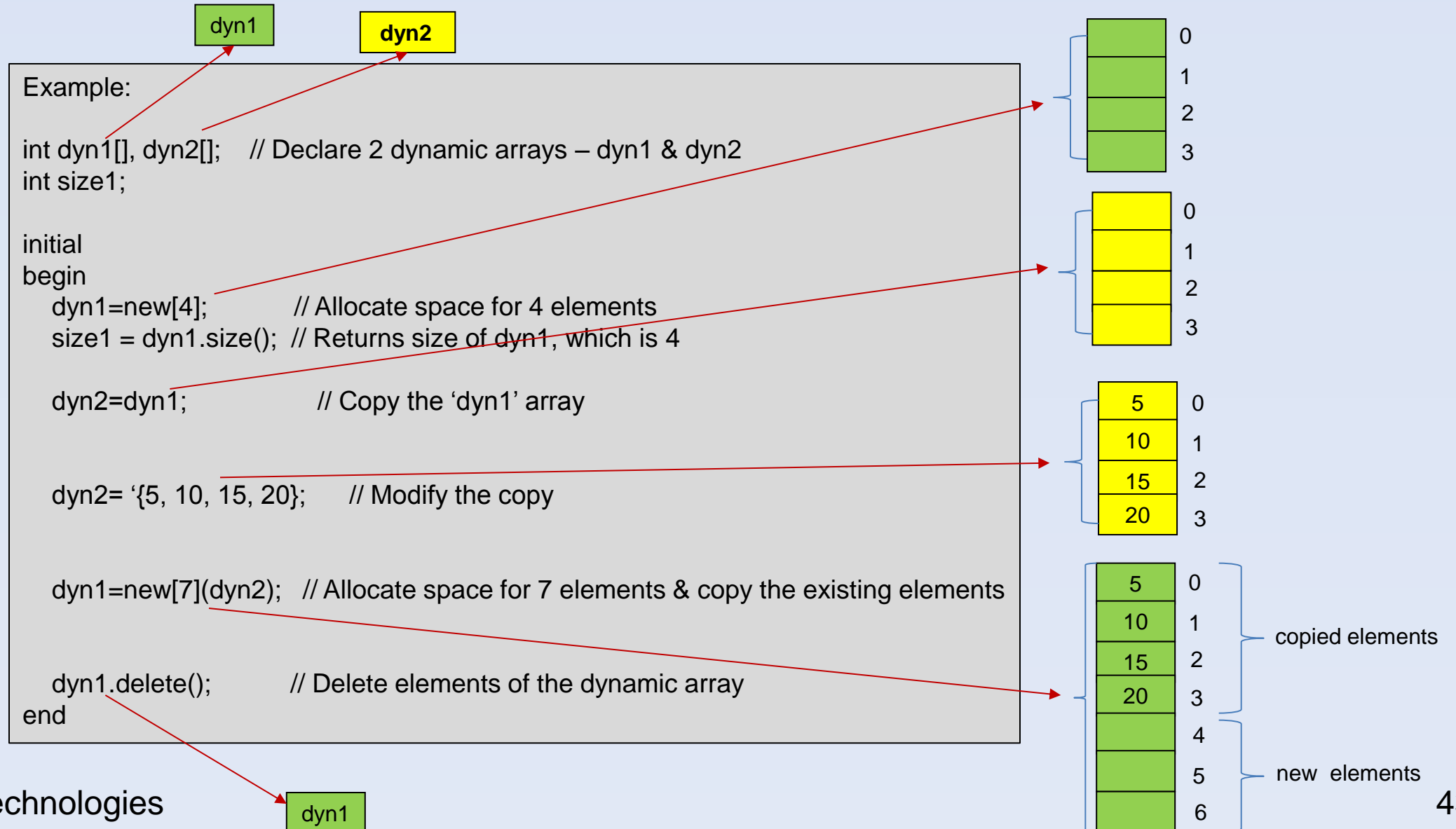
- Say random control signals need to be generated by the testbench during simulation. If a fixed-size array is used, it would have to be sized large enough to hold the maximum number of signals possible. In reality, only a subset of control signals would be used, thus wasting the remaining memory. A dynamic array that can be allocated and re-sized during simulation will avoid this unnecessary memory allocation.

Dynamic Array Methods

SystemVerilog provides a constructor and built-in methods to manipulate dynamic arrays. The methods or functions are called after the dynamic array object name with a period “.” between the object name and the function name.

- Constructor `new[value]`: Sets the size of a dynamic array and initializes its elements or changes the size of the array at run-time. If the value is zero, the array shall become empty. If the value is negative, then it is an error.
- Function `int size()`: The `size()` method returns the current size of a dynamic array or returns zero if the array has not been created
- Function `void delete()`: The `delete()` method empties the array, resulting in a zero-sized array.

Dynamic Array Methods Continued...



Associative Arrays

Need for the Introduction of Associative Arrays in SV

In addition to being able to handle to a very large address space, SystemVerilog introduces associative arrays that are used to store sporadic or sparse data entries. SV allocates memory for an element only when data is written into the associative array. This is a big advantage in that it saves tremendous amounts of memory space in comparison with dynamic arrays where the full memory is allocated even if only one element is written into it. In Verilog array indices can only be integer values. However, associative array indices in SV can be of any type.

Definition: An associative array is an unpacked array data-type in SV. It does not have any storage allocated until it is used, and the index type used to access elements is not restricted to integers.

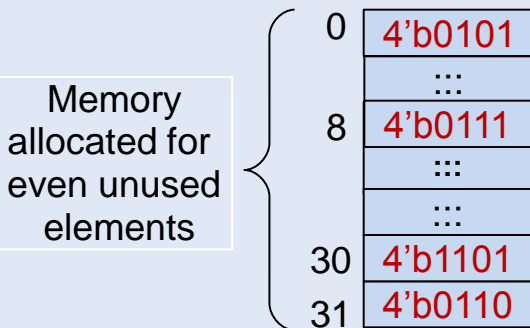
Syntax : `<data_type> array_name [index_type];`
where, index_type is the data_type to be used or can be the wildcard “*”

Examples:

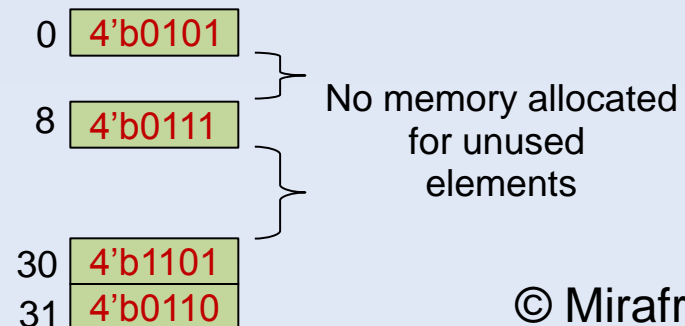
```
integer i_array[*];           // Un-initialized associative array of integers (unspecified index)
integer array_s[string] = `{“INIT”:0, “DEEPSLEEP”:30, “FULLPOWER”:10, “MEMSLEEP”:20, default:-1};
                                // Initialized associative array of integers, indexed by string
bit [3:0] array_a[int];       // Un-initialized associative array of a 4-bit vector, indexed by int
initial
begin
    array_a[0]  = 4'b0101;
    array_a[8]  = 4'b0111;
    array_a[30] = 4'b1101;
    array_a[31] = 4'b0110;
end
```

Memory allocation depiction if “array_a” from the example was declared using:

A standard array (Contiguous)



An associative array (Non-Contiguous)



Associative Arrays Continued...

Characteristics of an Associative Array:

- It can be stored by the simulator as a hash (lookup) table for extremely fast access of its elements. A hash table contains an array of groups of elements. A function called hash function generates a unique key to compute the index into this array, from which the correct array element value is obtained
- Associative array elements are unpacked
- The type of the array index used to access elements is restricted to int, integer, string, class and the wildcard “*”
 - For a wildcard associative array, non-integral index values are illegal and a string literal index is automatically cast to a bit vector of equivalent size
- Elements in an associative array can be accessed in a similar way as those in a one-dimensional array
- If an attempt is made to read an invalid (non-existent) associative array entry, then the simulator will issue a warning and will return the value ‘x’ for a 4-state integral type or a value ‘0’ for a 2-state integral type

Applications of Associative Arrays:

- They are used to model Content Addressable Memories (CAMs)
- Random read or write tests for verification of memories could use associative arrays for storing data only for addresses which have been written. This would take up significantly much lesser memory than the entire array normally used in Verilog.

Associative Array Methods

SystemVerilog provides several built-in methods to manipulate associative arrays as well as iterate over its indices. These methods or functions are called after the associative array object name with a period “.” between the object name and the function name. The function arguments (if any) are inputs.

- Function `int num()` and Function `int size()`: Return the number of entries in the associative array. If the array is empty, then return 0.

Example 1:

```
integer imem[*]; // Un-initialized associative array of integers (unspecified index)
imem[ 2'b3 ] = 1; // Populating element number 2'b3 of imem
imem[ 16'hffff ] = 2; // Populating element number 16'hffff of imem
imem[ 4b'1000 ] = 3; // Populating element number 4'b1000 of imem
$display( "%0d entries\n", imem.num()); // prints "3 entries"
```

- Function `void delete()`: The `delete()` function removes all the elements in the array
- Function `void delete(data_type index)`: Removes the entry at the specified index. If the entry to be deleted does not exist, the function issues no warning.

Example 2:

```
integer map[string];
map[ "data" ] = 1;
map[ "address" ] = 2;
map[ "control" ] = 3;
map.delete( "control" ); // Remove entry whose index is "control" from array "map", i.e., value 3
map.delete(); // Remove all entries from the associative array "map"
```

Associative Array Methods Continued...

- Function `int exists(data_type index)`: Checks if an element exists at the specified index within the given array. It returns 1 if the element exists, otherwise it returns 0

Example 3:

```
integer map[string];
integer count = 0;
map[ "data" ] = 1;
map[ "address" ] = 2;
map[ "control" ] = 3;
if(map.exists( "data" )) // Checks to see whether an element exists at the index 'data' in array 'map'.
                        // In this case the value 3 exists and hence the function returns a 1
    count = count + 1;
```

- Function `int first(data_type index)`: Assigns to the given index variable the value of the first (smallest) index in the associative array. It returns 0 if the array is empty, and 1 otherwise

Example 4:

```
string s;
integer map[string];
map[ "address" ] = 1;
map[ "control" ] = 2;
map[ "data" ] = 3;
if(map.first(s)) // Assigns the first index "address" to the string 's' and returns 1 as the array is not empty
    $display( "First entry is : map[ %s ] = %0d\n", s, map[s] ); // Display statement is executed to show "First entry is : map[address] = 1"
```

Associative Array Methods Continued...

- Function `int last(data_type index)`: Assigns to the given index variable the value of the final (largest) index in the associative array. It returns 0 if the array is empty, and 1 otherwise

Example 5:

```
string s;  
integer map[string];  
map[ "address" ] = 1;  
map[ "control" ] = 2;  
map[ "data" ] = 3;  
if(map.last(s)) // Assigns the last index "data" to the string 's' and returns 1 as the array is not empty  
    $display( "Last entry is : map[ %s ] = %0d\n", s, map[s] ); // Display statement is executed to show "Last entry is : map[data] = 3"
```

- Function `int next(data_type index)`: Finds the smallest index whose value is greater than the given index argument. If there is a next entry, the index variable is assigned the index of the next entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0

Example 6:

```
string s;  
integer map[string];  
map[ "address" ] = 1;  
map[ "control" ] = 2;  
if(map.first(s)) // Assigns the first index "address" to the string 's' and returns 1 as the array is not empty  
do  
    $display( "%s : %d\n", s, map[s] );  
while(map.next(s)); // This do-while loop iterates twice, the first time for "address" and the second time for "control".  
// After that the map.next(s) function returns a 0 as there are no more indices, and the loop terminates.
```


Associative Array Methods Continued...

- Function `int prev(data_type index)`: Finds the largest index whose value is smaller than the given index argument. If there is a previous entry, the index variable is assigned the index of the previous entry, and the function returns 1. Otherwise, the index is unchanged, and the function returns 0

Example 7:

```
string s;  
integer map[string];  
map[ "address" ] = 1;  
map[ "control" ] = 2;  
if(map.last(s)) // Assigns the last index "control" to the string 's' and returns 1 as the array is not empty  
do  
    $display( "%s : %d\n", s, map[s]);  
while(map.prev(s)); // This do-while loop iterates twice, the first time for "control" and the second time for "address".  
                    // After that the map.prev(s) function returns a 0 as there are no more indices, and the loop terminates.
```

Queues

Need for the Introduction of Queues in SV

The queue data type was introduced in SystemVerilog to combine the dynamic storage property and fast access of elements. The main advantage is that one can add or remove elements anywhere in a queue, without the storage penalty of a dynamic array which has to allocate a new array and copy its entire contents. Also, as done in a fixed-size array, any element in the queue can be accessed directly using an index without any need for accessing elements before it.

Definition: A queue is a variable-size, ordered collection of homogeneous (same type) unpacked array elements. It supports constant time access to all its elements as well as constant time insertion at the beginning and removal at the end of the queue respectively.

Syntax : <data_type> queue_name [\$];

Examples :

```
string ctrl_names[$] = { "control", "enable"}; // An initialized queue of
// strings called "ctrl_names"
// with two elements

integer Q[$] = { 4, 2, 7 }; // An initialized integer queue called "Q"
// containing three integers

byte bytq[$]; // A queue of bytes
```

ctrl_names

Position 0	Position 1
control	enable

Q

Position 0	Position 1	Position 2
4	2	7

Characteristics of a Queue:

- Each element in a queue is identified by a number that represents its position within the queue, with 0 representing the first, and \$ representing the last. In the second example shown, the integer queue 'Q' stores the value 4 at position 0. This is the first element of the queue. Value 2 is stored at position 1 and finally value 7 is stored at position 2, which is the last element of the queue
- A queue can have variable length, including a length of zero. This makes the queue data type an ideal candidate that can be shrunk or grown as elements are deleted or added to it, without fixing an upper limit on its size

Queues Continued...

Characteristics of a Queue (Cont'd):

- The empty array literal {} is used to denote an empty queue
- If known, an upper limit index 'N' of the queue can be set. This is known as a *bounded queue* and shall be limited to have indices not greater than 'N' (its size shall not exceed N+1)
 - For e.g.: `bit limitq[$:255];` // A bounded queue whose maximum size is 256 bits
- SystemVerilog automatically allocates space if the queue runs out of memory when adding additional elements. Hence, the `new[]` constructor used in allocating memory for arrays, is never used for queues
- A queue is similar to a one-dimensional unpacked array, except that it grows and shrinks automatically. Therefore, like arrays, queues can be manipulated using the indexing, concatenation, slicing and equality operators
- Elements of a queue are stored in contiguous locations in memory
 - Adding an element to the front of the queue or deleting an element from the back of the queue therefore takes a fixed amount of time
 - Adding or deleting an element from the middle of the queue requires shifting the data already present in the queue. The time to do this depends on the size of the queue

Applications of Queues:

- They are used to model Last In First Out (LIFO) and First In First Out (FIFO) behavior, typically buffers
- Queues are commonly used in various testbench components like scoreboard, monitor, driver – explained further in a later segment of this course

Queue Operations

Queues support the same operations that are performed on unpacked arrays. In addition to those, queues also support the following operations:

- It will resize itself to accommodate any value that is written to it, provided that it is within its maximum bound size
- In a queue slice expression, the slice bounds may be arbitrary integral expressions and are not required to be constant expressions
- Queues support additional built-in methods described in the next few slides

Consider a queue 'Q' whose range is bounded between 'a' and 'b'. The following rules govern queue operators:

- If $b > a$, then $Q[a : b]$ yields a queue with " $b - a + 1$ " elements
- If $a > b$, then $Q[a : b]$ yields the empty queue $\{\}$
- If $a = b$, then $Q[a : b]$ is the same as $Q[a : a]$ and it yields a queue with one item, the one at position 'a'.
- If 'n' lies outside Q's range ($n < 0$ or $n > \$$), then $Q[n:n]$ yields the empty queue $\{\}$
- $Q[a : b]$ where $a < 0$, is the same as $Q[0 : b]$
- $Q[a : b]$ where $b > \$$, is the same as $Q[a : \$]$
- An invalid index value will cause a queue read operation to return the value 'x' for a 4-state integral type or a value '0' for a 2-state integral type
- An invalid index value will cause a queue write operation to be ignored and a run-time warning to be issued

Queue Operations Continued...

Examples of queue operations:

```
int q[$] = {2, 4, 8}; // Initialized integer queue 'q' with 3 elements – 2, 4 and 8
int p[$];           // Un-initialized integer queue 'p'
int e, pos;

e = q[0]; // read the first (left-most) element of 'q'; 'e' gets the value 2
e = q[$]; // read the last (right-most) element of 'q'; 'e' gets the value 8
q[0] = e; // write the value of 'e' as the first element in 'q'
p = q; // copy contents of 'q' to 'p'
q = { q, 6 }; // insert '6' at the end of the queue 'q' (append the value 6)
q = { e, q }; // insert the value of 'e' at the beginning of the queue 'q' (prepend the value of 'e')
q = q[1:$]; // delete the first (left-most) element of 'q'
q = q[0:$-1]; // delete the last (right-most) element of 'q'
q = q[1:$-1]; // delete the first and last elements of 'q'
q = {}; // clear the queue contents, i.e., delete all elements of 'q'
q = { q[0:pos-1], e, q[pos,$] }; // insert 'e' in the queue 'q' at the position 'pos'
q = { q[0:pos], e, q[pos+1,$] }; // insert 'e' in the queue 'q' after the position 'pos'
```

Queue Methods

SystemVerilog provides several built-in methods to manipulate queues. These methods or functions are called after the queue object name with a period “.” between the object name and the function name. The function arguments (if any) are inputs.

- Function `int size()`: Returns the number of items in the queue. If the queue is empty, it returns 0.
 - `Q.size()` returns 4 for the integer queue, `Q[$] = {8, 2, 1, 20}`; containing 4 integer values - 8, 2, 1 and 20
- Function `void insert (int index, queue_type item)`: Inserts the given item at the specified index position
 - `Q.insert (i, e)` is equivalent to `Q = {Q[0:i-1], e, Q[i:$]}`
- Function `void delete (int index)`: Deletes the item from the specified index position
 - `Q.delete (i)` is equivalent to `Q = {Q[0:i-1], Q[i+1:$]}`
- Function `queue_type pop_front()`: Removes and returns the first element of the queue
 - `e = Q.pop_front ()` is equivalent to `e = Q[0]; Q = Q[1:$]`
- Function `queue_type pop_back()`: Removes and returns the last element of the queue
 - `e = Q.pop_back ()` is equivalent to `e = Q[$]; Q = Q[0:$-1]`
- Function `void push_front (queue_type item)`: Inserts the given element at the front of the queue
 - `Q.push_front (e)` is equivalent to `Q = {e, Q}`
- Function `void push_back (queue_type item)`: Inserts the given element at the end of the queue
 - `Q.push_back (e)` is equivalent to `Q = {Q, e}`

Use of queue methods enables one to perform queue operations in a much easier manner

Queue Methods Continued...

Examples of queue methods:

```
integer int_q[$] = {20, 30, 40, 50}; // Initialized integer queue 'int_q' with 4 elements – 20, 30, 40 and 50
integer qs, val;
```

```
qs = int_q.size(); // The queue 'int_q' has 4 elements and hence the size is 4. 'qs' gets the value 4.
```

```
int_q.push_front(10); // Inserts the value 10 at the beginning of the queue 'int_q'. Now 'int_q' has
                      // 5 elements – 10, 20, 30, 40 and 50 in that order.
```

```
int_q.push_back(60); // Inserts the value 60 at the end of the queue 'int_q'. Now 'int_q' has
                    // 6 elements – 10, 20, 30, 40, 50 and 60 in that order.
```

```
int_q.insert(3, 35); // New element 35 added at index position 3 in the queue 'int_q'. Now 'int_q' has
                    // 7 elements – 10, 20, 30, 35, 40, 50 and 60 in that order.
```

```
int_q.delete(3); // Deletes the element from index position 3 in the queue 'int_q'. Now 'int_q' has
                 // 6 elements – 10, 20, 30, 40, 50 and 60 in that order.
```

```
val = int_q.pop_front(); // Removes and returns the first element of the queue 'int_q'. 'val' now holds the
                        // value 10, while 'int_q' has 5 elements – 20, 30, 40, 50 and 60 in that order.
```

```
val = int_q.pop_back(); // Removes and returns the last element of the queue 'int_q'. 'val' now holds the
                       // value 60, while 'int_q' has 4 elements – 20, 30, 40 and 50 in that order.
```

Array Classification – Quick Reference

Array Type	Example	Application(s)
Packed Array	bit [31:0] word_arr; // Packed array of 32-bit vectors	Such arrays could be used to store packet structures on which bit-select and part-select operations could be performed
Unpacked Array	real float_point[0:1023]; // Unpacked array of real type	Such arrays could be used to model Random Access Memories (RAMs), Read Only Memories (ROMs) and register files where one element is accessed at a time
Dynamic Array	integer i_dynarr[]; // Dynamic array of integers	Say random control signals need to be generated by the testbench during simulation. If a fixed-size array is used, it would have to be sized large enough to hold the maximum number of signals possible. In reality, only a subset of control signals would be used, thus wasting the remaining memory. Hence a dynamic array that can be allocated and re-sized during simulation will avoid this unnecessary memory allocation
Associative Array	bit [3:0] array_a[int]; // Un-initialized associative array of a 4-bit vector, indexed by int	Such arrays could be used to model CAMs. Random read or write tests for verification of memories could use associative arrays for storing data only for addresses which have been written
Queue	integer Q[\$] = { 4, 2, 7 }; // An initialized integer queue called "Q" containing three integers	Such arrays could be used to model Last In First Out (LIFO) and First In First Out (FIFO) behavior, typically buffers

Array Locator Methods

These are methods that traverse the array in an unspecified order and search an array for elements or their indices based on a given search (filter) expression. The array locator methods operate on any unpacked array, including queues, but their return type is a queue. If no elements satisfy the given filter expression or the array is empty, then an empty queue is returned. These methods or functions are called after the array object name with a period “.” between the object name and the function name.

The **‘with’** clause is mandatory in the following methods:

- `find()` with (filter_expr): Returns all the elements satisfying the given expression
- `find_index()` with (filter_expr): Returns the indices of all the elements satisfying the given expression
- `find_first()` with (filter_expr): Returns the first element satisfying the given expression
- `find_first_index()` with (filter_expr): Returns the index of the first element satisfying the given expression
- `find_last()` with (filter_expr): Returns the last element satisfying the given expression
- `find_last_index()` with (filter_expr): Returns the index of the last element satisfying the given expression

For the following methods, the **‘with’** clause can be omitted if the relational operators (<, >, ==) are defined for the element type of the given array. If the **‘with’** clause is specified, the relational operators shall be defined for the type of the expression.

- `min()`: Returns the element with the minimum value or whose expression evaluates to a minimum
- `max()`: Returns the element with the maximum value or whose expression evaluates to a maximum
- `unique()`: Returns all elements with unique values or whose expression evaluates to a unique value
- `unique_index()`: Returns the indices of all elements with unique values or whose expression evaluates to a unique value

Array Locator Methods Continued...

Examples of Array Locator Methods:

```
string SA[10], qs[$];
int IA[*], qi[$];

// Find all items in IA that are greater than 30
qi = IA.find(x) with (x > 30);

// Find indices of all items in IA that are equal to 8
qi = IA.find_index with (item == 8);

// Find the first item in SA that is equal to "address"
qs = SA.find_first with (item == "address");

// Find the last item in SA that is equal to "data"
qs = SA.find_last(z) with (z == "data");

// Find the index of last item in SA that is greater than "Z"
qi = SA.find_last_index(p) with (p > "Z");

// Find the smallest item in IA
qi = IA.min;

// Find the string with largest numerical value in SA
qs = SA.max with (item.atoi);

// Find all unique string elements in SA
qs = SA.unique;

// Find all unique strings in upper-case in the array SA
qs = SA.unique(rv) with (rv.toupper);
```

Array Ordering Methods

These are methods that change the order of elements in any unpacked array, excluding associative arrays. An important point to note is that unlike array locator methods, these array ordering methods change the contents of the original array. These methods or functions are called after the array object name with a period "." between the object name and the function name.

- `reverse()`: Reverses the order of the elements in an array. Specifying a '**with**' clause shall be a compiler error.
- `sort()`: Sorts the array in ascending order. An optional condition can be specified using the expression in the '**with**' clause.
- `rsort()`: Sorts the array in descending order. An optional condition can be specified using the expression in the '**with**' clause.
- `shuffle()`: Randomizes the order of the elements in the array. Specifying a '**with**' clause shall be a compiler error.

Consider an array consisting of a packet of elements used as input stimuli in the verification of a design.

Next, the input stimuli needs to be given in a reverse order to test a particular corner case during verification. So the `reverse()` method is used.

For exhaustive testing, say the input stimuli need to be given to the test bench in an ascending order. The `sort()` method is used to achieve this.

Examples:

```
int da[] = '{10,2,7,3,4,4}';
da.reverse();
// output is {4,4,3,7,2,10}
da.sort();
// output is {2,3,4,4,7,10}
da.rsort();
// output is {10,7,4,4,3,2}
da.shuffle();
// output is {4,2,4,3,10,7}
```

Next the array needs to be sorted in descending order. Then the `rsort()` method is used.

Say the contents of the array needs to be randomized after an interrupt. Then the `shuffle()` method is used to randomize the array.

Array Reduction Methods

These are methods that may be applied to any unpacked array of integral values to reduce the array to a single value. The expression within the optional '**with**' clause is used to specify the values to use in the reduction. The method returns a single value of the same type as the array element type or, if specified, the type of the expression in the '**with**' clause.

- `sum()`: Returns the sum of all the array elements. If a '**with**' clause is specified, returns the sum of the values yielded by evaluating the expression for each array element
- `product()`: Returns the product of all the array elements. If a '**with**' clause is specified, returns the product of the values yielded by evaluating the expression for each array element
- `and()`: Returns the bit-wise AND (`&`) of all the array elements. If a '**with**' clause is specified, returns the bit-wise AND of the values yielded by evaluating the expression for each array element
- `or()`: Returns the bit-wise OR (`|`) of all the array elements. If a '**with**' clause is specified, returns the bit-wise OR of the values yielded by evaluating the expression for each array element
- `xor()`: Returns the bit-wise XOR (`^`) of all the array elements. If a '**with**' clause is specified, returns the bit-wise XOR of the values yielded by evaluating the expression for each array element

Examples:

```
byte da[] = {1,2,3,4};  
int ans;
```

```
ans = da.sum ;           // ans is 10 => 1 + 2 + 3 + 4  
ans = da.product ;       // ans is 24 => 1 * 2 * 3 * 4  
ans = da.xor with (item + 6); // ans is 12 => 7 ^ 8 ^ 9 ^ 10
```

Structures

Need for the Introduction of Structures in SV

The big limitation in Verilog is the lack of a data type that can group a collection of different data types together. SystemVerilog introduces the concept of structures to alleviate this limitation. The concepts are similar to what is available in the 'C' programming language.

Definition: A structure is a user-defined data type that represents a collection of various data types that can be referenced as a whole group, or as individual data types which make up the structure

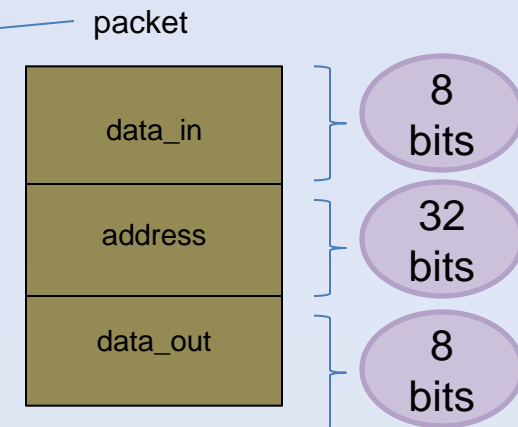
```
Syntax : struct packed(optional) {<data_type> <range(optional)> element1,element2,...;  
                                     <data_type> <range(optional)> element1,element2,...;  
                                     ...  
                                     } structure_name;
```

Examples:

```
struct {byte data_in;  
    logic [31:0] address;  
    bit [7:0] data_out;} packet; // structure data type, packet containing a collection of byte, logic and bit  
                                // data types respectively
```

```
packet ETH; // Defining a variable ETH of the packet structure type
ETH.data_in = 10; // Access of the data_in element using the ETH variable
```

```
struct packed {int a;  
               shortint b;  
               longint c, d;  
               bit [7:0] e, f;  
               byte g;} pack2state; // 2-state variable collection as a packed structure
```



Structures Continued...

Characteristics of a Structure:

- It is a convenient method of handling a group of related data items of different data types under a single name and stores all the values of the data types it contains
- Each of the constituent members of a structure is called a 'field'. In the packet structure example shown, data_in, address and data_out are fields of the packet structure
- By default, structures are unpacked
- A structure is just a collection of data and hence it can be synthesized
- A 'struct' data type has a subset of the functionality of the class data type (explained in the next Chapter) – it only groups data fields together and does not have routines to manipulate such grouped data. Classes are therefore preferred over structures when writing testbenches

Applications of Structures:

- An ethernet packet can be declared as a struct with its constituent elements like packet length, header, footer etc. as fields of the packet

Unions

Need for the Introduction of Unions in SV

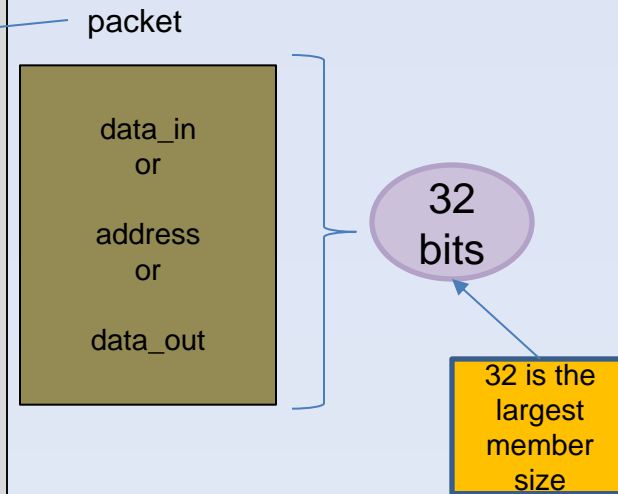
Unions are similar to structures, but with one major difference. The size of an object of the union data type is the size of its largest field. In contrast, the size of an object of the structure data type is the sum of the size of all its fields put together. Unions are therefore more memory efficient than structures and that is why they were introduced in SV.

Definition: A union is a data type that can be pictured as a chunk of memory that is used to store fields of different data types. Once a new value is assigned to a field, the existing data is overwritten with the new data.

```
Syntax : union packed(optional) {<data_type> <range(optional)> element1,element2,...;
                                     <data_type> <range(optional)> element1,element2,...;
                                     ...
                                     } union_name;
```

Examples:

```
union {byte data_in;  
    logic [31:0] address;  
    bit [7:0] data_out;} packet; // union data type, packet containing a collection of byte, logic and bit  
                                // data types respectively  
  
struct packed {int a;  
    shortint b;  
    longint c, d;} ipack2state; // Incomplete 2-state variable collection as a packed structure  
  
union packed {ipack2state incomplete; // Structure data type used here. Its 2-state data types are incomplete  
    bit [7:0] e, f; // Bit and byte data types complete the full list of 2-state data types  
    byte g;} cpack2state; // Complete 2-state variable collection as a packed union
```



Unions Continued...

Characteristics of a Union:

- The declaration of a union is similar to that of a structure and its various fields are referenced the same way as structures
- A union can store only one value – the size of the largest data field in it. Hence the memory usage is reduced
- By default, a union is unpacked
- It can be synthesized

Applications of Unions:

- Unions are useful when frequent reads and writes are done to a register in several different formats – the largest format determines the storage size.

Strings

Need for the Enhancement of Strings in SV

Verilog 2001 supports only string literals. SystemVerilog adds support for string as a built-in data type. Also added are a lots of operators and functions for string manipulation.

Definition: A string type is a variable-length ordered collection of characters. The length of a string is the number of characters in the collection.

Syntax: `string variable_name = initial_value(optional);`

Examples:

```
string soc_name;           // Un-initialized string variable 'soc_name'  
string module_name = "video_RAM"; // Initialized string variable  
string blp = "BLENDED learning PROgram";
```

Characteristics of Strings:

- The memory space for strings is dynamically allocated
- The indices of string variables shall be numbered from 0 to $N-1$ (where N is the length of the string) so that index 0 is the first (leftmost) character of the string and index $N-1$ is the last (rightmost) character of the string
- An un-initialized or empty string is represented with the special value `""`. An empty string has 0 length
- An individual character is of type 'byte'
- Unlike the 'C' programming language, there is no "null" character at the end of a SystemVerilog string

Applications of Strings:

- They are used to report the status of verification, to trace how far along in its execution a test case is, and most importantly to display error messages that helps in debug

String Operators

SystemVerilog provides a set of operators that can be used to manipulate combinations of string variables and string literals. They are listed in the table below:

Operator	Description
Str1 == Str2	Equality Operation: Checks whether the two string operands are equal. Result is 1 if both strings are composed of the same character sequence and 0 if not
Str1 != Str2	Inequality Operation: This is the exact opposite of the equality operation
Str1 < Str2 Str1 <= Str2 Str1 > Str2 Str1 >= Str2	Comparison Operation: Returns a 1 if the result of the given comparison is true
{Str1,Str2,...,Strn}	Concatenation Operation: Appends the 'n' individual string operands together. The operands can be of string type or can be string literals
{multiplier{Str}}	Replication Operation: Str is a string literal or a string expression and the multiplier is an integer 'N'. The result of this operation is a string containing 'N' concatenated copies of Str
Str[index]	Indexing Operation: Str is a string literal or a string expression and index is an integer from 0 to 'N-1', where 'N' is the number of characters in the string. The result returned is a byte, which is nothing but the character at the given index

String Methods

SystemVerilog provides special built-in methods to work with strings. These methods or functions are called after the string object name with a period "." between the object name and the function name.

- function int len(): Returns the length of the string, i.e., the number of characters in the string as an integer value
- function void putc(int i, byte c): Replaces the ith character in the string with character 'c'
- function byte getc(int i): Returns the ith character of the string
- function string toupper(): Returns a string with characters in the string converted to uppercase. The original string remains unchanged
- function string tolower(): Returns a string with characters in the string converted to lowercase. The original string remains unchanged
- function int compare(string s): str1.compare(str2) – compares strings str1 and str2. Return '0' if equal, a positive value if str1 comes after str2, and a negative value if str1 comes before str2
- function int icompare(string s): str1.icompare(str2) – case insensitive comparison of strings str1 and str2
- function string substr(int i, int j): str.substr(i,j) - returns a new string that is a sub-string consisting of characters in position i through j in string str
- function integer atoi(): String to integer conversion – returns the integer corresponding to the ASCII decimal representation of the string
- function void itoa(integer i): Integer to string conversion. This is the inverse of the function atoi()

String Methods Continued...

Examples of String Methods:

```
string a = "AEIOU";  
string b = "alonso";  
string c = "123";  
byte subs1;  
string subs2, lc, uc;
```

```
integer asiz, val;
```

```
asiz = a.len(); // asiz is assigned the value 5 as the string 'a' initialized to "AEIOU" has 5 characters
```

```
subs1 = b.getc(5); // subs1 gets 'o' which is the 5th character of string 'b' initialized to "alonso"
```

```
uc = b.toupper(); // uc gets "ALONSO" which is all uppercase of string 'b'
```

```
lc = a.tolower(); // lc gets "aeiou" which is all lowercase of string 'a'
```

```
subs2 = a.substr(1,3); // subs2 gets "EIO" which is the 1st to 3rd characters of string 'a'
```

Events

Need for the Enhancement of Events in SV

Events are used to synchronize processes or threads. In Verilog 2001 there is a possibility of a race condition where one process blocks on an event at the same time that another process triggers it. SystemVerilog introduces a function to check for the triggered status of an event. A process can wait on this function rather than block operation, thereby avoiding the race condition.

Definition: An event is a powerful means of synchronizing two or more concurrently active processes. This synchronization is done through an object and the event data type provides a handle it.

Syntax: `event event_name = initial_value(optional);`

Examples:

```
event done; // Declare a new event object called done
event redone = done; // Declare redone as an alias to done
event emptyevnt = null; // event variable emptyevnt has no synchronization object
```

Characteristics of Events:

- The synchronization object referenced by an event variable can be explicitly triggered and waited for.
- When one event variable is assigned another event variable, both event variables refer to the same synchronization object
- When assigned null, the association between the synchronization object and the event variable is broken
- An identifier declared as an event data type is called a named event
- The duration of an event persists throughout the time-step in which it was triggered
- If there is a need to send multiple synchronization notifications in a single time slot, do not use events. Instead use other built-in synchronization methods like semaphores and mailboxes (explained later in this course)

Applications of Events:

- They are used to synchronize the testbench and the DUV. Sending a burst of data from the testbench and waiting on an event response from the DUV are some common scenarios encountered.

Event Operators and Triggered Property

SystemVerilog enhances Verilog events by providing a set of operators that can be used to manipulate events. They are listed below:

- **Operators for triggering an event (-> and ->>)**
 - Named events are triggered using the -> operator
 - Nonblocking events are triggered using the ->> operator
 - Triggering an event unblocks all processes currently waiting on that event
- **Operator for waiting for an event (@)**
 - The basic mechanism to wait for an event to be triggered is via the event control operator, @
 - '@ hierarchical_event_identifier;' is the syntax and the @ operator blocks the calling process until the given event is triggered
 - For a trigger to unblock a process waiting on an event, the waiting process must execute the @ statement before the triggering process executes the trigger operator, ->. If the trigger executes first, then the waiting process remains blocked

Persistent triggered state: Triggered property

- SystemVerilog can distinguish between the event trigger itself, which is instantaneous, and the event's triggered state, which persists throughout the simulation time-step
- The triggered event property allows users to examine this state. The triggered property is invoked as follows:

`hierarchical_event_identifier.triggered`

The triggered event property evaluates to true if the given event has been triggered in the current time-step and false otherwise

Event Operations and Triggered Property Continued...

Persistent triggered state: Triggered property (Cont'd)

- The triggered event property is mostly used with a **wait** construct as shown below:

wait (hierarchical_event_identifier.triggered)

Using this mechanism, an event trigger shall unblock the waiting process whether the **wait** executes before or at the same simulation time as the trigger operation. This prevents any race condition likely to occur.

Examples of Event Operations and Triggered Property:

```
event done, sync; // declare two new events
event redone = done; // declare redone as an alias to done

task trigg(event ev); // task called "trigg" to trigger some event
    -> ev; // trigger ev
endtask
....
....

fork
    @ redone; // wait for done through redone
    #100 trigg(done); // trigger 'done' using the 'task' trigg
join

fork
    -> sync; // trigger sync
    wait (sync.triggered); // wait on the persistent trigger
join
```

User-defined Data Type

Need for the Introduction of User-defined Data Type in SV

Verilog 2001 does not allow users the freedom to extend net and variable types to create their own data types for some specific purpose. SystemVerilog introduces user-defined data types like those present in the 'C' programming language. Here users can create and manipulate their own complex data types for specific purposes like simulation, emulation etc.

Definition: User-defined types allow new type definitions to be created from existing data types. A 'typedef' declaration is used to define a user-defined type as similarly done in the 'C' programming language

Syntax : typedef <base_data_type> <user_defined_type_name>;

Examples:

```
typedef int unsigned unsint; // Define a new 32-bit unsigned 2-state integer type
typedef bit [31:0] uint; // Equivalent to the above definition – a new 32-bit
                        // unsigned 2-state integer type
```

```
unsint a, b; // Declaring variables with the new 'unsint' type
uint c, d; // a, b, c and d are 32-bit unsigned 2-state integers
```

```
typedef logic [15:0] bits16; // Defining a new 16-bit type called 'bits16'
bits16 word1, word2; // These are 2 16-bit words
```

```
typedef logic [3:0] nibble; // Defining a new 4-bit type called 'nibble'
nibble [15:0] vec; // vec is a 64-bit vector made from 16 nibble types
```

Characteristics of User-defined Data Types:

- They can be defined locally or in a package
- All of SystemVerilog's data types can be extended with user-defined type declarations
- The new type names are intended to provide additional descriptive information for the object declarations that use them
- Sometimes a user-defined type needs to be declared before the contents of the type have been defined. Support for this is provided by a **forward** typedef. More information about this can be obtained from page 77 of reference [3], the SystemVerilog IEEE 1800-2012 LRM

User-defined Data Type Continued...

Applications of User-defined Data Types:

- Most values in a testbench are positive integers and so having a signed integer could cause problems. So the unsigned, 2-state 32-bit integer type 'uint' or 'uint' (as shown in the example) is used in such testbenches.

Enumerated Data Type

Need for the Introduction of Enumerated Data Type in SV

An enumerated type in SystemVerilog allows one to create a named set of related but unique constants. Verilog 2001 has text macros for names and parameters to define constants, but relating both of them is extremely difficult.

