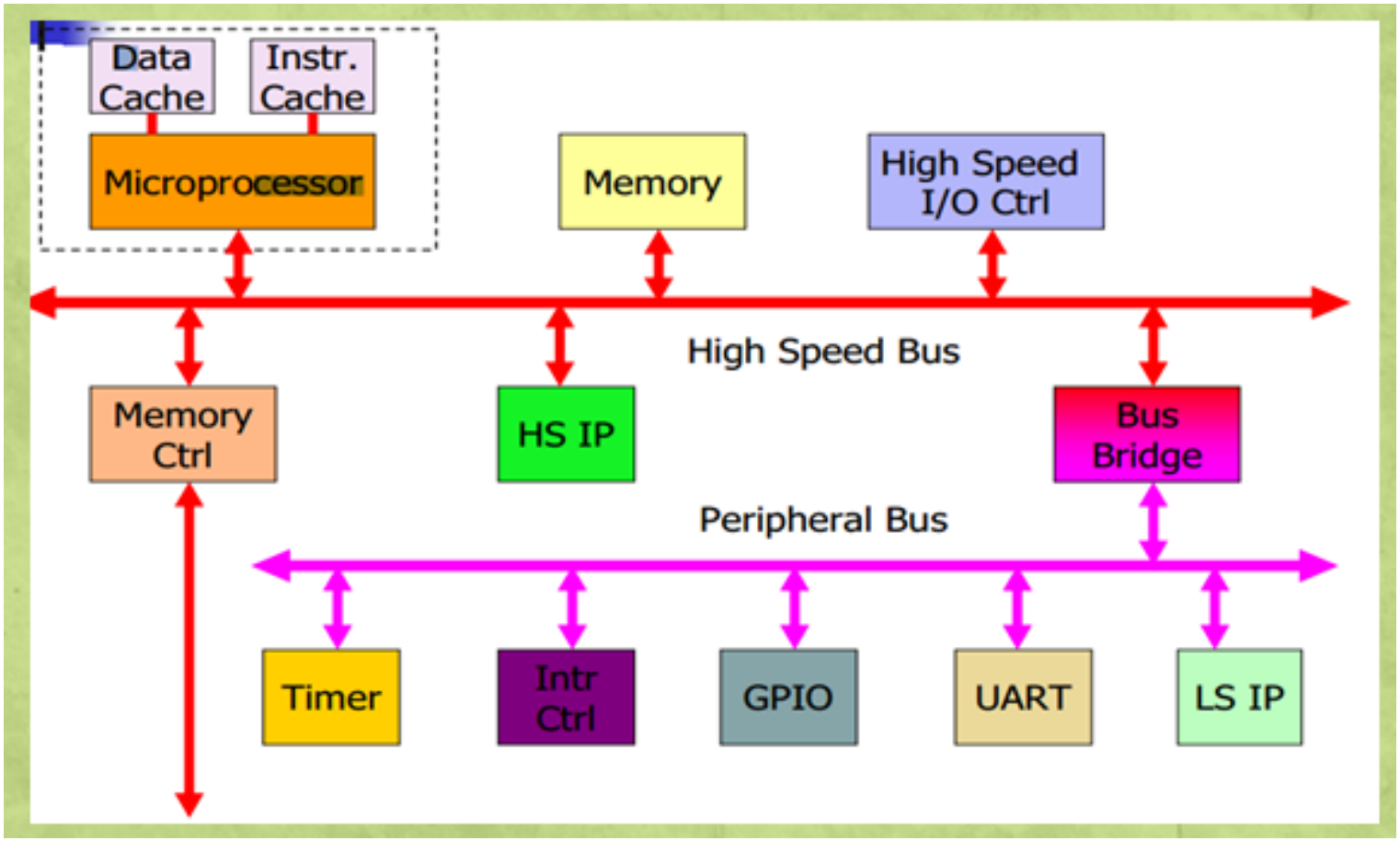
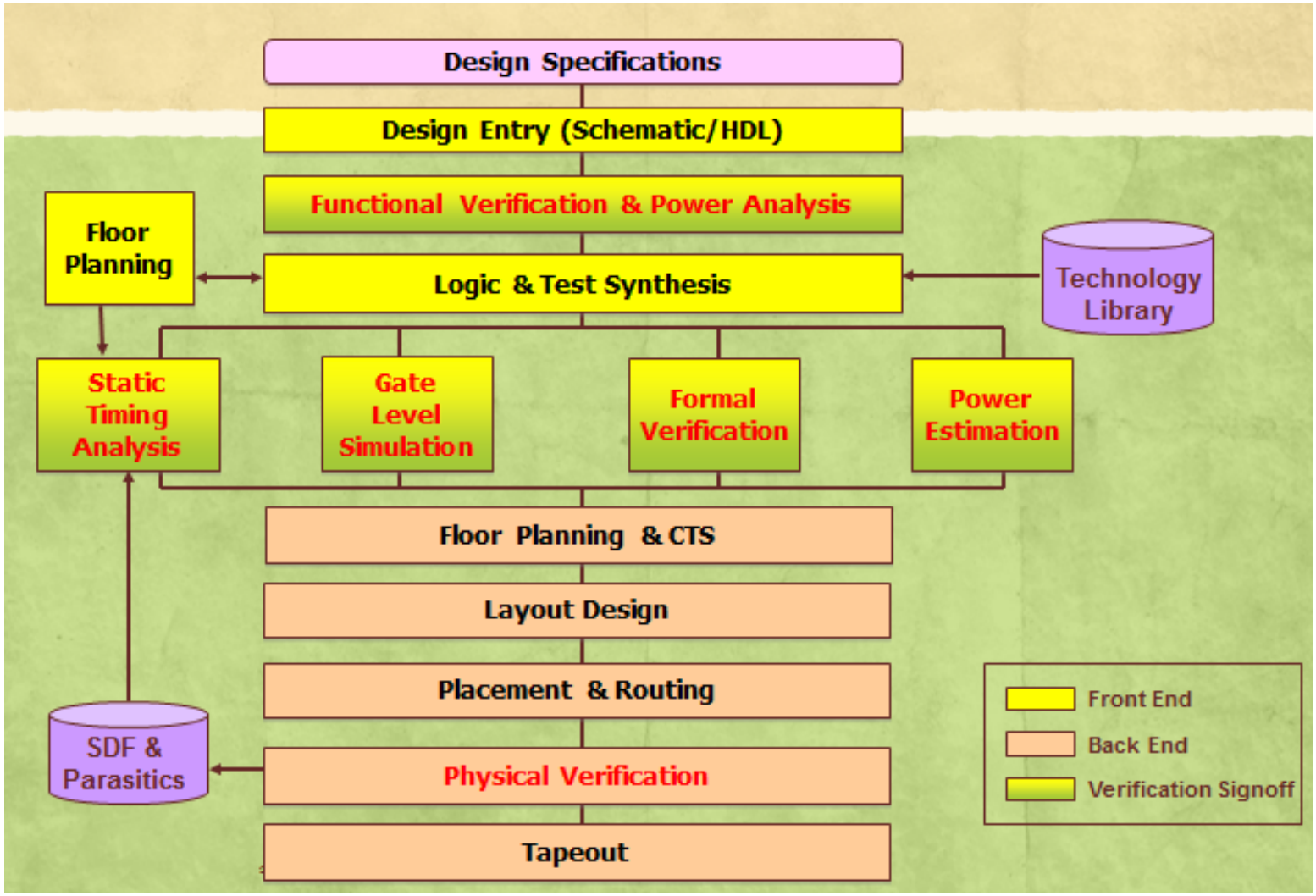
Verification Using System Verilog (IEEE standard1800)