The enumeration index values for INIT, START, STOP, FINAL and ABORT are 0,1, 2, 3 and 4 respectively – anonymous int types

If a value is not specified for a name, it gets the value of the previous name in the list incremented by 1. The value of W is 1. X is not specified and hence it is 1+1, i.e., 2. Similarly Y is 8 and Z is not specified. Hence its value is 8+1, i.e., 9

Definition: Enumerated data types defines a set of named values. It assigns a symbolic name to each legal value taken by the data type

Syntax : enum enum_base_type(optional) { <enum_name_declaration> = constant_expr(optional),
<enum_name_declaration> = constant_expr(optional),
:
<enum_name_declaration> = constant_expr(optional)
} <enum_type_identifier>;

Examples:

```
enum {red,amber,green} tlc; // anonymous int type used for a traffic light controller

enum {INIT,START,STOP,FINAL,ABORT} power_ctrl_fsm; // States of a finite state machine
// Explicit values defined below to do one-hot encoding
enum {idle = 1, start = 2, pause = 4, load = 8, done = 16} onehotstate;

// Encoding must match the length of the 3-bit enum data type defined below
enum bit [2:0] {ST0 = 3'b001, ST1 = 3'b010, ST2 = 3'b100} state_encoding;

// Error codes can be stored as an enum declaration
enum logic {underflow, overflow, divbyzero, io_err} error_codes;

// Opcodes can be stored as an enum declaration
enum logic [2:0] {ADD, SUB, MUL, MOV, LD, AND, XOR, NEG} opcode;

// Declaring a boolean type below
enum bit {FALSE=1'b0, TRUE=1'b1} boolean;

enum {W=1, X, Y=8, Z} missing_val; // Missing values for enum names
```

Enumerated Data Type Continued...

Characteristics of Enumerated Data Types:

- An enumerated type is stored as type 'int' unless specified as something else
- This type automatically gives a unique value to every name in the list
- Values default to the 'int' type starting at 0 and then incrementing by 1
- If a value is not specified for a name, it gets the value of the previous name in the list incremented by 1
- They are strongly typed, i.e., a variable of type 'enum' cannot be directly assigned a value that lies outside the enumeration set
- Elements of enumerated type variables can be used in numerical expressions. The value used in the expression is the numerical value associated with the enumerated value

Applications of Enumerated Data Types:

- These can be used to define processor opcodes, error codes and FSM states (as shown in the examples)
- Explicit values can be defined in the enumerated list to allow gray encoding, one-hot encoding etc. (as shown in the examples)

Enumerated Type Methods

SystemVerilog provides a set of specialized methods to iterate over and manipulate the values of enumerated types. These methods or functions are called after the enumerated object name with a period "." between the object name and the function name.

- function `enum first()`: Returns the value of the first member of the enumeration
- function `enum last()`: Returns the value of the last member of the enumeration
- function `enum next(int unsigned N = 1)`: Returns the N^{th} -next enumeration value (default is the next one) starting from the current value of the given variable
- function `enum prev(int unsigned N = 1)`: Returns the N^{th} -previous enumeration value (default is the previous one) starting from the current value of the given variable
- function `int num()`: Returns the number of elements in the given enumeration
- function `string name()`: Returns the string representation of the given enumeration value. If the given value is not a member of the enumeration, the `name()` method returns the empty string

Example of Enumerated Data Type Methods:

```
enum {init,start,stop} fsm_states;  
fsm_states ctrl;  
int total;  
  
total = ctrl.num(); // total gets the value 3  
  
ctrl = ctrl.first();  
$display("%s",ctrl.name); // Displays 'init'  
  
ctrl = ctrl.next();  
$display("%s",ctrl.name); // Displays 'start'  
  
ctrl = ctrl.last();  
$display("%s",ctrl.name); // Displays 'stop'  
  
ctrl = ctrl.prev();  
$display("%s",ctrl.name); // Displays 'start'
```

4. Object-Oriented Programming(OOP) Concepts

Introduction to OOP

- Programming languages such as C and hardware description languages such Verilog 2001 are structured languages i.e., data types and any algorithms or routines that manipulate them are segregated
- Object-Oriented Programming (OOP) helps to create complex data types and also to tie them together with routines that manipulate them. It helps combine or encapsulate manipulation routines and data as a single unit
- OOP helps in isolating the testbench from the design details. Therefore the code is easier to maintain, can be extended and is re-usable for system-level testing and also for use in testing of future projects
- OOP makes it very easy to partition a testbench into specific blocks that work together in accomplishing the verification task

OOP Terminology

Terminology	Example
<u>Class</u> : This forms the base of OOP. The class description describes all the properties, behavior and identity of objects present within it	“4-Wheeler” is a class which describes the different kinds of 4-wheeler vehicles present, properties like make, color etc. and how each of them behave
<u>Object</u> : This is an instance of the class	“Car”, “Truck” and “Mini-Bus” are instances of “4-Wheeler” vehicles
<u>Handle</u> : This is a pointer to the object that cannot be modified	“Registration Number” is unique to every 4-wheeler vehicle and in a way is its handle
<u>Properties</u> : These are variables that define the contents and characteristics of the instance of a class	“Make”, “Color”, “Mileage”, “Speed” and “Electric/Petrol/Diesel/Hybrid Type” are common characteristics that will be defined for all instances of 4-wheeler vehicles
<u>Methods</u> : These are tasks or functions that operate on the properties in an instance of the class	“Gears” to increase or decrease the engine revs and “Drive Mode” to control the fuel consumption are functions that operate on and control the “speed” and “mileage” properties respectively

OOP Terminology Continued...

Terminology	Example
<p><u>Inheritance:</u> OOP allows for extension of the characteristics and functionality of existing classes – this is inheritance</p>	<div><div><div><div>Class</div><div>4-Wheeler</div></div><div><div>Sub-Class</div><div>Passenger</div><div>Cargo</div></div><div><div>Sub-Class</div><div>Compact</div><div>Sedan</div><div>SUV</div><div>Pickup Truck</div><div>Van</div><div>Mini-Lorry</div></div></div><p>“Passenger” and “Cargo” are sub-classes of the “4-Wheeler” vehicle class – they extend the “4-Wheeler” class. This means that they automatically get the properties and methods of the “4-Wheeler” class. In addition to that, they can have their own extra properties and methods. “SUV” is a sub-class of the “Passenger” sub-class. This means that it can have its own properties and methods, in addition to getting the properties and methods of the “Passenger” sub-class, and also the properties and methods of the “4-Wheeler” class.</p></div> <div><p><u>Polymorphism:</u> OOP enables binding of data with functions at runtime – this is polymorphism</p><p>The “Drive Mode” method to control the fuel consumption had “Economy” and “Rally” settings built-in, which would limit the fuel injection to the slowest pace or the fastest pace respectively. While running the vehicle if one wanted to change the fuel injection to a medium pace, then they would just provide this data to the “Drive Mode” method and the function would accordingly adjust the rate of fuel injection.</p></div>

Classes, Properties & Methods

Definition: A **class** is a user-defined data type that contains a collection of data items and a set of subroutines that operate on that data. These data items in a class are referred to as class **properties**, and its subroutines are called **methods**.

Syntax:

```
class class_name;  
  <data type 1 / property 1 declaration>;  
  :  
  <data type p / property p declaration>;  
  <(method 1) task 1 / function 1 declaration>;  
  :  
  <(method q) task q / function q declaration>;  
endclass
```

Example:

```
class int_operations;  
  
  //Properties  
  int i;  
  int j;  
  
  //Methods  
  task addprint();  
    $display("i + j = %d", i+j);  
  endtask  
  task mulprint();  
    $display("i * j = %d", i*j);  
  endtask  
  task subprint();  
    $display("i - j = %d", i-j);  
  endtask  
  
endclass
```

2 properties i and j
are of type int

3 methods
called
addprint(),
mulprint()
and
subprint()

Classes, Properties & Methods Continued...

Features and Characteristics:

- The class properties and methods, define the contents and capabilities of that class
- A class is used to combine or encapsulate manipulation methods and data properties as a single unit
- Any data type can be used in the class property declaration
- The properties and functions/tasks in a class are also referred to as “members” of a class
- Classes provide a template or framework for use in testbenches during verification
- Classes in SystemVerilog are typically defined in either a program block or module or a package, but its definition is not limited to only these

Handles & Objects

Definition: An **object** is an instance of the class. A **handle** is nothing but a variable declaration of the class type – it is a pointer to the object that cannot be modified.

Syntax:

// For declaring a handle

```
class_name var_handle_1_name, ... var_handle_n_name;
```

// For creating an object

```
var_handle_1_name = new();    // Object 1
```

```
:
```

```
var_handle_n_name = new();    // Object n
```

Features and Characteristics:

- Every class has an inbuilt function **new()** called as the class constructor. This function allocates memory to store the variables of an object
- An object of a class is created by first declaring a variable of that class name, which is nothing but its handle and then calling the **new()** function and assigning it to the variable
- The compiler does the following when an object is created with the handle using the function **new()**:
 - Allocates space for all the properties in the class
 - Initializes all the properties to their default value ('0' for 2-state variables and 'x' for 4-state ones)
 - Returns the address where the object is stored - the handle gets this address
- Uninitialized object handles are set by default to the special value **null**
- An uninitialized object can be detected by comparing its handle with **null**

Handles & Objects Continued...

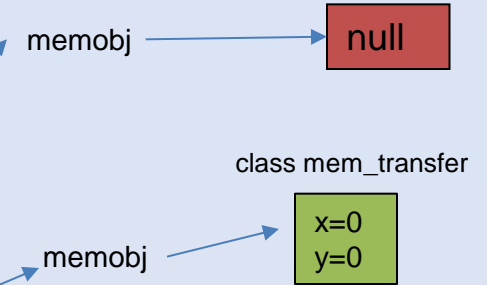
Features and Characteristics Continued...

- All handles are cleared by manually setting them to null
- SystemVerilog de-allocates memory for an object **only** if there are no handles pointing to it
- The properties or methods of any object can be accessed by using a period "." between the object name and the property or method name
- OOP allows for objects to be created and destroyed dynamically

Example 1:

```
class mem_transfer;  
    int x, y;  
endclass
```

```
// Declaring handle  
mem_transfer memobj;  
  
if(memobj == null)  
    // Allocate memory to  
    // create the object  
    memobj = new();
```



Example 2:

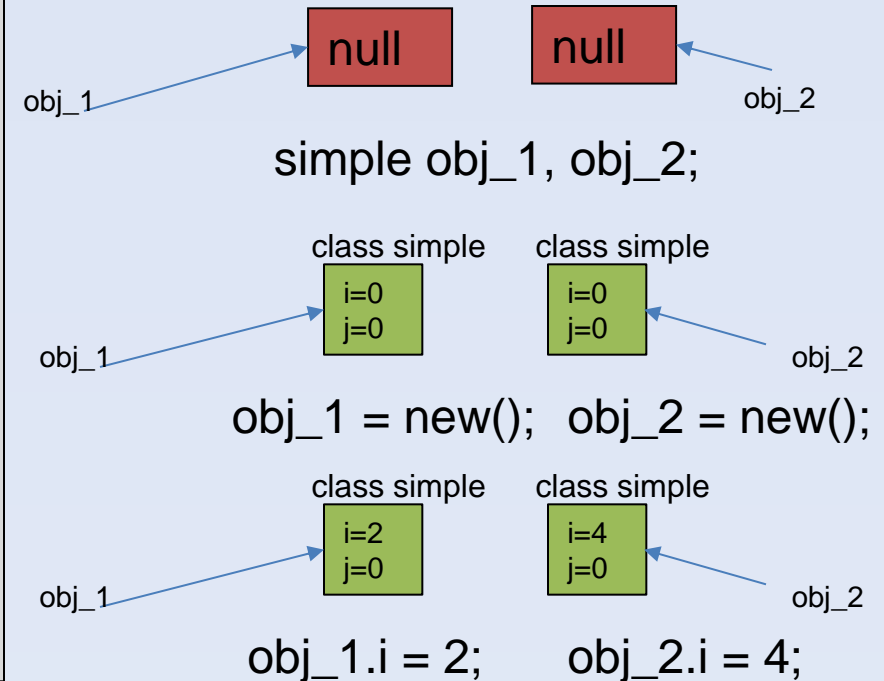
```
class simple;  
    //Properties  
    int i;  
    int j;  
  
    //Methods  
    task printf();  
        $display("%d, %d", i, j);  
    endtask  
endclass
```

Handles `obj_1` & `obj_2` for the class `simple`

Creating objects for the handles `obj_1` & `obj_2`

Accessing properties and methods using objects `obj_1` & `obj_2`

```
module simple_class;  
    initial  
    begin  
        //Declaring handles  
        simple obj_1;  
        simple obj_2;  
  
        //Creating objects  
        obj_1 = new();  
        obj_2 = new();  
  
        //Accessing properties &  
        //methods using objects  
        obj_1.i = 2;  
        obj_2.i = 4;  
        obj_1.printf();  
        obj_2.printf();  
    end  
endmodule
```



Parameterized Classes

Defining a generic class whose handles can be instantiated to have different array sizes or data types is very useful. This avoids writing similar code for each size or type and allows a single specification to be used for handles that are fundamentally different.

Features and Characteristics:

- The Verilog parameter mechanism is used to parameterize a class
- The parameterized class definition provides a template of the actual class. The actual class definition is created only when a handle of the class is declared
- A generic class is not a type declaration. Only a concrete specialization or initialization where the parameter value has been applied, represents a type
- Not all parameterized classes have a default specialization since it is legal for a class to not provide parameter defaults

Example 1:

```
class busvect #(int bus_size = 1); // Default specialization
    bit [bus_size-1:0] databus;    // for bus_size is 1
endclass
...
busvect #(10) vten;                // handle with databus vector of size 10
busvect #(.bus_size(2)) vtwo;      // handle with databus vector of size 2
busvect #(1024) vlK;              // handle with databus vector of size 1024
...
typedef busvect #(6) vsix; // class with databus vector of size 6
```

Example 2:

```
class queue #(type q_t = int); // Default specialization
    q_t queue_arr[];           // for q_t is int
    task push(q_t elem);
        :
    endtask
    task pop(q_t retval);
        :
    endtask
endclass
...
queue intq; // default: handle to a queue of integers
queue #(real) realq; // handle to a queue of real numbers
queue #(bit[7:0]) bitvecq; // handle to a queue of 8-bit vectors
```

Static Properties

Definition: By default, all instances (objects) of a class have their own copies of the data variables or properties. When it is required that all instances of a class have only one shared copy of some property, then that property has to be declared as static.

Features and Characteristics:

- These class properties are created by first specifying the keyword 'static'
- A static property is a variable of the class that is associated with the class definition, and not with an object (instance) of the class
- This variable is shared by all objects of the class - when its value is changed, the change is reflected in all instances of the class
- These properties can be used without creating an object

Example:

```
class Bluetooth;  
    // Counter for # of objects created  
    static int objcnt = 0;  
    // Unique instance ID  
    int uid;  
    function new();  
        // Set unique ID, incr the obj count  
        uid = objcnt++;  
    endfunction  
endclass  
  
// Declare handles of 2 Bluetooth objects here  
Bluetooth bt1, bt2;  
  
initial  
begin  
    bt1 = new; // Object1 created here  
    $display("First id=%d, Object count=%d"  
            ,bt1.uid,bt1.objcnt);  
  
    bt2 = new; // Object2 created here  
    $display("Second id=%d, Object count=%d,  
            Object count=%d"  
            ,bt2.uid,bt1.objcnt,bt2.objcnt);  
end
```

bt1.uid=0

bt1.objcnt=1

int variable uid is not static and so instances bt1 and bt2 have individual copies of uid

int variable objcnt is static and so only one copy of it is created, irrespective of however many Bluetooth instances are created

bt2.uid=1

bt1.objcnt=2

bt2.objcnt=2

Static Methods

Definition: These are nothing but static tasks or static functions within a class that can manipulate (read and write) the static properties. Multiple invocations of the same static method will reference the same local variables. Changing the local variable value in one invocation will make all others also see the change.

Features and Characteristics:

- A static method behaves like a regular task or function that can be called outside the class, even with no class instantiation
- They have an important restriction in that they can only access static class properties or static methods of their own class
- They do not allow access to non-static class properties or methods

Example 1:

```
class test_id;
    static int val = 0;                // Static Property
    int delay;                        // Non-static Property
    static function int inc_val(string objname); // Static Method (function)
        inc_val = ++val;              // Accessing the static class property val is OK
        $display("%s's val = %0d", objname, val); // Illegal to access non-static class property delay
    endfunction
endclass

test_id handle1, handle2; // 2 handles of the class test_id
int v1, v2;

initial
begin
    v1 = handle1.inc_val("handle1"); // calling the static function inc_val without actually
    v2 = handle2.inc_val("handle2"); // creating an object (class instance) here. Its legal !
end
```

Static Methods Continued...

Example 2:

```
class test_id;
    static int id = 0;                // Static Property
    static task gen_id(string objname); // Static Method (task)
        int val;
        val = ($urandom() % 100);
        id = val + 100;
        $display("%s's id = %0d, val = %0d",objname,id,val);
    endtask
endclass

test_id handle1, handle2;           // 2 handles of the class test_id

initial
begin
    handle1.gen_id("handle1");       // calling the static task gen_id here without actually
                                    // creating an object (class instance) here. Its legal !

    handle2 = new();
    handle2.gen_id("handle2");       // calling the static task gen_id here after creating the object
end
```

Since the 'gen_id' task is defined as static, all invocations of it will see the same value for 'val'. This means that changing 'val' in one invocation will make all the others also see the change.

If 'gen_id' was a normal task in the class, i.e., an automatic task, then each invocation of it would have its own local copy of 'val', totally independent of the others. This means that changing 'val' in one invocation won't affect any of the other invocations.

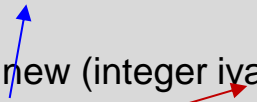
'this' Keyword

Features and Characteristics:

- The **this** keyword is used to unambiguously refer to class properties, parameters or methods of the current instance
- The **this** keyword denotes a predefined object handle
- The SystemVerilog IEEE 1800-2012 LRM states that the **this** keyword shall only be used within non-static class methods, or else an error is generated

Example:

```
class example_this;  
integer ivar;  
  
function new (integer ivar)  
    this.ivar = ivar;  
    // ivar = this.ivar; is also allowed  
endfunction  
  
endclass
```



'ivar' is both a property of the class and an argument to the function new.

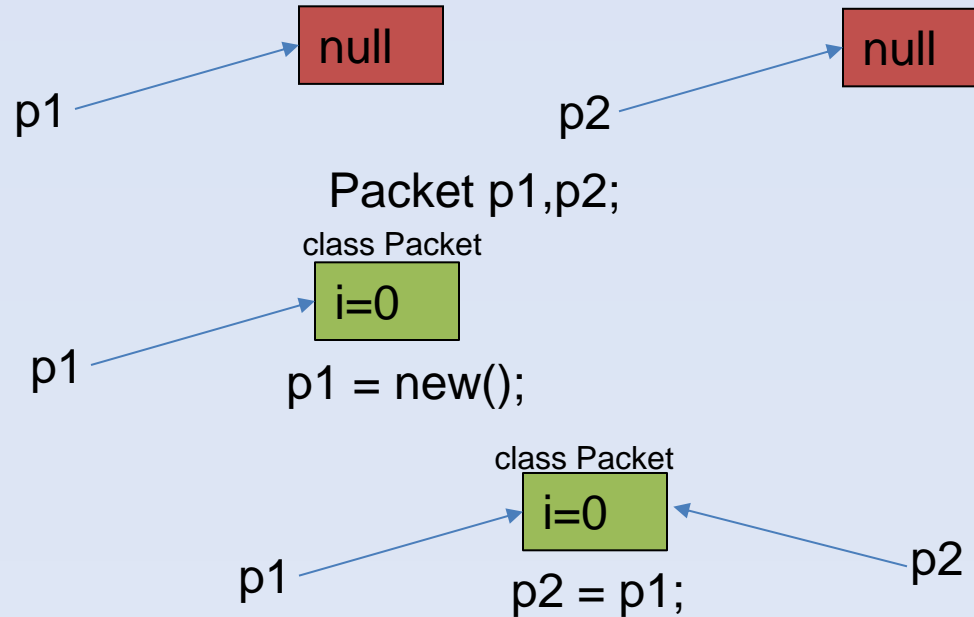
In the function new, an unqualified reference to ivar (without the 'this' keyword) is resolved by looking at the innermost scope of the variable. Here, it refers to the function argument declaration.

A qualified reference to ivar (with the 'this' keyword). refers to the instance class property for the current instance.

Object Assignment to Handle – Pointer Assignment

Example showing Pointer Assignment:

```
class Packet;  
  int i;  
endclass  
  
Packet p1,p2;  
  
initial  
begin  
  p1 = new();  
  p2 = p1;  
end
```



Recall handle declaration and the creation of an object of a class. In the example shown,

- The line of code (**Packet p1,p2;**) is the declaration of 2 handles p1 & p2 of the class Packet. Each of these can point to an object of the class. However, with just this declaration no memory is allocated and the initial value of p1 & p2 is null.
- The line of code (**p1 = new();**) is the creation of the object (instance). The compiler allocates space for all properties of Packet and initializes 'i' to 0.
- The line of code (**p2 = p1;**) is a simple pointer assignment operation. The handle p2 is assigned the object p1. Here the handle p2 just points to the object p1. Since **new()** was executed only once, only one object (instance) has been created. Although there is only one object, it can be referred to either with the handle p1 or p2.

Shallow Copy

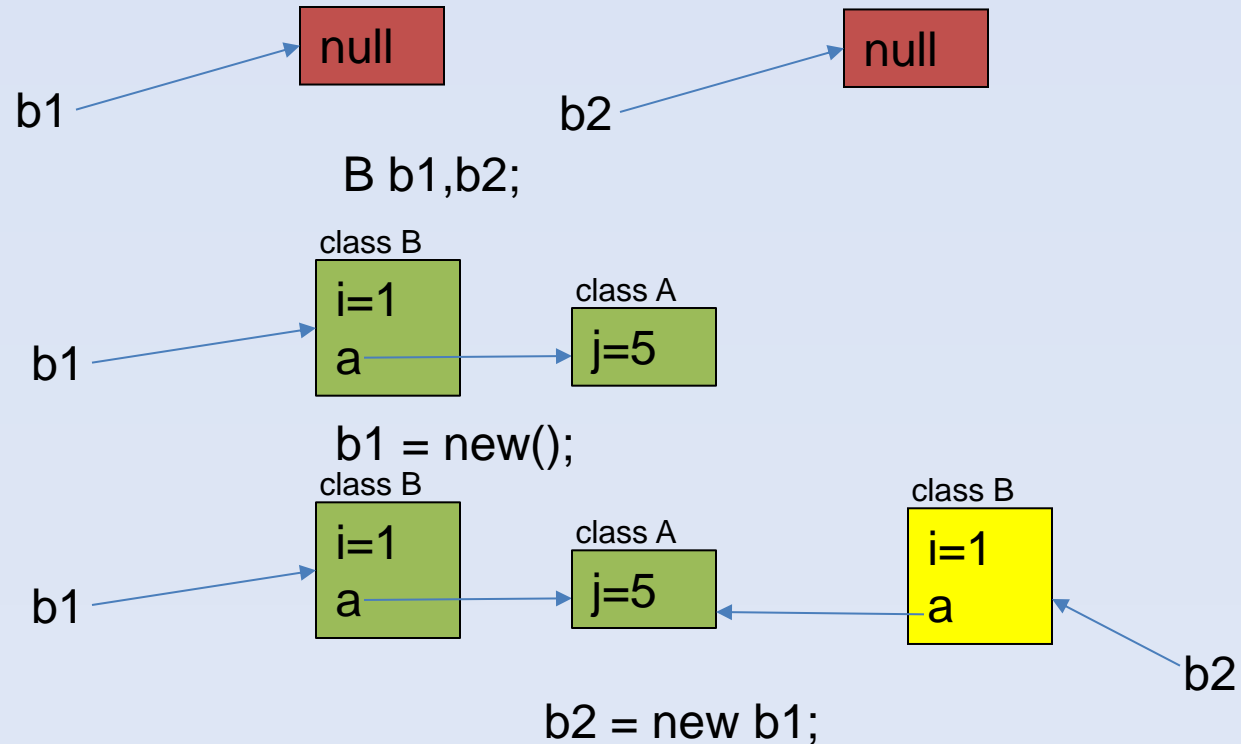
Example showing Shallow Copy:

```
class A;
  integer j = 5;
endclass

class B;
  integer i = 1;
  A a = new();
endclass

B b1,b2;

initial
begin
  b1=new();
  b2=new b1;
end
```



In a shallow copy operation only the first-level properties of the source are copied to the destination. If the first-level property of the source is an object, then the same object in the destination shares the existing memory allocated for it in the source. In the example shown,

- Class B has 2 properties - the integer 'i' and an object 'a' of class A
- The line of code (**B b1,b2;**) is the declaration of 2 handles b1 & b2 of the class B. Each of these can point to an object of the class. However, with just this declaration no memory is allocated and the initial value of b1 & b2 is null
- The line of code (**b1 = new();**) is the creation of the object (instance). The compiler allocates memory for all properties of the instance b1. It initializes 'i' to 1 and since object 'a' is also a property of class B, it allocates memory for 'a' and initializes the properties of 'a' as well. Hence 'j' is initialized to 5
- The line of code (**b2 = new b1;**) is the shallow copy operation. An object b2 is created first and then the first-level properties of b1 are copied to b2. Since 'a' is an object in b1, the object 'a' in b2 shares the existing memory allocated for it in b1

Deep Copy

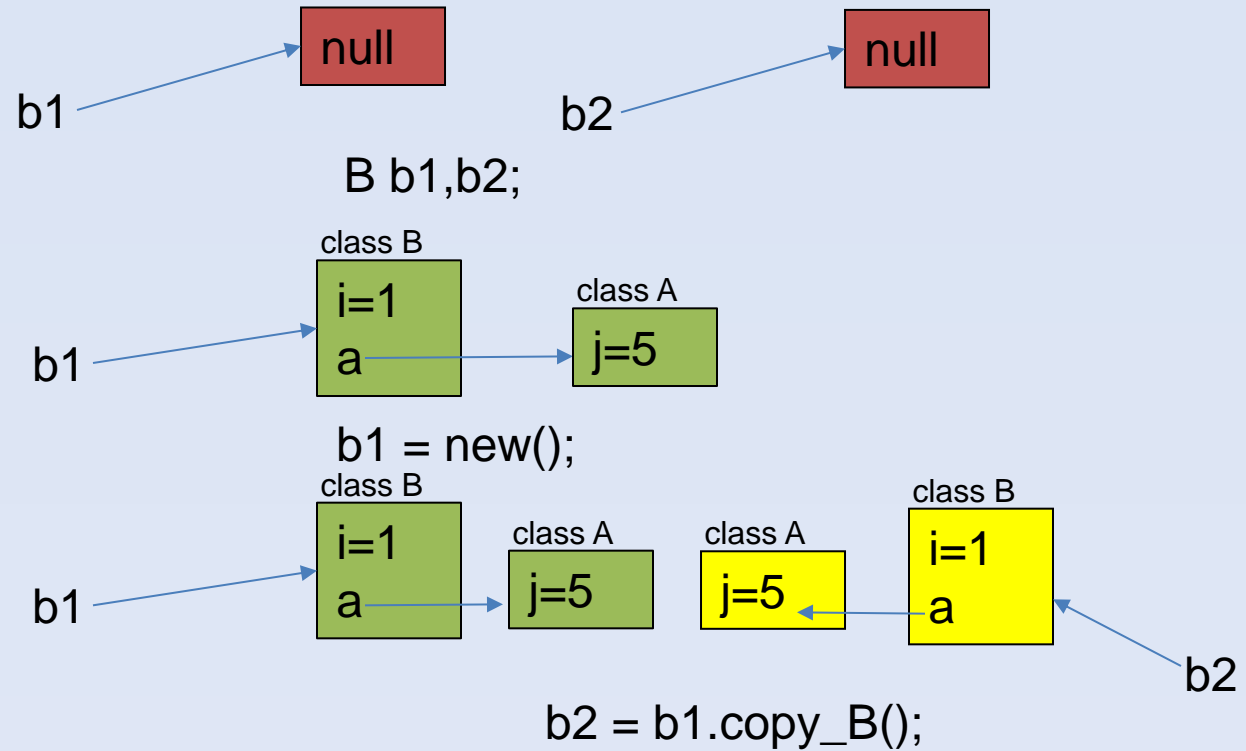
Example showing Deep Copy:

```
class A;
integer j = 5;
function A copy_A();
  copy_A=new();
  copy_A.j=this.j;
endfunction
endclass

class B;
integer i = 1;
A a = new();
function B copy_B();
  copy_B=new();
  copy_B.i=this.i;
  copy_B.a=a.copy_A();
endfunction
endclass

B b1,b2;

initial
begin
  b1=new();
  b2=b1.copy_B();
end
```



In a deep copy operation all the properties of the source object are copied to the destination object. This is a complete, non-restrictive, nested copy of all the properties - including objects and dynamically allocated memory pointed to by some properties. To do a deep copy, custom code is typically needed.

Deep Copy Continued...

In the example shown,

- `copy_A` and `copy_B` are customized functions whose output is stored in the function name itself and its return type is the class type where the function is declared. They help perform the deep copy operation.
- Class A has 1 property - the integer 'j' along with a function called 'copy_A' whose return type is class A. The 'copy_A' function allocates memory for the `copy_A` object and copies the property 'j' of class A to this object
- Class B has 2 properties - the integer 'i' and an object 'a' of class A, along with a function called 'copy_B'. The 'copy_B' function allocates memory for the `copy_B` object, copies the property 'i' of class B to this object and calls the function 'copy_A' and assigns its output 'copy_A' to the object 'a' in class B.
- The line of code (**B b1,b2;**) is the declaration of 2 handles b1 & b2 of the class B. Each of these can point to an object of the class. However, with just this declaration no memory is allocated and the initial value of b1 & b2 is null
- The line of code (**b1 = new();**) is the creation of the object (instance). The compiler allocates memory for all properties of the instance b1. It initializes 'i' to 1 and since object 'a' is also a property of class B, it allocates memory for 'a' and initializes the properties of 'a' as well. Hence 'j' is initialized to 5
- The line of code (**b2 = b1.copy_B();**) is the deep copy operation. The call to the function `copy_B` executes it, thereby assigning its output 'copy_B' to the object b2. This is a fully nested copy of all the properties of object b1 to b2

Inheritance

Definition: Inheritance is a way of automatically extending an existing class (called parent-class) by adding new properties and methods while deriving the original properties and methods of the parent-class

Features and Characteristics:

- If class X inherits from class Y, then
 - Y is called the parent-class of X
 - X is called the sub-class of Y
- The sub-class adds to the parent-class using the 'extends' keyword, i.e., it extends the parent-class
- The methods of the parent-class can be overridden by methods using the same name in the sub-class
- SystemVerilog restricts inheritance across only one level of parent-class and sub-class hierarchy
- Common code is typically written in the parent-class. New additions and changes to the common code is placed in the sub-class
- If the parent and sub classes have members with same name, the 'super' keyword is used from within the sub-class to refer to members of the parent-class. This is discussed with an example later on in this topic.
- Inheritance enables code re-use and allows for easy to maintain code. It is heavily used in testbenches

Packet is the parent-class containing the integer properties payload and checksum

NetworkPacket is an extension of Packet and hence is a sub-class. It contains new integer properties header, trailer and a function called linkpackets in addition to inheriting the members of the parent-class Packet as well, i.e., the integer properties payload and checksum



Example 1:

```
class Packet;                                // Parent-Class
    int payload;
    int checksum;
endclass

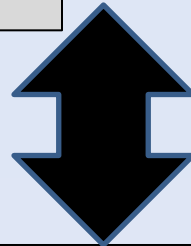
class NetworkPacket extends Packet; // Sub-Class
    // Other properties & methods of class NetworkPacket;
    int header, trailer;
    function linkpackets()
        :
        :
    endfunction
endclass
```

Inheritance Continued...

Example 2:

```
module exmpl_inherit;
  class A;
    int i=1;
    function int getA();
      getA = i;
    endfunction
  endclass
  class B extends A;
    int j=10;
    function int getB();
      getB = j;
    endfunction
  endclass
  . . .
```

```
. . .
  //Internal variables
  int x,y,z,zz;
  //Handle for class B
  B b;
  initial
  begin
    b = new();
    x = b.i;      //x = 1
    y = b.j;      //y = 10
    z = b.getA;   //z = 1
    zz = b.getB;  //zz = 10
  end
endmodule
```



Class A is the parent-class containing the integer property *i* and the function *getA*

Class B is an extension of class A and hence is a sub-class. It contains a new integer property *j* and the function *getB* in addition to inheriting the members of the parent-class A as well, i.e., the integer property *i* and the function *getA*

The assignment, **b = new()** creates an object for class B by first allocating memory for properties of class A, (since class B extends class A) and then allocating memory for properties of class B as well

Can use the “b” object to access properties and functions of both the classes A and B

Composition

Definition: The manual extension of a class is called composition

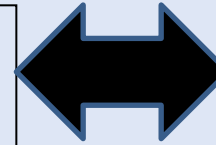
Features and Characteristics:

- No 'extends' keyword here. A handle of the parent-class is declared in the sub-class to accomplish the manual extension
- The methods of the parent-class cannot be overridden by methods using the same name in the sub-class

Packet is the parent-class containing the integer properties payload and checksum

NetworkPacket is a manual extension of Packet and hence is a sub-class. It contains a handle of Packet called intranet_packet, integer properties header, trailer and a function called linkpackets

The handle intranet_packet declared in the class NetworkPacket is used to access properties or methods of class Packet from NetworkPacket. All references to the Packet class are made through this handle.



Example:

```
class Packet;           // Parent-Class
  int payload;
  int checksum;
endclass

class NetworkPacket;    // Sub-Class
  // Handle to the parent-class declared below
  Packet intranet_packet;
  // Other properties & methods of class NetworkPacket
  int header, trailer;
  function linkpackets()
  :
  :
  endfunction
endclass
```


Comparison of Inheritance & Composition

Inheritance

- 1) All the properties and methods of a parent-class are also available to the sub-class if the sub-class has inherited everything from the parent-class
- 2) Possible to extend/override the methods of the parent-class from a sub-class
- 3) SystemVerilog limits inheritance to only one level of hierarchy – the parent-class and its sub-class
- 4) The implemented functionality inherited from a parent-class cannot be changed at run-time, as inheritance is defined at compile-time

Composition

All the properties and methods of a parent-class are also available to the sub-class if the sub-class has inherited everything from the parent-class

Not possible to extend/override the methods of the parent-class from a sub-class

Composition allows for extension across multiple levels of hierarchy

Functionality is acquired dynamically at run-time by objects referring to the methods. Its implementation can be replaced at run-time, with one object just replaced by another, as long as they have the same type

'super' Keyword

Features and Characteristics:

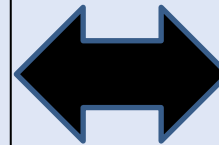
- If the parent and sub classes have members with same name, i.e., the members of the parent-class have been overridden by those in the sub-class, then the **super** keyword is used from within the sub-class to unambiguously refer to members of the parent-class
- When using the super keyword within the new function, **super.new** shall be the first statement executed in the constructor. This is because the parent-class must be initialized before the sub-class
- If the user code doesn't provide an initialization, the compiler will automatically insert a call to **super.new**

Packet is the parent-class containing the integer property dvalue and a function delay

EthernetPacket extends Packet and hence is a sub-class. It has an integer property dvalue and a function delay. These override the parent-class property as well as function as they use the same name

Note that inside the EthernetPacket delay function, the assignment statement has super.delay() and super.dvalue as part of the expression. These refer to the delay function and the dvalue integer property of the parent-class Packet respectively

The dvalue used in the assignment expression without the super keyword refers to the integer dvalue property of the EthernetPacket sub-class



Example:

```
class Packet;                                // Parent-Class
integer dvalue;
function integer delay();
    delay = dvalue * dvalue;
endfunction
endclass

class EthernetPacket extends Packet; // Sub-Class
integer dvalue;
function integer delay();
    delay = super.delay() + dvalue * super.dvalue;
endfunction
endclass
```

Diagram illustrating the use of the **super** keyword in Verilog code. The code defines two classes: **Packet** (Parent-Class) and **EthernetPacket** (Sub-Class). The **EthernetPacket** class extends **Packet**. Both classes have an integer property **dvalue** and a function **delay()**. The **Packet** class's **delay()** function calculates **delay = dvalue * dvalue**. The **EthernetPacket** class's **delay()** function calculates **delay = super.delay() + dvalue * super.dvalue**. Arrows indicate the resolution of **super.delay()** and **super.dvalue** to the parent class **Packet**.

Scope Resolution Operator ::

Definition: The class scope resolution operator :: is used to specify an identifier defined within the scope of a class. As classes and other scopes like packages or typedefs can have the same member names, the class scope resolution operator is used to uniquely identify a particular member.

Syntax: <scope_identifier> :: {scope_identifier ::} <member_identifier>;

Features and Characteristics:

- Scope identifiers on the left side of the :: scope resolution operator can be class names, package names or typedef names
- The scope resolution operator applies to all static elements of a class - static class properties, static methods, typedefs, enumerations, structures, unions, and nested class declarations
- It can be used to access members of a parent-class from within the sub-class
- Such class scope resolved member identifiers can be
 - Read in expressions
 - Written in assignments or subroutine calls
 - Triggered in event expressions

Scope Resolution Operator :: Continued...

Example:

```
class Base;
    typedef enum {bin,oct,dec,hex,frac} radix;
    static task print(int n, radix r);
        ...
    endtask
endclass

...
Base b = new;
int bin ← 321;
b.print(bin, Base::bin); // Base::bin and bin are different
Base::print(666, Base::hex);
```

Static Casting

Data type conversions in SystemVerilog are done using the cast operator. Using casting one can assign values that might not normally be valid because of differing data types between the source and destination variables.

Syntax: <casting_type> ' (<expression>)

Features and Characteristics:

- Static casting is done using the cast (') operation
- A static cast checks for type validity at compile time – no checks are done for out-of-bounds values

Examples:

```
int v1;  
shortint v2;
```

```
v1 = int'(2.0 * 3.6);
```

```
v2 = 2 * shortint'(8'hFE);
```

Float values are changed to int data type (output is of type int)

Hexadecimal to shortint (output is of type shortint)

Dynamic Casting

Syntax: \$cast (destination_variable, source_expression);

Features and Characteristics:

- Dynamic casting is done using the \$cast operation
- At run-time, a dynamic cast checks whether the assignment is legal and also checks for out-of-bounds values
- **\$cast** can be called either as a task or as a function. It attempts to assign the source expression to the destination variable.
 - function int \$cast(destination_variable, source_expression);
When called as a function, it returns 1 if the cast is legal. If the cast fails, the function does not make the assignment and returns 0 - no run-time error occurs, and the destination variable is left unchanged
 - task \$cast(destination_variable, source_expression);
When \$cast() is used as a task and the assignment is invalid, a run-time error occurs and the destination variable is left unchanged

Example 1:

```
typedef enum {RED , BLUE , GREEN, YELLOW, BLACK} rainbow;
rainbow colour1, colour2;
int colval;

initial
begin
    colour1 = YELLOW;           // Set to known valid value
    colval = colour1;           // Convert from enum to int (value is 3)
    colval++;                   // Increment (value is 4)
    if(!$cast(colour1,colval))  // Cast int back to enum. Called as a function, checks for return value 1
        $display("cast failed for colval=%0d", colval);
    $display("colour1 is %0d, %s", colour1, colour1.name);
    colval++;                   // Increment – This 5 value is out of bounds for the enum type
    colour2=rainbow'(colval);    // No type checking – use of static cast here
    $display("colour2 is %0d, %s", colour2, colour2.name);
end
```

\$cast converts colval which is of int type to colour1 which is of enum type. \$cast checks for out-of-bounds values, where as static cast does not check for out-of-bounds values

\$display output:

colour1 is 4, BLACK

colour2 is 5,

Dynamic Casting Continued...

Rules when casting classes:

- 1) It is *legal* to assign a sub-class object to an object of its parent-class
- 2) It is *illegal* to directly assign a parent-class object to an object of its sub-class
- 3) However, it is *legal* to assign a parent-class handle to a sub-class object using **\$cast** if the parent-class object handle previously pointed to an object of the given subclass

Example 2:

```
class B;
  task print();
    $display("CLASS B");
  endtask
endclass

class E_1 extends B;
  task print();
    $display("CLASS E_1");
  endtask
endclass
```

Illegal assignment
since \$cast is not
used. This is an error

```
module no_casting();
  initial
  begin
    B b;
    E_1 e1,e2;
    e1 = new();
    e2 = new();
    b = new();
    //legal assignment
    b=e1;
    //Illegal assignment since
    //$cast is not used
    e2 = b; //error
    b.print;
    e1.print;
  end
endmodule
```

Legal assignment
since \$cast is used.
This is not an error

```
module casting();
  initial
  begin
    B b;
    E_1 e1,e2;
    e1 = new();
    e2 = new();
    b = new();
    //legal assignment
    b=e1;
    //legal assignment since
    //$cast is used
    if($cast(e2,b))
    begin
      b.print;
      e1.print;
    end
  end
endmodule
```

Parent-class handle

Sub-class handle

Data Hiding - Local & Protected

Definition: In SystemVerilog, by default all the members of a class - class properties and methods are public, i.e., they are accessible in the scope in which the object is visible. However, in many cases, it is desirable to restrict access to class properties and methods from outside the class by hiding their names. This is called **data hiding or encapsulation**. To prevent accidental modifications to class properties that are internal to the class and also incorrect invocation of methods, they can be declared using the local or protected reserved words as necessary.

Features and Characteristics:

- Class members declared as **local** are available to methods inside the class. They are not visible within sub-classes and cannot be inherited
- A **protected** class property or method has all of the characteristics of a local member, except that it can be inherited and is visible only to sub-classes and not outside in the main module or block of code
- It is an error to define class members as being both **local** and **protected**

```
# ** Error:
C:/test/loc.sv(17): Access to local
member 'i' from outside a class context
is illegal.
# ** Error:
C:/test/loc.sv(18): Access to local
member 'i' from outside a class context
is illegal.
```

Example:

```
module hide_dat;
    class A;
        local int i = 5;
        protected int j = 10;
    endclass
    class B extends A;
        int k = j; // Protected property j is inherited
                    // from class A
    endclass
    A a;
    B b;
    int x,y,z;
    initial
    begin
        b = new();
        a = new();
        x = b.i;      //line 17 -- Illegal
        y = a.i;      //line 18 -- Illegal
        z = b.k;      //line 19 -- Legal
        display("x= %d, y= %d, z = %d", x,y,z);
    end
endmodule
```


Abstract Classes (Virtual Classes)

Definition: Abstract or virtual classes are a way of creating a template for a common base-class that is never instantiated, but extended to derive useful sub-classes.

Features and Characteristics:

- The common base-class is characterized as abstract by specifying it along with the 'virtual' keyword
- In an abstract class, the properties and methods are only specified and are not fully defined – this class is therefore incomplete. The virtual keyword is used to express the fact that derived classes could re-define the properties and must override the methods to complete the required functionality
- Virtual classes create a template from which real classes are derived
- Virtual classes can have virtual methods that are overridden in derived classes. Once a virtual method is defined, all derived classes that define the same method must use the same number and type of arguments, as well as the return value type (if any)

Example:

```
virtual class BasePacket;
    bit [1:0] frame_delimiter;
    virtual function integer send_pkt(bit[63:0] payload_data); // No implementation – just function declaration
endclass

class InternetPacket extends BasePacket;
    virtual function integer send_pkt(bit[63:0] payload_data);
    // Internal implementation details here in the body of the function
    ...
    endfunction
endclass
```

A common base-class of type BasePacket that defines the structure of packets and methods to work on it, is incomplete and is never going to be instantiated. However, from this base-class, a number of useful sub-classes can be derived, such as ethernet packets, internet packets, and satellite packets to name a few. Each of these packets might look very similar, all needing the same set of methods, but they could vary significantly in terms of their internal implementation details.

Polymorphism

Definition: A virtual method can be implemented using the same name and implemented differently for different sub-classes of the base-class. When the class object executes the virtual method, the correct implementation is chosen and executed at run-time. Simply put, the OOP term for multiple methods sharing a common name is called polymorphism.

Features and Characteristics:

- Polymorphism allows the use of a variable of the parent-class type to hold sub-class objects and to reference the methods of those sub-classes directly from the parent-class variable
- To achieve polymorphism the 'virtual' identifier must be used when defining the base class and methods within that class
- Similar to the concept found in the C++ programming language
- Different data types for e.g., integers, bits, and strings, cannot be stored in a single array. However, with polymorphism this can be done

Example:

```
virtual class BasePacket;
    bit [1:0] frame_delimiter;
    // No implementation – just function declaration below
    virtual function integer get_pkt_value(bit[63:0] payload_data);
    endfunction
endclass

class EthernetPacket extends BasePacket;
    virtual function integer get_pkt_value(bit[63:0] payload_data);
    // Internal implementation details here for this function
    ...
    get_pkt_value = x * y;
    endfunction
endclass

class InternetPacket extends BasePacket;
    virtual function integer get_pkt_value(bit[63:0] payload_data);
    // Internal implementation details here for this function
    ...
    get_pkt_value = a + b;
    endfunction
endclass

class SatellitePacket extends BasePacket;
    virtual function integer get_pkt_value(bit[63:0] payload_data);
    // Internal implementation details here in the body of the function
    ...
    get_pkt_value = ((i + 10) – (j + 20));
    endfunction
endclass
```

Polymorphism Continued ...

Example Continued...:

```
integer pkt_retval;

// Abstract BasePacket
BasePacket router_pkts[50]; // Router packet array of BasePackets

EthernetPacket ep = new(); // EthernetPacket object creation
InternetPacket ip = new(); // InternetPacket object creation
SatellitePacket sp = new(); // SatellitePacket object creation

// Put object instances created above into the router packet array
router_pkts[0] = ep;
router_pkts[1] = ip;
router_pkts[2] = sp;

// Calling the get_pkt_value() function
pkt_retval = router_pkts[2].get_pkt_value();

// At run-time, the compiler correctly binds the method from the appropriate
// class. Hence, it invokes the get_pkt_value() function associated with the
// SatellitePacket class.
```

Extern Out-of-Block Declaration

For better understanding and easier readability, it would be convenient to be able to move method definitions out of the body of the class declaration. SystemVerilog allows one to achieve this in 2 steps – (1) By breaking a method into the prototype (method name and arguments) defined inside the class, and (2) By declaring its body (the functional code) outside the class.

Step (1), which is defining a method prototype inside the class, involves adding the ‘extern’ keyword at the beginning to the method name along with its arguments. The extern qualifier indicates that the implementation of the method is to be found outside the declaration.

Step (2), involves declaring the method body outside the class. It must match the prototype declaration exactly and must be declared in the same scope as the class declaration. Here the entire method functionality is moved to after the class body. The class name and two colons (::) which is the scope resolution operator are added before the method name.

It is an error if more than one out-of-block declaration is provided for a particular extern method. An out-of-block method declaration should be able to access all declarations of the class in which the corresponding prototype is declared.

Example of Normal Method Declaration in a Class:

```
class Packet;
  bit [31:0] addrs;
  int value;

  function int get_next(int value);
    // get_next functionality here
    . . .
  endfunction

  function void printf();
    $display("Packet Address = %h", addrs);
  endfunction
endclass
```

extern keyword allows for out-of-body method declaration

Example of Out-of-Block Method Declaration in a Class:

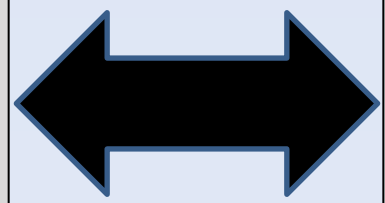
```
class Packet;
  bit [31:0] addrs;
  int value;

  extern function int get_next(int value);
  extern function void printf();
endclass

function int Packet::get_next(int value);
  // get_next functionality here
  . . .
endfunction

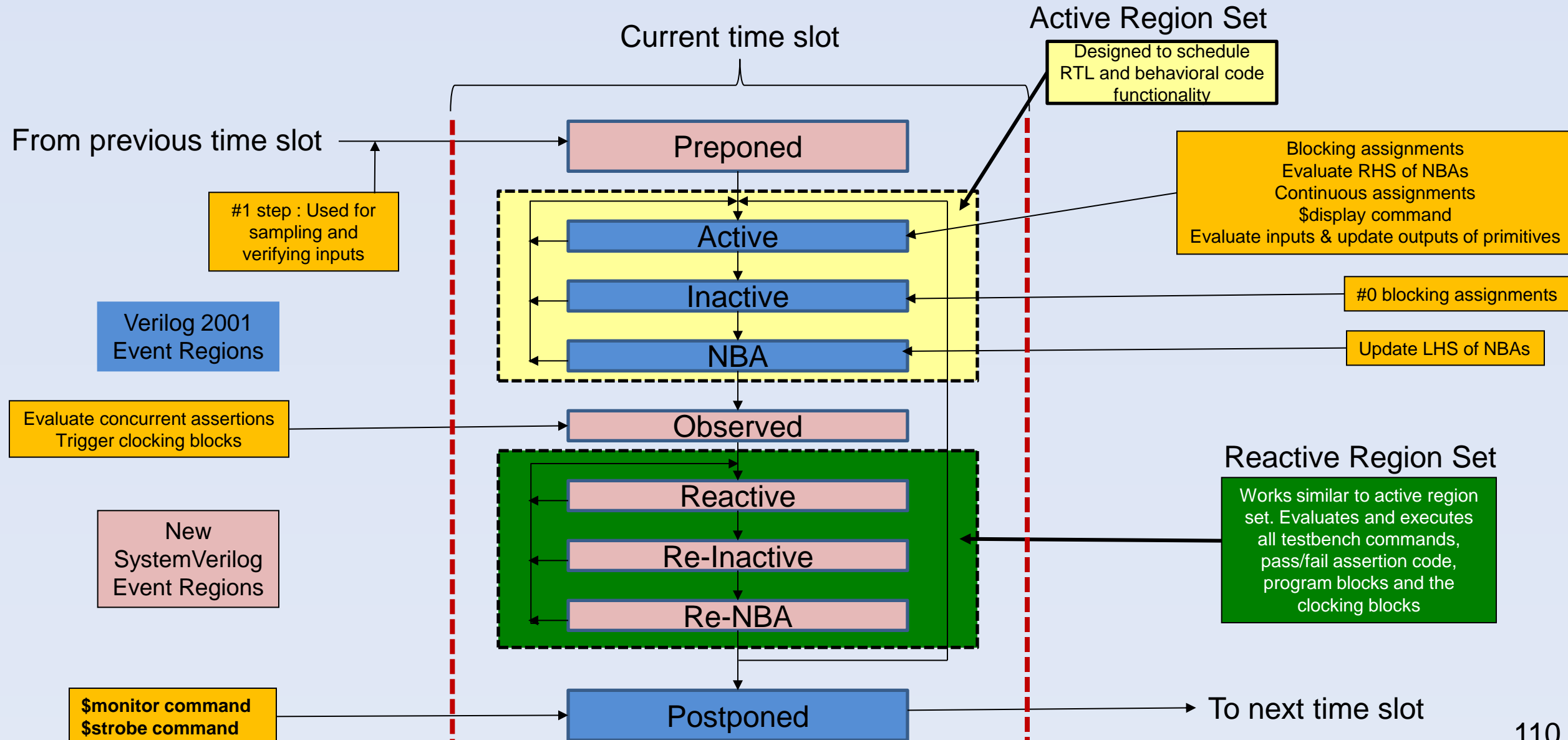
function void Packet::printf();
  $display("Packet Address = %h", addrs);
endfunction
```

“::” scope resolution operator links method declaration to class definition



5. SV Stratified Event Queue

SV Stratified Event Queue



Regions of the SV Stratified Event Queue

- **Preponed Region**
 - The function of this region is to sample values that are used by concurrent assertions.
 - The Preponed region is executed only once in each time slot
- **Active Region (Simulation of design code)**
 - The function of this region is to evaluate and process the current active region set event activity in any order. The event activity includes:
 - Execution of all module blocking assignments
 - Evaluation of the Right-Hand-Side (RHS) expression of all nonblocking assignments and schedule updates into the NBA region
 - Execution of all the continuous assignments
 - Evaluation of the inputs and updating outputs of Verilog primitives
 - Execution of the \$display and \$finish commands
- **Inactive Region**
 - This is the region that holds the events to be evaluated after all the active events are processed
 - #0 blocking assignments are scheduled here
 - #0 procedural assignments should be avoided as per coding guidelines and hence ideally the inactive region should not be used
- **NBA (NonBlocking Assignment) Region**
 - This is the region that holds the events to be evaluated after all the inactive events are processed
 - It executes updates to the Left-Hand-Side (LHS) variables that were scheduled in the active region for all currently executing nonblocking assignments
- **Observed Region (Assertions evaluated after design executes)**
 - The function of this region is to evaluate the concurrent property assertions using the values sampled in the Preponed region
 - Assertions that execute a pass or fail action block, actually schedule a process associated with the pass and fail code into the reactive regions and not in the observed region. This is because concurrent assertions are designed to behave strictly as monitors and are not allowed to modify the state of the design

Regions of the SV Stratified Event Queue Continued...

- **Reactive Region** (Execution of the testbench)
 - This is the set dual of the corresponding active region in the same time slot
 - The function of this region is to evaluate and execute all current program activity in any order. The activity includes:
 - Execution of all program blocking assignments
 - Execution of the pass/fail code from concurrent assertions
 - Evaluation of the Right-Hand-Side (RHS) expression of all program nonblocking assignments and schedule updates into the Re-NBA region
 - Execution of all the program continuous assignments
 - Execution of the \$exit and implicit \$exit commands
 - Execution of the verification process spawned by program blocks
 - The processes that execute when processing the reactive region typically drive back stimulus into the design
- **Re-Inactive Region**
 - This is the set dual of the corresponding inactive region in the same time slot
 - This is the region that holds the events to be evaluated after all the reactive events are processed
 - It iterates with the reactive region until all reactive/re-inactive events have completed
 - Events are scheduled in this region by executing a #0 in a program process
 - Ideally the re-inactive region should not be used as per coding guidelines
- **Re-NBA Region**
 - The Re-NBA region is the set dual of the corresponding NBA region in the same time slot
 - This is the region that holds the events to be evaluated after all the Re-Inactive events are processed
 - The function of this region is to execute the updates to the Left-Hand-Side (LHS) variables that were scheduled in the reactive region for all currently executing nonblocking assignments that were evaluated in the reactive region
 - It iterates with the reactive and re-inactive regions until all reactive/re-inactive events have completed
- **Postponed Region** (Sampling signals after all design activity)
 - The 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
 - No new value changes are allowed to happen in the current time slot once the postponed region is reached

Design, Verification and Assertion Activity Scheduled in Regions

- Regions that are designed to implement RTL functionality
 - Active region set (Active, Inactive and NBA regions - but avoid Inactive region events)
- Regions that are designed to implement verification execution
 - Preponed region, Reactive region set (Reactive, Re-Inactive, Re-NBA) and Postponed region
- Regions that are designed to implement concurrent assertion checking
 - Preponed, Observed, and Reactive regions
- Region that should be avoided
 - Inactive region

6. SV Tasks & Functions

Tasks and Functions

Tasks and functions are commonly referred to as sub-routines. They provide the ability to execute some common procedure from different places in the code. SystemVerilog inherits these constructs from Verilog 2001 and adds new features to them and extends their capabilities. The differences between tasks and functions are tabulated here.

Tasks	Functions
A task can call other tasks and functions	A function cannot call other tasks, but can call other functions
A task can contain an event, delay or timing control statements. Hence, a task can consume time	Within a function, no event, delay or timing control statements are permitted. Hence, a function cannot consume time and the statements in the body of a function will execute in one simulation time unit
A task does not return a value	A function has to return a single value
A task invocation can have zero or more arguments passed	In the function invocation at least one argument needs to be passed
Tasks are non-synthesizable, if they contain timing control statements	Functions are synthesizable

Capabilities of Tasks and Functions

In Verilog 2001:

- Automatic tasks and functions are supported
- Support for recursion (nested calls) in tasks and functions is present
- Local variables are already allocated when the function starts execution
- Multiple statements in a task or function need to be grouped within a begin ... end block
- Arguments are passed by value

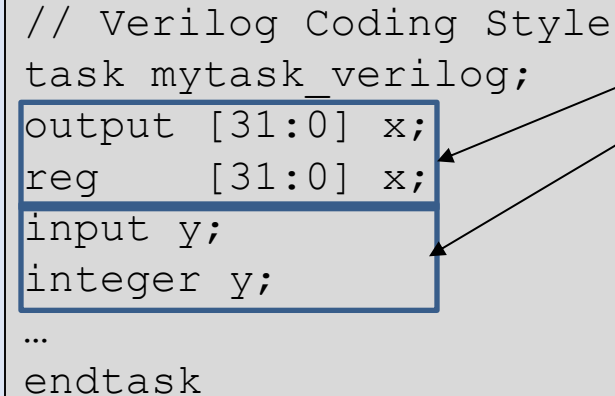
Have been extended in SystemVerilog:

- Multiple statements in a task or function do not require the begin ... end block
- Local variables are dynamically allocated at run time
- By default, the tasks and functions that are defined within a class are automatic, and those defined within a module, interface, program, or package are static
- A return statement has been added to a task
- Return from a task or function before reaching the end of the task or function is allowed
- A function declaration shall implicitly declare a variable, internal to the function, with the same name as the function. This variable has the same type as the function return value
- Functions can be declared as type void – these do not have a return value
- Functions can have the same formal arguments as tasks. The default argument direction is input. In addition to this, argument directions of output, inout and ref are allowed. Once a direction is given, subsequent formals default to the same direction
- C-programming style argument declaration syntax introduced makes the code more compact with lesser repetition while coding
- Each formal argument has a data type that can be explicitly declared or inherited from the previous argument. If the data type is not explicitly declared, then the default data type is logic
- Arguments can be passed by name, position and reference as well
- Arguments can be assigned some default value

C-programming Style Argument Declaration

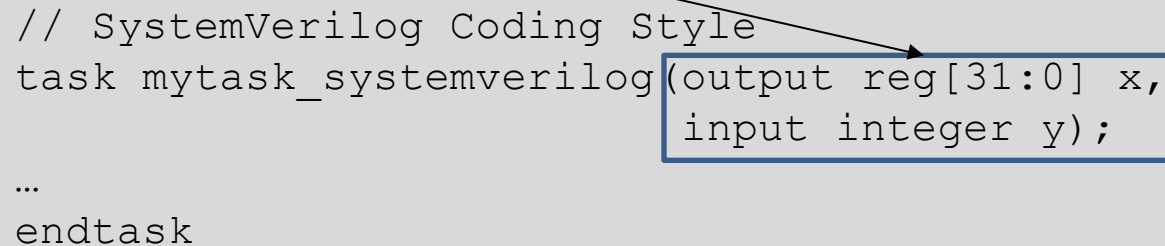
Verilog tasks require one to declare some arguments twice: once for the direction, and once for the type

```
// Verilog Coding Style
task mytask_verilog;
output [31:0] x;
reg      [31:0] x;
input y;
integer y;
...
endtask
```

A diagram showing a Verilog task declaration. The code is enclosed in a light gray box. A blue rectangle highlights the argument declarations: 'output [31:0] x;', 'reg [31:0] x;', 'input y;', and 'integer y;'. Two arrows originate from the text 'once for the direction, and once for the type' and point to the 'output' and 'input' keywords respectively.

With SystemVerilog, one can use C-style argument declarations, making the code compact with lesser repetition

```
// SystemVerilog Coding Style
task mytask_systemverilog(output reg[31:0] x,
input integer y);
...
endtask
```

A diagram showing a SystemVerilog task declaration. The code is enclosed in a light gray box. A blue rectangle highlights the combined argument declarations: '(output reg[31:0] x, input integer y);'. An arrow originates from the text 'one can use C-style argument declarations' and points to the opening parenthesis of the argument list.

SystemVerilog is backward compatible, i.e., Verilog style declarations can still be used in SystemVerilog code and these will compile fine

Formal Argument Direction and Type

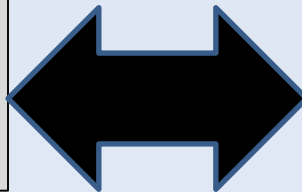
Every formal argument has one of the following directions. The default is input.

- **input**
- **output**
- **inout**
- **ref** (explained later)

Each formal argument has a data type that can be explicitly declared or inherited from the previous argument. If the data type is not explicitly declared, then the default data type is logic.

The default direction and type “input logic” are sticky, so subsequent formals do not have to repeat these until a change is seen.

```
// Verilog Coding Style
task mytask_v();
input a,b;
wire a,b;
output [15:0] u,v;
bit u,v;
...
endtask
```



The direction and type default to “input logic” are sticky and hence it is not repeated for similar arguments

```
// SystemVerilog Coding Style
task mytask_sv(a,b, output bit [15:0] u,v);
...
endtask
```

Function Return Value and Void Functions

A function's return value can be specified in two ways, either by using a return statement or by assigning a value to the internal variable with the same name as the function

```
// Way 1 --- Return using the same name as the function
function [8:0] addfunc (input [7:0] x,y);
    addfunc = x + y; // Return value assigned to function name
endfunction

// Way 2 --- Return using the 'return' statement
function [8:0] addfunc (input [7:0] x,y);
    return(x + y); //Return value is specified using return statement
endfunction
```

Functions can be declared as type void – these do not have a return value

```
function void printf (int a);
    $display("The value passed to the printf function is %0d", a);
endfunction
```

A non-void function (say it is called 'nv_fn_name') can be used as a statement and the return value discarded without any warning from the compiler, by casting the function call to the void type as shown below:

`void'(nv_fn_name());`

Early Return from a Sub-Routine

Verilog 2001 has a primitive way of ending a sub-routine. After executing the `endtask` or `endfunction` keyword, it will return control to the calling code

SystemVerilog adds the `'return'` statement to make it easier to control the flow in sub-routines. The early return of control from a task or function, at any point before reaching the `endtask` or `endfunction` keyword, is now allowed

Example 1:

```
task div5k(input int ipvar);
    $display("Entered task div5k with ipvar = %0d", ipvar);
    if (ipvar <= 0)
        begin
            $display("0 and negative numbers cannot be used for division");
            return; // Early return here before remaining code executes
        end
    $display("5000 divided by %0d = %f", ipvar, (5000/ipvar));
endtask
```

Note that multiple statements in a task do not require to be grouped within a `begin ... end` block

Example 2:

```
function [31:0] retvalfunc(input [9:0] x,y);
    if ((x < 40) && (y < 10))
        return x * y; // No brackets around the expression to be returned
    else if ((x >= 40) && (y < 10))
        return (x + y); // Could use brackets around the expression as well
    return y;
endfunction
```


Sub-Routine Call & Passing Arguments by Reference

Using the argument type **ref** to pass the array 'mem' to the 'print_parity' function via reference

SystemVerilog allows memories to be passed as arguments – Verilog allowed only scalars to be passed as arguments

The example is code to print the parity of each element of the mem array that is passed by reference to the print_parity function

Only addition of the **ref** keyword is needed to pass arguments by reference. In the example shown, the compiler knows that 'mem' is now addressed via a reference, and users do not need to make these references explicit at the point of the call

Combining **ref** with any other directional qualifier is illegal. E.g.:
~~task something(ref input int plk);~~

Example:

```
function automatic void print_parity(ref reg[7:0]mem[]);  
    bit parity = 0;  
    for(int i = 0; i< mem.size(); i++)  
    begin  
        parity = parity ^ mem[i];  
        $display("mem[%0d] = %0d. Its parity = %0b",i, mem[i], parity);  
        parity = 0;  
    end  
endfunction  
  
reg [7:0]mem[];  
  
initial  
begin  
    mem = new[6]; // Setting array size to 6  
    $display("Size of the mem array is = %0d",mem.size());  
    $display("Initializing the array with default values");  
    for (int i = 0; i<mem.size(); i ++)  
    begin  
        mem[i] = i;  
    end  
    print_parity(mem); // function call  
end
```

Default Values for Sub-Routine Arguments

SystemVerilog allows a sub-routine declaration to specify a default value for each of its arguments

When the subroutine is called, arguments with defaults can be omitted from the call, and the compiler will insert those corresponding default values

Empty arguments can be used as placeholders for default arguments. However, if an empty argument is used for an argument that does not have any default value specified, the compiler issues an error message

Default Values for Sub-Routine Arguments Continued...

Consider the example shown on the previous slide (Sub-routine class and passing arguments by reference page). For the `print_parity` function which calculates the parity of each array element, say one wants to print the parity of just a particular range of bits of the array element. Going back and rewriting every function call to add extra arguments is not feasible

The code shown on this page shows how to add minimum bit and maximum bit arguments to the original `print_parity` function in order to print the parity for only a range of bits of the array element

The `print_parity` function can be called in a variety of ways as shown here

Example:

```
function automatic void print_parity(ref reg[7:0]mem[],
                                     input int min = 0,
                                     input int max = -1);

bit parity = 0;
if ((max == -1) || (max > mem.size()))
    max = mem.size()-1;
for (int i=min; i<=max; i++)
begin
    parity = parity ^ mem[i];
    $display("mem[%0d] = %0d. Its parity = %0b",i, mem[i], parity);
    parity = 0;
end
$display("\n");
endfunction

// Assume 'mem = new[10];' i.e., set the mem array size to 10
// Various print_parity function calls with different arguments are shown below

print_parity(mem);           // print_parity mem[0:mem.size-1] | Default
print_parity(mem,3,5);       // print_parity mem[3:5] | Start at 3 and end at 5
print_parity(mem,1);         // print_parity mem[1:mem.size-1] | Start at 1
print_parity(mem, ,3);       // print_parity mem[0:3] | End at 3
print_parity(mem, 5, 15);    // print_parity mem[5:mem.size-1] | Start at 5 and
                             //                               | end at mem.size-1
print_parity();              // compile error: mem has no default initialization
```

Sub-Routine Call & Passing Arguments by Name

The Syntax for named argument passing is the same as Verilog's syntax for named port connections

SystemVerilog allows arguments to be bound both by name as well as by position, called mixed style argument passing. If both positional and named arguments are specified in a single sub-routine call, then all the positional arguments have to come before the named arguments

Example:

```
task printval(input int v1 = 10, v2 = 200, v3 = 3000, v4 = 40000);
    $display("v1 = %0d, v2 = %0d, v3 = %0d, v4 = %0d", v1, v2, v3, v4);
endtask

initial
begin
    printval() ;           // Passing arguments - default values
    printval(44,52,61,7);  // Passing arguments by value based on their position
    printval(.v2(25)) ;    // Passing arguments by name only to v2. Others take default values
    printval(,8,.v3(69));  // Mixed style argument passing - positional to v2 and named to v3
end
```

7. Verification Specific SV Constructs

Loop Statements - Foreach loop

The foreach loop construct allows for iteration over the elements of an array. The array name and an index variable is specified in square brackets, and SystemVerilog automatically steps through all the elements of the array. The index variable is automatically declared by the compiler and is local to the loop.

Example 1:

```
module foreach_loop();
  byte a[10] = {0,11,22,33,44,55,66,77,88,99};
  initial
  begin
    foreach (a[i])
    begin
      $display ("Value of a[%0d] is = %0d",i,a[i]);
    end
    #1 $finish;
  end
endmodule
```

The type of variable 'i' is int which is declared automatically and its scope & life is restricted only inside the foreach loop

The mapping of the loop variables to array indices is determined by the dimensions of the array. The number of loop variables should not be greater than the number of dimensions of the array variable. Loop variables may be omitted to indicate no iteration over that dimension of the array.

Example 2:

```
//      1 2 3      3 4      1 2 → Array Dimension Numbers
int IEEE [3][4][5]; bit [4:0][7:1] PLK [3:1][6];

foreach(IEEE [ i, j, k ] ) ...
foreach(PLK [w,x , , z ] ) ...
```

The first foreach loop for IEEE causes 'i' to iterate from 0 to 2, 'j' from 0 to 3, and 'k' from 0 to 4. The second foreach loop for PLK causes 'w' to iterate from 3 to 1, 'x' from 0 to 5, and 'z' from 7 to 1 (iteration over the third dimension is skipped).

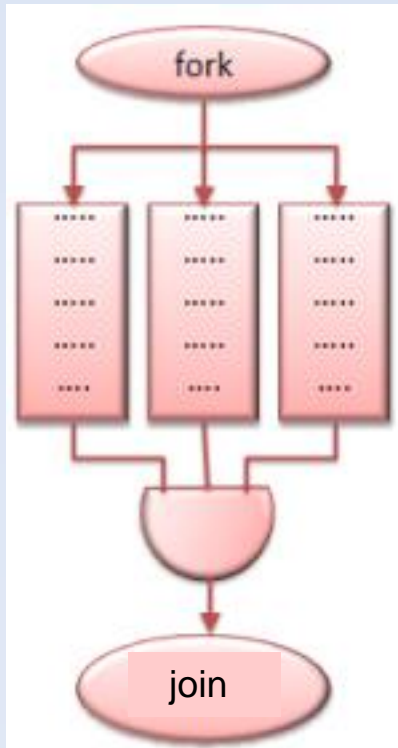
Thread Control - Parallel block constructs: fork ... join (Verilog)

- Verilog groups statements for execution

Sequentially: Within a begin ... end block

In Parallel: Within a fork ... join block

- In the fork ... join block of code, the parent process blocks further execution until all the processes spawned by this fork complete. Every statement (thread) inside the fork ... join block has to finish before the rest of the block can continue execution. Hence, Verilog testbenches rarely use this coding feature.



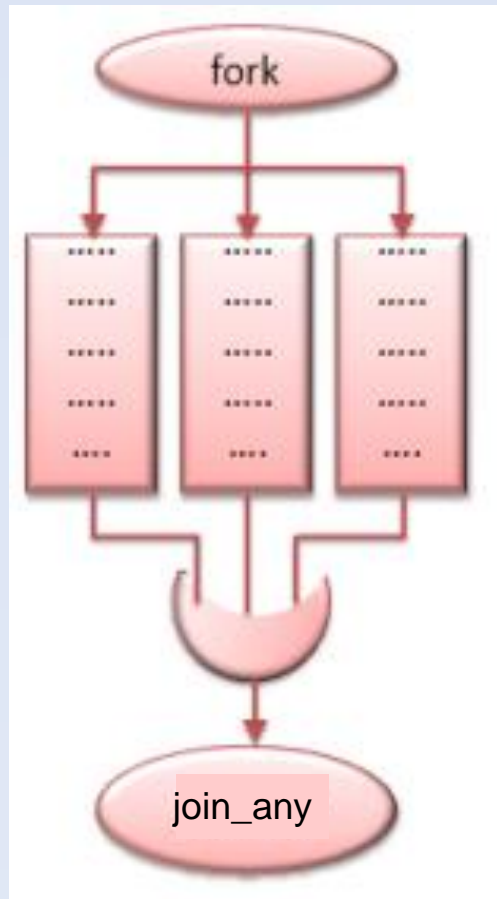
Example:

```
initial
begin
  clk = 0;
  #5
  fork
    #5 a = 0;
    #10 b = 0;
  join
  clk = 1; //clk becomes one at t=15
end
```

clk becomes one at
t=15

Thread Control - Parallel block constructs: fork ... join_any (SystemVerilog)

- SystemVerilog has introduced a new construct to create parallel processes – the fork ... join_any block
- In the fork ... join_any block of code, the parent process blocks further execution until any of the processes spawned by this fork completes



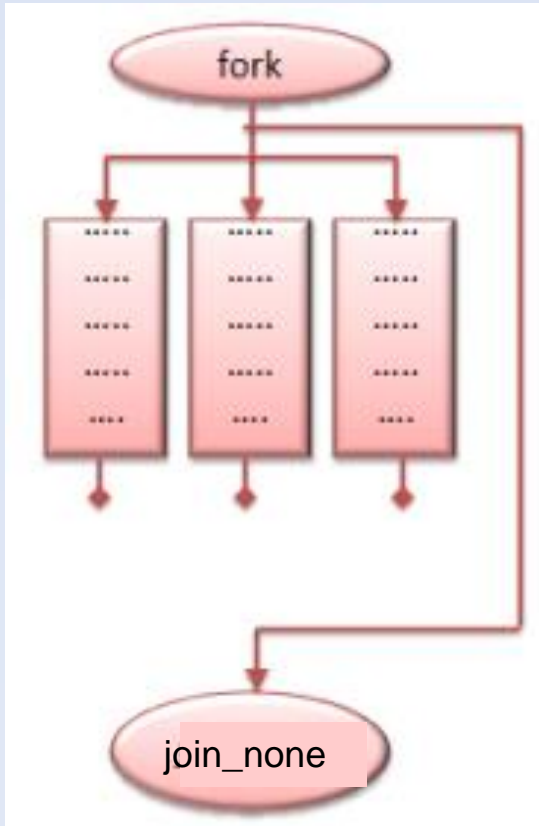
Example:

```
initial
begin
  clk = 0;
  #5
  fork
    #5 a = 0;
    #10 b = 0;
  join_any
  clk = 1; //clk becomes one at t=10
end
```

clk becomes one at
t=10

Thread Control - Parallel block constructs: fork ... join_none (SystemVerilog)

- SystemVerilog has introduced a new construct to create parallel processes – the fork ... join_none block
- In the fork join_none block of code, the parent process continues to execute concurrently with all the child processes spawned by the fork. The spawned processes do not start executing until the parent thread executes a blocking statement or terminates.

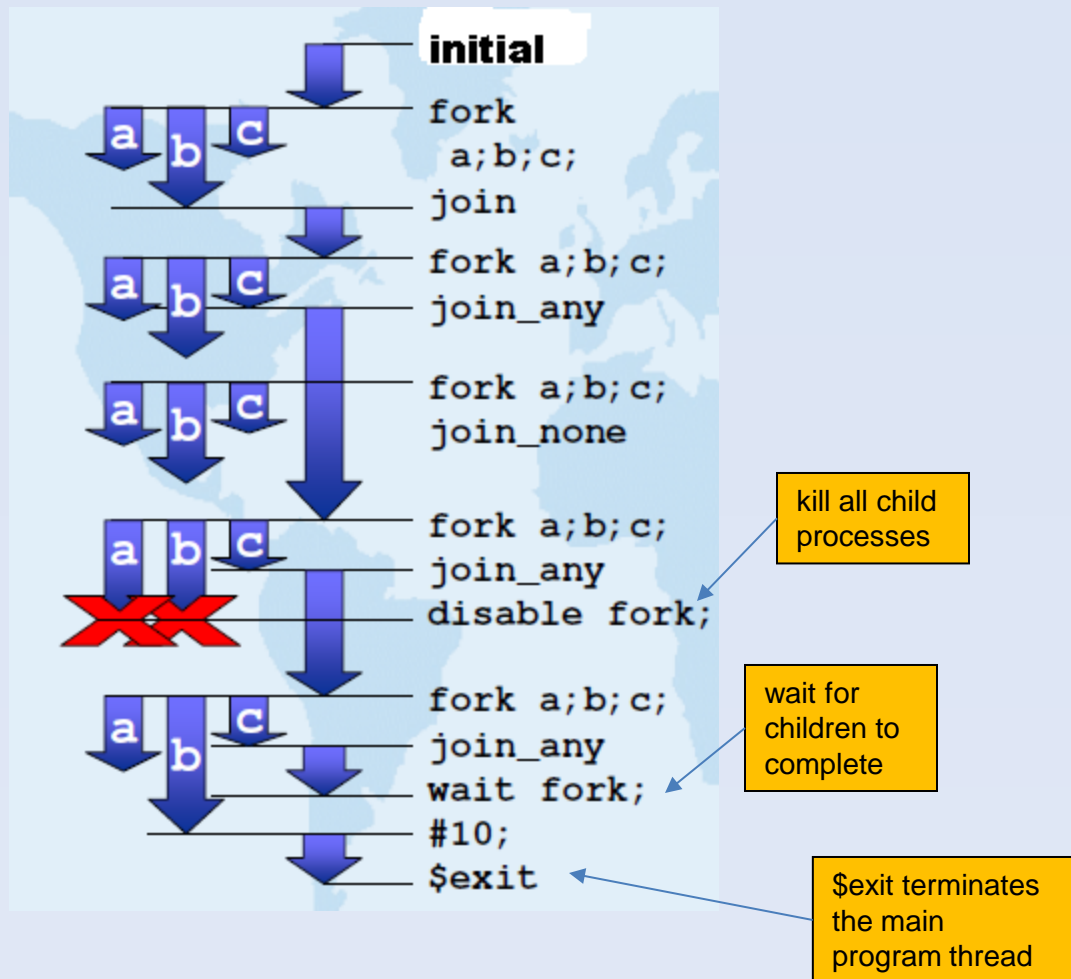


Example:

```
initial
begin
  clk = 0;
  #5
  fork
    #5 a = 0;
    #10 b = 0;
  join_none
  clk = 1; //clk becomes one at t=5
end
```

clk becomes one at
t=5

Thread Control - Parallel block constructs: wait fork, disable and disable fork (SystemVerilog)



The *wait fork* statement is used to ensure that all immediate child sub-processes (processes created by the current process, excluding their descendants) have completed their execution.

The *disable* statement provides the ability to terminate the activity associated with concurrently active processes, while maintaining the structured nature of procedural descriptions. It is useful for handling exception conditions such as hardware interrupts and global resets.

The *disable fork* statement terminates all active descendants (sub-processes) of the calling process. In other words, if any of the child processes have descendants of their own, the *disable fork* statement shall terminate them as well.

Communication Encapsulation - Interface

- In order to overcome the following disadvantages of standard Verilog module port connections, SystemVerilog adds a powerful new port type called an interface
 - ❖ Same declarations have to be repeated in multiple modules
 - ❖ Any change in design specification resulting in ports being added or removed, will require a modification to the port list of multiple modules and also changes in the names in the port list of the connecting modules
 - ❖ Risk of mismatched positional or named port maps due to pilot or scripting errors
- The interface construct captures connectivity and synchronization between various design blocks, and also between design and verification blocks of code
- It is basically a named bundle of signals in a single location, which is instantiated in a design and can be connected to interface ports of other instantiated modules, interfaces and program blocks (explained later)
- Each module that uses these signals will now have a single port of the interface type instead of multiple ports - one for every signal. The connections here are cleaner, less prone to mistakes and helps make code maintenance easy
- References to a signal in an interface is done by making a hierarchical reference using the port interface or instance name
- Interface signals should always be driven using nonblocking assignments

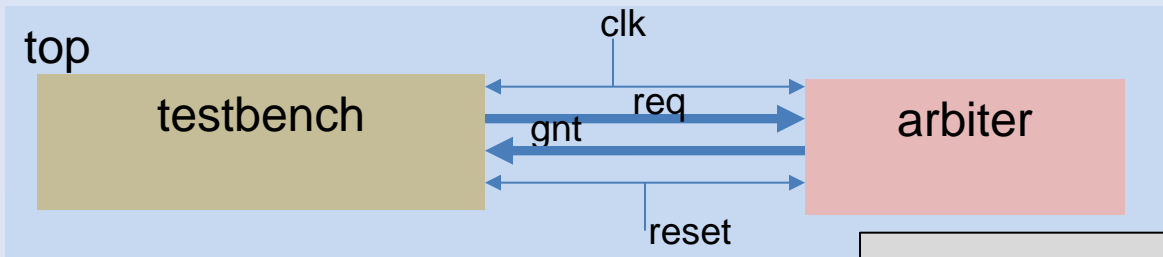
Example:

```
interface bus();    // interface definition for bus containing req, gnt, addr, data and mode signals
    logic req, gnt;
    logic [7:0] addr, data;
    logic [1:0] mode;
endinterface

module apple_soc(bus amba); // bus interface port called amba
:
    amba.gnt <= 1'b1; // hierarchical reference to signal name
                        // nonblocking assignments should always drive interface signals
endmodule
```

Communication Encapsulation – Interface: Arbiter Example using Verilog Port Connections

The following Verilog code example is that of an arbiter communicating with its testbench in the top module via the `clk`, `req`, `gnt` and `reset` ports using module instances and port maps as can be seen



Testbench with Verilog port connections

DUV (arbiter) with Verilog port connections

```
module arbiter (clk, reset, req, gnt);
```

```
    input clk, reset;
    input [1:0] req;
    output [1:0] gnt;
    wire clk, reset;
    wire [1:0] req;
    reg [1:0] gnt;
```

```
    always @(posedge clk)
```

```
    if (!reset)
        if (req == 2'b01)
            gnt <= 2'b10;
        else
            gnt <= 2'b01;
```

```
    else
        gnt <= 2'b00;
```

```
endmodule
```

```
module testbench (clk, reset, gnt, req);
```

```
    input clk, reset;
    input [1:0] gnt;
    output [1:0] req;
    wire clk, reset;
    wire [1:0] gnt;
    reg [1:0] req;
    reg [1:0] gnt;
```

```
    initial
```

```
    begin
```

```
        @(posedge clk);
```

```
        req <= 2'b10;
```

```
        $display ("Driven request to 10 at %0t", $time);
```

```
        repeat(6) @(posedge clk);
```

```
        if (gnt == 2'b01)
```

```
            $display ("Recvd grant 01 at %0t", $time);
```

```
        else
```

```
            $display ("Error at %0t. Wrong grant recvd", $time);
```

```
            $finish;
```

```
    end
```

```
endmodule
```

Top Module with Verilog port connections

```
module top();
```

```
    reg clk = 1'b0;
    reg reset = 1'b1;
    reg [1:0] gnt, req;
```

```
    initial
```

```
    begin
```

```
        // Clock Generation Logic
```

```
        forever #25 clk = ~clk;
```

```
        // Reset Generation Logic
```

```
        #150 reset = 1'b0;
```

```
    end
```

```
    arbiter duv(.clk(clk), // Arbiter DUV Instantiation
```

```
                .reset(reset),
```

```
                .req(req),
```

```
                .gnt(gnt));
```

```
    testbench tb(.clk(clk), // Testbench Instantiation
```

```
                .reset(reset),
```

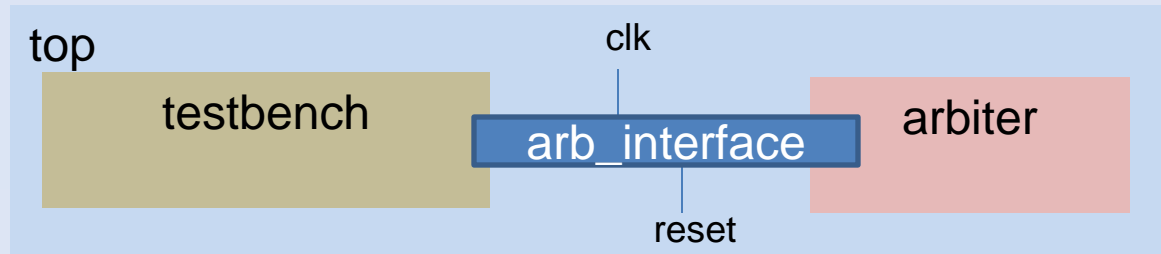
```
                .gnt(gnt),
```

```
                .req(req));
```

```
endmodule
```

Communication Encapsulation – Interface: Arbiter Example using Interface Declaration

The following Verilog code example is that of the same arbiter example shown in the previous page, but communicating with its testbench in the top module via an `arb_interface` interface using module and interface instances as can be seen



Arbiter Interface declaration

```
interface arb_interface(input bit clk, reset);  
    logic [1:0] gnt, req;  
endinterface
```

Testbench using the interface

```
module testbench(arb_interface arbif);  
    initial  
    begin  
        @(posedge arbif.clk);  
        arbif.req<=2'b10;  
        $display ("Driven request to 10 at %0t", $time);  
        repeat(6) @(posedge arbif.clk);  
        if(arbif.gnt==2'b01)  
            $display ("Recvd grant 01 at %0t", $time);  
        else  
            $display ("Error at %0t. Wrong grant recvd", $time);  
        $finish;  
    end  
endmodule
```

DUV (arbiter) using the interface

```
module arbiter(arb_interface arbif);  
    always @(posedge arbif.clk)  
    if (!arbif.reset)  
        if(arbif.req == 2'b01)  
            arbif.gnt <= 2'b10;  
        else  
            arbif.gnt <= 2'b01;  
    else  
        arbif.gnt <= 2'b00;  
endmodule
```

Top Module using the interface

```
module top();  
    logic clk = 1'b0;  
    logic reset = 1'b1;  
  
    initial  
    begin  
        // Clock Generation Logic  
        forever #25 clk = ~clk;  
        // Reset Generation Logic  
        #150 reset = 1'b0;  
    end  
    arb_interface arbif(clk, reset); // Interface Instantiation  
    arbiter duv(arbif);              // Arbiter DUV Instantiation  
    testbench tb(arbif);             // Testbench Instantiation  
endmodule
```

Communication Encapsulation - Interface -- Modports

- SystemVerilog interfaces provide a means to define different directions for the interface signal, i.e., whether the connecting module sees the signal direction as input, output or bidirectional. These are accomplished using corresponding modport lists
- modport is an abbreviation for "module port". It indicates that the directions are declared as if inside the module
- An interface can have any number of modport definitions

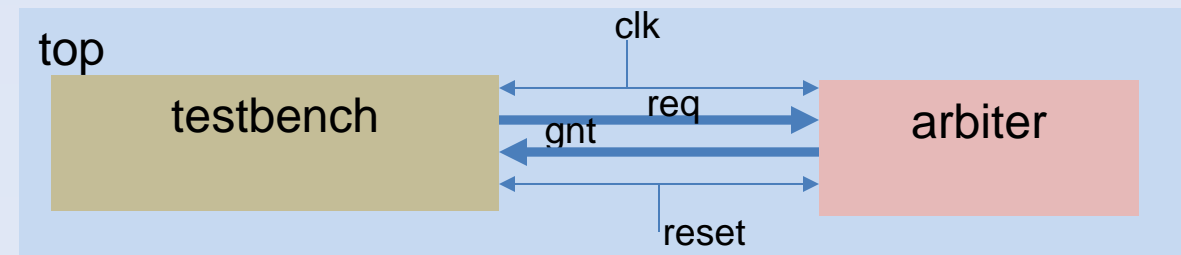
The arbiter interface declaration code from the example shown on the prior page is re-written as follows with modports:

Arbiter Interface declaration with modports

Example:

```
interface arb_interface(input bit clk, reset);  
  logic [1:0] gnt, req;  
  modport FOR_TB(input reset, gnt, clk, output req);  
  modport FOR_DUV(input req, reset, clk, output gnt);  
  modport FOR_MONITOR(input reset, clk, req, gnt);  
endinterface
```

Modports specifies the directions



Communication Encapsulation - Interface -- Modports Continued...