System on Chip



VLSI/Chip Design Flow



What is SystemVerilog ?

- Hardware Description Language (HDL) like Verilog and VHDL are used to describe hardware behavior so that it can be converted to digital blocks made up of combinational gates and sequential elements.

- In order to verify that the hardware description in HDL is correct, there is a need for a language with more features in OOP that will support complicated testing procedures and is often called a Hardware Verification Language.

- SystemVerilog is an extension of Verilog with many such verification features that allow engineers to verify the design using complex testbench structures and random stimuli in simulation.

Why is Verilog not preferred ?

- Back in the 1990's, Verilog was the primary language to verify functionality of designs that were small, not very complex and had less features.
- As design complexity increases, so does the requirement of better tools to design and verify it.
- SystemVerilog is far superior to Verilog because of its ability to perform constrained random stimuli, use OOP features in testbench construction, functional coverage, assertions among many others.

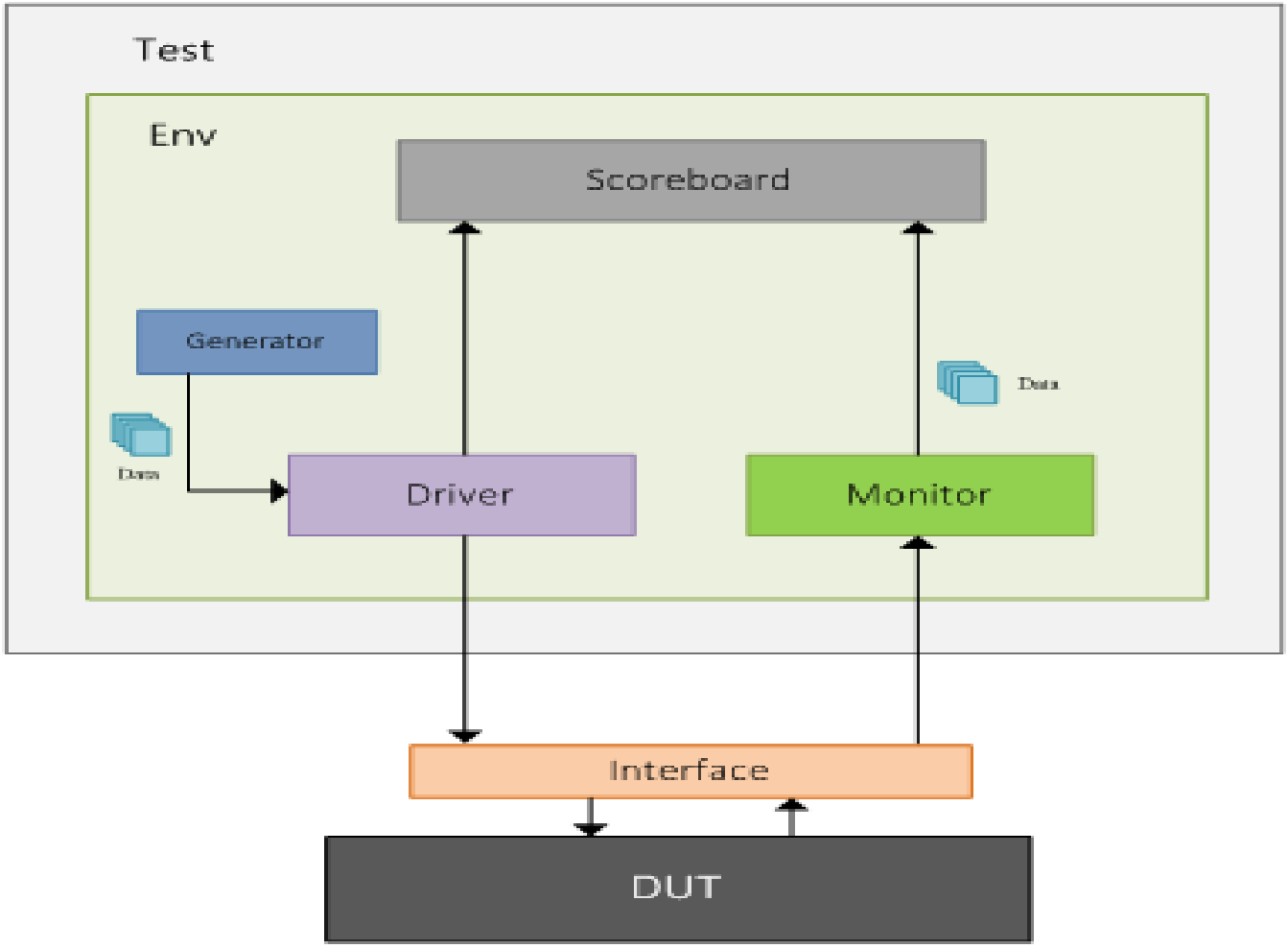
What is verification ?

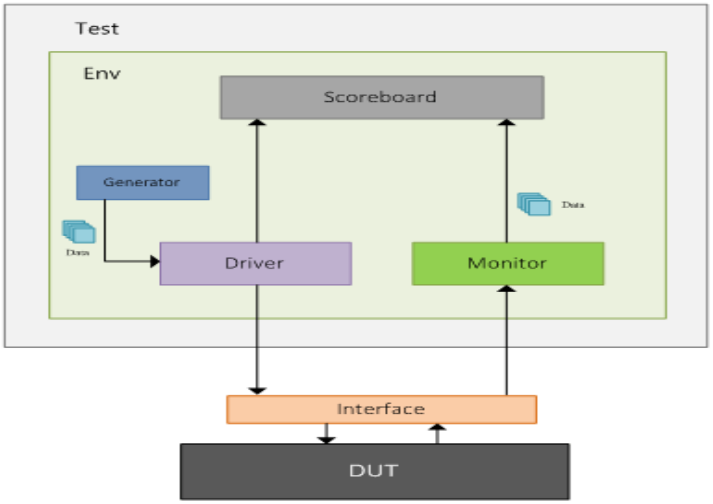
- Verification is the process of ensuring that a given hardware design works as expected.
- Chip design is a very extensive and time consuming process and costs millions to fabricate.
- Functional defects in the design if caught at an earlier stage in the design process will help save costs.
- If a bug is found later on in the design flow, then all of the design steps have to be repeated again which will use up more resources, money and time.
- If the entire design flow has to be repeated, then its called a *respin* of the chip.

What is the purpose of a testbench ?

- A testbench allows us to verify the functionality of a design through simulations. It is a container where the design is placed and driven with different input stimulus.
- Generate different types of input stimulus
- Drive the design inputs with the generated stimulus
- Allow the design to process input and provide an output
- Check the output with expected behavior to find functional defects
- If a functional bug is found, then change the design to fix the bug
- Perform the above steps until there are no more functional defects

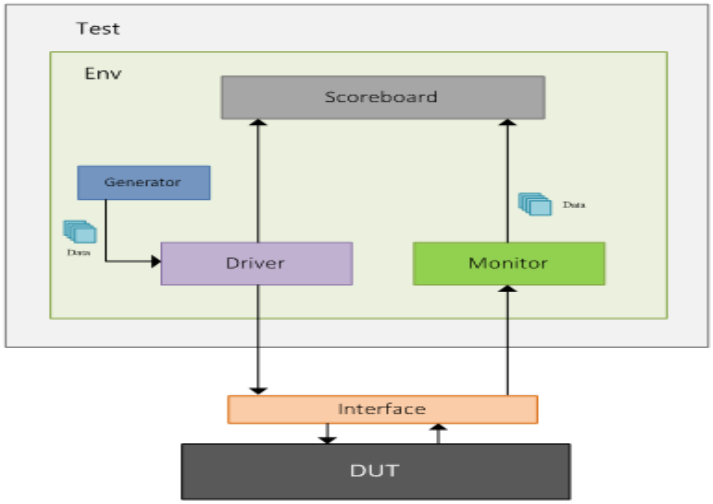
Components of a testbench





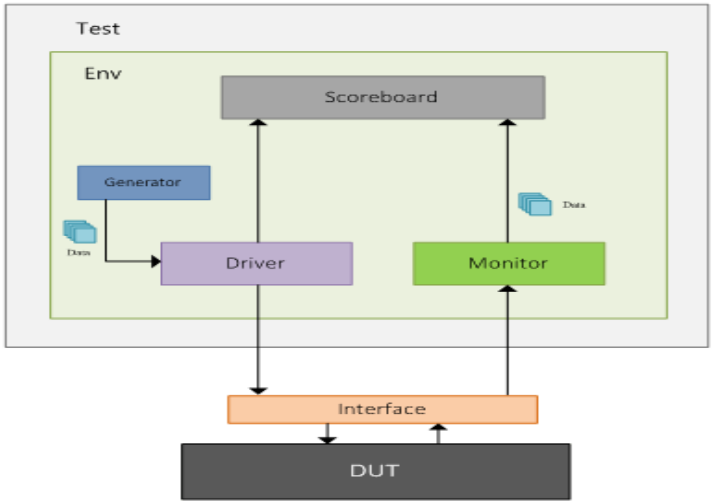
What is DUT ?

- **DUT** stands for *D*esign *U*nder *T*est and is the hardware design written in Verilog or VHDL.
- DUT is a term typically used in post validation of the silicon once the chip is fabricated.
- In pre validation, it is also called as *D*esign *U*nder *V*erification, DUV in short.



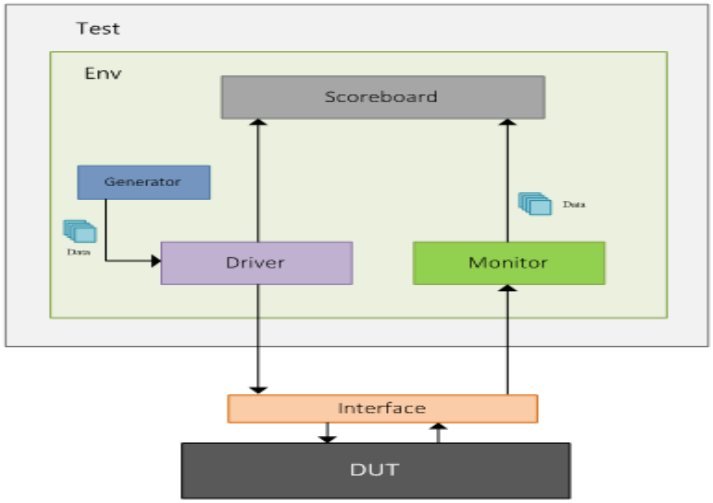
What is an interface ?

- If the design contained hundreds of port signals it would be cumbersome to connect, maintain and re-use those signals.
- Instead, we can place all the design input-output ports into a container which becomes an *interface* to the DUT.
- The design can then be driven with values through this interface.



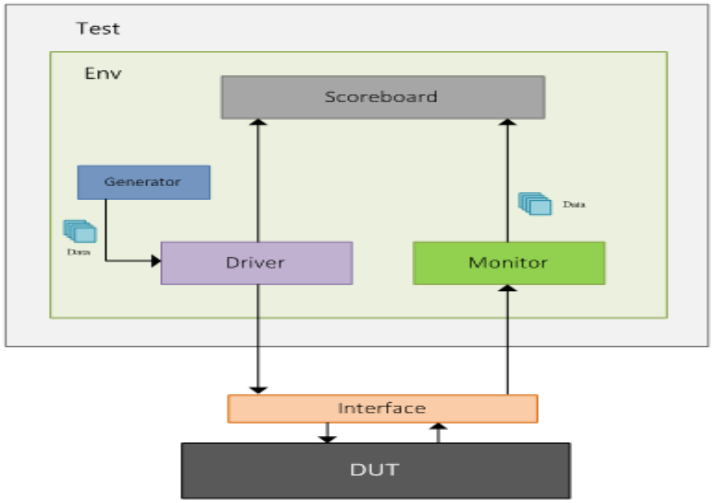
What is a driver ?