- The syntax of **interface_name.modport_name reference_name** gives a hierarchical reference to a local name. This can be generalized to any interface with a given modport name
- There are 2 ways to specify modport names in a design

(a) In the modules that connect to the interface signals

Here, only the module interface syntax changes to include the modport name as well. *The top module code remains the same and does not change.*

The DUV and Testbench code for the example shown on the prior page is re-written as follows with modports:

DUV (arbiter) with the interface using modports

```
module arbiter (arb_interface.FOR_DUV arbif);  
    : // same code as before – no changes  
endmodule
```

Testbench with the interface using modports

```
module testbench (arb_interface.FOR_TB arbif);  
    : // same code as before – no changes  
endmodule
```

Communication Encapsulation - Interface -- Modports Continued...

(b) The modport is specified directly when instantiating the modules that connect to the interface signals in the top level module

Here, only the top module code changes. The modports are defined for the interface instance and given in the module instance port list.
The code for the modules that connect to the interface signals does not change.

The top module code for the example shown on the prior page is re-written as follows with modports:

Top Module with the interface using modports

```
module top();  
    logic clk = 1'b0;  
    logic reset = 1'b1;  
  
    initial  
    begin  
        // Clock Generation Logic  
        forever #25 clk = ~clk;  
        // Reset Generation Logic  
        #150 reset = 1'b0;  
    end  
    arb_interface arbif(clk, reset); // Interface Instantiation  
    arbiter duv(arbif.FOR_DUV); // Arbiter DUV Instantiation  
    testbench tb(arbif.FOR_TB); // Testbench Instantiation  
endmodule
```


Communication Encapsulation - Virtual Interface

Definition: A virtual interface is a pointer to an actual physical interface instance. Interfaces are static in nature, while classes and objects are dynamic. Virtual interfaces help bridge the dynamic world of objects with the static world of modules and interfaces.

Syntax: virtual interface_name interface_identifier;

Features and Characteristics:

- It is most often used in classes to provide a connection point to allow classes to access the signals in the interface through the virtual interface
- Virtual interfaces can be declared as class properties, which can be initialized procedurally or by an argument to new(). This allows the same virtual interface to be used in different classes
- Instead of referring directly to the actual set of signals, users can manipulate a set of virtual signals using virtual interfaces
- Like any other data type, virtual interfaces can be passed to tasks or functions
- A virtual interface must be initialized before it can be used. By default, it points to null. Attempting to use an uninitialized virtual interface will result in a run-time error
- Virtual interfaces should not be used as ports, interface items, or as members of unions

Communication Encapsulation - Virtual Interface Continued...

The following example shows how the same class can be used to interact with various different devices using a virtual interface:

Static interface declaration

```
interface VBus; // VBus interface definition
  logic req, gnt;
  logic [31:0] address, data;
endinterface
```

Virtual Interface defined in a class (Dynamic)

```
class VBusxtor; // VBus transactor class

  virtual VBus bus; // Virtual interface of type Vbus

  function new(virtual VBus v);
    bus = v; // Initialize the virtual interface
  endfunction

  task sendbusreq(); // Request for the bus
    bus.req <= 1'b1;
  endtask

  task waitbusgnt(); // Wait for the bus to be granted
    @(negedge bus.gnt);
  endtask

endclass
```

Modules that use the VBus interface

```
// Multiple devices that use the VBus interface

module device1(VBus v)
  ....
endmodule

module device2(VBus v)
  ....
endmodule
```

The VBusxtor class is a reusable component. It can interact with any number of devices (6 are shown in the example) that adhere to the VBus interface protocol. A dynamic virtual interface is connected to a static interface in the soctop module. The virtual interface is brought from the lower levels in the testbench hierarchy (e.g.: driver or monitor) to the soctop module by passing the virtual interface as an argument to the class constructor - the function new()

soctop module instantiates the interfaces and the devices

```
module soctop;

  VBus v[1:6] (); // Instantiate 6 interfaces

  device1 ram1(v[1]); // instantiate 6 devices --
  device2 spi1(v[2]); // 3 from device1 and 3
  device1 ram2(v[3]); // from device2
  device2 spi2(v[4]);
  device1 ram3(v[5]);
  device2 spi3(v[6]);

  initial
  begin
    // Create 6 bus-transactors and bind them here
    VBusxtor trans[1:6];

    trans[1] = new(v[1]);
    trans[2] = new(v[2]);
    trans[3] = new(v[3]);
    trans[4] = new(v[4]);
    trans[5] = new(v[5]);
    trans[6] = new(v[6]);
  end

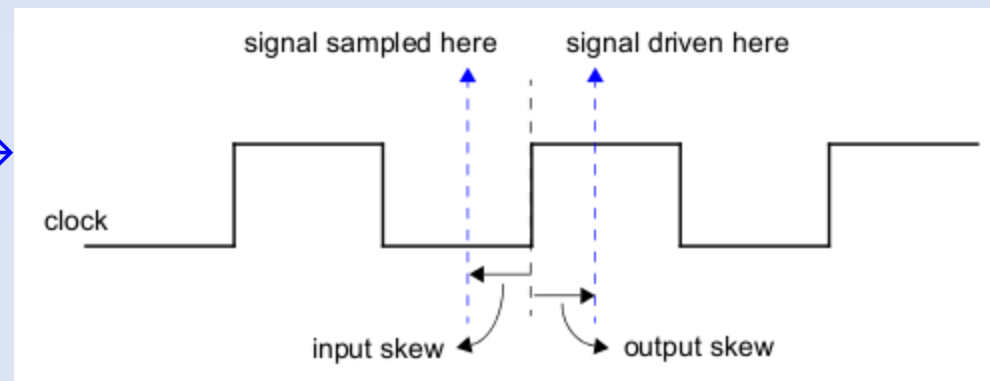
endmodule
```

Timing and Synchronization Capture - Clocking Blocks

- The timing between a design and its testbench should be synchronized in order to avoid race conditions. Clocking blocks have been introduced in SV to provides a means for grouping signals that are synchronous to a particular clock and making their timing explicit
- They can only be declared within a module, interface or a program block (explained later)
- A testbench can contain one or more clocking blocks, each containing its own clock along with an arbitrary number of signals
- Each clocking block must have at least one clock associated with it
- The clocking block separates the timing and synchronization details from the structural, functional, and procedural details of a testbench
- Signals in a clocking block are driven or sampled synchronously, ensuring that the testbench interacts with the signals at the right time
- Clocking blocks cannot be synthesized
- The *clocking_skew* determines how many time units away from the clock event a signal is to be sampled or driven. Input skews are implicitly negative, that is, they always refer to a time before the clock edge, whereas output skews always refer to a time after the clock edge

Timing and Synchronization Capture - Clocking Blocks Continued...

Input and Output Skew w.r.t the Positive Edge of 'clock' →



Example:

```
clocking grp_sysclk@(posedge sysclk);  
  default input #10ns output #3ns; // Clocking skews defined here  
  input request, enable, data;  
  output negedge grant; // Override the default output skew - grant is driven on the negative edge of sysclk  
  input #1step addr; // Override the default input skew - address is sampled one step before the positive edge of sysclk  
endclocking
```

- In the example shown, **default** keyword defines the default skews for inputs (10ns) and outputs (3ns)
- Unless otherwise specified, the default input skew is 1step and the default output skew is 0. A 1step skew is the negative time delay from the edge of a clock when all inputs are steady and ready to be sampled
- Default skews can be overridden for any signal with what the user specifies

Timing and Synchronization Capture - Clocking Blocks Continued...

- Clocking block signals must always be driven using nonblocking assignments
- Signals in a clocking block can be referenced using the following syntax:

```
[module_or_interface_or_program_block_name.]clocking_block_name.signal_name
```

The arbiter interface declaration with modports from the example shown on the prior page can have clocking blocks as shown in the code below. Also seen is how these clocking block signals are referenced in the testbench

```
interface arb_interface(input bit clk, reset);
  logic [1:0] gnt, req;

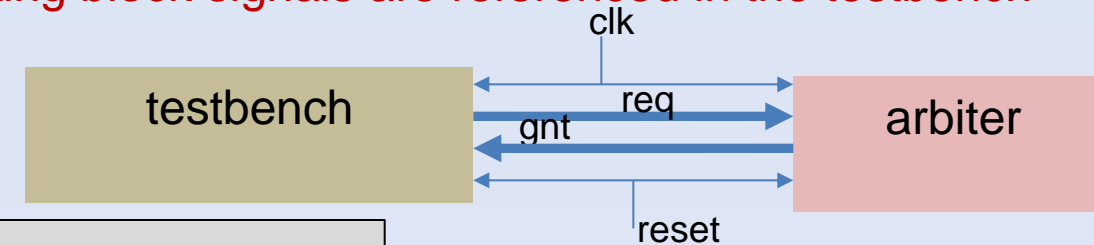
  clocking clk_blk_tb@(posedge clk);
    default input #2ns output #3ns;
    input gnt;
    output req;
  endclocking

  clocking clk_blk_duv@(posedge clk);
    default input #2ns output #2ns;
    input req;
    output gnt;
  endclocking

  modport FOR_TB(input reset, clk, clocking clk_blk_tb);
  modport FOR_DUV(input reset, clk, clocking clk_blk_duv);
  modport FOR_MONITOR(input reset, clk, req, gnt);

endinterface
```

```
module testbench(arb_interface.FOR_TB arbif);
  initial
  begin
    @(posedge arbif.clk);
    arbif.clk_blk_tb.req<=2'b10;
    $display ("Drove request to 01 at %0t", $time);
    repeat(6) @(posedge arbif.clk);
    if(arbif.clk_blk_tb.gnt==2'b10)
      $display ("Recvd grant 10 at %0t", $time);
    else
      $display ("Error at %0t. Wrong grant recvd", $time);
    $finish
  end
endmodule
```



Program Block

- The testbench environment is typically written in a program block within the program ... endprogram keywords
- It is an entry point for the execution of testbenches and can contain data declarations, class definitions, subroutine definitions, object instances and one or more initial blocks. It cannot contain always blocks, primitive instances, module instances, interface instances, or other program instances – no hierarchy is allowed
- They are typically instantiated at the top-level
- Clocks are never generated in program blocks
- In conjunction with clocking blocks, program blocks schedule events in the “Reactive region” of the stratified queue thereby helping prevent race conditions that exist between the design and the testbench
- Interfaces & ports can be connected in the same manner as any other module
- The program block separates the DUV from the testbench
- Program blocks can read and write signals present in modules, but modules have no visibility into program blocks
- An implicit \$finish happens when all initial blocks end inside the program block

Example:

```
program exmplprogblk(arb_interface.FOR_TB arbif);  
  
initial  
begin  
    @arbif.clk_blk_tb;  
    wait (arbif.clk_blk_tb.gnt == 2'b01);  
    :  
    :  
end  
  
initial  
begin  
    #(10);  
    $display(" BEFORE fork time = %d ",$time);  
    fork  
    begin  
        # (20);  
        $display("fork 1 time = %d # 20 ",$time);  
    end  
    begin  
        #(10);  
        $display("fork 2 time = %d # 10 ",$time);  
    end  
    join_any  
    $display("AFTER fork time = %d", $time);  
end  
  
endprogram
```

Interprocess Synchronization and Communication

- Semaphores

Definition: Semaphores are used for synchronization and mutual exclusion of shared resources, thereby providing access control to shared resources.

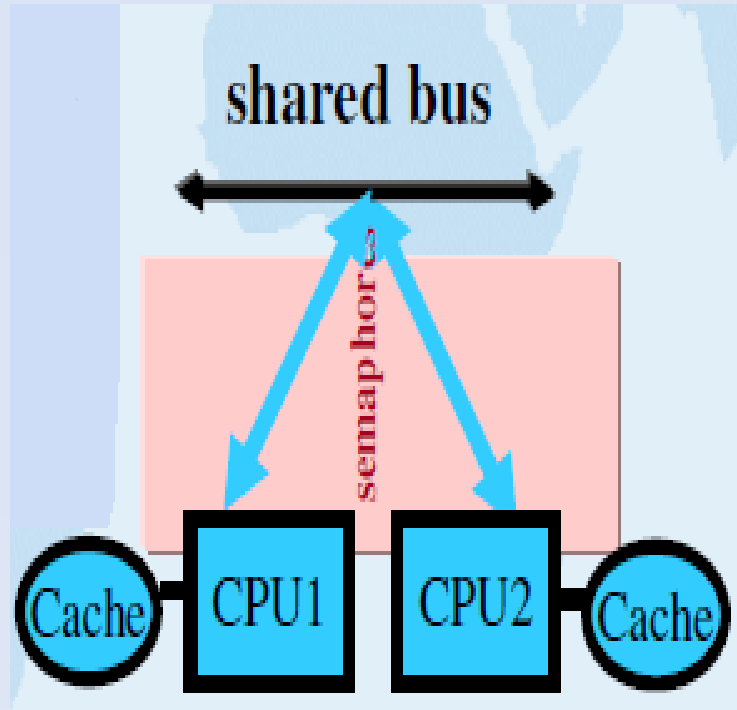
Syntax: semaphore semaphore_name;

E.g.: Consider the case where 2 people 'A' and 'B' share one room that has a lock to access it. Only 1 person can occupy it at a time after opening the lock using a key. When 'A' is done accessing the room (s)he hands over the key to 'B' who then unlocks the room and uses it. Here, the 'key' is the semaphore that makes sure only one person has access to the room at a time.

Conceptually, a semaphore is like a bucket. When a semaphore is allocated, a bucket that contains a fixed number of keys is created. Processes using semaphores must first procure a key from the bucket before they can execute. In reality, a semaphore is a built-in class that provides the following pre-defined methods. These methods are called after the semaphore object name with a period "." between the object name and the method name.

- new(num_keys): Create a semaphore with a specified number of keys
- get(num_keys): Obtain one or more keys from the semaphore bucket and blocks execution until the required number of keys are available
- put(num_keys): Return one or more keys to the semaphore bucket
- try_get(num_keys): Obtain one or more keys from the semaphore without blocking. If the required number of keys are not available, continue execution without waiting

Interprocess Synchronization and Communication – Semaphores Continued...



The shared bus contention problem occurs when multiple CPUs try to access the bus at the same time. Semaphores help CPUs gain access to the shared bus by providing mutual exclusion.

Example:

```
program bus_contention;
    semaphore sbus = new(1);
    task block_bus(string name, integer busval);
        $display("[%0t]: %s gets key and is going to block the shared bus for 50 time units to write the value %0d", $time,
        name, busval);
        #50;
    endtask
    initial
    begin
        fork
        begin
            #55;
            $display("[%0t]: CPU1 requests key", $time);
            sbus.get(1);
            block_bus("CPU1",1000);
            $display("[%0t]: CPU1 returns key", $time);
            sbus.put(1);
            #10
            $display("[%0t]: CPU1 requests key", $time);
            sbus.get(1);
            block_bus("CPU1",3000);
            $display("[%0t]: CPU1 returns key", $time);
            sbus.put(1);
        end
        begin
            #25;
            $display("[%0t]: CPU2 requests key", $time);
            sbus.get(1);
            block_bus("CPU2",2000);
            $display("[%0t]: CPU2 returns key", $time);
            sbus.put(1);
            #25;
            $display("[%0t]: CPU2 requests key", $time);
            sbus.get(1);
            block_bus("CPU2",4000);
            $display("[%0t]: CPU2 returns key", $time);
            sbus.put(1);
        end
    join
    #25;
    $display("[%0t]: FINISHED Simulation", $time);
end
endprogram
```

1 key

Mutual exclusion using semaphore methods

Interprocess Synchronization and Communication – Semaphores Continued...

Program Output (Simulated):

```
[25]: CPU2 requests key
[25]: CPU2 gets key and is going to block the shared bus for 50 time units to write the value 2000
[55]: CPU1 requests key
[75]: CPU2 returns key
[75]: CPU1 gets key and is going to block the shared bus for 50 time units to write the value 1000
[100]: CPU2 requests key
[125]: CPU1 returns key
[125]: CPU2 gets key and is going to block the shared bus for 50 time units to write the value 4000
[135]: CPU1 requests key
[175]: CPU2 returns key
[175]: CPU1 gets key and is going to block the shared bus for 50 time units to write the value 3000
[225]: CPU1 returns key
[250]: FINISHED Simulation
```

Interprocess Synchronization and Communication

- Mailboxes

Definition: A mailbox is an inter-process communication mechanism that allows messages to be exchanged between processes. It is a built-in class around a queue that uses semaphores to control access to the ends of a queue, when there are multiple process threads that are reading and writing data.

Syntax: mailbox mailbox_name;

Features and Characteristics:

- It is a FIFO queue of any data type with a source and a sink. The source sends data into the mailbox while the sink retrieves it
- The mailbox acts as a synchronizer between one or more processes
- The mailbox queue size may be bounded (can become full) or unbounded (never becomes full)
- If the bound value is 0, then the mailbox is unbounded and can theoretically hold an infinite number of messages
- A bounded mailbox becomes full when it reaches the maximum number of messages
- A process that attempts to place a message into a full mailbox shall be suspended until enough room becomes available in the mailbox queue
- Unbounded mailboxes never suspend a thread in a send operation (explained later)

Interprocess Synchronization and Communication

– Mailbox Methods

Like a semaphore, a mailbox is a built-in class and has a set of pre-defined methods that operate on it. These methods are called after the mailbox object name with a period “.” between the object name and the method name.

- `new(bound)`: Creates a mailbox with a capacity of the specified ‘bound’ number of messages. bound has to be a positive value or 0.
- `num()`: Returns the number of messages in a mailbox
- `put()`: Places a message in the mailbox. If the mailbox is full, the process suspends and blocks until there is enough place in the queue to place the message.
- `try_put()`: Attempts to place a message in the mailbox. The process does not block if the mailbox is full.
- `get()`: Retrieves and removes a message from the mailbox, if available. If the mailbox is empty, then the current process blocks until a message is placed in the mailbox.
- `try_get()`: Retrieves and removes a message from the mailbox. The process does not block if the mailbox is empty.
- `peek()`: Copies a message from a mailbox, if available. If not, block. It does not remove the message from the mailbox queue.
- `try_peek()`: Copies a message from a mailbox without blocking

Interprocess Synchronization and Communication

– Mailbox Methods Continued...

Example:

```
program mymail;
  mailbox mndata = new();

  initial
  begin
    fork
      put_value();
      get_value();
    join_any
    #75;
  end

  task put_value();
  begin
    static bit [31:0] value = 0;

    for(integer i = 0; i < 6; i++)
    begin
      #4;
      value = $random();
      $display("[%0t]: Put value %0d into mailbox", $time, value);
      mndata.put(value);
    end
  end
endtask
```

The task "get_value" attempts to retrieve a 32-bit data value from the mailbox

```
task get_value();
begin
  static bit [31:0] value = 0;

  while(1)
  begin
    #2;
    if (mndata.num() > 0)
    begin
      → mndata.get(value);
      $display("[%0t]: Got value %0d from mailbox", $time, value);
    end
    else
    begin
      #5;
    end
  end
endtask

endprogram
```

The task "put_value" places a 32-bit data value into the mailbox

Interprocess Synchronization and Communication

– Mailbox Methods Continued...

Program Output (Simulated):

```
[4]: Put value 303379748 into mailbox  
[8]: Put value 3230228097 into mailbox  
[9]: Got value 303379748 from mailbox  
[11]: Got value 3230228097 from mailbox  
[12]: Put value 2223298057 into mailbox  
[13]: Got value 2223298057 from mailbox  
[16]: Put value 2985317987 into mailbox  
[20]: Put value 112818957 into mailbox  
[22]: Got value 2985317987 from mailbox  
[24]: Put value 1189058957 into mailbox  
[24]: Got value 112818957 from mailbox  
[26]: Got value 1189058957 from mailbox
```

Interprocess Synchronization and Communication – Parameterized Mailboxes

By default a mailbox is typeless, i.e., it can send and receive any type of data. This could result in run-time errors due to type mismatches between a message and the type of the variable used to retrieve the message. It would be useful to detect type mismatches at compile time itself. Parameterized mailboxes enable this feature.

Features and Characteristics:

- A parameterized mailbox is declared by specifying the type of data
- Parameterized mailboxes provide all the same built-in methods as the default mailboxes – new(), num(), put(), get(), peek(), try_get() etc.
- The only difference between a default mailbox and a parameterized mailbox is that for a parameterized mailbox, the compiler ensures that the calls to put(), try_put(), peek(), try_peek(), get(), and try_get() methods use argument types equivalent to the mailbox type, so that all type mismatches are caught at compile time and not at run time

Example:

```
string str;  
int ival;  
  
mailbox #(string) string_mbox;  
  
string_mbox gotanymail = new(); // Unbounded mailbox  
  
gotanymail.put( "PLK and RP" );  
gotanymail.get(str); // Type match – Correct  
  
gotanymail.get(ival); // Type mismatch string and integer,  
                      // leading to Compile Error
```

Mailbox string_mbox
is parameterized with
the string data type. It
accepts only string
data elements.

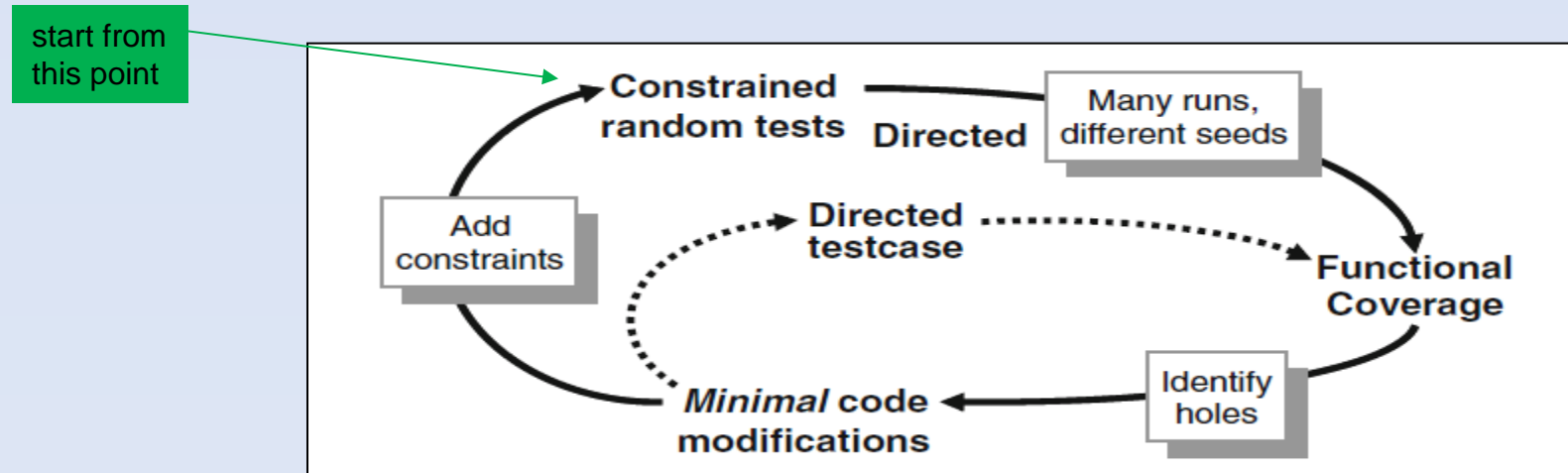
Constrained Randomization - Introduction

With the complex functionality of present-day designs and as chips continue to grow larger, it becomes extremely difficult to create a complete suite of directed test cases to check their functionality. Random testing can be more effective than a traditional, directed testing approach. Constraint-driven randomized test generation allows users to automatically generate tests for functional verification.

Features and Characteristics:

- By specifying constraints, it is easy to create tests that can find hard-to-reach corner cases that neither verification engineers nor designers have thought about
- SystemVerilog allows users to specify constraints in a compact and declarative manner. The constraints determine the legal values that can be assigned to the random variables
- The specified constraints are processed by a solver embedded in the simulator. If the constraints are met, the solver returns random values that adhere to those constraints
- Constraints should be written carefully so as to target precise scenarios and not overload the constraint solver. Long constraint solver overhead times ultimately results in additional time required for verification
- The random stimulus generation capabilities and the object-oriented constraint-based verification methodology enables users to quickly develop tests that cover complex functionality and assures better design correctness

Constrained Randomization - Introduction Continued...



Functional Verification Cycle

Procedural Steps:

- (1) Start by writing basic constrained-random tests for the design. Run these tests with different seeds to cover as many functional scenarios as possible
- (2) After the test run, check the functional coverage reports (explained in the next topic). If the desired functional coverage metric is not met, identify holes in the test scenarios for corresponding gaps in the coverage
- (3) Based on the holes identified, make minimal code changes by
 - ❖ Adding new constraints to the tests already present, and/or,
 - ❖ Writing directed tests for only the new features that are not likely to be hit using random tests
- (4) Repeat the above steps until the desired functional coverage level is reached, with the majority of the time spent in the constrained random testing loop

Constrained Randomization - Declaring Random Variables using rand

Variables that are declared with the **rand** keyword are called standard random variables

- Their values are uniformly distributed over the specified range of its type when there is no control on distribution
- If unconstrained, the variable will be assigned any value in the specified range with equal probability
- Rand variables are randomized only with the call of the randomize() function. It returns a value '1' on success and '0' on failure. Every class has a built-in randomize() virtual function.

random variables
declared using the
rand keyword

Example:

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;
endclass

Bus bus = new;

repeat (10)
begin
  if (bus.randomize() == 1) // Randomize variables inside the Bus class
    $display ("addr = %d, data = %d\n", bus.addr, bus.data);
  else
    $display ("Randomization failed\n");
end
```

The .randomize()
function is used to
randomize
properties (that are
declared as rand) of
an object of a class

Constrained Randomization - Declaring Random Variables using randc

Variables that are declared with the **randc** keyword are called standard random cyclic variables

- They cycle through all the values in a random permutation of their declared range
- No value is repeated within an iteration. When an iteration finishes, a new iteration automatically starts.
- randc is applicable only for bit and enumerated data types
- randc variables are randomized only with the call of the randomize() function. It returns a value '1' on success and '0' on failure. Every class has a built-in randomize() virtual function.

random variables
can also be declared
using the randc
keyword

Example:

```
class Bus;
  randc bit[1:0] addr;
  rand bit[3:0] data;
endclass

Bus bus = new;

repeat (10)
begin
  if (bus.randomize() == 1) // Randomize variables inside the Bus class
    $display ("addr = %d, data = %d\n", bus.addr, bus.data);
  else
    $display ("Randomization failed\n");
end
```

The .randomize()
function is used to
randomize
properties (that are
declared as rand or
randc) of an object
of a class

bus.addr value:

initial permutation 2 -> 0 -> 3 -> 1

next permutation 0 -> 3 -> 2 -> 1

Constrained Randomization - Constraint Block

The values of random variables can be bound using constraint expressions that are declared using constraint blocks

Syntax: `constraint constraint_identifier_name {constraint_block_expression(s);}`

Features and Characteristics:

- Constraint identifier names shall be unique within a class
- Constraint blocks are members of a class, similar to properties, tasks and functions
- Calling `randomize()` causes new values to be selected for all of the random variables specified within the constraint block so that the given constraint expression evaluates to true. All other unconstrained random variables are assigned any value in their range

Example:

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;

  constraint word_align {addr[1:0] == 2'b0;}
endclass

Bus bus = new;

repeat (10)
begin
  if (bus.randomize() == 1) // Randomize variables inside the Bus class
    $display ("addr = %d, data = %d\n", bus.addr, bus.data);
  else
    $display ("Randomization failed\n");
end
```

random variables
declared using the
rand keyword

Constraint block

The `.randomize()` function is used to randomize the upper 14 bits of `addr` (`addr[15:2]`) with its lower 2 bits (`addr[1:0]`) set to 0, and to randomize all 32-bits of `data`

constraint block
expression states that
address `addr[1:0]`
should always be equal
to zero

Constrained Randomization - Inline Constraint

Users can declare in-line constraints at the point where the `randomize()` function is called, by using the `randomize()` **with** construct. These additional constraints are applied along with the object constraints.

Syntax: `class_object.randomize() with {constraint_block_expression(s);}`

Example:

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;

  constraint word_align {addr[1:0] == 2'b0;}
endclass

Bus bus = new;
repeat (10)
begin
  if ((bus.randomize() with {data==32'b1;} == 1)
    $display ("addr = %d, data = %d\n", bus.addr, bus.data);
  else
    $display ("Randomization failed\n");
end
```

Objects can be further constrained using the `randomize()` **with** construct, which declares additional constraints in line with the call to `randomize()`. In addition to the lower 2-bits of `addr` i.e., `addr[1:0]` being zero, it is a must that `data` has to be equal to `32'b1`. Hence here only the upper 14-bits of `addr`, i.e., `addr[15:2]` will be randomized

Constrained Randomization - Set Membership and Distribution

Constraint block expressions support integer value sets using the set membership operator **inside**. All values have equal probability of being chosen provided there are no other constraints.

Constraint block expressions also support sets of weighted values called distributions. The weights are not a percentage and do not have to add up to a 100. Some values need to have a weight more than the others. This description helps specify a statistical distribution function for returning values appropriately.

Example:

```
class Bus;
  rand bit[10:0] addr;
  rand bit[7:0] data;

  constraint myaddr_space {
    addr inside {0, 2, 6, [64:127], [255:302]};           // Set Membership
  };

  constraint mydata_weight {
    data dist {                                           // Distribution
      [0:31]    := 3,    // Weight of 3/72
      [32:63]   := 5,    // Weight of 5/72
      [64:127]  := 1,    // Each element has weight of 1/72
    };
  };
endclass
```

addr is constrained to ONLY values 0, 2, 6 and anything in the range between 64 to 127 and anything in the range between 255 to 302

data can take values between 0 and 127. This constraint distributes the weights of the occurrence of a particular value. For e.g.: The value '4' is 3 times more likely to occur than the value '72'. Similarly a value '46' is 5 times more likely to occur than the value '100'.

Constrained Randomization - The `constraint_mode()` Method

The `constraint_mode()` method can be used to enable or disable a named constraint identifier block. When a constraint block is inactive, it is not considered by the `randomize()` function. By default the `constraint_mode()` is ON, i.e., 1.

`constraint_mode()` is a built-in method and cannot be overridden.

Syntax:

```
class_object.constraint_identifier_name.constraint_mode(0 or 1);
```

Example:

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;

  constraint word_align {addr[1:0] == 2'b0;}
  constraint addr_zero {addr==0;}
endclass

Bus bus = new;

repeat (10)
begin
  bus.word_align.constraint_mode(0);

  bus.addr_zero.constraint_mode(1);

  if ((bus.randomize() with {data==32'b1;})) == 1)
    $display ("addr = %d, data = %d\n", bus.addr, bus.data);
  else
    $display ("Randomization failed\n");

  bus.addr_zero.constraint_mode(0);

  bus.word_align.constraint_mode(1);

  ...
end
```

The word_align constraint is turned OFF

The addr_zero constraint is turned ON

The addr_zero constraint is turned OFF

The word_align constraint is turned ON

Constrained Randomization - The rand_mode() Method

The rand_mode() method is used for dynamic constraint modification. Randomization of all random properties in a class can be dynamically enabled or disabled anytime using the rand_mode() method. This method can also be applied to dynamically enable or disable individual random properties of a class. When a random variable is inactive, it is treated the same way as if it had not been declared with the rand or randc keywords. Inactive variables are not randomized by the randomize() function. All random variables are initially active.

rand_mode() is a built-in method and cannot be overridden.

Syntax:

```
class_object.[random_variable_name.]rand_mode(0 or 1);
```

Example:

```
class Bus;
  rand bit[15:0] addr;
  rand bit[31:0] data;
  constraint word_align {addr[1:0] == 2'b0;}
endclass
```

```
Bus bus = new;
```

```
bus.rand_mode(0);
```

```
repeat (10)
```

```
begin
```

```
  if (bus.randomize() == 1)
```

```
    $display ("addr = %d, data = %d\n", bus.addr, bus.data);
```

```
  else
```

```
    $display ("Randomization failed\n");
```

```
end
```

```
...
```

```
bus.rand_mode(1);
```

```
bus.addr.rand_mode(0);
```

```
...
```

```
bus.addr.rand_mode(1);
```

```
...
```

bus.rand_mode(0) makes all the random variables as normal variables and disables their randomness property. One can now use them as any normal variable in the class

Calling randomize() function after rand_mode(0) will not change the values of rand variables

bus.rand_mode(1) enables the randomness to the variables. Calling randomize() now will assign appropriate random values to the variables

bus.addr.rand_mode(0/1) will disable / enable the randomness on the addr variable

Constrained Randomization - The pre_randomize() & post_randomize() Methods

Sometimes it is necessary to perform an action immediately before every randomize call or immediately afterwards. SystemVerilog contains two built-in empty methods pre_randomize() and post_randomize() that are automatically called before and after the randomize() method is called

The pre_randomize() and post_randomize() methods can be overridden

- pre_randomize() in any class can be overridden to perform initialization and set pre-conditions before the object is randomized
- post_randomize() in any class can be overridden to perform cleanup, print diagnostics, and check post-conditions after the object is randomized

Example:

```
program pre_post_exmpl;
class Bus;
    bit parity = 1'b0;
    rand bit[15:0] addr;
    rand bit[31:0] data = 32'b0;
    constraint word_align {addr[1:0] == 2'b0;}
```

```
function void pre_randomize();
    $display("\nIn the pre_randomize overridden function");
    addr = 16'hFFFF;
    $display("\npre_randomize: addr = %0h, data = %0d, parity = %0b", addr, data, parity);
endfunction

function void post_randomize();
    $display("\nIn the post_randomize overridden function");
    parity = ^data;
    $display("\npost_randomize: addr = %0h, data = %0d, parity = %0b", addr, data, parity);
endfunction
endclass
```

```
Bus bus = new;
```

```
initial
begin
    if (bus.randomize() == 1)
        $display ("Next statement after randomize function: addr = %0h, data = %0d", bus.addr, bus.data);
    else
        $display ("Next statement after randomize function: Randomization failed");
    end
endprogram
```

The pre_randomize() and post_randomize() overridden functions will be called before and after the randomize() call respectively.

Constrained Randomization - The pre_randomize() & post_randomize() Methods Continued...

Output after running the pre_post_exmpl program:

In the pre_randomize overridden function

pre_randomize: addr = ffff, data = 0, parity = 0

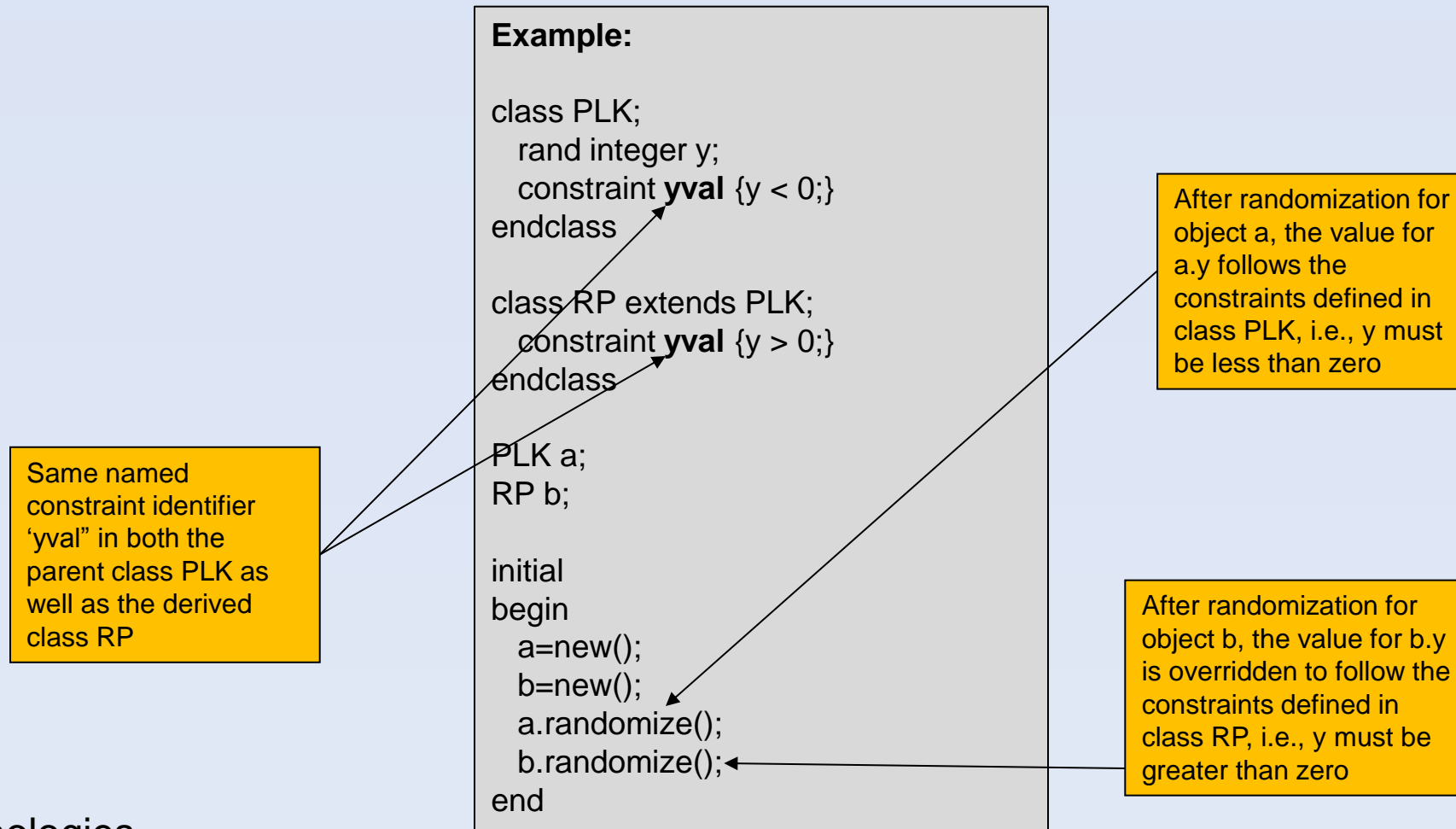
In the post_randomize overridden function

post_randomize: addr = da48, data = 13, parity = 1

Next statement after randomize function: addr = da48, data = 13

Constrained Randomization - Constraint Overriding

A constraint in a derived class that uses the same identifier name as the constraint in its parent class overrides the parent class constraints. In other words, when a named constraint is redefined in an extended class, the previous definition is overridden



Constrained Randomization - Constraint Extension

A constraint in a derived class can easily extend the constraint of its parent class. This is illustrated in the example shown.