- The *driver* is the verification component that does the pin-wiggling of the DUT, through a task defined in the interface.
- When the driver has to drive some input values to the design, it simply has to call this pre-defined task in the interface, without actually knowing the timing relation between these signals.



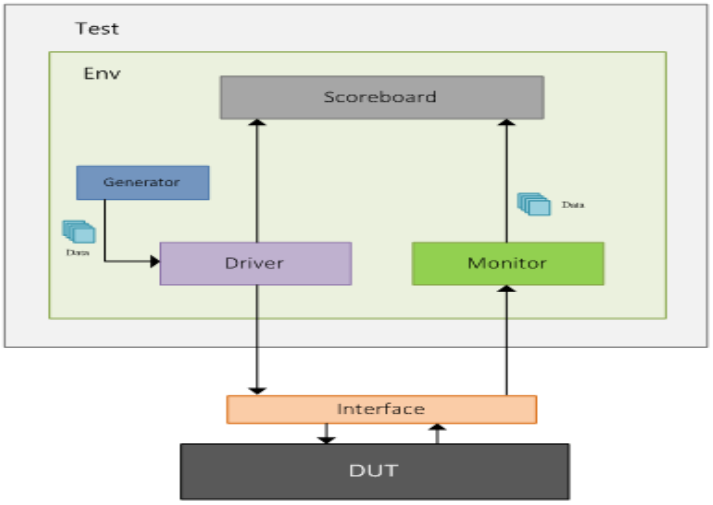
What is a generator

- The *generator* is a verification component that can create valid data transactions and send them to the driver.
- The driver can then simply drive the data provided to it by the generator through the interface.
- Data transactions are implemented as class objects shown by the blue squares in the image above.
- It is the job of the driver to get the data object and translate it into something the DUT can understand.



Why is a monitor required?

- Until now, how data is driven to the DUT was discussed. But that's only half way through, because our primary aim is to verify the design.
- The DUT processes input data and sends the result to the output pins.
- The *monitor* picks up the processed data, converts it into a data object and sends it to the scoreboard.



What is the purpose of a scoreboard?

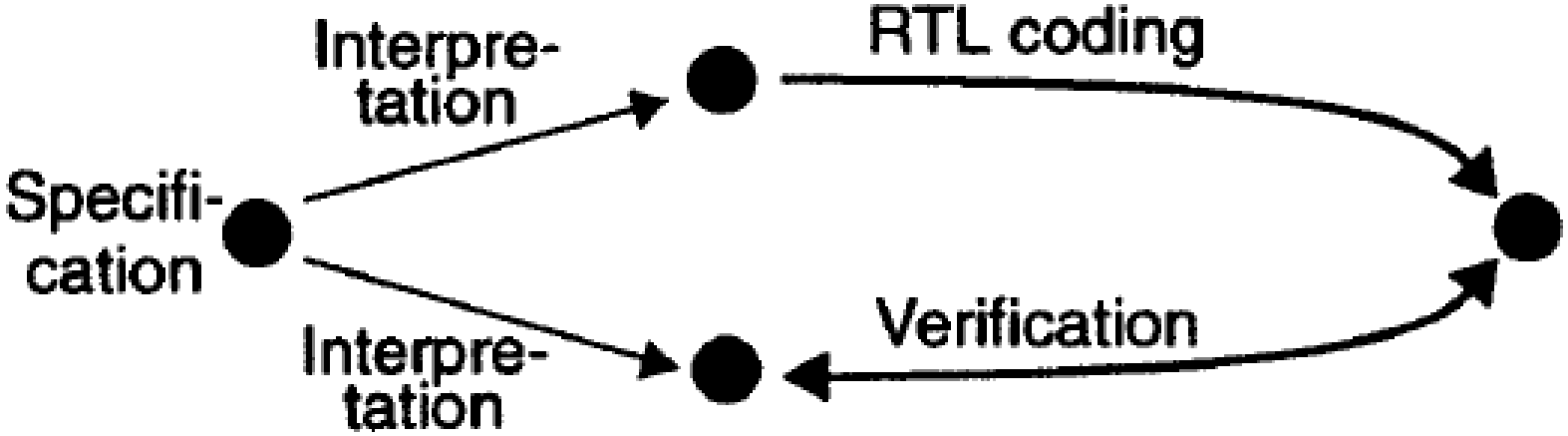
- The *Scoreboard* can have a reference model that behaves the same way as the DUT.
- This model reflects the expected behavior of the DUT.
- Input sent to the DUT is also sent to this reference model.
- So if the DUT has a functional problem, then the output from the DUT will not match the output from our reference model.
- Comparison of outputs from the design and the reference model will tell us if there is a functional defect in the design.

Why is an environment required and What does the test do ?

- It makes the verification more flexible and scalable because more components can be plugged into the same environment for a future project.
- The test will instantiate an object of the environment and configure it the way the test wants to.
- Remember that we will most probably have thousands of tests and it is not feasible to make direct changes to the environment for each test.
- Instead we want certain knobs/parameters in the environment that can be tweaked for each test. That way, the test will have a higher control over stimulus generation and will be more effective.
- Here, we have talked about how a simple testbench looks like. In real projects, there'll be many such components plugged in to do various tasks at higher levels of abstraction.
- If we had to verify a simple digital counter with maximum 50 lines of RTL code, yea, this would suffice. But, when complexity increases, there will be a need to deal with more abstraction.

What is verification

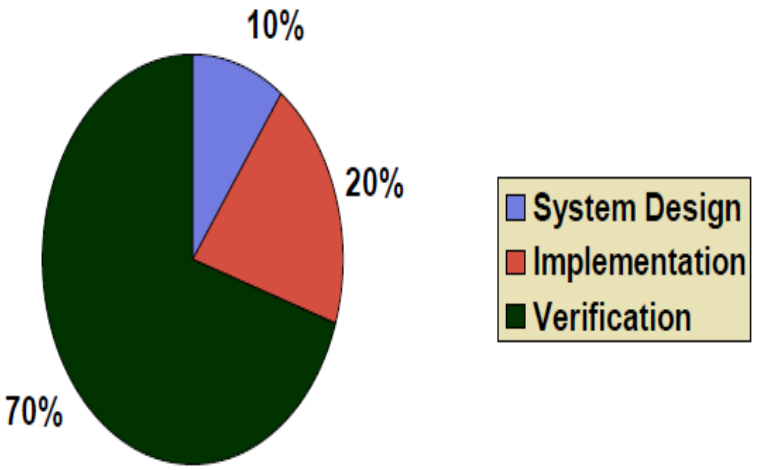
- Process of demonstrating functional correctness of a design.



What if we don’t verify?

- Is it possible to guarantee a design without verification?
  - Incorrect/Insufficient Specifications
  - Misinterpretation and Misunderstandings
  - Incorrect interaction between IPs/Cores
  - Unexpected behavior of the system

The Importance of Verification



- Bug escapes to silicon can be costly including re-spin
- 70 % of design cycles is spent in verifying design
- Ever increasing complexity of designs makes this harder
- Hence Verification is always on critical path for any product design

Verification Space

- **What to Verify?**
  - Functional Verification
  - Timing Verification
  - Performance Verification
- **How to Verify?**
  - Simulation based Verification
  - Emulation/FPGA based Verification
  - Formal Verification
  - Semi-Formal Verification
  - HW/SW Co- Verification

What is a Verification Plan?

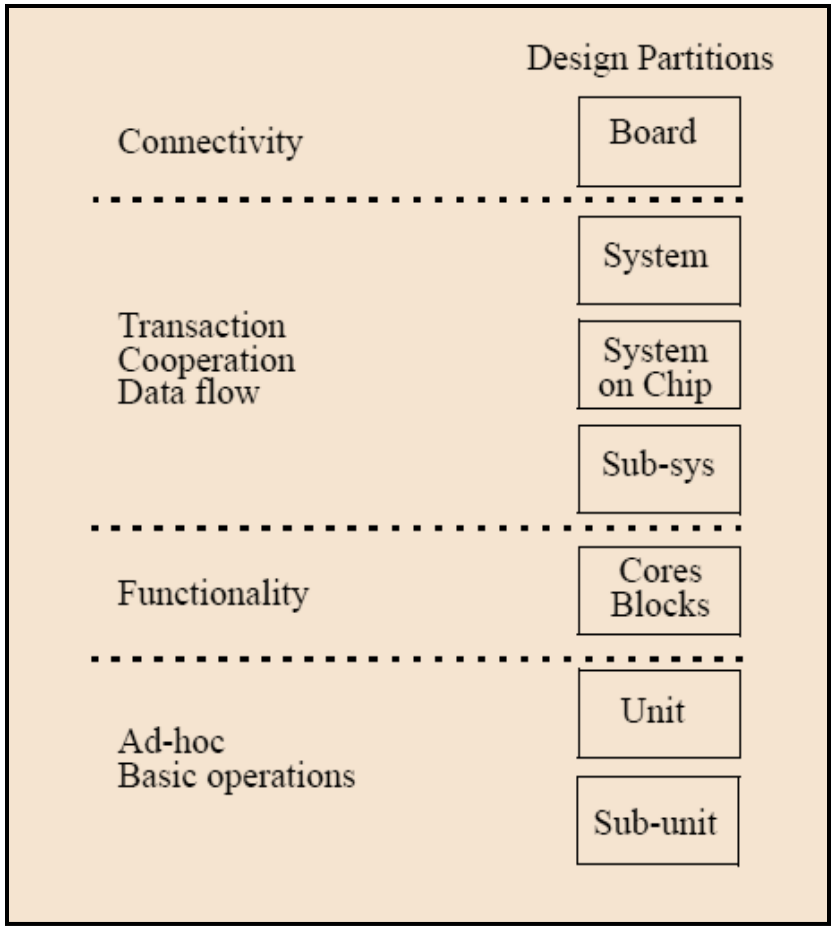
- Specification document for verification effort
  - Mechanism to ensure all essential features are verified as needed
    - What to verify ?
      - Features and under what conditions to verify them
    - How to verify ?
      - What methodologies to use – Formal, Checking, Coverage etc.
      - What should be - Stimulus, Checkers, Coverage
    - Priority for features to be verified

Verification Approaches

- Black-Box
  - ☺ Verification without knowledge of design implementation
  - ☹ Lack of visibility and observability
  - ☺ Tests are independent of implementation
  - ☹ Impractical in todays designs
- White Box
  - ☺ Intimate knowledge of design implementation
  - ☺ Full visibility and observability
  - ☹ Tests are tied to a specific implementation
- Grey Box
  - ☺ Compromise between above approaches

Levels of Verification

- Each level of Verification will be suited for a specific objective
  - Lower levels have better controllability and visibility
- Block level Verification helps designs to be verified independently and in parallel
- System Level Verification focuses more on interactions