Example:

```
class PLK;  
  rand integer y;  
  constraint yval1 {y > 0; y < 200;}  
endclass
```

```
class RP extends PLK;  
  constraint yval2 {y > 100;}  
endclass
```

```
PLK a;  
RP b;
```

```
initial  
begin  
  a=new();  
  b=new();  
  a.randomize();  
  b.randomize();  
end
```

After randomization for object a, the value for a.y follows the constraints defined in class PLK, i.e., y must be greater than 0 and lesser than 200

After randomization for object b, the value for b.y follows the constraints of class PLK and that of class RP as well. Here, both (y > 0 and y < 200) AND (y > 100) constraints need to be honored by the constraint solver. Thus (y > 100 and y < 200) is the legal range of values that the solver should return in order to satisfy both constraints. The RP class constraint yval2 has extended the yval1 constraint of its parent class PLK.

8. Functional Coverage

Coverage

Definition: Coverage is defined as the percentage of verification objectives that have been met. It is used as a metric for evaluating the progress of a verification project in order to reduce the number of simulation cycles spent in verifying a design.

Coverage is used as a guide for directing verification resources by identifying tested and untested portions of a design. There are 2 types of coverage metrics used in validation:

1) Code Coverage – Introduced in Verilog 2001

- ❖ This is automatically extracted from the RTL code of a design
- ❖ It is a measure of how much of the RTL implementation of the design specification has been successfully hit by test cases during a verification run
- ❖ Line, Expression, Toggle, FSM and Branch coverage are the different categories of code coverage that one tests for

2) Functional Coverage – Introduced in SystemVerilog

- ❖ This is a user-defined metric that measures how much of the design specification, as enumerated by the features in a test plan, has been exercised
- ❖ It is based on the design specification and therefore is independent of the actual RTL code of a design

***** Achieving 100% code coverage in a design does not imply that 100% functional coverage is achieved *****

Example:

```
module d_flip_flop(input logic d, clk, rst, output logic q);  
    always @(posedge clk or negedge rst)  
    begin  
        q <= d;  
    end  
endmodule
```

100% code coverage will be achieved for this code.

The actual reset 'rst' functionality is missing within the begin ... end block of code, and hence 100% functional coverage will not be achieved.

Imagine what would happen if this buggy D flip-flop was actually manufactured as part of a design !!

Functional Coverage

Functional coverage is the measure of how much of the design specification has been exercised by the features in the verification plan. It can be used to gauge whether interesting scenarios, corner cases, or other applicable design conditions have been observed, validated, and tested.

It is fully specified by the user and hence takes up much more time in the beginning to setup and write the model. It captures the design intent and hence requires a more structured approach to verification. These shortcomings are overcome by providing language constructs for easy specification of functional coverage models. The SystemVerilog functional coverage constructs enable:

- ❖ Coverage of variables and expressions, as well as cross coverage between them
- ❖ Automatic as well as user-defined coverage bins
- ❖ Bins with associated sets of values, transitions, or cross products
- ❖ Filtering conditions
- ❖ Procedural activation and query coverage
- ❖ Optional directives to control and regulate coverage

SystemVerilog Coverage

SystemVerilog functional coverage is achieved by doing the following:

- ❖ Defining a coverage model (***covergroup***)
- ❖ Defining the coverage points for the model (***coverpoint***)
- ❖ Optionally specifying coverage point bins for tracking hits and misses (***bins***)
- ❖ Defining cross-coverage points (***cross-coverage***)
- ❖ Instantiating the coverage model(s) in the Testbench

Defining the Coverage Model - covergroup

Definition: The covergroup construct is a user-defined type that encapsulates the specification of a coverage model.

Syntax:

```
covergroup covergroup_identifier [(argument_list)] [coverage_event];  
    {coverage_spec_or_option;}  
endgroup[: covergroup_identifier]  
  
where,  
    coverage_spec ::= cover_point | cross_cover          --- Explained Later  
    coverage_option ::= (option.member_identifier = expression) --- Explained Later
```

Features and Characteristics:

- The covergroup construct can be defined in a package, module, program, interface or class
- A class can have more than one covergroup
- Once a covergroup definition is written, multiple instances of it can be created wherever necessary. A covergroup must be instantiated for it to collect data
- Like a class, once a covergroup is defined, its instance can be created using the new() operator
- It is triggered by a clocking event or a procedural sample() command or by execution of a named block, task or function
- A covergroup can contain one or more coverage points (explained later). A coverage point can cover a variable or an expression

Coverage sampling event

Example:

```
covergroup cgexmpl @(posedge sysclk); // cgexmpl covergroup definition  
:  
endgroup: cgexmpl  
  
cgexmpl cgexmpl_inst1 = new(); // Covergroup instance  
cgexmpl cgexmpl_inst2 = new(); // Another covergroup instance
```


Defining Coverage Points - coverpoint

Definition: A coverage point specifies an integral expression that is to be covered. Each coverage point includes a set of bins associated with the sampled values or value transitions of the covered expression (explained later).

Syntax:

```
[[data_type_or_implicit] cover_point_identifier :] coverpoint expression [iff (guard_expression)] [bins_or_empty];
```

Features and Characteristics:

- One or more coverage points are contained in a covergroup
- A coverage point creates a hierarchical scope and can be optionally labeled
- Each coverage point includes a set of bins associated with the sampled values or value transitions of the covered expression (explained later)
- Evaluation of the coverage point expression (and its enabling 'iff' filter condition, if present) takes place when the covergroup is sampled
- The guard expression within the 'iff' construct specifies an optional condition that disables coverage for that coverpoint. If this expression evaluates to false at a sampling point, the coverage point is ignored

Example 1:

```
enum{white, green, blue, red, black} color;  
  
covergroup clrgp @(posedge sysclk);  
  c: coverpoint color iff(!reset);  
endgroup
```

filter condition - If and only if reset is de-asserted, then the bins for coverpoint color will be created. In other words coverage point color is covered only if the value reset is false.

Defining Coverage Points – coverpoint Continued...

Example 2:

```
module multiple_cover_group_ex(logic clk, reset);
class c1;
    int x;
endclass

class c2;
    c1 c1_obj;
    logic[3:0] a;
    integer b;
    logic[63:0] c;
    bit sel;

    covergroup cvr0 @(c1_obj.x);      // embedded covergroup cvr0 within class c2
        coverpoint a;
    endgroup

    covergroup cvr1 @(posedge clk);  // embedded covergroup cvr1 within class c2
        label: coverpoint b;
    endgroup

    covergroup cvr2 @sel;            // embedded covergroup cvr2 within class c2
        cvr2_cp: coverpoint c[63:56] iff(!reset); // creates coverpoint "cvr2_cp" covering the high order 8-bits of 'c'
    endgroup

    function new();
        c1_obj = new;
        cvr0 = new; // The embedded coverage group --
        cvr1 = new; // -- can be explicitly instantiated --
        cvr2 = new; // -- in the new method
    endfunction
endclass
endmodule
```

coverpoint – Bins

- A coverage point expression or variable is sampled and its sampled values or its value transitions are accounted for in a set of bins
- A coverage point bin associates a name and a count with a set of values or a sequence of value transitions
- Bins associated with a set of values are referred to as *state bins*
 - The count is incremented every time the coverage point matches one of the values in the set
 - State bins are used where value coverage is critical, typically in the datapath of a design
- Bins associated with a sequence of value transitions are referred to as *transition bins*
 - The count is incremented every time the coverage point matches the entire sequence of value transitions
 - Transition bins are used where transition coverage is critical, typically in controllers in a design
- Bins are used as a basic unit of measurement for functional coverage. An analysis tool reads the bin database at the end of simulation and generates a report with coverage for each part of the design and also for the total coverage
- Bins can be explicitly defined by the user or automatically created by the SystemVerilog compiler
- One can impose a limit on the number of automatic bins created by specifying a value for the *auto_bin_max* option (explained later)

coverpoint – Bins Continued...

- The **bins** construct allows creating a separate bin for each value in the given range list or a single bin for the entire range of values
 - To create a separate bin for each value, i.e., an array of bins, the square brackets [] should follow the bin name
 - To create a fixed number of bins for a set of values, a single positive integral expression is specified inside the square brackets.
 - If a fixed number of bins is specified and that number is smaller than the specified number of values, then the possible bin values are uniformly distributed among the specified bins. If the number of values is not divisible by the number of bins, then the last bin will include the remaining items
 - If the number of bins exceeds the number of values, then some of the bins will be empty

Syntax:

bins name [[]] = {value_set} [iff(expression)];

Defining state bins

bins name [[]] = (transitions) [iff(expression)];

Defining transition bins

Example of a value set -- {1, 5, [7:14], 25}

Example of transitions -- (4=>5, 6=>7, 8=>9=>10, [6:9]==>[1:2]);

Example:

bins separate[] = {100, 200, 300};
bins fixed[4] = {[1:10], 11, 12, 0};

3 bins are created – separate[100], separate[200], and separate[300]

4 bins are created and the 13 values are distributed as follows:
fixed[1] -- <1,2,3>
fixed[2] -- <4,5,6>
fixed[3] -- <7,8,9>
fixed[4] -- <10,11,12,0>

- The **default** specification defines a bin that is associated with none of the defined value bins. The default bin catches the values of the coverage points that do not lie within any of the defined bins

coverpoint - Examples of State Bins

Example 1:

```
bit[7:0] v_a, v_b;

covergroup cg1 @(posedge sysclk);
  coverpoint v_a + v_b
  {
    bins a = {[64:127],200}; // user-defined bins
    bins b[] = {0,10,100,220} iff(!reset);
    bins bad = default; // all remaining values
  }
endgroup
```

Scalar bin

Vector bin

'a' creates one bin

- ❖ All values in the range 64 to 127, i.e., [64:127] and the value 200 are covered

'b' creates one bin per value: b[0], b[10], b[100], b[220]

- ❖ Only covered when reset == 0

'bad' catches all other cases (in one bin)

- ❖ All values in the ranges [1:9], [11:63], [128:199], [201:219], [221:255] are covered

This coverpoint generates a set of mod5 bin values between 0 and 255 that are divisible by 5. The range is 0 to 255 because val is an 8-bit variable.

Example 2:

```
bit[7:0] val;

covergroup cg2 @(posedge sysclk);
  cname: coverpoint val
  {
    bins mod5[] = cname with (item % 5 == 0);
  }
endgroup
```

Example 3:

```
enum{white, green, blue, red, black} color;
bit[5:0] pixel;

covergroup cg3 @(negedge derivedclk);
  coverpoint color;
  coverpoint pixel;
endgroup
```

sampling event

5 bins for color created automatically

64 bins for pixel created automatically

Automatic Bin Creation

coverpoint - Examples of Transition Bins

Example 1:

```
bit[7:0] v_a;

covergroup cg1 @evnt1;
  coverpoint v_a
  {
    bins u[] = (1,2=>2,3);
    bins v = (4=>5=>6, [7:9]==>[1:2]);
    bins w = (6=>7), ([2:4],10=>16,27);
  }
endgroup
```

1->2, 1->3, 2->2, 2->3

4->5->6,
7->1, 8->1, 9->1
7->2, 8->2, 9->2

6->7,
2->16, 3->16, 4->16, 10->16
2->27, 3->27, 4->27, 10->27

'u' creates one bin per transition: 4 bins

- ❖ 1->2, 1->3, 2->2, 2->3 are the transitions covered

'v' creates one bin for all 7 transitions

- ❖ 4->5->6, 7->1, 8->1, 9->1, 7->2, 8->2, 9->2 are the transitions covered

'w' creates one bin for all 9 transitions

- ❖ 6->7, 2->16, 3->16, 4->16, 10->16, 2->27, 3->27, 4->27, 10->27 are the transitions covered

1->2, 1->9, 4->2, 4->9

'trans_set' creates one bin per transition: 4 bins

- ❖ 1->2, 1->9, 4->2, 4->9 are the transitions covered

Example 2:

```
logic[0:6] mclaren;

covergroup cg2 @(posedge clk);
  coverpoint mclaren
  {
    // Single value transition below
    bins trans_78 = (7=>8);

    // Sequence of transitions below
    bins trans_12569 = (1=>2=>5=>6=>9);

    // Set of transitions below
    bins trans_set[] = (1,4=>2,9);

    // Consecutive repetitions of transitions below.
    // 4 is repeated 6 times. This is the same as
    // 4=>4=>4=>4=>4=>4
    bins trans_4_6_times = (4[*6]);

    // Range of repetitions below. This is the same as
    // 2=>2=>2, 2=>2=>2=>2, 2=>2=>2=>2=>2
    bins trans_2_range_3_to_5 = (2[*3:5]);
  }
endgroup
```

coverpoint - Wildcard Bins

The bins construct can be qualified with the **wildcard** keyword. Then, the wildcard character, '?', can be used. A wildcard bin definition causes 'x', 'z', or '?' to be treated as a '0' or '1' value.

Example:

```
bit[3:0] data;  
  
covergroup cg @(negedge clk);  
  coverpoint data  
  {  
    wildcard bins p = {4'b11??};  
    wildcard bins s[] = (4'b000? => 4'b001?);  
    wildcard bins v = (2'b0x => 2'b1x);  
    wildcard bins even = {3'b???0};  
  }  
endgroup
```

- Bin 'p' (4'b11??) maps to a box containing: 4'b1100, 4'b1101, 4'b1110, 4'b1111
- Bin 's' (4'b000? => 4'b001?) maps to a box containing: 0000->0010, 0000->0011, 0001->0010, 0001->0011
- Bin 'v' (2'b0x => 2'b1x) maps to a box containing: 00->10, 00->11, 01->10, 01->11
- Bin 'even' (3'b???0) maps to a box containing: 3'b000, 3'b010, 3'b100, 3'b110

'p' creates one bin for the 4 values

- ❖ 4'b1100, 4'b1101, 4'b1110, 4'b1111 are the values covered

's' creates one bin for each of the transitions

- ❖ 0000->0010, 0000->0011, 0001->0010, 0001->0011 are the transitions covered

'v' creates one bin for all 4 transitions

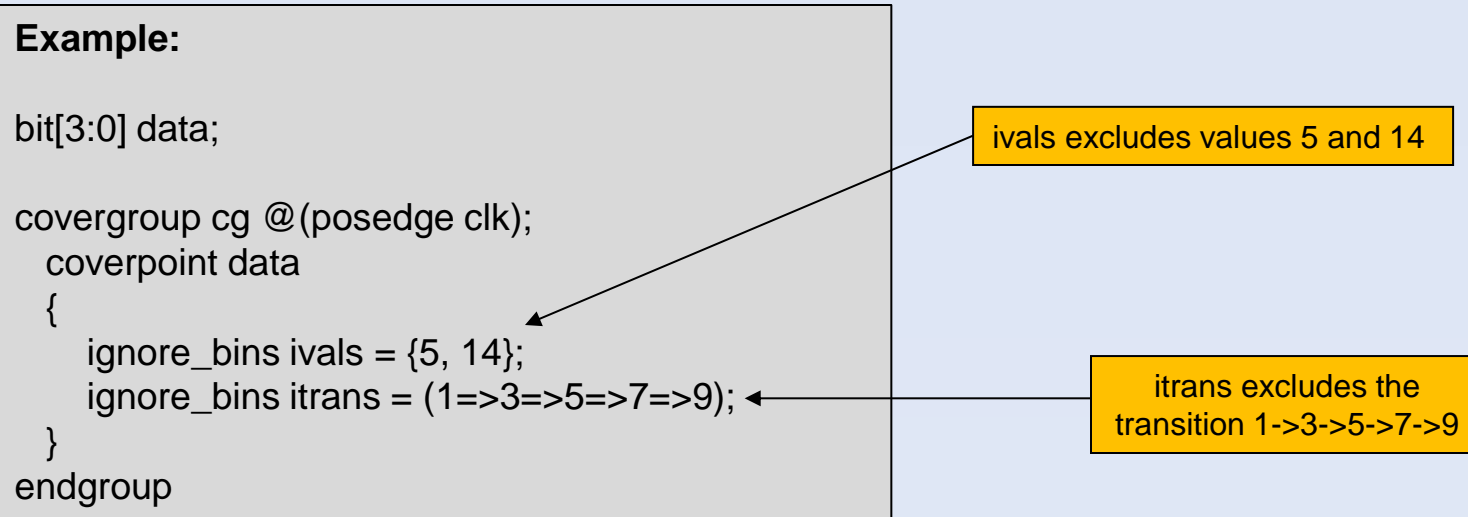
- ❖ 00->10, 00->11, 01->10, 01->11 are the transitions covered

'even' creates one bin for the 4 values

- ❖ 3'b000, 3'b010, 3'b100, 3'b110 are the values covered

coverpoint - Ignore Bins

A set of values or transitions associated with a coverage point can be explicitly excluded from coverage by specifying them as **ignore_bins**. Certain sections of the stimuli space might not be relevant and hence can be specified using ignore_bins. All values or transitions associated with ignore_bins are excluded from coverage.



coverpoint - Illegal Bins

A set of values or transitions associated with a coverage point can be marked as illegal by specifying them as **illegal_bins**. All values or transitions associated with illegal bins are excluded from coverage. When a value matches that specified in `illegal_bins`, a runtime error is generated. Illegal bins take precedence over any other bins, i.e., they will result in a run-time error even if the value or transition is included in another bin.

Example:

```
bit[3:0] data;

covergroup cg @(posedge clk);
  coverpoint data
  {
    illegal_bins bad_vals = {1, 2, 4, 8};
    illegal_bins bad_trans = (2=>1=>9, 5=>3);
  }
endgroup
```

coverpoint – Bins Classification – Quick Reference

Bin Type	Use Case	Example
State Bins	Bins associated with a set of values are referred to as state bins . State bins are used where value coverage is critical, typically in the datapath of a design.	<pre> bit[7:0] v_a, v_b; covergroup cg1 @(posedge sysclk); coverpoint v_a + v_b { bins a = {[64:127],200}; // user-defined bins bins b[] = {0,10,100,220} iff(!reset); bins bad = default; // all remaining values } endgroup </pre>
Transition Bins	Bins associated with a sequence of value transitions are referred to as transition bins . Transition bins are used where transition coverage is critical, typically in controllers in a design.	<pre> bit[7:0] v_a; covergroup cg1 @evnt1; coverpoint v_a { bins u[] = (1,2=>2,3); bins v = (4=>5=>6, [7:9]==>[1:2]); } endgroup </pre>
Wildcard Bins	The bins construct can be qualified with the wildcard keyword. Then, the wildcard character, '?' can be used. A wildcard bin definition causes 'x', 'z', or '?' to be treated as a '0' or '1' value.	<pre> bit[3:0] data; covergroup cg @(negedge clk); coverpoint data { wildcard bins p = {4'b11??}; wildcard bins v = (2'b0x ==> 2'b1x); } endgroup </pre>
Ignore Bins	A set of values or transitions associated with a coverage point can be explicitly excluded from coverage by specifying them as ignore bins . All values or transitions associated with ignore bins are excluded from coverage.	<pre> bit[3:0] data; covergroup cg @(posedge clk); coverpoint data { ignore_bins ival = {5, 14}; ignore_bins itrans = (1==>3==>5==>7==>9); } endgroup </pre>
Illegal Bins	A set of values or transitions associated with a coverage point can be marked as illegal by specifying them as illegal bins . All values or transitions associated with illegal bins are excluded from coverage.	<pre> bit[3:0] data; covergroup cg @(posedge clk); coverpoint data { illegal_bins bad_vals = {1, 2, 4, 8}; illegal_bins bad_trans = (2==>1==>9, 5==>3); } endgroup </pre>

Cross Coverage

A coverage group can specify cross coverage between two or more coverage points or variables. Any combination of more than two variables or previously declared coverage points is allowed. Cross coverage is specified using the **cross** construct.

Syntax:

```
[cross_identifier :] cross list_of_coverpoint_identifier(s)_or_variable_identifier(s) [iff (expression)] bins_selection_or_coverage_option;
```

Features and Characteristics:

- A cross coverage bin associates a name and a count with a set of cross products. The count of the bin is incremented every time any of the cross products match
- Cross coverage of a set of N coverage points is defined as the coverage of all combinations of all bins associated with the N coverage points, i.e., the cartesian product of the N sets of coverage point bins
- User-defined cross bins and automatically generated bins can co-exist in the same cross
- No cross coverage bins shall be created for coverpoint bins that are specified as default, ignored, or illegal bins
- Attempting to cross items from different coverage groups shall result in a compiler error
- Expressions cannot be used directly in a cross - a coverage point must be explicitly defined first

Cross Coverage Continued...

Example 1:

```
enum{white, blue, black} color;  
bit[4:0] pixel1, pixel2;
```

```
covergroup cg1 @(posedge sysclk);  
  coverpoint color;  
  coverpoint pixel1;  
  coverpoint pixel2;
```

```
  colXpix: cross color, pixel1;  
  pixel1Xpixel2: cross pixel1, pixel2;  
endgroup
```

3 bins for color

32 bins for pixel1

32 bins for pixel2

96 cross product bins

1024 cross product bins

Example 2:

```
enum{white, green, blue, red, black} color;  
bit [3:0] pixel_address, pixel_offset, pixel_hue;
```

```
covergroup cg2 @(posedge sysclk);  
  Hue: coverpoint pixel_hue;  
  Offset: coverpoint pixel_offset;
```

```
  CxA: cross color, pixel_address; // cross between 2 variables (implicitly declared coverpoints)
```

```
  HxO: cross Hue, Offset; // cross between 2 coverpoints
```

```
  all: cross color, Hue, Offset; // cross between 1 variable and 2 coverpoints
```

```
endgroup
```

Coverage Options

Options control the behavior of the **covergroup**, **coverpoint**, and **cross**. There are two types of options:

(1) Instance-Specific Coverage Options		
Option Name	Default Value	Description
weight = <i>number</i>	1	If set at the covergroup level, it specifies the weight of this covergroup instance for computing the overall instance coverage of the simulation. If set at the coverpoint (or cross) level, it specifies the weight of a coverpoint (or cross) for computing the instance coverage of the enclosing covergroup. The specified weight should be a non-negative integral value.
name = <i>string</i>	Unique Name	Specifies a name for the covergroup instance. If unspecified, a unique name for each instance is automatically generated by the tool.
comment = <i>string</i>	""	This is a comment that appears with a covergroup instance or with a coverpoint or cross of the covergroup instance. The comment is saved in the coverage database and included in the coverage report.
at_least = <i>number</i>	1	Minimum number of hits for each bin. A bin with a hit count that is less than number is not considered covered.
detect_overlap = <i>boolean</i>	0	When true, a warning is issued if there is an overlap between the range list (or transition list) of two bins of a coverpoint.
auto_bin_max = <i>number</i>	64	Maximum number of automatically created bins when no bins are explicitly defined for a coverpoint.
per_instance = <i>boolean</i>	0	Each instance contributes to the overall coverage information for the covergroup type. When true, coverage information for this covergroup instance shall be saved in the coverage database and included in the coverage report. When false, implementations are not required to save instance-specific information.

Coverage Options Continued...

(2) Coverage Group Type Options		
Option Name	Default Value	Description
<code>weight = constant_number</code>	1	If set at the covergroup syntactic level, it specifies the weight of this covergroup for computing the overall cumulative (or type) coverage of the saved database. If set at the coverpoint (or cross) syntactic level, it specifies the weight of a coverpoint (or cross) for computing the cumulative (or type) coverage of the enclosing covergroup. The specified weight shall be a non-negative integral value.
<code>comment = string_literal</code>	""	This is a comment that appears with the covergroup type or with a coverpoint or cross of the covergroup type. The comment is saved in the coverage database and included in the coverage report.
<code>strobe = boolean</code>	0	When true, all samples happen at the end of the time slot, like the \$strobe system task.
<code>merge_instances = boolean</code>	0	When true, cumulative (or type) coverage is computed by merging instances together as the union of coverage of all instances. When false, type coverage is computed as the weighted average of instances.

Coverage Options Continued...

Example:

```
covergroup cg (int wt, string inst_comment) @(negedge sysclk) ;

    // Track coverage information for each instance of cg in addition
    // to the cumulative coverage information for covergroup type cg
    option.per_instance = 1;

    // Comment for each instance of this covergroup
    option.comment = inst_comment;

    a: coverpoint var1
    {
        // Create 100 automatic bins for coverpoint 'a' of each instance of cg
        option.auto_bin_max = 100;
    }

    b: coverpoint var2
    {
        // This coverpoint 'b' contributes 'wt' times more to the coverage of an
        // instance of cg than coverpoints 'a' and 'c'
        option.weight = wt;
    }

    c: cross var1, var2;

endgroup
```

Coverage Control – Pre-defined Methods

These methods can be invoked procedurally at any point of time and are called after the covergroup instance name with a period between the instance name and the method name.

- `void sample():` Triggers sampling of the covergroup
- `real get_coverage():` Calculates the type coverage number (1..100)
- `real get_inst_coverage(ref int, ref int):` Calculates the coverage number (1..100)
- `void set_inst_name(string):` Sets the instance name to the given string
- `void start():` Starts collecting coverage information
- `void stop():` Stops collecting coverage information

Example 1:

```
module functional_coverage1();
logic[3:0] val1, val2;
assign val2 = val1*2;
covergroup value_cov;
v1: coverpoint val1
{
    // Create 6 automatic bins for coverpoint 'v1' of each instance of value_cov
    option.auto_bin_max = 6;
}
v2: coverpoint val2
{
    // Create 8 automatic bins for coverpoint 'v2' of each instance of value_cov
    option.auto_bin_max = 8;
}
endgroup
value_cov vcov = new;
initial
begin
    // Setting at least 2 hits for each bin
    vcov.v1.option.at_least = 2;
    vcov.v2.option.at_least = 2;
    // Start collecting coverage
    vcov.start();
    // Set the instance name
    vcov.set_inst_name("WEARABLE SoC");
    $monitor("val1 = %0d, val2 = %0d",val1, val2);
    repeat(25)
    begin
        val1 = $urandom_range(2,7);
        // Sample the covergroup
        vcov.sample();
        #15;
    end
    // Stop collecting coverage
    vcov.stop();
    // Display the coverage details
    $display("Instance coverage is %e", vcov.get_coverage());
end
endmodule
```


Coverage Control – Pre-defined Methods Continued...

Example 2:

```
program functional_coverage2;

class Xaction;
    rand bit [5:0] data; // Sixty-four data numbers
    rand bit [2:0] addr; // Eight addr numbers
endclass

Xaction trans;

covergroup cgdata; // Covergroup for data
    coverpoint trans.data; // Measure data coverage
endgroup

covergroup cgaddr; // Covergroup for addr
    coverpoint trans.addr; // Measure addr coverage
endgroup

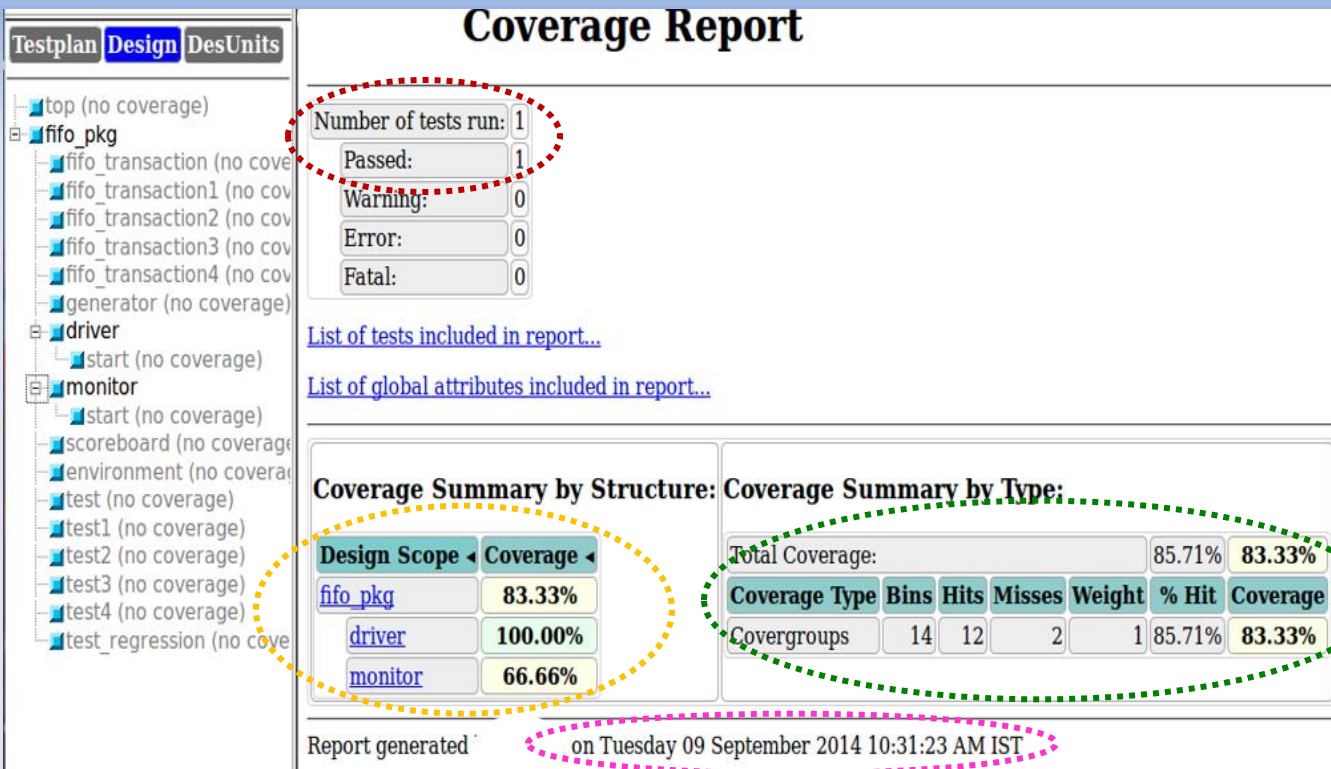
initial
begin
    cgdata cd; // Instantiate coverage data group
    cgaddr ca; // Instantiate coverage addr group
    cd = new();
    ca = new();
    repeat(10) // Repeat for a few cycles
    begin
        trans = new();
        if (trans.randomize() == 1) // Create a Transaction
        begin
            XYZ <= trans.addr; // XYZ is assumed to be defined somewhere
            ABC <= trans.data; // ABC is assumed to be defined somewhere
        end
        cd.sample(); // Trigger the covergroup and collect coverage
        ca.sample(); // Trigger the covergroup and collect coverage
    end
end

endprogram
```

Coverage Control – Pre-defined System Tasks

SystemVerilog provides the following system tasks to help manage file i/o during coverage data collection:

- **\$set_coverage_db_name(filename)**: Sets the file name of the coverage database into which coverage information is saved at the end of a simulation run
- **\$load_coverage_db(filename)**: Loads the cumulative coverage information for all coverage group types from the given file name



Sample Coverage Report

Covergroups						
Name	Coverage	Goal	% of Goal	Status	Merge_instances	Get_inst_coverage
/fifo_pkg/monitor						
TYPE mon_cg	66.6%	100	66.6%	<div><div></div></div>	auto(1)	
CVP mon_cg::E...	50.0%	100	50.0%	<div><div></div></div>		
bin emp[0]	250	1	100.0%	<div><div></div></div>		
bin emp[1]	0	1	0.0%	<div><div></div></div>		
CVP mon_cg::F...	50.0%	100	50.0%	<div><div></div></div>		
bin full[0]	250	1	100.0%	<div><div></div></div>		
bin full[1]	0	1	0.0%	<div><div></div></div>		
CVP mon_cg::D...	100.0%	100	100.0%	<div><div></div></div>		
bin dout	250	1	100.0%	<div><div></div></div>		
/fifo_pkg/driver						
TYPE drv_cg	100.0%	100	100.0%	<div><div></div></div>	auto(1)	
CVP drv_cg::W...	100.0%	100	100.0%	<div><div></div></div>		
bin wrt[0]	128	1	100.0%	<div><div></div></div>		
bin wrt[1]	122	1	100.0%	<div><div></div></div>		
CVP drv_cg::RE...	100.0%	100	100.0%	<div><div></div></div>		
bin rd[0]	128	1	100.0%	<div><div></div></div>		
bin rd[1]	122	1	100.0%	<div><div></div></div>		
CVP drv_cg::DA...	100.0%	100	100.0%	<div><div></div></div>		
bin data	250	1	100.0%	<div><div></div></div>		
CROSS drv_cg::...	100.0%	100	100.0%	<div><div></div></div>		
bin <wrt[0],r...	66	1	100.0%	<div><div></div></div>		
bin <wrt[1],r...	62	1	100.0%	<div><div></div></div>		
bin <wrt[0],r...	62	1	100.0%	<div><div></div></div>		
bin <wrt[1],r...	60	1	100.0%	<div><div></div></div>		

187

9. Verification Plan & SV Testbench Architecture

Verification Plan

The verification plan is a specification for the verification effort. It is used to define what is first-time success and how a design is verified. It is prepared by reviewing the design specification.

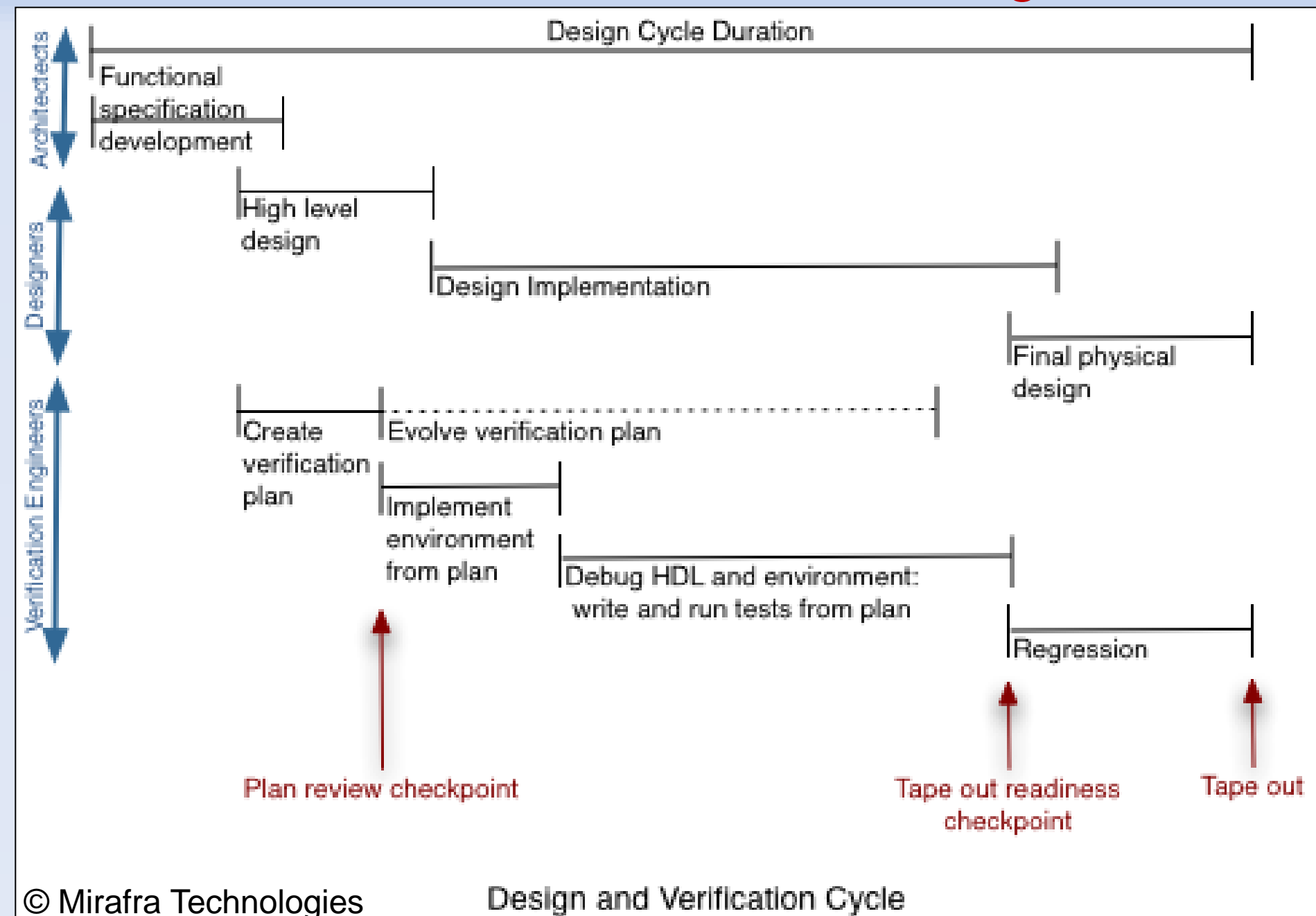
The verification plan document should contain the following features:

- Scope:
 - Information about the extent of the design to be verified and target audience
- Purpose:
 - Bring out the necessity of this plan - finding bugs in the specific design and verifying that the implementation exactly matches its specification
- Resource, Budget Allocation and Tools Used:
 - The allocation of verification team members, i.e., the modules that they are working on, as well as dependencies on their work/for their work is to be captured here
 - The allocation of test equipment, need for new equipment, cost of licenses for software tools/simulators used in simulation, funds for FPGA boards to be used in emulating the design needs to be captured in a requirements list
 - Typical tools used such as HDL simulators, formal verification software, assertion based tools and debuggers need to be listed here
- Risks and Dependencies:
 - HDL arrival dependency
 - Architecture specification closure – no unresolved issues should remain

Verification Plan Continued...

- Test bench Architecture Details:
 - The various components of the test bench architecture are described in detail and their interconnection architecture specified
- Feature Extraction and Test Strategy:
 - Features are implicitly or explicitly defined in the design specification
 - The features to be tested are extracted from the design specification. For each feature of the design, a directed test strategy is recognized, and a test sequence is identified
 - Black box, white box and/or grey box testing
 - For corner case testing, constrained random stimuli will be applied after all the directed testing is completed
 - The template shown in the next page is used for feature extraction and test plan strategy for any design
- Order of Features to be Verified:
 - For complex SoCs, there is a definitive order in the way the various features are to be verified. These are captured in the verification plan
 - Critical features are verified first before secondary features (non-critical to tapeout)
- Verification Criteria:
 - Expected and actual results for each design feature is documented
 - The feature definition, test strategy, test sequence, and verification criteria form the pillars of the functional verification plan
- Coverage Goals and Priorities
- Time Lines and Deliverables

Evolution of the Verification Plan during the Verification Cycle



Slide ref. source:

“Comprehensive Functional Verification” by Bruce Wile, John Goss and Wolfgang Roesner

ISBN-10: 0127518037

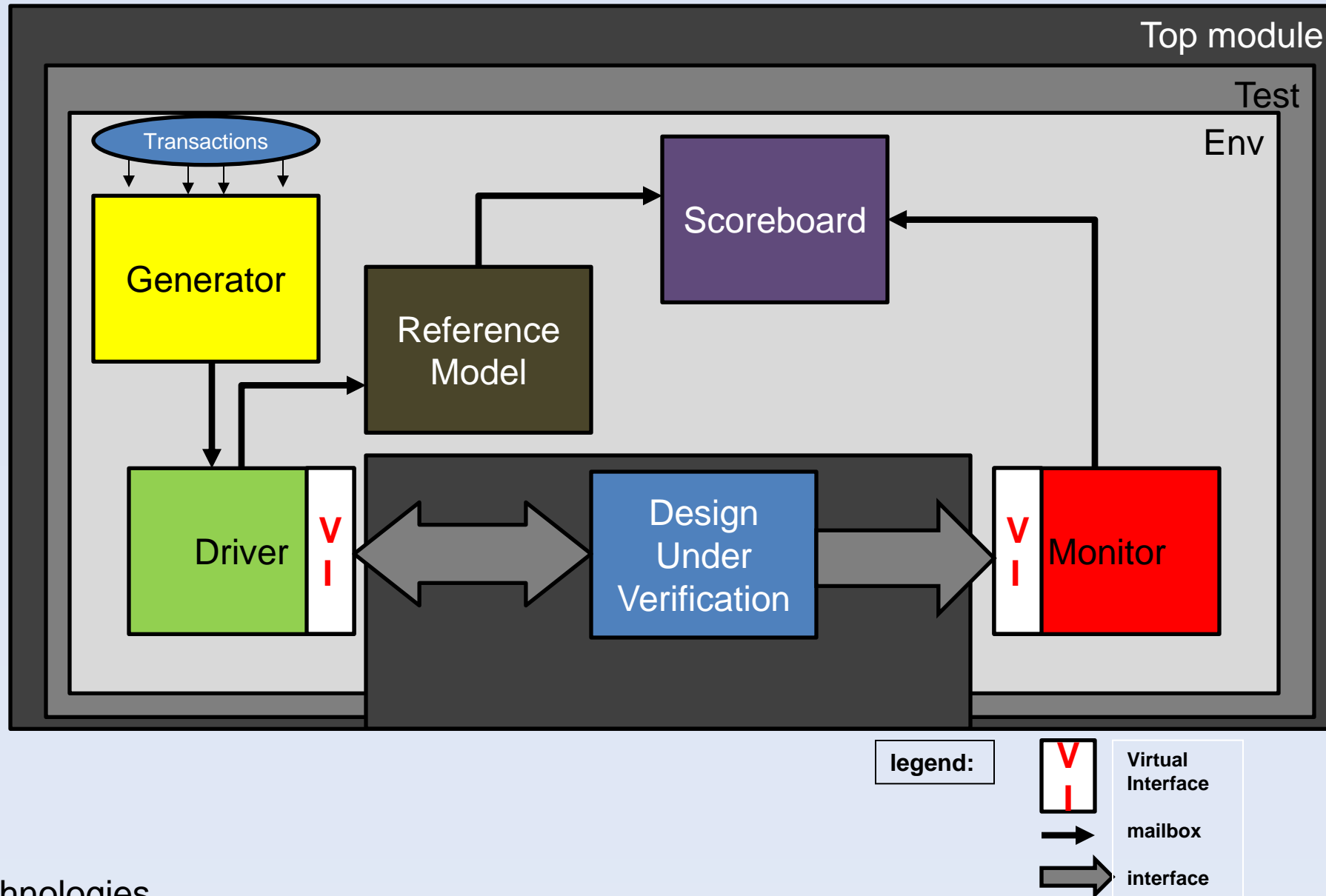
ISBN-13: 978-0127518039

Morgan Kaufmann Publications

Feature Extraction and Test Strategy Section of the Verification Plan – Template

Test #	Feature and Directed test strategy	Spec Section #	Priority	Test Sequence	Verification criteria
1					
2					
3					
4					
5					

Block Diagram of the SV Testbench Architecture



Building Blocks of the SV Testbench Architecture

- **Transactions**

- All the inputs and outputs of the Design Under Verification (DUV) are written inside the transaction class, excluding the clock and reset signals that are generated in the top module
- The transaction class can optionally have constraints for the inputs

- **Generator**

- This is a component of the testbench which generates constrained random stimuli (transactions) for the DUV
- The generator sends the generated stimuli to the driver through a mailbox
- Its code is written inside a class

- **Driver**

- This is a component which converts transactions into pin-level activity at the inputs of the DUV
- It receives the generated transactions from the generator through a mailbox and drives it to the DUV through a virtual interface according to the DUV protocol
- It also sends the received transactions from the generator to the reference model (described later) through another mailbox
- Its code is written inside a class

- **Monitor**

- This is a component which converts pin-level activity into transactions at the outputs of the DUV
- It collects the outputs of the DUV through a virtual interface and sends it to the scoreboard through a mailbox
- Its code is written inside a class

- **Reference Model**

- This is a 'golden' model that mimics the functionality of the DUV and generates reference results used for comparison against simulation results
- It is typically a non-synthesizable high-level model that is written as part of the architectural exploration of the design specification. Such models are used to analyze the performance of the system and make trade-offs between actual hardware component creation versus testing with its equivalent software functions
- It accepts the inputs from the driver through a mailbox and sends the outputs to the scoreboard through another mailbox
- Its code is written inside a class

Building Blocks of the SV Testbench Architecture Continued ...