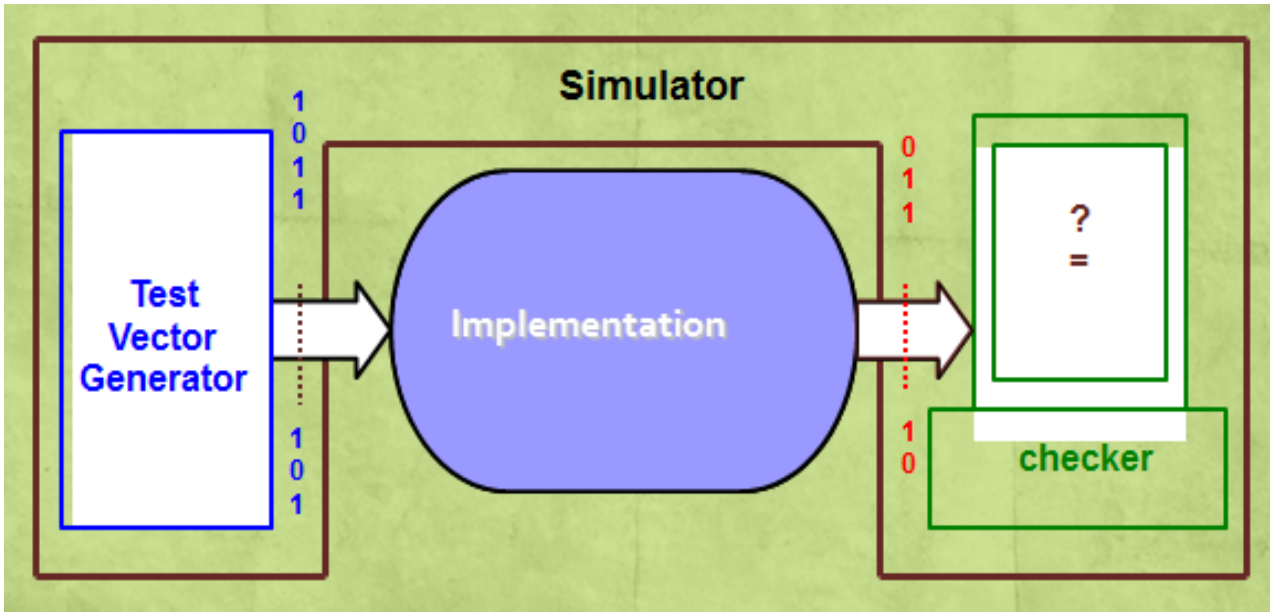
Verification Metrics

- Metrics
  - Measurements that provide visibility into a process
  - Help identify issues so corrective action can be taken
  - Historical trend data facilitates future project planning
- Metrics can help us
  - Understand how the design was verified
  - Measure the productivity of various stimulus methods
  - Ensure that the desired checkers are in place and performing checks
  - Ensure regressions are run effective and productive
  - Coverage metrics helps in identifying stimulus holes
  - Bug metrics can provide insight into the simulation, coverage and overall progress

Verification Methods

- Simulation Based
- Formal Verification
- Semi Formal Verification
- Assertions

Simulation Based Verification

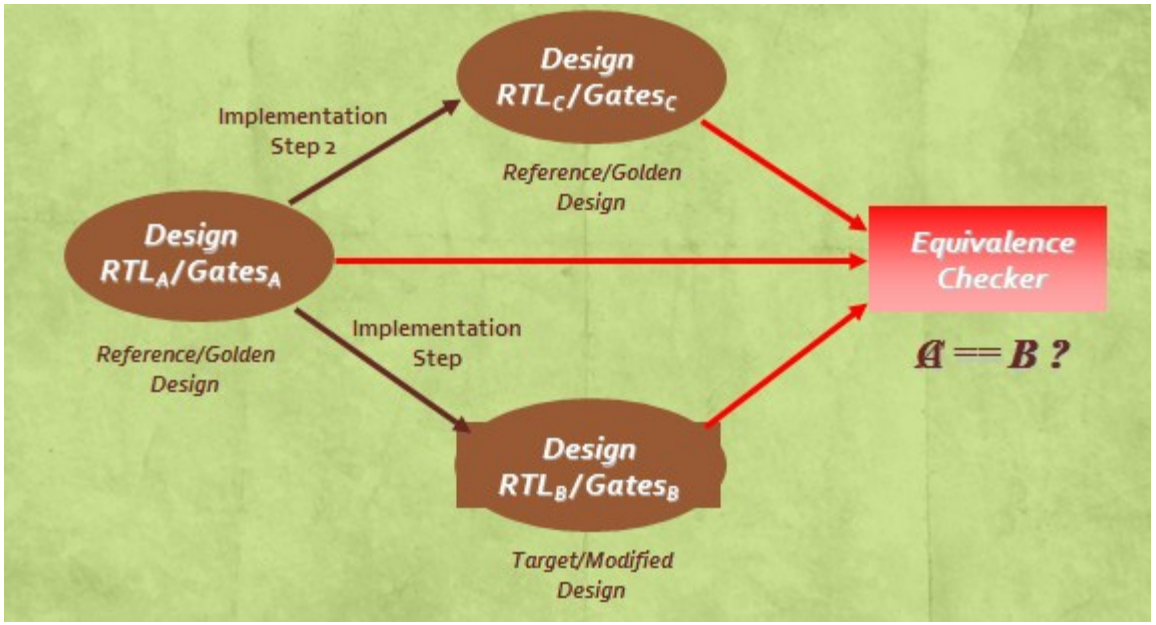


- Generate Input sequences
  - Directed/Random/Constrained-Random sequences
- Generate expected output sequence (golden)
- Simulate the DUT with the generated input sequences
- Verify DUT output against golden output
- Use Coverage Metrics for ensuring completeness

Formal Verification

- What is Formal Verification?
  - It is the method by which we prove or disprove a design implementation against a formal specification or property
  - Uses mathematical reasoning
  - It algorithmically and exhaustively explores all possible input values over time.
  - works well for small designs where the number of inputs, outputs and states is small

Formal - Equivalence Checking



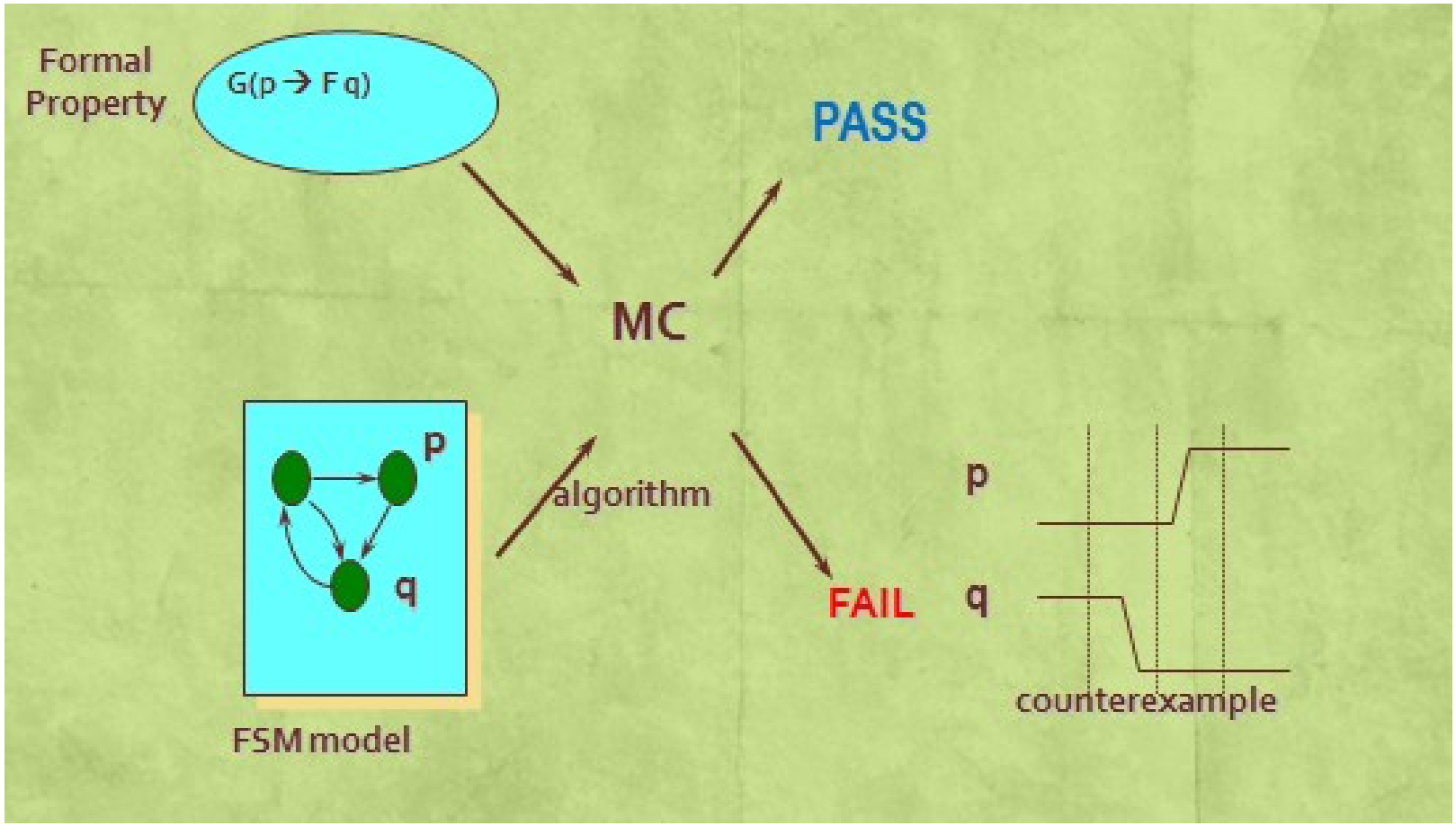
- Equivalence checker checks the logical equivalence of the final version of the netlist with the RTL

- Equivalence checker checks for functional equivalence and not functional correctness

Formal – Model Checking

- Exhaustive search of FSM for state/property violations
- Two inputs to the algorithm
  - A finite state transition system (FSM) representing the implementation
  - A formal property representing the specification
- The algorithm checks whether the FSM *"models"* the property correctly

Formal – Model Checking



Advantages of Formal Verification

- Covers exhaustive state space, something difficult to achieve by simulations
- Does not require generation of input stimulus, since exhaustive stimulus can be generated automatically by tool
- No need to generate expected output sequences
- Correctness of design guaranteed mathematically

Disadvantages of Formal Verification

- Scalability
  - each added flop doubles the state space. Limited to designs with small state spaces like interfaces, small FSMs.

- What properties to verify?
  - For some methods, needs the specification to be translated into properties to verify that properties hold under all inputs and all states.

Semi-Formal Verification

- Best of both worlds (Simulation & Formal)
- Use simulation to reach interesting states in the design
- Then fork off formal to do exhaustive analysis around the interesting state
- Finds bugs sitting deep in the design
- Useful for bug hunting

Assertion Based Verification

❖What is Assertion?

- An Assertion is a statement about a design's intended behavior ,which must be verified

❖Benefits of Assertions:

- Improving Observability and debug ability
- Improving integration through correct usage checking
- Improving verification efficiency
- Improving communication through documentation
- Useful in both static and dynamic simulations

Assertion Types

- Immediate Assertion

```
assert (A == B) else $error("It's gone wrong");
```

- Concurrent Assertion

```
property p1;
  @(posedge clk) disable iff (Reset) not b ##1 c;
endproperty
assert property (p1) else  $error(“B ##1C failed”);
```

Assertion Based Verification

- Using Assertions in Formal Verification
  - Formally specify all functional requirements and behavior of design in a language
  - Use a procedure to prove that all specified properties hold true
- Using Assertions in Dynamic Simulation
  - Specified properties are useful as checks and for coverage

Summary

- Simulation based Verification is used to find most bugs
- Formal verification is useful in select areas
- Assertions are a great value add

Directed vs Random Testing

- What is Directed Testing ?
- What is Random Testing ?

Directed vs Random Testing

Directed

- ☹ Can only cover scenarios thought through planning
- ☹ High maintenance cost
- ☺ Works good when condition space is finite
- ☺ No need of extensive coverage coding