- **Scoreboard**
 - This is the component that collects the transactions (expected results) from the reference model through a mailbox
 - It collects the transactions (actual results) from the monitor through another mailbox
 - It compares the expected transactions with the actual transactions and generates a report
 - Its code is written inside a class
- **Environment**
 - This is the component of the testbench which instantiates the generator, driver, monitor, reference model and scoreboard
 - Here, the various sub-components are built and properly connected
 - Its code is written inside a class
- **Test**
 - This is the component where different test cases are written and run
 - Here the environment is instantiated and built
 - Its code is written inside a class
- **Top**
 - This is the top most module of the SV testbench architecture where control signals like clock and reset are generated
 - Its code is written inside a module and the interface, the DUV and the test are instantiated in it
 - Here a particular test is executed and run from inside an initial block

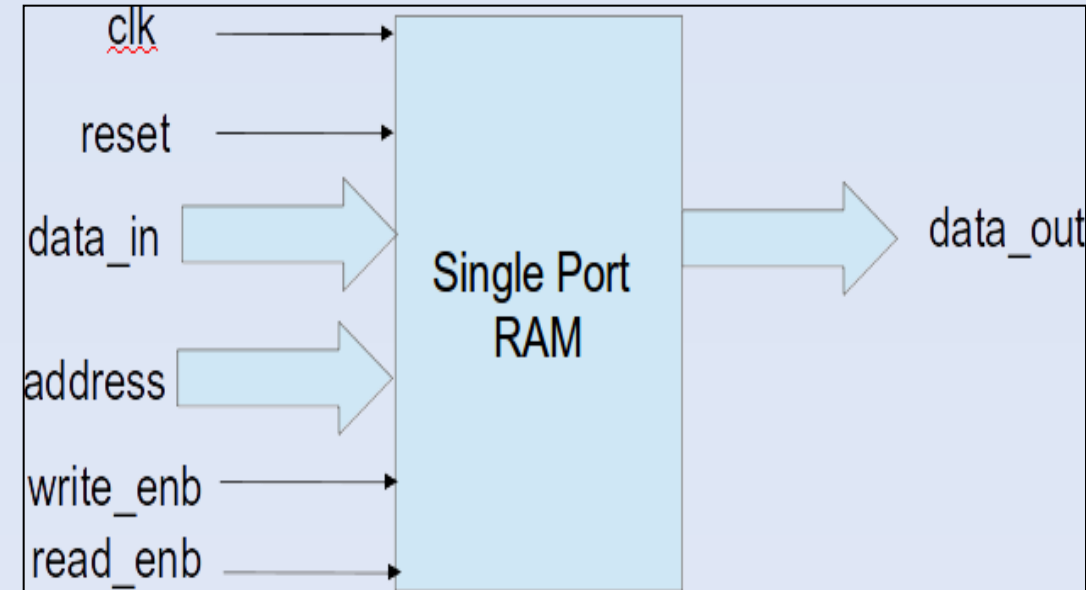
Extraction of Features from Design Specification Document - RAM

Read the design specification document for a RAM (Random Access Memory) by clicking here:

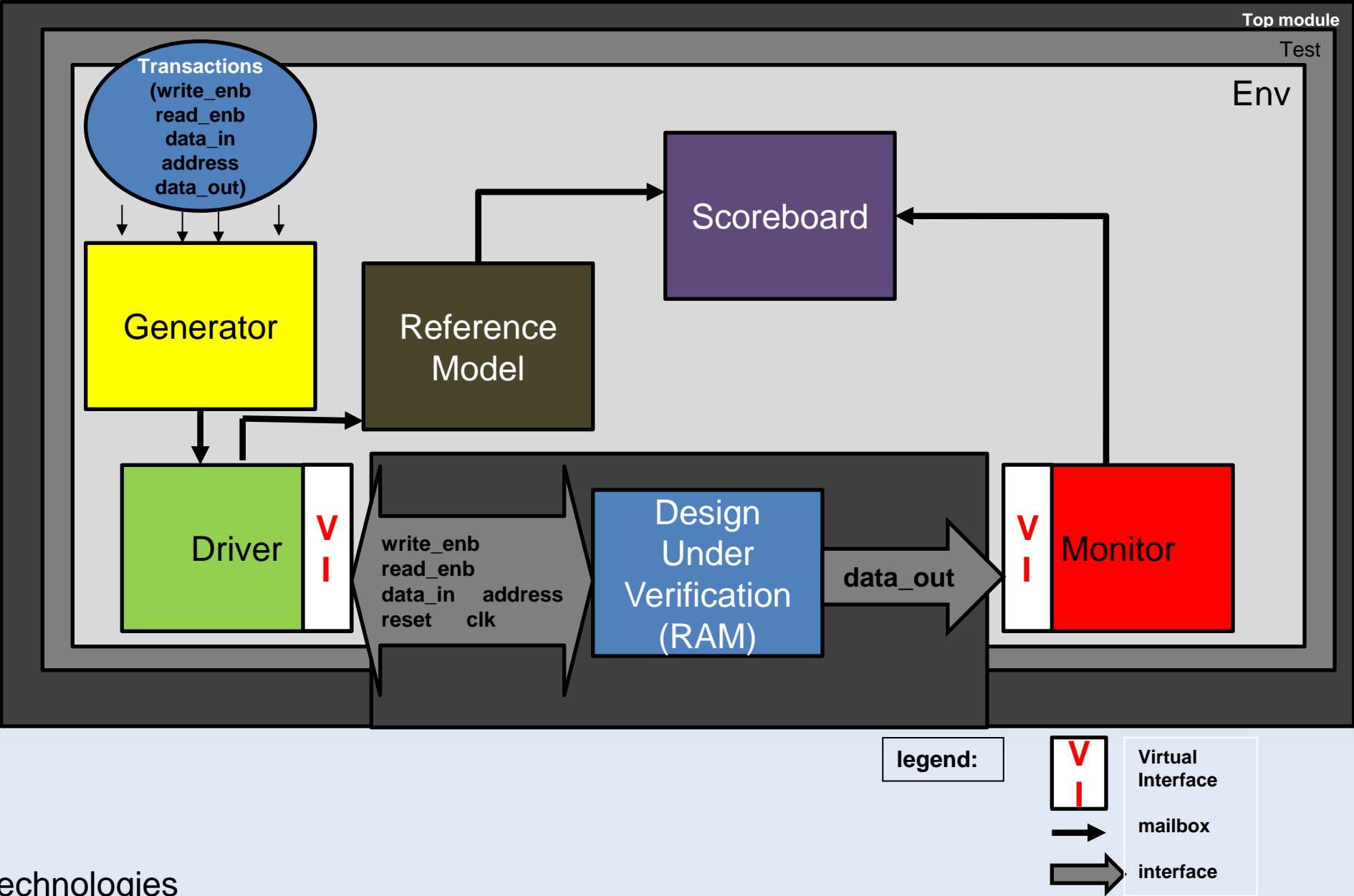


The key features of the RAM are as follows:

- The design is a single port RAM
- The RAM clock frequency is 50 MHz
- The depth of the RAM is 32 locations
- The data width is 8-bits
- Changes in the RAM happen at positive edge of clock
- Synchronous active high reset. On assertion, the data output becomes 'z'
- The write operation into the RAM is supported. For invalid address, the RAM does nothing
- The read operation from the RAM is supported. For invalid address, i.e., address value > that supported by the address width, the invalid address is truncated to fit in the address width and the data read from the RAM will be done using that truncated location
- The RAM does not support simultaneous (concurrent) read and write operations



Block Diagram of the SV Testbench Architecture - RAM

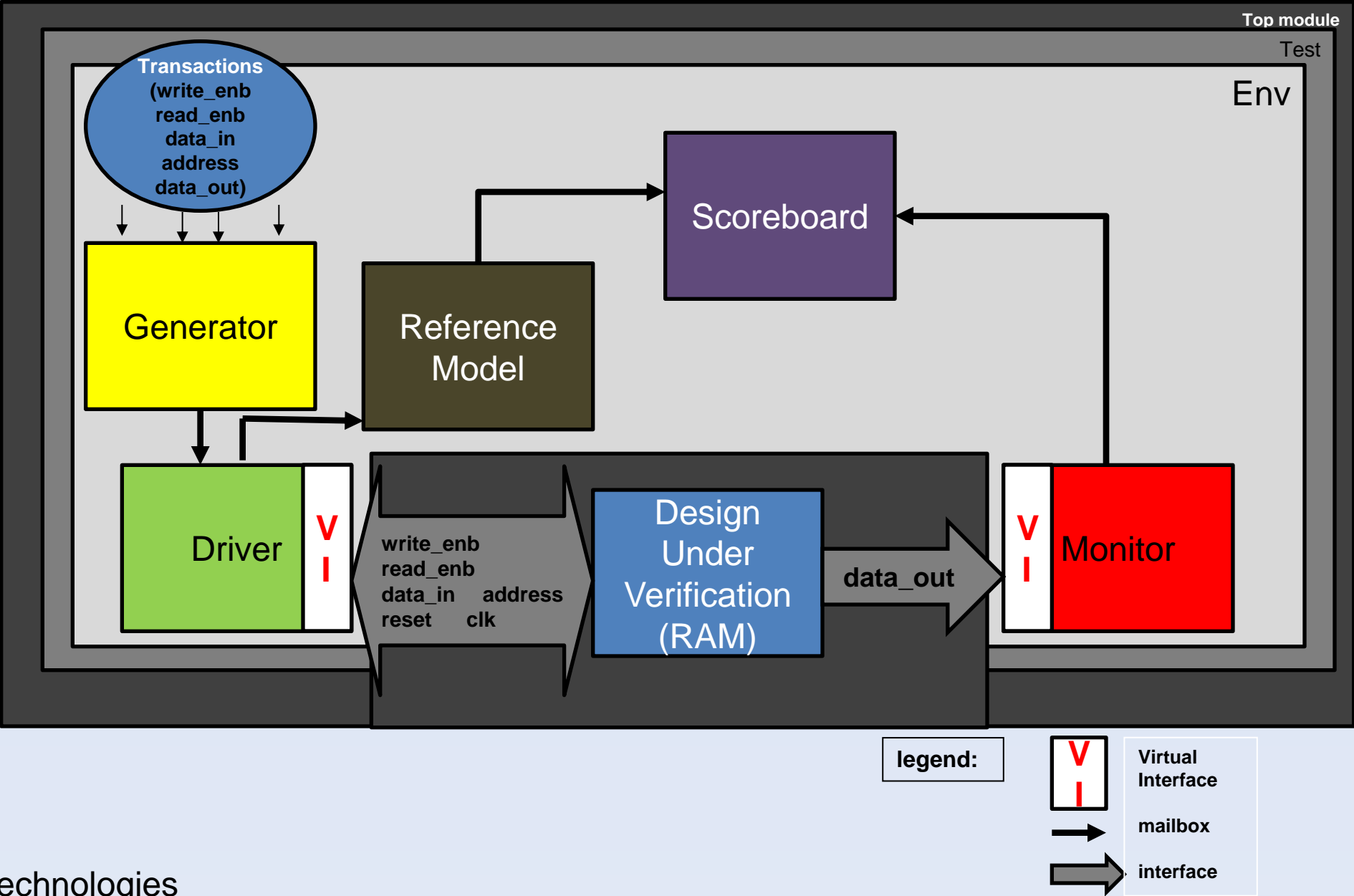


10. Modelling Testbench Blocks for the RAM Design

Disclaimer:

There are multiple ways and/or styles of coding SV testbenches. Shown here is one particular implementation for illustration purposes only. The verification engineers are encouraged to come up with other implementations of the testbench as a learning exercise.

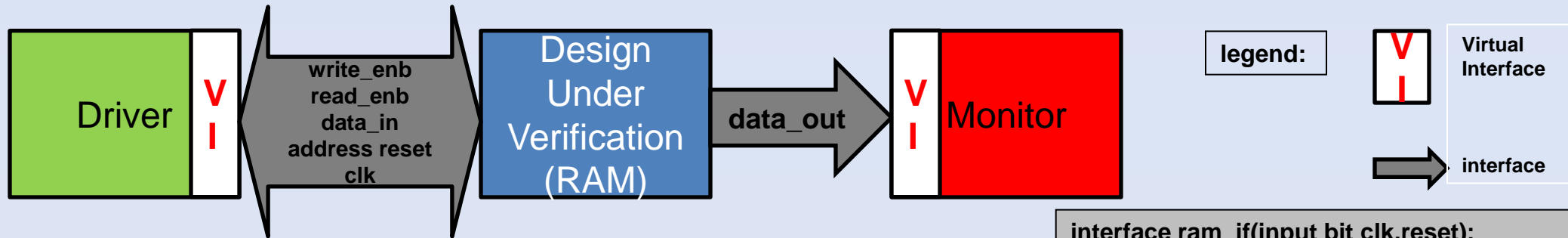
Block Diagram of the SV Testbench Architecture - RAM



defines.svh File

```
`timescale 1ns/100ps /* Time Unit (ns) / Precision(ps) */  
  
`define DATA_WIDTH 8  
`define DATA_DEPTH 32  
  
`define num_transactions 8  
  
function integer log2;  
    input integer n;  
    begin  
        log2 = 0;  
        while(2**log2 < n)  
            begin  
                log2=log2+1;  
            end  
        end  
endfunction  
  
parameter ADDR_WIDTH = log2(`DATA_DEPTH);
```


Interface



- The interface consists of all the input and output pins of the RAM
- The interface communicates with the driver and monitor in the testbench architecture
- The interface is static, whereas the testbench components driver and monitor which are written inside classes are dynamic. Therefore to connect a static interface with a dynamic testbench component, a Virtual Interface (VI) is used

clocking block for driver
which consists of the input
and output signals w.r.t
the testbench

clocking block for monitor
which consists of the input
signals w.r.t the testbench

```
interface ram_if(input bit clk,reset);  
  //Declaring signals with width  
  logic[7:0] data_in,data_out;  
  logic  write_enb,read_enb;  
  logic[4:0] address;  
  
  //Clocking block for driver  
  clocking drv_cb@(posedge clk);  
    //Specifying the values for input and output skews  
    default input #0 output #0;  
    //Declaring signals without widths, but specifying the direction  
    output write_enb,read_enb,data_in,address;  
    input reset;  
  endclocking  
  
  //Clocking block for monitor  
  clocking mon_cb@(posedge clk);  
    //Specifying the values for input and output skews  
    default input #0 output #0;  
    //Declaring signals without widths, but specifying the direction  
    input data_out;  
  endclocking
```

Interface Continued...

clocking block for reference model which consists of the input and output signals w.r.t the testbench

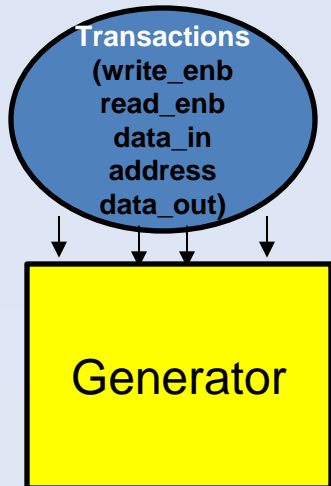
Modports for each clocking block driver, monitor and reference model which takes the clocking block as an input argument

```
//clocking block for reference model  
clocking ref_cb@(posedge clk);  
  //Specifying the values for input and output skews  
  default input #0 output #0;  
endclocking
```

```
//modports for driver, monitor and reference model  
modport DRV(clocking drv_cb);  
modport MON(clocking mon_cb);  
modport REF_SB(clocking ref_cb);  
endinterface
```

Transactions [ram_transaction.sv File]

- Transactions consists of the RAM input stimuli which will have to be randomized in the generator, as well as the RAM outputs that are non-randomized
- The transaction class also consists of constraints for the input stimuli



```
`include "defines.svh"

class ram_transaction;
//PROPERTIES
//INPUTS declared as rand variables
rand logic [`DATA_WIDTH-1:0] data_in;
rand logic write_enb,read_enb;
rand logic [ADDR_WIDTH-1:0] address;
//OUTPUTS declare as non-rand variables
logic [`DATA_WIDTH-1:0] data_out;
//CONSTRAINTS for write_enb and read_enb
constraint wr_rd_constraint {{write_enb,read_enb} inside
{[0:3]}};
constraint wr_not_equal_rd {{write_enb,read_enb}!=2'b11;}
//METHODS
//Copying objects
virtual function ram_transaction copy();
    copy = new();
    copy.data_in=this.data_in;
    copy.write_enb=this.write_enb;
    copy.read_enb=this.read_enb;
    copy.address=this.address;
    return copy;
endfunction
endclass
```

INPUTS are declared as rand variables

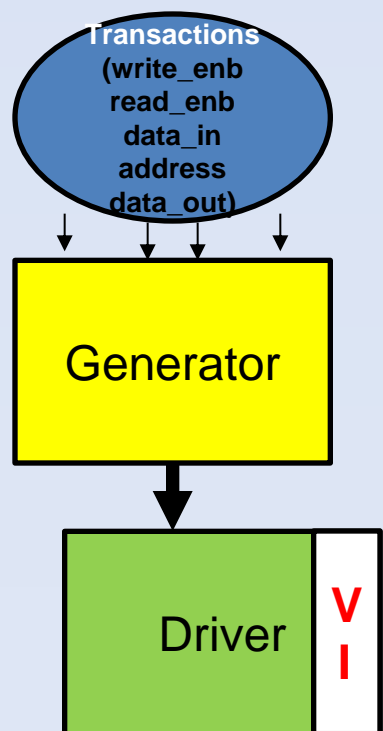
OUTPUTS are declared as non-rand variables

CONSTRAINTS for write_enb and read_enb

copy() function is used for running different test cases using the same transaction class object in the test class

NOTE: This will be discussed later.

Generator [ram_generator.sv File]



legend:

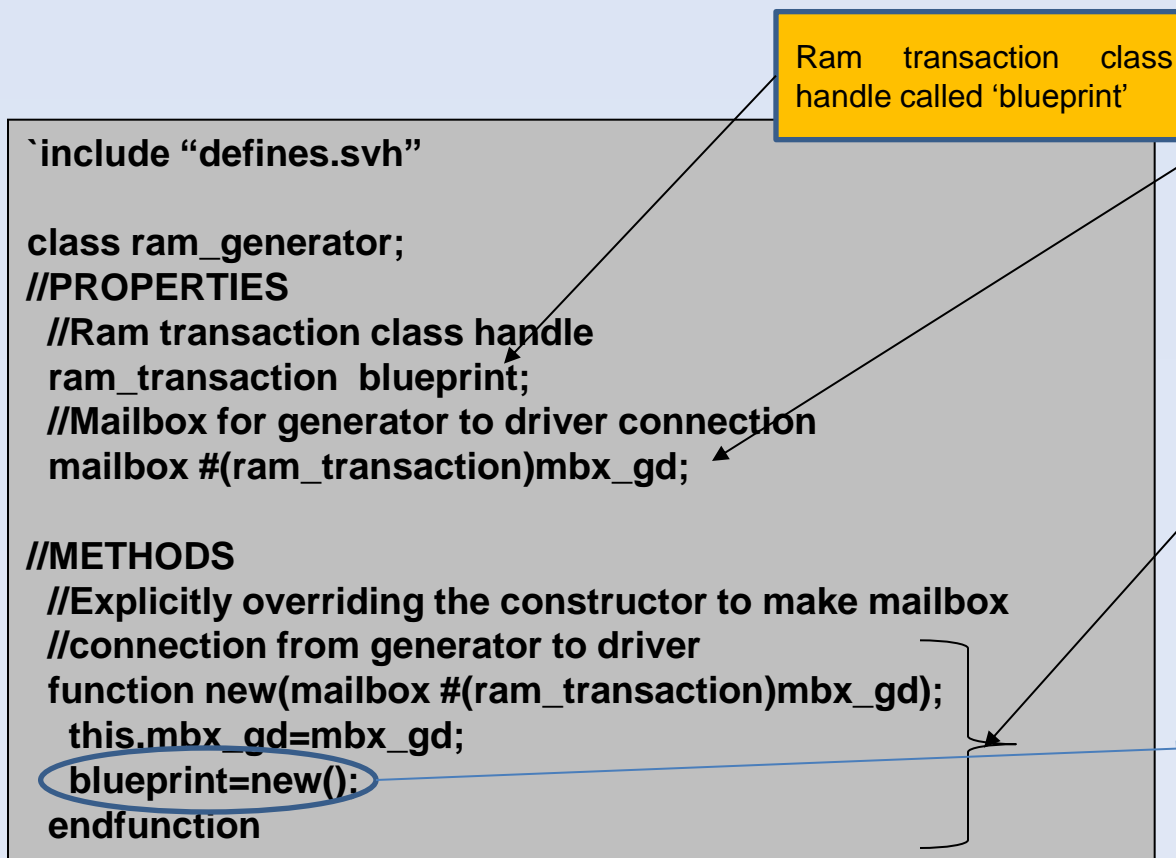


Virtual
Interface



mailbox

The generator consists of the transaction class handle, the mailbox handle which has to be connected to the driver and the task start() which randomizes the transactions and sends the randomized transactions to the driver through the mailbox



Mailbox for generator to driver connection

The mailbox is taken as an argument to the function new() in order to connect it to the driver

NOTE: The object for the transaction class handle is created in the function new(). This allows different test cases to be implemented. More about how different test cases are created and run will be discussed later.

Generator Continued...

```
//Task to generate the random stimuli
```

```
task start(); ←
```

```
for(int i=0;i<`num_transactions;i++)
```

```
begin
```

```
    //Randomizing the inputs
```

```
    assert(blueprint.randomize() == 1); // More about assertions in the next chapter
```

```
    //Putting the randomized inputs to mailbox
```

```
    mbx_gd.put(blueprint.copy());
```

```
    $display("GENERATOR Randomized transaction data_in=%0h, write_enb=%0d, read_enb=%0d, address=%0h",  
blueprint.data_in, blueprint.write_enb, blueprint.read_enb, blueprint.address, $time);
```

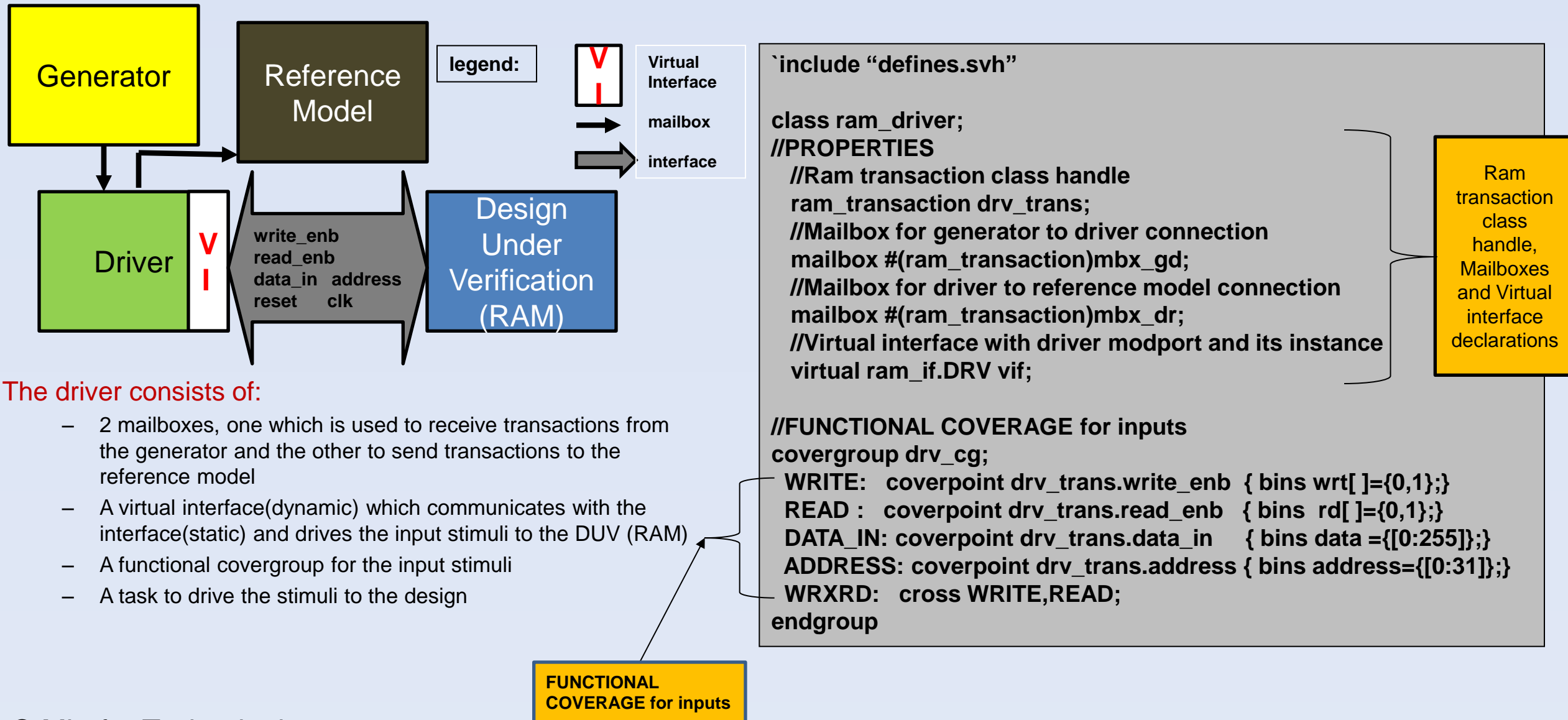
```
end
```

```
endtask
```

```
endclass
```

Task which generates the random stimuli

Driver [ram_driver.sv File]



The driver consists of:

- 2 mailboxes, one which is used to receive transactions from the generator and the other to send transactions to the reference model
- A virtual interface(dynamic) which communicates with the interface(static) and drives the input stimuli to the DUV (RAM)
- A functional covergroup for the input stimuli
- A task to drive the stimuli to the design

Driver Continued...

Explicitly overriding the constructor to make mailbox connection from driver to generator, to make mailbox connection from driver to reference model and to connect the virtual interface from driver to environment

Task to drive the stimuli

Getting the transaction from generator

//METHODS

//Explicitly overriding the constructor to make mailbox connection from driver
//to generator, to make mailbox connection from driver to reference model and
//to connect the virtual interface from driver to environment

```
function new(mailbox #(ram_transaction)mbx_gd,  
            mailbox #(ram_transaction)mbx_dr,  
            virtual ram_if.DRV vif);
```

```
    this.mbx_gd=mbx_gd;  
    this.mbx_dr=mbx_dr;  
    this.vif=vif;
```

```
    //Creating the object for covergroup  
    drv_cg=new();  
endfunction
```

//Task to drive the stimuli

```
task start();  
    repeat(3) @(vif.drv_cb);  
    for(int i=0;i<`num_transactions;i++)  
    begin  
        drv_trans=new();  
        //Getting the transaction from generator  
        mbx_gd.get(drv_trans);
```

Driver Continued...

```
if(vif.drv_cb.reset==1)
repeat(1) @(vif.drv_cb)
begin
vif.drv_cb.write_enb<=0;
vif.drv_cb.read_enb<=0;
vif.drv_cb.data_in<=8'bz;
vif.drv_cb.address<=0;
mbx_dr.put(drv_trans);
repeat(1) @(vif.drv_cb);
$display("DRIVER DRIVING DATA TO THE INTERFACE data_in=%0h, write_enb=%0d, read_enb=%0d, address=%0h",
vif.drv_cb.data_in, vif.drv_cb.write_enb, vif.drv_cb.read_enb, vif.drv_cb.address, $time);
end
else
repeat(1) @(vif.drv_cb)
begin
vif.drv_cb.write_enb<=drv_trans.write_enb;
vif.drv_cb.read_enb<=drv_trans.read_enb;
vif.drv_cb.data_in<=drv_trans.data_in;
vif.drv_cb.address<=drv_trans.address;
repeat(1) @(vif.drv_cb);
$display("DRIVER WRITE OPERATION DRIVING DATA TO THE INTERFACE data_in=%0h, write_enb=%0d, read_enb=%0d,
address=%0h", vif.drv_cb.data_in, vif.drv_cb.write_enb, vif.drv_cb.read_enb, vif.drv_cb.address, $time);
vif.drv_cb.write_enb<=0;
//Putting the randomized inputs to mailbox
mbx_dr.put(drv_trans);
//Sampling the covergroup
drv_cg.sample();
$display("INPUT FUNCTIONAL COVERAGE = %0d", drv_cg.get_coverage());
end
end
endtask
endclass
```

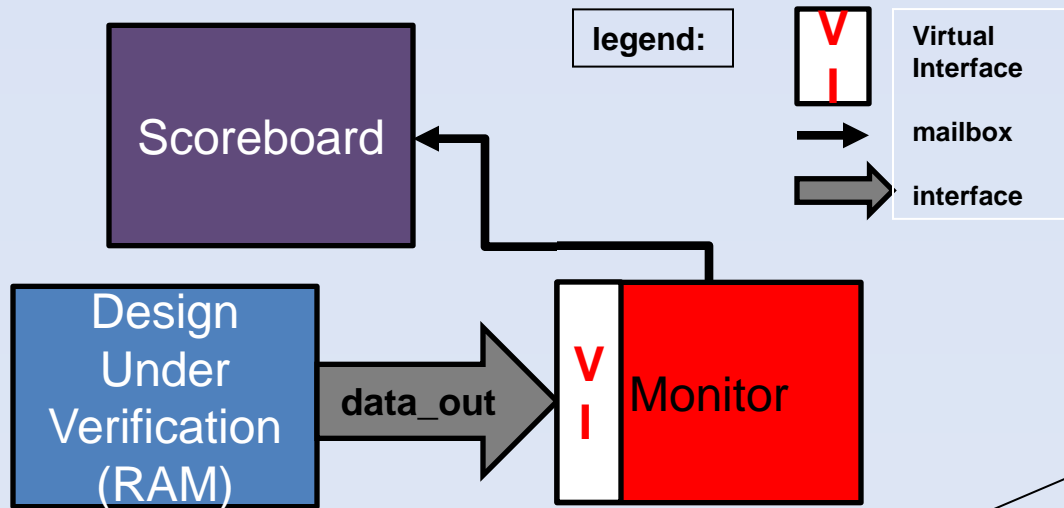
Initialize on reset

Driving the stimuli to the interface

Putting the randomized inputs to mailbox

Sampling the covergroup

Monitor [ram_monitor.sv File]



```
`include "defines.svh"

class ram_monitor;
//PROPERTIES
    //Ram transaction class handle
    ram_transaction mon_trans;
    //Mailbox for monitor to scoreboard connection
    mailbox #(ram_transaction) mbx_ms;
    //Virtual interface with monitor modport and its instance
    virtual ram_if.MON vif;

//FUNCTIONAL COVERAGE for outputs
    covergroup mon_cg;
        DATA_OUT: coverpoint mon_trans.data_out {bins dout = {[0:255]}};
    endgroup

//METHODS
    //Explicitly overriding the constructor to make mailbox connection from monitor
    //to scoreboard, and to connect the virtual interface from monitor to environment
    function new( virtual ram_if.MON vif,
                  mailbox #(ram_transaction) mbx_ms);

        this.vif=vif;
        this.mbx_ms=mbx_ms;
        //Creating the object for covergroup
        mon_cg=new();
    endfunction
```

Ram transaction class handle, Mailboxes and virtual interface declarations

FUNCTIONAL COVERAGE for outputs

The monitor consists of:

- 1 mailbox, which is used to send the transactions to the scoreboard
- A virtual interface(dynamic) which communicates with the interface(static) and collects the output from the DUV (RAM).
- A functional covergroup for the output
- A task to collect the output of the design

Explicitly overriding the constructor to make mailbox connection from monitor to scoreboard, and to connect the virtual interface from monitor to environment

Monitor Continued...

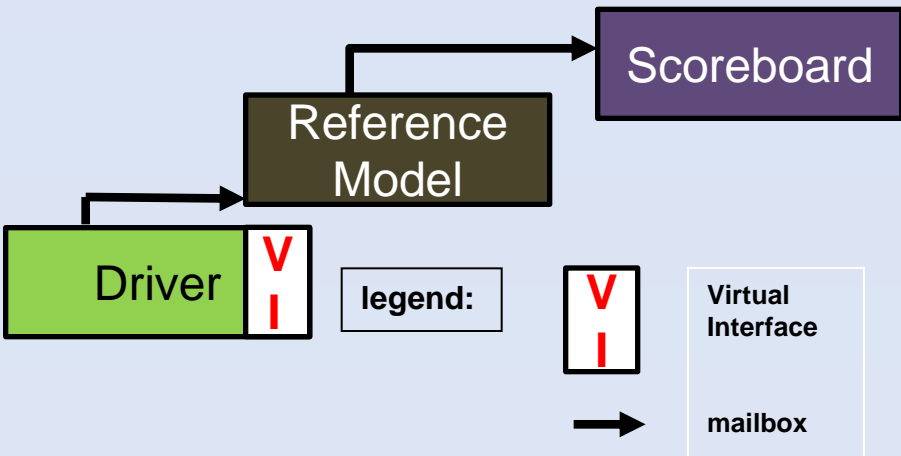
```
//Task to collect the output from the interface
task start();
repeat(4) @(vif.mon_cb);
for(int i=0;i<`num_transactions;i++)
begin
    mon_trans=new();
    repeat(1) @(vif.mon_cb)
    begin
        mon_trans.data_out=vif.mon_cb.data_out;
    end
    $display("MONITOR PASSING THE DATA TO SCOREBOARD data_out=%0h",
mon_trans.data_out, $time);
    //Putting the collected outputs to mailbox
    mbx_ms.put(mon_trans);
    //Sampling the covergroup
    mon_cg.sample();
    $display("OUTPUT FUNCTIONAL COVERAGE = %0d", mon_cg.get_coverage());
    repeat(1) @(vif.mon_cb);
end
endtask
endclass
```

Monitoring the output

Putting the collected transaction to the mailbox

Sampling the covergroup

Reference Model [ram_reference_model.sv File]



The reference model consists of:

- 2 mailboxes, one which is used to receive transactions from the driver and the other which is used to send transactions to the scoreboard
- A task to mimic the functionality of the design. In this case a 2-D array is used to model the RAM storage

```
`include "defines.svh"
```

```
class ram_reference_model;
```

```
//PROPERTIES
```

```
//Ram transaction class handle
```

```
ram_transaction ref_trans;
```

```
//Mailbox for reference model to scoreboard connection
```

```
mailbox #(ram_transaction) mbx_rs;
```

```
//Mailbox for driver to reference model connection
```

```
mailbox #(ram_transaction) mbx_dr;
```

```
//Virtual interface with reference model modport and its instance
```

```
virtual ram_if.REF_SB vif;
```

```
//2-D array used for RAM storage
```

```
reg [`DATA_WIDTH-1:0] MEM [`DATA_DEPTH-1:0];
```

```
//METHODS
```

```
//Explicitly overriding the constructor to make a mailbox connection from driver
```

```
//to reference model, a mailbox connection from reference model to scoreboard
```

```
//and to connect the virtual interface from reference model to environment
```

```
function new(mailbox #(ram_transaction) mbx_dr,  
             mailbox #(ram_transaction) mbx_rs,  
             virtual ram_if.REF_SB vif);
```

```
    this.mbx_dr=mbx_dr;
```

```
    this.mbx_rs=mbx_rs;
```

```
    this.vif=vif;
```

```
endfunction
```

Ram transaction class handle, Mailboxes and virtual interface declarations

A 2-D array is used to model the RAM storage

Explicitly overriding the constructor to make mailbox connection from driver to reference model, to make mailbox connection from reference model to scoreboard and to connect the virtual interface from reference model to environment

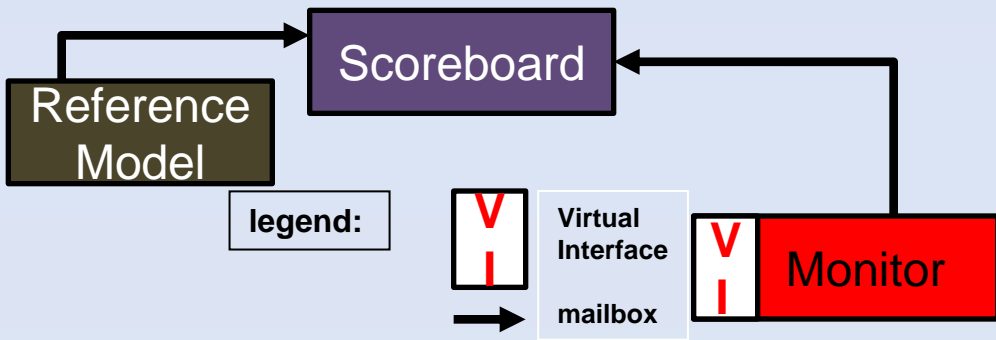
Reference Model Continued...

```
//Task which mimics the functionality of the RAM
task start();
for(int i=0;i<`num_transactions;i++)
begin
    ref_trans=new();
    //getting the driver transaction from mailbox
    mbx_dr.get(ref_trans);
    repeat(1) @(vif.ref_cb)
    begin
        if(ref_trans.write_enb)
            MEM[ref_trans.address]=ref_trans.data_in;
            $display("REFERENCE MODEL DATA IN MEMORY MEM[%0h]=%0h",
ref_trans.address,MEM[ref_trans.address],$time);
        if(ref_trans.read_enb)
            ref_trans.data_out=MEM[ref_trans.address];
            $display("REFERENCE MODEL DATA OUT FROM MEMORY data_out=%0h",ref_trans.data_out,$time);
        end
    end
    //Putting the reference model transaction to mailbox
    mbx_rs.put(ref_trans);
end
endtask
endclass
```

Getting the transaction from the driver's mailbox and mimicking the functionality of the RAM using the associative array declared

Putting the transaction on the mailbox

Scoreboard [ram_scoreboard.sv File]



The scoreboard consists of:

- 2 mailboxes, one which is used to receive expected transactions from the reference model, and the other which is used to receive the actual transactions from the monitor
- 2 tasks, one to collect the transactions from the reference model and the monitor, and the other to compare these transactions and report whether the data matches or mismatches
- 2-D arrays are used to store the collected data, one from the reference model and the other from the monitor

```
`include "defines.svh"
```

```
class ram_scoreboard;
```

```
//PROPERTIES
```

```
//Ram transaction class handles
```

```
ram_transaction ref2sb_trans,mon2sb_trans;
```

```
//Mailbox for from reference model to scoreboard connection  
mailbox #(ram_transaction) mbx_rs;
```

```
//Mailbox for from monitor to scoreboard connection  
mailbox #(ram_transaction) mbx_ms;
```

```
//2-D arrays used for storing data_out (from reference model, EXPECTED  
//RESULT) w.r.t address in reference model memory and storing data_out (from  
//DUV,ACTUAL RESULT) w.r.t address in monitor memory
```

```
logic [ `DATA_WIDTH-1:0] ref_mem [ `DATA_DEPTH-1:0];
```

```
logic [ `DATA_WIDTH-1:0] mon_mem [ `DATA_DEPTH-1:0];
```

```
//Variables to indicate number of matches and mismatches
```

```
int MATCH = 0;
```

```
int MISMATCH = 0;
```

```
//METHODS
```

```
//Explicitly overriding the constructor to make mailbox connection from monitor
```

```
//to scoreboard, to make mailbox connection from reference model to scoreboard
```

```
function new(mailbox #(ram_transaction) mbx_rs,  
             mailbox #(ram_transaction) mbx_ms);
```

```
    this.mbx_rs=mbx_rs;
```

```
    this.mbx_ms=mbx_ms;
```

```
endfunction
```

Ram transaction
class handles,
Mailboxes
declarations

2-D arrays used for storing data_out (from
reference model, the EXPECTED RESULT)
w.r.t address in reference model memory
and storing data_out (from DUV, the
ACTUAL RESULT) w.r.t address in monitor
memory

Scoreboard Continued...

```
//Task which collects data_out from reference model and monitor
//and stores them in their respective memories
task start();
  for(int i=0;i<`num_transactions;i++)
    begin
      ref2sb_trans=new();
      mon2sb_trans=new();
      fork
        begin
          //getting the reference model transaction from mailbox
          mbx_rs.get(ref2sb_trans);
          ref_mem[ref2sb_trans.address]=ref2sb_trans.data_out;
          $display("#####SCOREBOARD REF data_out=%0h, ADDRESS=%0h
#####", ref_mem[ref2sb_trans.address], ref2sb_trans.address, $time);
        end
        begin
          //getting the monitor transaction from mailbox
          mbx_ms.get(mon2sb_trans);
          mon_mem[mon2sb_trans.address]=mon2sb_trans.data_out;
          $display("!!!!!!!!!!SCOREBOARD MON data_out=%0h, ADDRESS=%0h !!!!!!!!!!!",
mon_mem[mon2sb_trans.address], mon2sb_trans.address, $time);
        end
      join
      if (i != (`num_transactions-1))
        compare_report();
    end
endtask
```

Getting the reference
model transaction from
mailbox and storing it into
the 2-D array

Getting the monitor
transaction from
mailbox and storing it
into the 2-D array

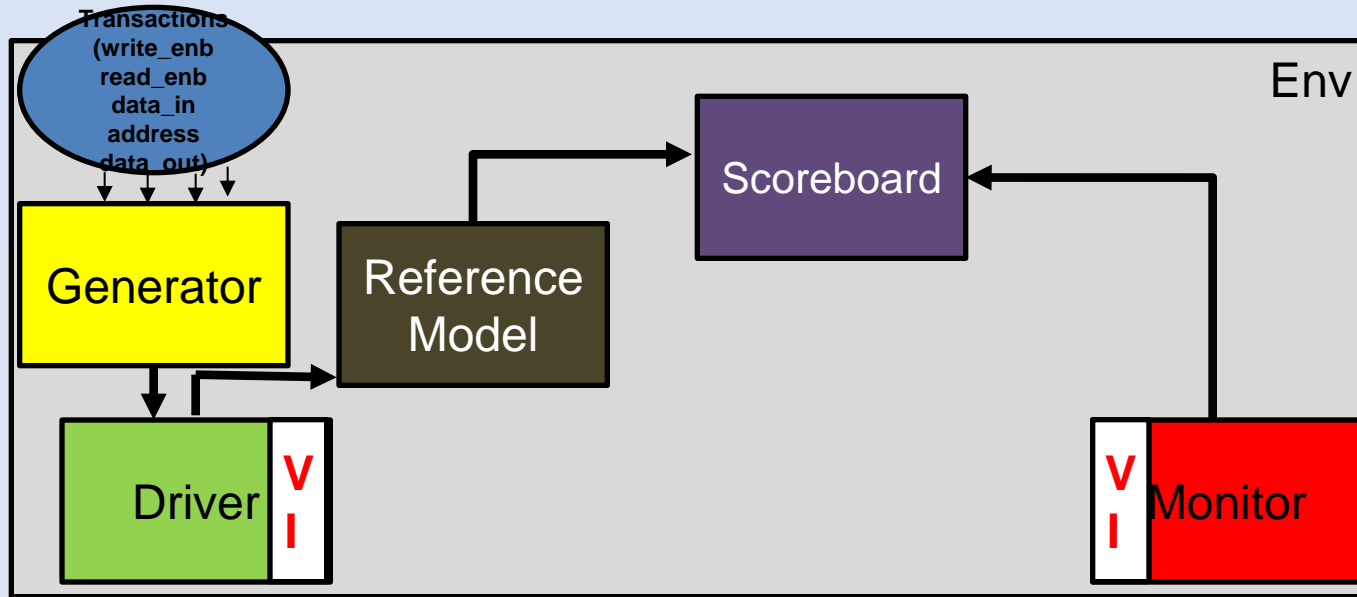
Calling compare and
report task

Scoreboard Continued...

```
//Task which compares the memories and generates the report
task compare_report(); ←
    if(ref_mem[ref2sb_trans.address] === mon_mem[mon2sb_trans.address])
        begin
            $display("SCOREBOARD REF data_out=%0h, MON data_out=%0h",
ref_mem[ref2sb_trans.address], mon_mem[mon2sb_trans.address], $time);
            ++MATCH;
            $display("DATA MATCH SUCCESSFUL. MATCH count = %0d",MATCH);
        end
    else
        begin
            $display("SCOREBOARD REF data_out=%0h, MON data_out=%0h",
ref_mem[ref2sb_trans.address], mon_mem[mon2sb_trans.address], $time);
            ++MISMATCH;
            $display("DATA MATCH FAILURE. MISMATCH count = %0d",MISMATCH);
        end
    endtask
endclass
```

Compare and report task

Environment [ram_environment.sv File]



Virtual interfaces for driver, monitor and reference model
Mailbox for generator to driver connection, mailbox for driver to reference model connection, mailbox for reference model to scoreboard connection and mailbox for monitor to scoreboard connection

```
`include "defines.svh"
```

```
class ram_environment;  
//PROPERTIES
```

```
//Virtual interfaces for driver, monitor and reference model
```

```
virtual ram_if drv_vif;
```

```
virtual ram_if mon_vif;
```

```
virtual ram_if ref_vif;
```

```
//Mailbox for generator to driver connection
```

```
mailbox #(ram_transaction) mbx_gd;
```

```
//Mailbox for driver to reference model connection
```

```
mailbox #(ram_transaction) mbx_dr;
```

```
//Mailbox for reference model to scoreboard connection
```

```
mailbox #(ram_transaction) mbx_rs;
```

```
//Mailbox for monitor to scoreboard connection
```

```
mailbox #(ram_transaction) mbx_ms;
```

```
//Declaring handles for components
```

```
//generator, driver, monitor, reference model and scoreboard
```

```
ram_generator          gen;
```

```
ram_driver             drv;
```

```
ram_monitor            mon;
```

```
ram_reference_model    ref_sb;
```

```
ram_scoreboard         scb;
```

Declaring handles for components
generator, driver, monitor, reference
model and scoreboard

legend:



The environment consists of:

- 3 virtual interfaces – one for the driver, one for the monitor and one for the reference model
- 4 mailboxes - a mailbox for the generator to driver connection, a mailbox for the driver to reference model connection, a mailbox for the reference model to scoreboard connection, and, a mailbox for the monitor to scoreboard connection
- Handles for generator, driver, monitor, reference model and scoreboard
- A task called `build()` which creates objects for all the mailboxes and components and a task `start()` which calls all the start methods

Environment Continued...

//METHODS

//Explicitly overriding the constructor to connect the virtual interfaces

//from driver, monitor and reference model to test

```
function new (virtual ram_if drv_vif,  
             virtual ram_if mon_vif,  
             virtual ram_if ref_vif);
```

```
    this.drv_vif=drv_vif;
```

```
    this.mon_vif=mon_vif;
```

```
    this.ref_vif=ref_vif;
```

```
endfunction
```

//Task which creates objects for all the mailboxes and components

```
task build();
```

```
    begin
```

//Creating objects for mailboxes

```
    mbx_gd=new();
```

```
    mbx_dr=new();
```

```
    mbx_rs=new();
```

```
    mbx_ms=new();
```

//Creating objects for components and passing the arguments

//in the function new() i.e the constructor

```
    gen=new(mbx_gd);
```

```
    drv=new(mbx_gd,mbx_dr,drv_vif);
```

```
    mon=new(mon_vif,mbx_ms);
```

```
    ref_sb=new(mbx_dr,mbx_rs,ref_vif);
```

```
    scb=new(mbx_rs,mbx_ms);
```

```
    end
```

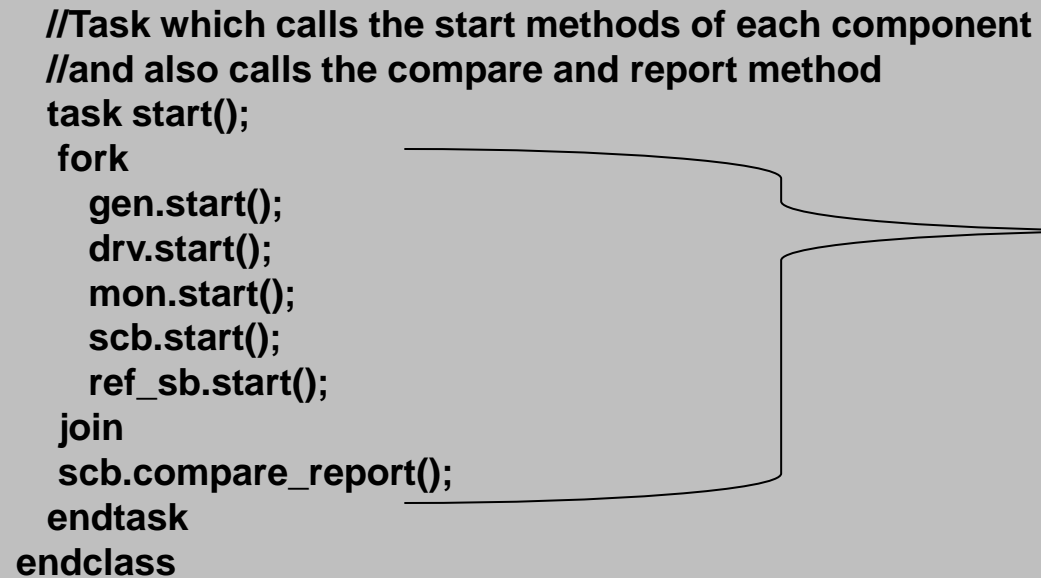
```
endtask
```

Explicitly overriding the constructor to connect the virtual interfaces from driver, monitor and reference model to test

Task which creates objects for all the mailboxes and components

Environment Continued...

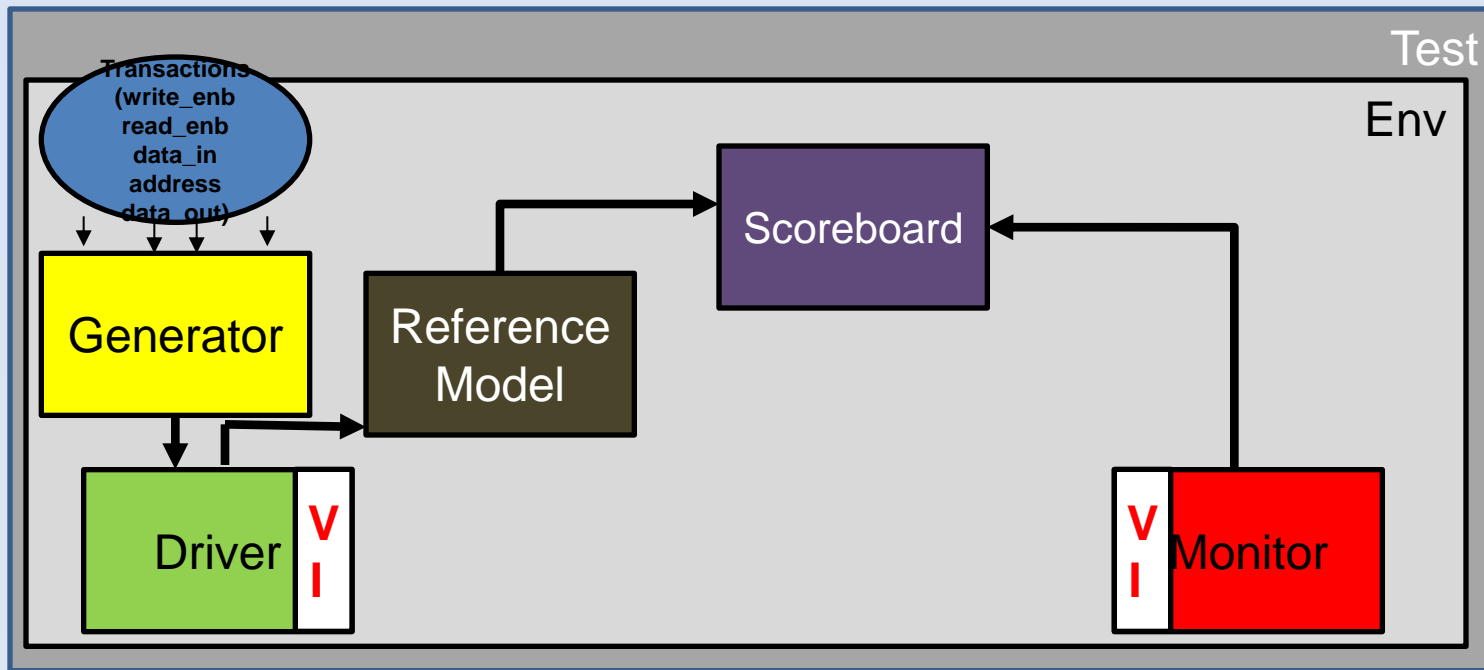
```
//Task which calls the start methods of each component  
//and also calls the compare and report method  
task start();  
  fork  
    gen.start();  
    drv.start();  
    mon.start();  
    scb.start();  
    ref_sb.start();  
  join  
    scb.compare_report();  
endtask  
endclass
```



This task calls the start methods of each component and also calls the compare and report method

This fork . . . join block runs each start method concurrently , thereby making each testbench component run independently

Test [ram_test.sv File]



The test consists of:

- 3 virtual interfaces, one for the driver, one for the monitor and one for the reference model
- A handle for the environment
- Code that explicitly overrides the new() constructor to connect the virtual interfaces from driver, monitor and reference model to test
- A task called run, which builds the object for the environment handle and calls the build and start methods of the environment

legend:



Virtual
Interface
→
mailbox

```
`include "defines.svh"
```

```
class ram_test;
```

```
//PROPERTIES
```

```
//Virtual interfaces for driver, monitor and reference model
```

```
virtual ram_if drv_vif;
```

```
virtual ram_if mon_vif;
```

```
virtual ram_if ref_vif;
```

```
//Declaring handle for environment
```

```
ram_environment env;
```

```
//METHODS
```

```
//Explicitly overriding the constructor to connect the virtual
```

```
//interfaces from driver, monitor and reference model to test
```

```
function new(virtual ram_if drv_vif,  
             virtual ram_if mon_vif,  
             virtual ram_if ref_vif);
```

```
    this.drv_vif=drv_vif;
```

```
    this.mon_vif=mon_vif;
```

```
    this.ref_vif=ref_vif;
```

```
endfunction
```

```
//Task which builds the object for environment handle and
```

```
//calls the build and start methods of the environment
```

```
task run();
```

```
    env=new(drv_vif,mon_vif,ref_vif);
```

```
    env.build;
```

```
    env.start;
```

```
endtask
```

```
endclass
```

RAM Package [ram_package.sv File]

The RAM package consists of:

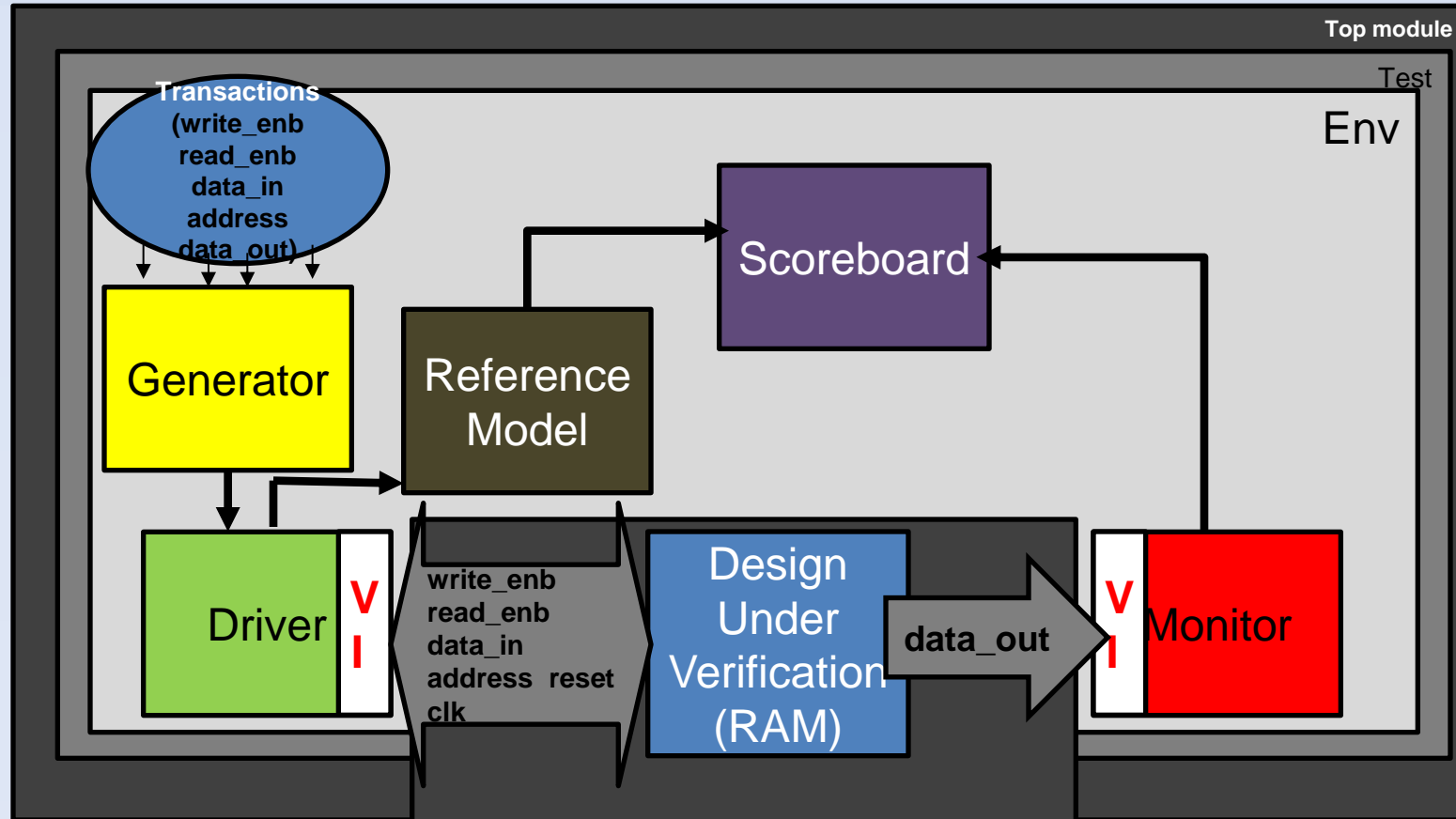
The transaction, generator, driver, monitor, reference model, scoreboard, environment and the test source files

This package as a whole will be imported in the top module which helps in compilation

This package includes all the files in the testbench architecture which will be imported in the top module

```
package ram_package;  
  `include "ram_transaction.sv"  
  `include "ram_generator.sv"  
  `include "ram_driver.sv"  
  `include "ram_monitor.sv"  
  `include "ram_reference_model.sv"  
  `include "ram_scoreboard.sv"  
  `include "ram_environment.sv"  
  `include "ram_test.sv"  
endpackage
```

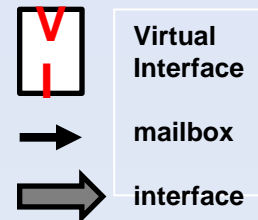
Top Module



The top module consists of:

- Importing the RAM package
- Clock generation code
- Reset generation code
- Interface, DUV (RAM) and test instantiation
- An initial block to run the test(s)

legend:



```
`include "defines.svh"
```

```
module top();
    //Importing the ram package
    import ram_package ::*;
    //Declaring variables for clock and reset
    logic clk;
    logic reset;
```

```
//Generating the clock
initial
begin
    forever #10 clk=~clk; // Period is 20ns -->
    Frequency is 50Mhz
end
```

```
//Asserting and de-asserting the reset
initial
begin
    @(posedge clk);
    reset=1;
    repeat(1)@(posedge clk);
    reset=0;
end
```

Top Module Continued...

```
//Instantiating the interface
ram_if intrf(clk,reset);

//Instantiating the DUV
RAM DUV(.data_in(intrf.data_in),
        .write_enb(intrf.write_enb),
        .read_enb(intrf.read_enb),
        .data_out(intrf.data_out),
        .address(intrf.address),
        .clk(clk),
        .reset(reset)
);

//Instantiating the Test
ram_test test= new(intrf.DRV,intrf.MON,intrf.REF_SB);

//Calling the test's run task which starts the execution of the testbench architecture
initial
begin
    test.run();
    $finish();
end
endmodule
```

Instantiating the interface,
DUV(RAM) and the test

Running the test

Procedure for Running Different Tests

- 1) To run different test cases, the transaction class has to be extended from the parent transaction class
- 2) Then the constraints in the transaction class have to be overridden for different features written in the verification plan

The code shown here is for the write and read features of the RAM design

```
`include "defines.svh"

class ram_transaction;
//PROPERTIES
//INPUTS declared as rand variables
rand logic [DATA_WIDTH-1:0] data_in;
rand logic write_enb,read_enb;
rand logic [ADDR_WIDTH-1:0] address;
//OUTPUTS declare as non-rand variables
logic [DATA_WIDTH-1:0] data_out;
//CONSTRAINTS for write_enb and read_enb
constraint wr_rd_constraint {{write_enb,read_enb} inside {[0:3]};}
constraint wr_not_equal_rd {{write_enb,read_enb}!=2'b11;}
//METHODS
//Copying objects
virtual function ram_transaction copy();
    copy = new();
    copy.data_in=this.data_in;
    copy.write_enb=this.write_enb;
    copy.read_enb=this.read_enb;
    copy.address=this.address;
    return copy;
endfunction
endclass
```

```
class ram_transaction_write extends ram_transaction;
//CONSTRAINTS OVERRIDING by extending the transaction class
constraint wr_rd_constraint {{write_enb,read_enb}==2'b10;}
//METHODS
//Copying objects
virtual function ram_transaction copy();
    ram_transaction_write copy1;
    copy1=new();
    copy1.data_in=this.data_in;
    copy1.write_enb=this.write_enb;
    copy1.read_enb=this.read_enb;
    copy1.address=this.address;
    return copy1;
endfunction
endclass
```

```
class ram_transaction_read extends ram_transaction;
//CONSTRAINTS OVERRIDING by extending the transaction class
constraint wr_rd_constraint {{write_enb,read_enb}==2'b01;}
//METHODS
//Copying objects
virtual function ram_transaction copy();
    ram_transaction_read copy2;
    copy2=new();
    copy2.data_in=this.data_in;
    copy2.write_enb=this.write_enb;
    copy2.read_enb=this.read_enb;
    copy2.address=this.address;
    return copy2;
endfunction
endclass
```

An important point to remember is that the return type for all the copy() functions in the parent transaction class and extended transaction classes, is the parent class type

Procedure for Running Different Tests Continued...

- 3) Next, the test class has to be extended from the parent test class
- 4) Then the extended transaction class object has to be assigned to the parent transaction class object

```
class ram_test;
//PROPERTIES
//Virtual interfaces for driver, monitor and reference model
virtual ram_if drv_vif;
virtual ram_if mon_vif;
virtual ram_if ref_vif;
//Declaring handle for environment
ram_environment env;
//METHODS
//Explicitly overriding the constructor to connect the virtual interfaces
//from driver, monitor and reference model to test
function new(virtual ram_if drv_vif,
             virtual ram_if mon_vif,
             virtual ram_if ref_vif);
    this.drv_vif=drv_vif;
    this.mon_vif=mon_vif;
    this.ref_vif=ref_vif;
endfunction
//Task which builds the object for environment handle and
//calls the build and start methods of the environment
task run();
    env=new(drv_vif,mon_vif,ref_vif);
    env.build;
    env.start;
endtask
endclass
```

```
class test_write extends ram_test;
    ram_transaction_write trans_write;
    function new(virtual ram_if drv_vif,virtual ram_if mon_vif,
                virtual ram_if ref_vif);
        super.new(drv_vif,mon_vif,ref_vif);
    endfunction
    task run();
        env=new(drv_vif,mon_vif,ref_vif);
        env.build;
        begin
            trans_write = new();
            env.gen.blueprint= trans_write;
        end
        env.start;
    endtask
endclass
```

```
class test_read extends ram_test;
    ram_transaction_read trans_read;
    function new(virtual ram_if drv_vif,virtual ram_if mon_vif,
                virtual ram_if ref_vif);
        super.new(drv_vif,mon_vif,ref_vif);
    endfunction
    task run();
        env=new(drv_vif,mon_vif,ref_vif);
        env.build;
        begin
            trans_read = new();
            env.gen.blueprint= trans_read;
        end
        env.start;
    endtask
endclass
```


Procedure for Running Different Tests Continued...

- 5) Next, In the top module the different tests are instantiated and objects are created for these tests
- 6) These tests are run in the initial block as and when required. Only one test can be run at a given instant of time

In order to run many tests using a single simulation session the concept of regression is used

```
module top();
    //Importing the ram package
    import ram_package::*;
    //Declaring variables for clock and reset
    bit clk;
    bit reset;
    //Generating the clock
    ...
    ...
    //Asserting and de-asserting the reset
    ...
    //Instantiating the interface
    ram_if intrf(clk,reset);
    //Instantiating the DUV
    RAM DUV(.data_in(intrf.data_in),
        ...
    );
    //Instantiating the Test
    ram_test t1 = new(intrf.DRV,intrf.MON,intrf.REF_SB);
    test_write t2 = new(intrf.DRV,intrf.MON,intrf.REF_SB);
    test_read t3 = new(intrf.DRV,intrf.MON,intrf.REF_SB);

    //Calling the test's run task which starts the execution of the
    //testbench architecture
    initial
    begin
        //t1.run(); // running some ram test
        //t2.run(); // running write test
        t3.run(); // running read test
        $finish();
    end
endmodule
```

Procedure for Regression Testing

Regression is the process of re-running verification of the DUV with an identified or known set of test cases before releasing the code for synthesis or for generating coverage reports. This is done in order to ensure that the functionality of the DUV is correct, and that new code changes have not introduced any new bugs in the design.

- 1) To run regression tests, the test class has to be extended from the parent test class.
- 2) The transaction class handles for each extended transaction class are declared.
- 3) The object for the extended transaction class handle is created. Then the extended transaction class object is assigned to the parent transaction class object and the start() method is called.
- 4) The above step is repeated sequentially for all extended transaction classes as shown

```
class test_regression extends ram_test;

    ram_transaction      trans;
    ram_transaction_write trans_write;
    ram_transaction_read  trans_read;

    function new(virtual ram_if drv_vif,
                 virtual ram_if mon_vif,
                 virtual ram_if ref_vif);
        super.new(drv_vif,mon_vif,ref_vif);
    endfunction

    task run();
        env=new(drv_vif,mon_vif,ref_vif);
        env.build();

        ///////////////////////////////////
        trans = new();
        env.gen.blueprint= trans;
        env.start();
        ///////////////////////////////////

        ///////////////////////////////////
        trans_write = new();
        env.gen.blueprint= trans_write;
        env.start();
        ///////////////////////////////////

        ///////////////////////////////////
        trans_read = new();
        env.gen.blueprint= trans_read;
        env.start();
        ///////////////////////////////////

    endtask
endclass
```

Procedure for Regression Testing Continued...

- 5) Next, In the top module the regression test is instantiated and the object is created for it.
- 6) Regression tests can be run from the initial block as and when required.

```
module top();  
    //Importing the ram package  
    import ram_package::*;  
    //Declaring variables for clock and reset  
    bit clk;  
    bit reset;  
    //Generating the clock  
    ...  
    ...  
    //Asserting and de-asserting the reset  
    ...  
    //Instantiating the interface  
    ram_if intrf(clk,reset);  
    //Instantiating the DUV  
    RAM DUV(.data_in(intrf.data_in),  
        ...  
    );  
    //Instantiating the Test  
    regression_test reg_tb = new(intrf.DRV,intrf.MON,intrf.REF_SB);  
  
    //Calling the test's run task which starts the execution of the  
    //testbench architecture  
    initial  
    begin  
        reg_tb.run(); // running regression test  
        $finish();  
    end  
endmodule
```

11. Assertions

SystemVerilog Assertions

- They are primarily used to validate the functionality/behavior of a design and capture the designer's intent
- They form the basis of the Assertion-Based Verification Methodology
 - White-box assertions (inside sub-blocks)
 - Black-box assertions (at interface boundaries)
- SystemVerilog assertions are represented as a property that defines intent and this property is enforced by using the assert statement
- They are applicable in both design and verification domains
 - SV assertions are easy to code, understand and are used by both design and verification engineers
- Assertions help in monitoring the design
 - Concurrent or standalone assertions (explained later)
 - Procedural or embedded assertions (explained later)
- Checking of SV assertions can be performed either,
 - Dynamically by simulation, or,
 - Statically by a separate property checker tool, i.e., a formal verification tool that proves whether or not a design meets its specification

SystemVerilog Assertions Continued...

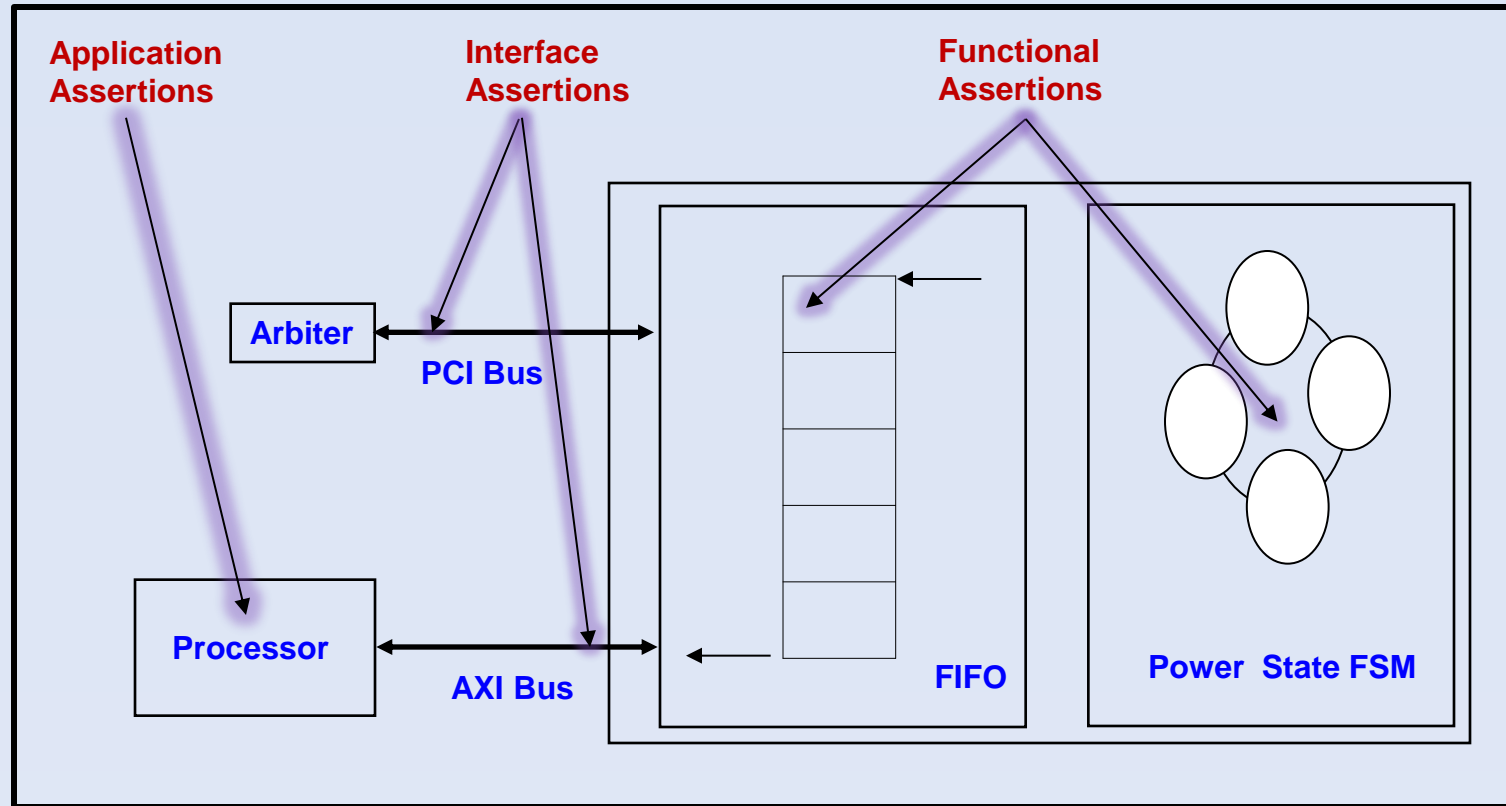
- There are 2 types of assertions:
 - Immediate (explained later)
 - Concurrent (explained later)
- SystemVerilog assertions allow the verification environment to be reactive. This means that based on results of an assertion (pass or fail), different actions may be taken

Since an assertion is a statement that something must be true, the failure of an assertion shall have a severity associated with it. By default, the severity of an assertion failure is “\$error”.

Any of the severity levels described below can be specified in the fail statement:

- **\$fatal**: Is a run-time fatal
- **\$error**: Is a run-time error
- **\$warning**: Is a run-time warning, which can be suppressed by EDA tools
- **\$info**: This indicates that the given assertion failure has no specific severity level

Assertion Types



Common System Functions used in Assertions

- **\$rose()**: Returns true if the LSB of the expression changes to 1, else it returns false
- **\$fell()**: Returns true if the LSB of the expression changes to 0, else it returns false
- **\$countones()**: Returns the number of 1's in a bit-vector
- **\$stable()**: If the signal is stable, then it returns 1
- **\$onehot()**: Returns true if only 1 bit of the given expression is high
- **\$onehot0()**: Returns true if at most 1 bit of the given expression is high
- **\$past()**: Returns the sampled value of the expression that was present a number of clock ticks prior to the time of evaluation of \$past()
- **\$isunknown()**: Returns true if any bit of the expression is 'x' or 'z'

Note: If an assertion expression evaluates to 'x', 'z' or '0', then it is interpreted as being false. Otherwise, it is true.

Immediate Assertion Type

An immediate assertion is non-temporal and is the test of an expression performed when a statement is executed in the procedural code. It,

- Appears in procedural blocks
- Is based on simulation events
- Is used without the “property” keyword (explained later)
- Follows simulation semantics like an “if-else” construct

Syntax:

```
[assertion_label:] assert (expression) action_block;
```

where,

```
action_block ::= statement_or_null
```

- Executes immediately and can contain system tasks to control severity
- Is used with only dynamic simulation tools

E.g.:

```
A1: assert (!(a & c))  
    $display("Meets the assert condition. Test Pass");  
    else  
    $error("Does not meet the assert condition. Test Fail");
```

The assert statement works like an if-else clause. If the assertion fails and no else clause is specified, the tool shall, by default, call \$error.

Sequences

Boolean logic does not have the concept of time. The temporal capabilities can be defined for boolean expressions using the sequence feature. A sequence is a list of boolean expressions in a linear order of increasing time. The sequence is true over time if the boolean expressions are true at the specific clock ticks.

The **##** operator followed by a number or range specifies the delay from the current clock tick to the beginning of the sequence that follows.

E.g.:

```
sequence seq_exmpl;  
  @(negedge clk) a ##1 b ##[2:$] (c+d);  
endsequence
```

The delay **##1** indicates that the beginning of the sequence that follows is one clock tick later than the current clock tick, i.e., 'b' occurs 1 clock cycle after 'a'

The sequence **seq_exmpl** requires that '(c+d)' occurs 2,3,4 ... or infinite clock cycles after 'b'

Syntax:

```
sequence sequence_identifier_name_decl[list_of_signal(s)_or_variable(s)];  
  [assertion_variable(s)_declaration(s);]  
  sequence_expr;  
endsequence [: sequence_identifier_name_decl]
```

where,

sequence_expr is a boolean expression that could contain various sequence operations (explained later)

Sequence Operations

- Consecutive Repetition ([*])

- Specifies finitely many iterative matches of the operand sequence, with a delay of one clock tick from the end of one match to the beginning of the next. The overall repetition sequence matches at the end of the last iterative match of the operand. The notation [*] is the equivalent representation of <0 to \$>, while [+] is an equivalent representation of <1 to \$>

E.g.:

```
req ##1 hld ##1 hld ##1 hld ##1 gnt
```

Equivalent to hld[*3]

```
req ##1 hld[*3] ##1 gnt
```

- Goto Repetition ([->])

- Specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and strictly no match of the operand in between. The overall repetition sequence matches at the last iterative match of the operand

E.g.:

```
req ##1 hld [->4:9] ##1 gnt
```

req should be true in the first clock tick

hld should be true in the penultimate clock tick and including the penultimate there are at least 4 and at most 9 not-necessarily consecutive clock ticks strictly between the first and last on which hld is true

gnt should be true in the last clock tick

Sequence Operations Continued...

- **Non-Consecutive Repetition ([=)**

- Specifies finitely many iterative matches of the operand boolean expression, with a delay of one or more clock ticks from one match of the operand to the next successive match and strictly no match of the operand in between. The overall repetition sequence matches at or after the last iterative match of the operand, but before any later match of the operand

E.g.: `req ##1 hld [=3:9] ##1 gnt`

req should be true on the first clock tick

There are at least 3 and at most 9 not-necessarily consecutive clock ticks strictly in between the first and last on which hld should be true

gnt should be true on the last clock tick

- **AND Operation**

- The two operands of AND are sequences. This is used when both the operand sequences are expected to match

E.g.: `(req1 ##1 hld ##1 gnt1) and (req2 ##1 hld ##1 gnt2)`

- **OR Operation**

- The two operands of OR are sequences. This is used when at least one of the operand sequences is expected to match

E.g.: `(sig_a ##4 sig_b) or (sig_1 ##1 sig_2 ##3 sig_3)`

Sequence Operations Continued...

- **First Match Operation**

- The `first_match` operator matches only the first of possibly multiple matches for an evaluation attempt of its operand sequence. All subsequent matches are discarded from consideration

E.g.:

```
sequence seq1;  
    (req1 ##[2:5] hld ##1 gnt1) and (req2 ##[1:9] hld ##1 gnt2);  
endsequence  
  
sequence seq2;  
    first_match(seq1);  
endsequence
```

- **Using the endpoint of a sequence**

- The end point of a sequence is reached whenever the ending clock tick of a match of the sequence is reached, regardless of the starting clock tick of the match

E.g.:

```
sequence seq1;  
    @(posedge clk) $rose(rdy) ##2 bus1 ##1 bus2;  
endsequence  
  
sequence seq2;  
    @(posedge clk) reset ##3 seq1.ended ##2 read_memory;  
endsequence
```

Multiply Clocked Sequences

Multiply clocked sequences are built by concatenating singly-clocked sub-sequences using the single delay concatenation operator `##1`. The `##1` operator is non-overlapping and acts as a synchronizer between the clocks of the sub-sequences

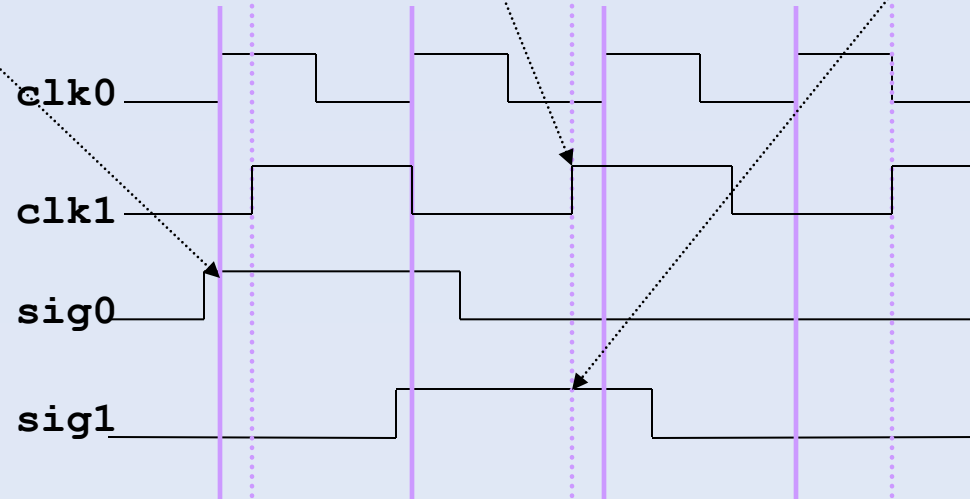
E.g.:

```
sequence mul_clk_seq;  
    @(posedge clk0) sig0 ##1 @(posedge clk1) sig1;  
endsequence
```

A match of this sequence starts with a match of sig0 at posedge clk0

Then `##1` moves the time to the nearest strictly subsequent posedge clk1

The match of the sequence ends at that point with a match of sig1



Properties

A property is a collection of logical and temporal relationships that represent the behavior and characteristics of a design

Properties are built using sequences

E.g.:

```
property prop_exmpl;  
  @(posedge clk) a | => (b or c);  
endproperty
```

Non-overlapped implication operator

The left hand side operand of the sequence expression is called **Antecedent**. If FALSE, then the property does not succeed.

The right hand side operand of the sequence expression is called **Consequent**.

Syntax:

```
property property_identifier_name_decl[list_of_signal(s)_or_variable(s)];  
  [assertion_variable(s)_declaration(s);]  
  [disable iff (expression)] sequence_expr;  
endproperty [: property_identifier_name_decl]
```

where,

sequence_expr is a boolean expression that could contain various sequence operations

Property Rules:

- If the antecedent is FALSE, then one need not care about the consequent and the property is considered to have a meaningless success
- If antecedent is TRUE and if consequent is FALSE, then the property has failed
- If antecedent is TRUE and if consequent is TRUE, then the property has succeeded

Recursive Properties

A named property is recursive if the declaration involves an instantiation of itself

E.g.1 : prop_always is a recursive property which says that the formal argument 'plk' must hold at every cycle

```
property prop_always(plk);  
    plk and (1'b1 ==> prop_always(plk));  
endproperty
```

E.g. 2: check_phase1 and check_phase2 are mutually recursive properties in the example shown

```
property check_phase1;  
    var1 ==> (phase1_property and (1'b1 ==> check_phase2));  
endproperty  
property check_phase2;  
    var2 ==> (phase2_property and (1'b1 ==> check_phase1));  
endproperty
```


Concurrent Assertion Type

A concurrent assertion describes behavior that spans over time – it is evaluated only at the occurrence of a clock tick. It,

- Appears in procedural blocks, interfaces, module or program definitions
- Is based on clock cycles and follows cycle semantics using sampled values
- Is used with the “property” keyword. This keyword distinguishes a concurrent assertion from an immediate assertion

Syntax:

```
[assertion_label:] assert property (property_spec) action_block;
```

where,

```
property_spec ::= [disable iff (expression)] sequence_expr;  
action_block ::= statement_or_null
```

- Executes in reactive region and can contain system tasks to control severity
- Is used with both dynamic simulation tools as well as formal verification checkers

E.g. 1:

```
property prop_abc;  
  @(posedge clk) disable iff (!reset_n)  
    a |-> (b ##1 c);  
endproperty  
  
assrt_chk: assert property(prop_abc);  
  $display("Conc assrt assrt_chk passed");  
else  
  $display("Conc assrt assrt_chk failed");
```

Concurrent assertions can be reset or disabled. They contain temporal behavior but embedded in a sequential environment and derive their activation from the simulation execution of the surrounding code. They contain sequential behavior and work on sampled values.

“(b ##1 c)” means that ‘b’ must be true in the current clock cycle and ‘c’ on the next clock cycle

|-> is the overlapped implication operator

Hence, a |-> (b ##1 c)

means that if ‘a’ is true, then ‘(b ##1 c)’ must also be true

Concurrent Assertion Type Continued...

E.g. 2:

```
property prop_exmpl;  
    @(posedge clk) a | => (b or c);  
endproperty  
  
assrt_exmpl1:  
    assert property (prop_exmpl);
```

E.g. 3:

```
sequence delay_exmpl(x, y, wire_delay);  
    x ##wire_delay y;  
endsequence  
  
assrt_exmpl2:  
    assert property (@(posedge clk) delay_exmpl(x,y,7));
```

Expect Statement

The expect statement is a procedural blocking statement that allows waiting on a property evaluation. The following features apply:

- It accepts the same syntax used to assert a property
- It causes the executing process to block until the given property succeeds or fails

E.g.:

```
program expect_example;

initial
begin
    #60ns;
    expect(@(posedge clk) req ##1 hld [*2] ##1 gnt) else $error("expect_example failed");
    :
    :
end

endprogram
```

The expect statement blocks until the sequence req ##1 hld [*2] ##1 gnt is matched or is determined not to match

Cover Property Statement

The cover property statement is used to monitor sequences and other behavioral aspects of the design for coverage

When the property for the cover statement is successful, the pass statements can specify a coverage function, such as monitoring all paths for a sequence

Coverage results are of two types:

- 1) Coverage for Sequences. The results shall include
 - Number of times attempted
 - Number of times matched
- 2) Coverage for Properties. The results shall include
 - Number of times attempted
 - Number of times succeeded
 - Number of times failed

Syntax:

[cover_label:] **cover property** (property_spec) statement_or_null;

where,

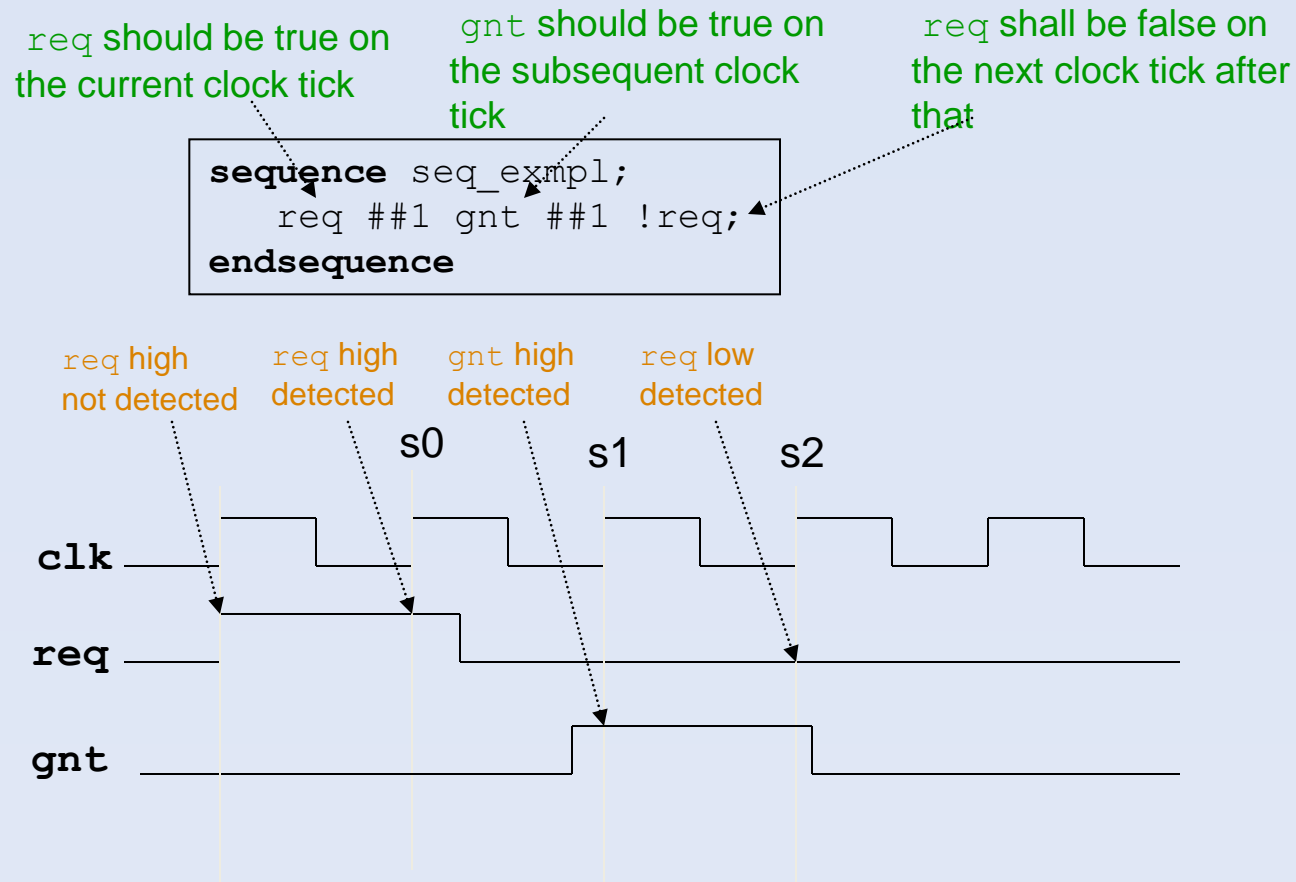
property_spec ::= [disable iff (expression)] sequence_expr;

E.g.:

```
module coverex(input bit clk);  
  logic req,hld,gnt;  
  
  property p1;  
    @(posedge clk) gnt |-> req ##2 hld;  
  endproperty  
  
  ap1: assert property (p1);  
  
  sequence s1;  
    @(posedge clk) req ##6 gnt;  
  endsequence  
  
  cs1: cover property (s1);  
endmodule
```

Assertions-Example 1

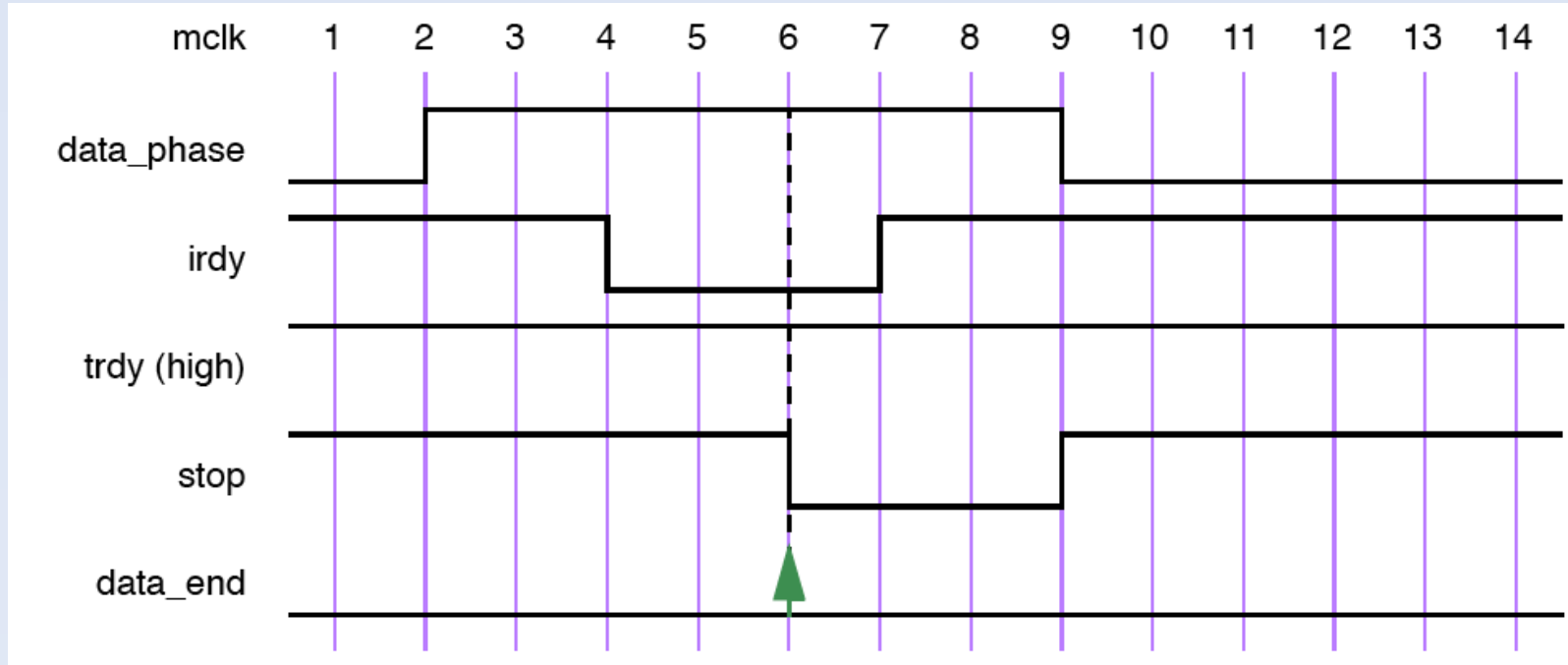
Sequence Example



Assertions-Example 2

Example of implication and conditional sequence matching

```
property data_end;  
  @(posedge mclk)  
  data_phase |-> ((irdy==0) && ($fell(trdy) || $fell(stop))) ;  
endproperty
```



Assertions-Example 3

Assertion to check for no overflow in a stack or queue structure

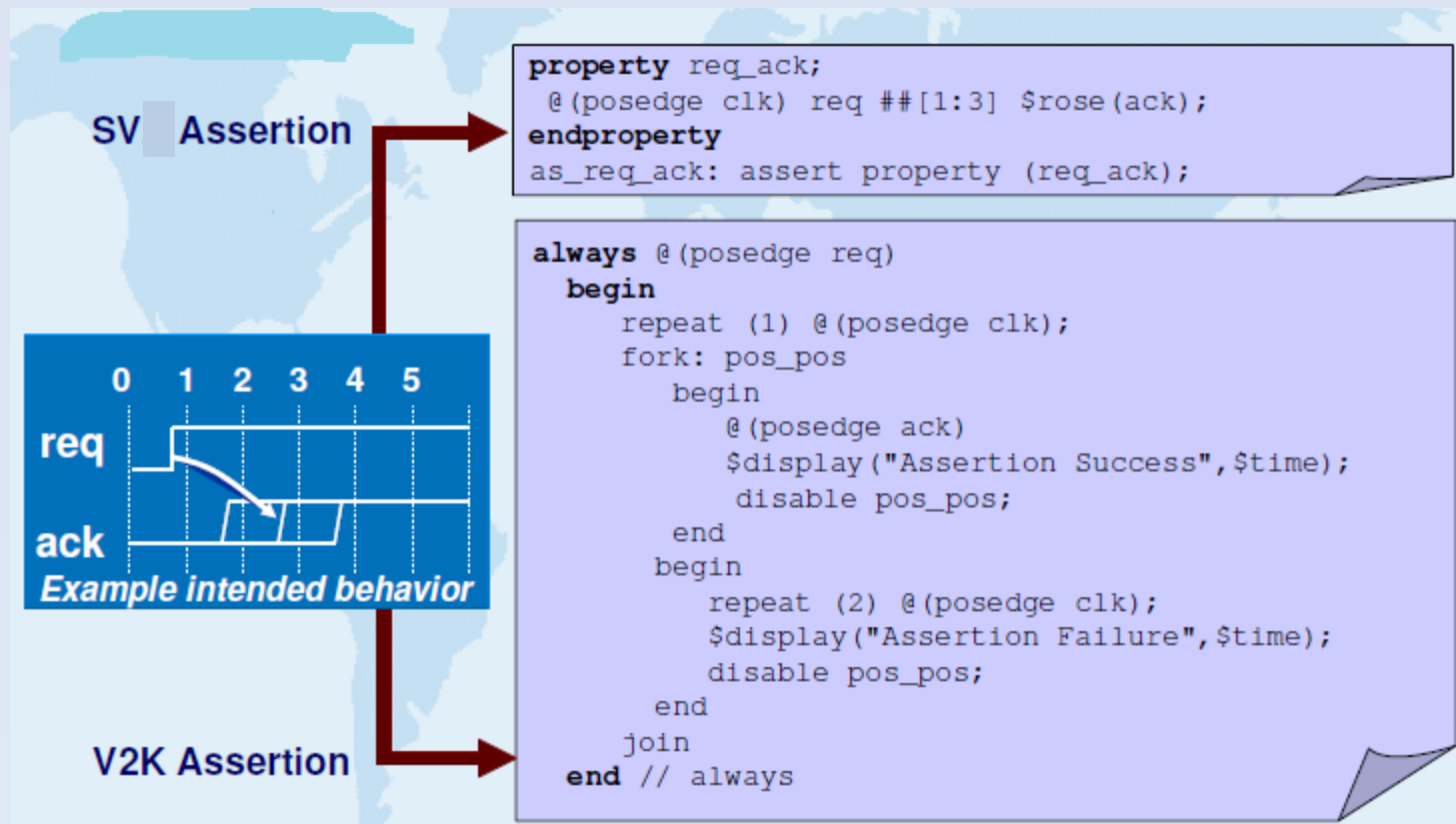
```
assert property (@ posedge mclk disable iff (!rst_n)
                !((count + push - pop) > `DEPTH));
```

Assertion to check for no underflow in a stack or queue structure

```
assert property (@ posedge mclk disable iff (!rst_n)
                !((count + push - pop) < 0));
```

Assertions-Example 4

"The "ack" signal must come within 1 to 3 cycles after the "req" signal is asserted"

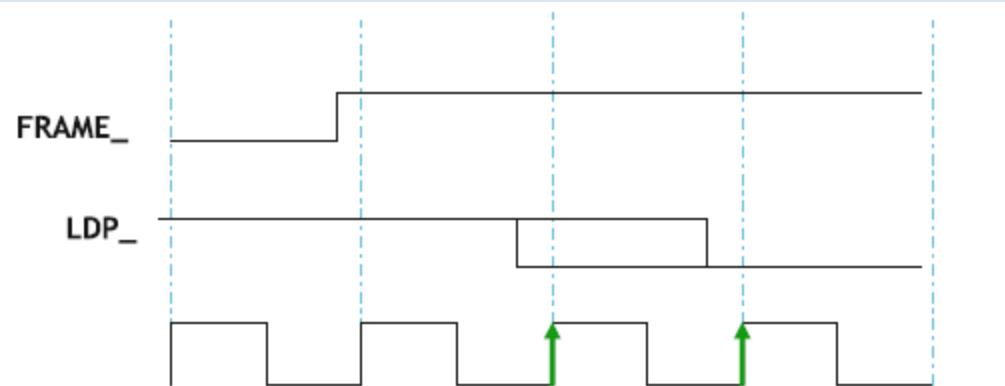


Assertions-Example 5

"When FRAME_ is de-asserted (i.e. it goes high), the Last Data Phase (LDP_) must be asserted (i.e. it must go low) within the next 2 clocks"

SystemVerilog Assertion

```
property ldpcheck;  
  @(posedge clk) $rose (FRAME_) |-> ##[1:2] $fell (LDP_);  
endproperty  
  
aP: assert property (ldpcheck) else $display("ldpcheck FAIL");  
cP: cover property (ldpcheck) $display("ldpcheck PASS");
```



When FRAME_ is de-asserted, LDP_ (last data phase) must be asserted within the next 2 clocks

Verilog 2001 Assertion

```
always @(posedge FRAME_)  
begin : ldpcheck  
  @(posedge clk);  
  fork  
    begin  
      @(negedge LDP_) disable ldpcheck;  
    end  
    begin  
      repeat (2) @(posedge clk); $display("ldpcheck FAIL");  
      disable ldpcheck;  
    end  
  join  
end
```

Assertions-Example 6

“Check whether a signal “sig1” goes high at any random point of time during the entire simulation duration”

```
property: is_sig1_high;  
    @(posedge clk) disable iff (rst) strong(##[1:$] $rose(sig1));  
endproperty  
  
as_sig1_chk: assert property (is_sig1_high)  
            else $display("sig1 never goes high. is_sig1_high FAILED");
```

The strong() sequence operator , i.e., strong(sequence_expression) evaluates to true, if and only if, there is a non-empty match of the sequence_expression

12. Direct Programming Interface (DPI)

SystemVerilog Direct Programming Interface (DPI)

- DPI is an interface between SystemVerilog and a foreign programming language. It consists of two separate layers: the SystemVerilog layer and a foreign language layer
- The motivation for DPI is two-fold:
 - The interface should allow a heterogeneous system to be built in which some components can be written in some foreign language (or languages) other than SystemVerilog
 - There is a practical need for an easy and efficient way to connect existing code, usually written in C or C++, without the knowledge and the overhead of VPI
- Neither the SystemVerilog compiler nor the foreign language compiler is required to analyze the source code in the other's language
- The SystemVerilog standard requires that its implementation shall support C protocols and linkage
- It is a mechanism that allows importing foreign language sub-routines, such as C-functions into SystemVerilog . Vice-versa, it also allows exporting SystemVerilog tasks and functions to foreign languages like C or C++
- The specification and the implementation of a component are clearly separated in DPI, and the actual implementation is transparent to the rest of the system
 - Thus any function can be treated as a black-box and implemented either in SystemVerilog or in the foreign language, without changing its corresponding function call

Two Layers of DPI

1) SystemVerilog Layer

- Does not depend on which language is used as the foreign language
- The actual function call protocol and argument passing mechanisms used in the foreign language are transparent and irrelevant to SystemVerilog
- SystemVerilog code shall look identical regardless of what code the foreign side of the interface is using
- The semantics of the SystemVerilog side of the interface is independent from the foreign side of the interface

For e.g.: SystemVerilog DPI-C allows direct inter-language function calls between SystemVerilog and any foreign programming language with a C function call protocol and linking model

2) Foreign Language Layer

- SystemVerilog defines a foreign language layer only for the C programming language
- It specifies how actual arguments are passed, how they can be accessed from the foreign code, how SystemVerilog-specific datatypes (such as logic and packed) are represented, and how they are translated to and from some predefined C-like types
- Different foreign languages can require that the SystemVerilog implementation shall use the appropriate function call protocol and argument passing and linking mechanisms
- The data types allowed for formal arguments and results of imported functions or exported functions are generally SystemVerilog types
- The SystemVerilog compiler or simulator shall generate and/or use the function call protocol and argument passing mechanisms required for the intended foreign language layer

Import and Export Sub-routines (Tasks and Functions)

- Sub-routines implemented in a foreign language can be called from and used in SystemVerilog; such sub-routines are referred to as *imported* tasks or functions
- Sub-routines implemented in SystemVerilog can be called from and used in a foreign language (C/C++ or System C); such sub-routines are referred to as *exported* tasks or functions

Properties:

- Every imported subroutine needs to be declared – this is called an *import declaration*
- The qualifier **ref** cannot be used in import declarations
- Imported sub-routines can have zero or more formal input, output and inout arguments
 - The formal input arguments cannot be modified
 - All SystemVerilog data types are allowed for formal arguments of imported functions
 - An imported function shall not assume any initial values of formal output arguments – it is undetermined and implementation dependent
 - An imported function can access the initial value of a formal inout argument
- An imported function shall complete its execution instantly and must consume zero simulation time. It can return a value or be declared as a void function. However, an imported task can consume time and never returns any value
- The memory space owned and allocated by the foreign code and SystemVerilog code are disjoint – each side is responsible for its own allocated memory
 - An imported function shall not free the memory allocated by SystemVerilog code, nor expect SystemVerilog code to free memory allocated by the foreign code

Import and Export Sub-routines (Tasks and Functions) Continued...

Properties (Cont'd):

- A call to an imported task can result in suspension of the currently executing thread. This occurs when an imported task calls an exported task, and the exported task executes a delay control, event control or wait statement
- In exported DPI subroutines, it is an error to declare formal arguments of dynamic array types
- Class member functions cannot be exported, but all other SystemVerilog functions can be exported
- It is illegal to call an exported task from within an imported function
- Result types of both imported and exported functions are restricted to small values, which include:
 - void, byte, shortint, int, longint, real, shortreal,chandle, and string
 - packed bit arrays up to 32 bits and all types that are eventually equivalent to packed bit arrays up to 32 bits
 - scalar values of type bit and logic

Pure and/or Context Sub-routines

- A function whose result depends solely on the values of its input arguments and with no side effects can be specified as *pure*
 - An imported task can never be declared pure
 - Only non-void functions with no output or inout arguments can be specified as pure
 - A pure function is assumed not to directly or indirectly (i.e., by calling other functions) perform the following:
 - a) File operations
 - b) Read or write anything including input/output (I/O), environment variables, objects from the operating system etc.
 - c) Access any persistent data like global or static variables
- An imported subroutine that is intended to call exported subroutines or to access SystemVerilog data objects other than its actual arguments (e.g., via VPI calls) shall be specified as *context*
 - Imported subroutine calls in SystemVerilog code are not instrumented unless the imported subroutine is specified as context
 - The SystemVerilog context of DPI export tasks and functions must be known when they are called, including when they are called by imports
 - A subroutine not specified as context shall not read or write any data objects from SystemVerilog other than its actual arguments
 - A context imported subroutine, however, can access (read or write) any SystemVerilog data objects by calling VPI or by calling an export subroutine
 - Only one export declaration is permitted per SystemVerilog function, and all export functions are always context functions

DPI Import and Export Declaration Syntax

DPI Import Declaration Syntax:

```
dpi_import_export ::=  
...  
| import dpi_spec_string [ dpi_function_import_property ] [ c_identifier = ] function_prototype;  
| import dpi_spec_string [ dpi_task_import_property ] [ c_identifier = ] task_prototype;  
...  
dpi_spec_string ::= "DPI-C" | "DPI"  
dpi_function_import_property ::= context | pure  
dpi_task_import_property ::= context  
function_prototype ::= function data_type_or_void function_identifier [ ( [ tf_port_list ] ) ]  
task_prototype ::= task task_identifier [ ( [ tf_port_list ] ) ]  
  
// The c_identifier is optional here. It defaults to function_identifier or task_identifier
```

DPI Export Declaration Syntax:

```
dpi_import_export ::=  
...  
| export dpi_spec_string [ c_identifier = ] function function_identifier;  
| export dpi_spec_string [ c_identifier = ] task task_identifier;  
...  
dpi_spec_string ::= "DPI-C" | "DPI"  
  
// The c_identifier is optional here. It defaults to function_identifier or task_identifier
```

Examples of Import and Export Declarations

```
//-----  
// import Examples  
//-----  
import "DPI-C" function void print_routine();  
  
// From the standard math library  
import "DPI-C" pure function real sin(real);  
  
// From the standard C library: memory management functions  
import "DPI-C" function chandle malloc(int size); // standard C function  
// chandle is a special SystemVerilog type that is used for passing C pointers as arguments to imported DPI functions  
import "DPI-C" function void free(chandle ptr); // standard C function  
  
// Miscellaneous  
import "DPI-C" function bit [31:0] get_ip_stimulus();  
import "DPI-C" context function void process_transaction(chandle elem, output logic [7:0] memarr [0:31]);  
import "DPI-C" task check_op_results(input string s, bit [1023:0] packet);  
import "DPI-C" function void f1(input logic [127:0]);  
  
//-----  
// export Examples  
//-----  
export "DPI-C" task MySVTask;  
export "DPI-C" function add_to_this;  
export "DPI-C" void function myfunc;  
export "DPI-C" Cfunc = function SVfunc;  
// The function is called SVfunc in SystemVerilog and Cfunc in C
```

Passing values through DPI – Type compatibility

A pair of matching type definitions is required to pass a value through DPI – the SystemVerilog definition and the C definition. SystemVerilog types which are directly compatible with C types are presented in the following table:

SystemVerilog Type	C Type
byte	char
int	int
shortint	short int
longint	long long
real	double
shortreal	float
chandle	void *
string	const char *
bit	unsigned char
logic/reg	unsigned char

There are SystemVerilog-specific types, including packed types (arrays, structures, unions), 2-state or 4-state, which have no natural correspondence in C. For these, the designers can choose the layout and representation that best suits their simulation performance.

Binary and Source Compatibility

- *Binary compatibility* means an application compiled for a given platform will work with every SystemVerilog simulator on that platform
- *Source compatibility* means an application needs to be re-compiled for each SystemVerilog simulator and implementation-specific definitions will be required for the compilation

Depending on the data types used for imported or exported functions, the C code can be binary-level or source-level compatible. Applications that are binary compatible:

- Do not use SystemVerilog packed types
- Do not mix SystemVerilog packed and unpacked types in the same data type
- Might use open arrays with both packed and unpacked dimensions.
 - An open array is an array with its packed, unpacked or both dimensions left unspecified. The symbol [] indicates open array dimensions.
 - Open arrays allow the use of generic code to handle different sizes
 - Please refer page 911 of [3] The SystemVerilog IEEE 1800-2012 LRM, for restrictions on the use of open arrays as formal arguments

The C-Layer “include” Files

The C-layer of the DPI provides two include files:

1) **svdpi.h**

- ❖ This file defines the canonical representation of 2-state (bit) and 4-state (logic) values, all basic types, all interface functions and passing references to SystemVerilog data objects
- ❖ It also provides function headers and defines a number of helper macros and constants
- ❖ The content of svdpi.h is implementation-independent -- all simulators use the same file
- ❖ Applications using only this include file are binary-compatible with all SystemVerilog simulators
- ❖ The source code of svdpi.h is present in Annex I, pages 1217-1225 of [3] The SystemVerilog IEEE 1800-2012 LRM

2) **svdpi_src.h (Deprecated)**

- ❖ This is a deprecated header that defines the implementation-dependent representation of packed values
- ❖ It defines the actual representation of 2-state and 4-state SystemVerilog packed arrays (of type bit or logic)
- ❖ Applications that need to include this file are not binary-level compatible, they are source-level compatible. They must be recompiled for each simulator on which they are to be run.

DPI Example 1

SV-CODE

```
module testdpi();

int retval;

import "DPI-C" context function int print_string(string str);
export "DPI-C" function add_to_this;

// The following unction is defined in SV, but used in C
function int add_to_this();
    add_to_this = 100;
    $info("In SV .... the add_to_this value = %0d", add_to_this);
    $info("Returning control to the C caller from SV");
endfunction

initial
begin
    // Calling the print_string() function below. This is defined in C
    retval = print_string("Miraфра Testing DPI Import Calls");
    $info("The integer value returned by the C function call print_string() is %0d", retval);
end
endmodule
```

DPI Example 1

Continued...

C-CODE

```
#include "svdpi.h"
#include "stdio.h"

// The following function is defined in SV, but called in C
extern int add_to_this();

// The following function is defined in C, but called in SV
int print_string(const char* fromsv)
{
    int result = 0;
    printf("\nIn C function");
    printf("\nThis statement is written in C inside a function called print_string()");
    printf("\nThe string passed from SV is --- %s\n", fromsv);
    printf("\nCalling the add_to_this() function in C\n");
    result = add_to_this(); // Calling the SV function
    printf("\nadd_to_this() defined in SV returned a value = %0d", result);
    result = result + 25;
    printf("\nResulting added value in C = %0d (expected value = 125)", result);
    printf("\nReturning control to SV caller from C\n");
    return(5775);
}
```

DPI Example 2

SV-CODE

```
program main;
export "DPI-C" task export_task;
import "DPI-C" context task import_task();

task export_task();
    $display("SV: Entered the export function . wait for some time : %0t\n", $time);
    #100;
    $display("SV: After waiting %0t\n", $time);
endtask

initial
begin
    $display("SV: Before calling import function %0t\n", $time);
    import_task();
    $display("SV: After calling import function %0t\n", $time);
end
endprogram
```

C-CODE

```
#include "svdpi.h"
extern void export_task();

void import_task()
{
    printf("C: Before calling export function\n");
    export_task();
    printf("C: After calling export function\n");
}
```


DPI Example 3

SV-CODE

```
module svmodule;  
import "DPI-C" context function void import_func();  
export "DPI-C" function export_func;  
  
initial  
    import_func();  
  
function void export_func();  
    $display("SV: This scope is %m\n");  
endfunction  
endmodule
```

C-CODE

```
#include "svdpi.h"  
#include "stdio.h"  
  
extern void export_func(void);  
  
void import_func()  
{  
    printf("C: This scope is %s\n", svGetNameFromScope(svGetScope()));  
    printf("C: Before calling function export_func\n");  
    export_func();  
    printf("C: After calling function export_func\n");  
}
```

DPI Example 4

Example from source

[http://www.testbench.in/
DP_07_DATA_TYPES.h
tml](http://www.testbench.in/DP_07_DATA_TYPES.html)

SV-CODE

```
program main;  
logic a;  
import "DPI" function void show(logic a);  
  
initial  
begin  
    a = 1'b0;  
    show(a);  
    a = 1'b1;  
    show(a);  
    a = 1'bx;  
    show(a);  
    a = 1'bz;  
    show(a);  
end  
endprogram
```

C-CODE

```
#include <stdio.h>  
#include <svdpi.h>  
  
void show(svLogic a) {  
    if (a == 0)  
        printf("From C: a is 0\n");  
    else if (a == 1)  
        printf("From C: a is 1\n");  
    else if (a == 2)  
        printf("From C: a is z\n");  
    else if (a == 3)  
        printf("From C: a is x\n");  
}
```

DPI Example 5

Example from source

[http://www.testbench.in/
DP_08_ARRAYS.html](http://www.testbench.in/DP_08_ARRAYS.html)

SV-CODE

```
program main;
int fxd_arr_1[8:3];
int fxd_arr_2[1:13];

import "DPI-C" context function void pass_array(input int dyn_arr[] );

initial
begin
    $display("SV: Passing fxd_arr_1 to C\n");
    pass_array(fxd_arr_1);
    $display("SV: Passing fxd_arr_2 to C\n");
    pass_array(fxd_arr_2);
end
endprogram
```

C-CODE

```
#include <stdio.h>
#include <svdpi.h>

void pass_array(const svOpenArrayHandle dyn_arr) {
    printf("C: Array Pointer is %0x\n", svGetArrayPtr(dyn_arr));
    printf("C: Lower index %0d\n", svLow(dyn_arr, 1));
    printf("C: Higher index %0d\n", svHigh(dyn_arr, 1));
    printf("C: Left index %0d\n", svLeft(dyn_arr,1), svRight(dyn_arr, 1));
    printf("C: Right index %0d\n", svRight(dyn_arr, 1));
    printf("C: Length of array %0d\n", svLength(dyn_arr,1));
    printf("C: Incremental %0d\n",svIncrement(dyn_arr,1));
    printf("C: Dimentions of Array %0d\n", svDimensions(dyn_arr));
    printf("C: Size of Array in bytes %0d\n", svSizeOfArray(dyn_arr));
}
```

Guidelines for Inclusion of Foreign Language Code into a SystemVerilog Application

The intention of these guidelines is to enable the re-distribution of C binaries in shared object form.

1) Location Independence

- All path names specified are intended to be location independent
- This is accomplished by using the switch **-sv_root**
- This switch can receive a single directory path name as the value, which is then prepended to any relative path name that has been specified
- In absence of this switch, or when processing relative file names before any -sv_root specification, the current working directory of the user shall be used as the default value

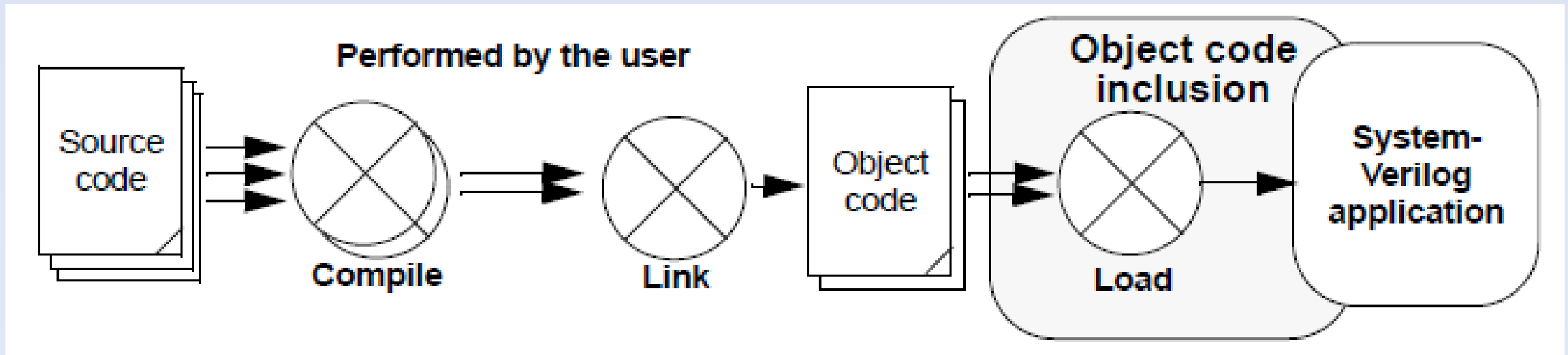
E.g.:

```
-sv_root /home/SoCproject/code  
-sv_root /home/usr10/shared_code  
-sv_root /home/share/project
```

Guidelines for Inclusion of Foreign Language Code into a SystemVerilog Application Continued...

2) Object Code Inclusion

- Compiled object code is required for cases where the compilation and linking of source code are fully handled by the user
- Therefore, the created object code only needs to be loaded in order to integrate the foreign language code into a SystemVerilog application



Guidelines for Inclusion of Foreign Language Code into a SystemVerilog Application Continued...

All SystemVerilog applications support integration of foreign language code in object code form. Compiled object code can be specified by any of the following two methods:

a) By an entry in a bootstrap file

- ❖ Its location will be specified with one instance of the switch **-sv_liblist pathname**
- ❖ This switch can be used multiple times to define the usage of multiple bootstrap files

a) By specifying the file with one instance of the switch **-sv_lib *pathname_without_extension* (i.e., the file name shall be specified without the platform-specific extension)**

- ❖ The SystemVerilog application is responsible for appending the appropriate extension for the actual platform
- ❖ This switch can be used multiple times to define multiple libraries holding object code

Any library shall only be loaded once. There are a list of conditions that apply, and they are listed on pages 1227-1228 of [3] The SystemVerilog IEEE 1800-2012 LRM.

Bootstrap File Syntax

- The first line contains the string `#!SV_LIBRARIES`
- An arbitrary amount of entries follow, one entry per line, where every entry holds exactly one library location
- Each entry consists only of the `pathname_without_extension` of the object code file to be loaded and can be surrounded by an arbitrary number of blanks; at least one blank shall precede the entry in the line. The value `pathname_without_extension` is equivalent to the value of the switch `-sv_lib`

E.g.:

```
#!SV_LIBRARIES  
commonlibs/lib10  
SoCproj/clibs/lib2code
```

Guidelines for Inclusion of Foreign Language Code into a SystemVerilog Application Continued...

Examples of specification of compiled object code

- a) If the path-name root has been set by the switch `-sv_root` to `/home/user` and the following object files need to be included:

```
/home/user/myclibs/lib1.so  
/home/user/myclibs/lib3.so  
/home/user/proj1/clibs/lib4.so  
/home/user/proj3/clibs/lib2.so
```

then use either of the methods in [Figure J.2](#). Both methods are equivalent.

```
#!/SV_LIBRARIES  
myclibs/lib1  
myclibs/lib3  
proj1/clibs/lib4  
proj3/clibs/lib2
```

Bootstrap file method

```
...  
-sv_lib myclibs/lib1  
-sv_lib myclibs/lib3  
-sv_lib proj1/clibs/lib4  
-sv_lib proj3/clibs/lib2  
...
```

Switch list method

Figure J.2—Using a simple bootstrap file or a switch list

Guidelines for Inclusion of Foreign Language Code into a SystemVerilog Application

Continued...

Examples of specification of compiled object code (Cont'd...)

Examples from source: [3] The SystemVerilog IEEE 1800-2012 LRM, Page 1229

b) If the current working directory is /home/user, using the series of switches shown in [Figure J.3](#) (left column) results in loading the following files (right column):

<pre>-sv_lib svLibrary1 -sv_lib svLibrary2 -sv_root /home/project2/shared_code -sv_lib svLibrary3 -sv_root /home/project3/code -sv_lib svLibrary4</pre>	<pre>/home/user/svLibrary1.so /home/user/svLibrary2.so /home/project2/shared_code/svLibrary3.so /home/project3/code/svLibrary4.so</pre>
Switches	Files

Figure J.3—Using a combination of -sv_lib and -sv_root switches

c) Further, using the set of switches and contents of bootstrap files shown in [Figure J.4](#):

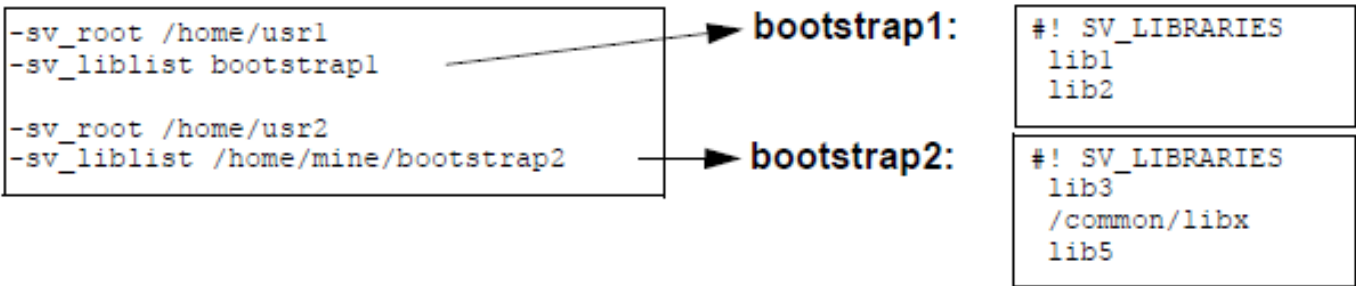


Figure J.4—Mixing -sv_root and bootstrap files

results in loading the following files:

```
/home/usr1/lib1.ext
/home/usr1/lib2.ext
/home/usr2/lib3.ext
/home/usr2 /common/libx.ext
/home/usr2/lib5.ext
```

where ext stands for the actual extension of the corresponding file.

Disabling DPI Tasks and Functions

It is possible for a **disable** statement to disable a block that is currently executing a mixed-language (SV & C) code call chain.

When a DPI import subroutine is disabled, the C code is required to follow a simple disable protocol. The protocol gives the C code the opportunity to perform any necessary resource clean-up, such as closing open file handles, closing open VPI handles, or freeing heap memory.

An imported subroutine is said to be in the disabled state when a **disable** statement somewhere in the design targets either it or a parent for disabling.

An imported subroutine can only enter the disabled state immediately after the return of a call to an exported subroutine.

An important aspect of the protocol is that disabled import tasks and functions shall programmatically acknowledge that they have been disabled. A subroutine can determine that it is in the disabled state by calling the API function `svIsDisabledState()`.

Disabling DPI Tasks and Functions Continued...

The protocol is composed of the following items:

- 1) When an exported task returns due to a disable, it returns a value 1. Otherwise, it returns 0.
- 2) When an imported task returns due to a disable, it returns a value 1. Otherwise, it returns 0.
- 3) Before an imported function returns due to a disable, it shall call the API function `svAckDisabledState()`.
- 4) Once an imported subroutine enters the disabled state, it is illegal for the current function call to make any further calls to exported subroutines.

Items (2), (3) and (4) are mandatory behavior for imported DPI tasks and functions. It is the responsibility of the DPI programmer to correctly implement the behavior.

Item (1) is guaranteed by SystemVerilog simulators.

In addition, simulators shall implement checks to verify that items (2), (3) and (4) are correctly followed by imported tasks and functions. If any protocol item is not correctly followed, a fatal simulation error is issued.

Disabling DPI Tasks and Functions Continued...

The foreign language side of the DPI contains a disable protocol that is realized by user code working together with a simulator. The disable protocol allows for foreign models to participate in SystemVerilog disable processing. The participation is done through special return values for DPI tasks and special API calls for DPI functions.

The special return values do not require a change in call syntax of either import or export DPI tasks in the SystemVerilog code. While the return value for an export task is guaranteed by the simulator, for the import task the return value has to be ensured by the DPI application.

If an exported task itself is the target of a disable, its parent imported task is not considered to be in the disabled state when the exported task returns. In such cases, the exported task shall return value 0, and calls to `svIsDisabledState()` shall return 0 as well.