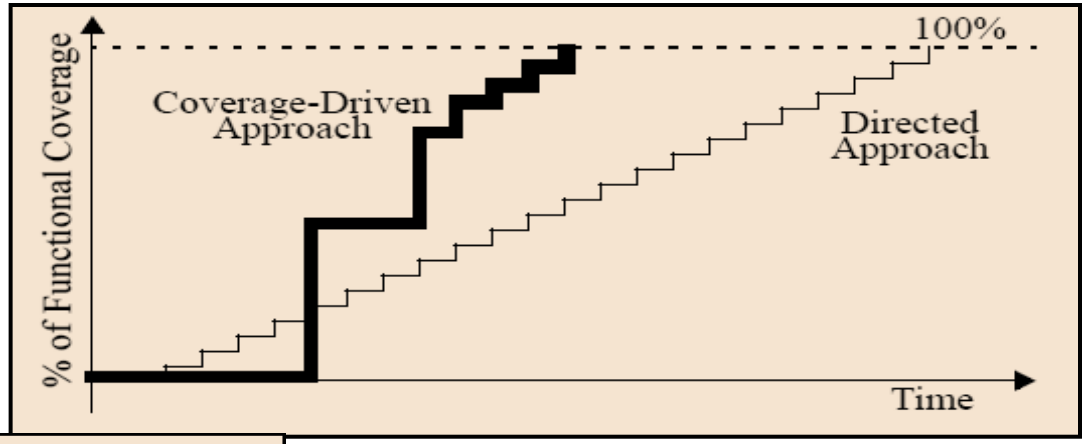
Constrained Random

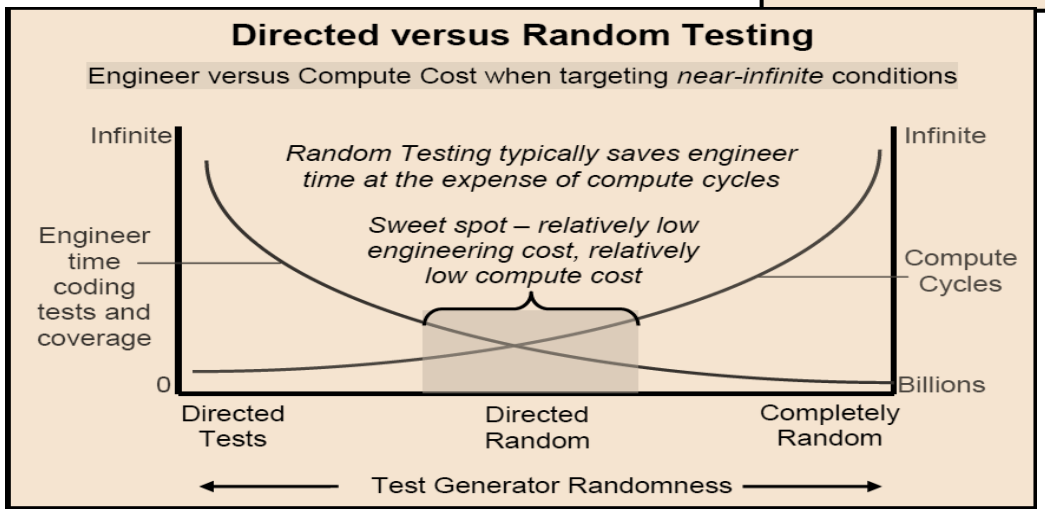
- ☹ High ramp up time to
- ☹ Need Coverage
- ☺ Deep User control. Complex Test generator
- ☺ Best balance between engineer time and compute time
- ☺ Can have static and dynamic (during test run) randomness

Pure Random

- ☹ Need infinite compute cycles to cover all condition space
- ☺ Less user control. Simple to build generator

Directed vs. Random Testing





Coverage

- What is Coverage?
  - Coverage is the metric of completeness of verification
- Why coverage?
  - Verification is based on samples
    - Cannot run all possible tests (2^n) to cover full space
    - Need to know that all areas of the design have been verified
- Functional Coverage and Code Coverage

Code Coverage

- Statement
  - Has each statement of the source code been executed?
- Branch
  - Has each control structure been evaluated to both true and false?
  - Example: if, case, while, repeat, forever, for, loop
- Condition
  - Has each boolean sub-expression evaluated both to true and false?

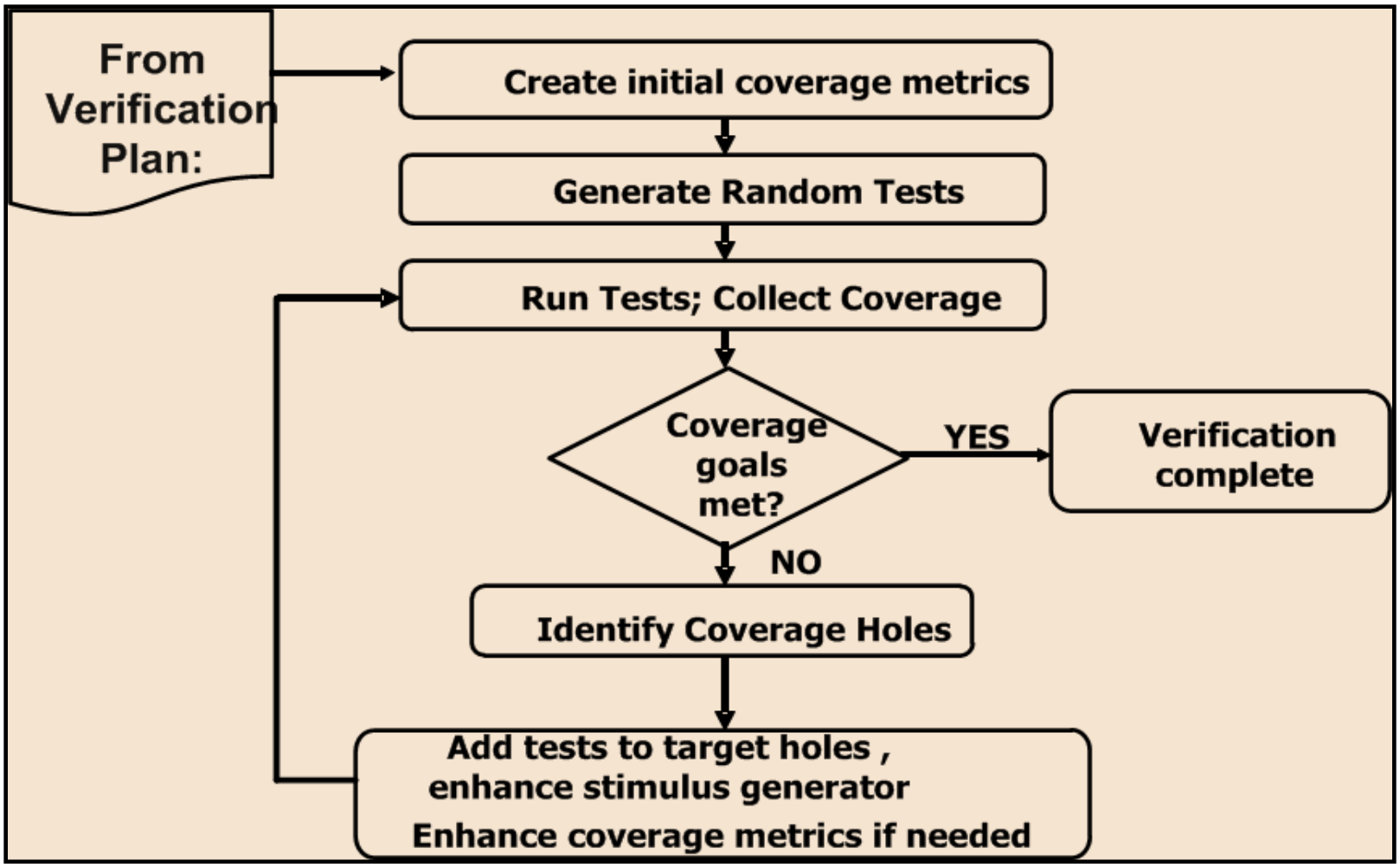
Code Coverage

- Expression
  - Covers the RHS of an assignment statement
  - Example: **x <= a xor (not b);**
- Toggle
  - Covers logic node transitions
  - Standard – covers 0->1 and 1->0 transitions
  - Extended – covers 0<->1, z<->1 and z<->0 transitions
- FSM
  - State coverage and FSM Arc coverage

Functional Coverage

- Covers the functionality of the DUT
- Functional Coverage is derived from design specification
  - DUT Inputs – Are all input operations injected?
  - DUT Outputs/functions – Are all responses seen from every output port?
  - DUT internals –
    - Are all interested design events being verified?
      - e.g. FIFO fulls, arbitration mechanisms etc.

Coverage Driven Verification

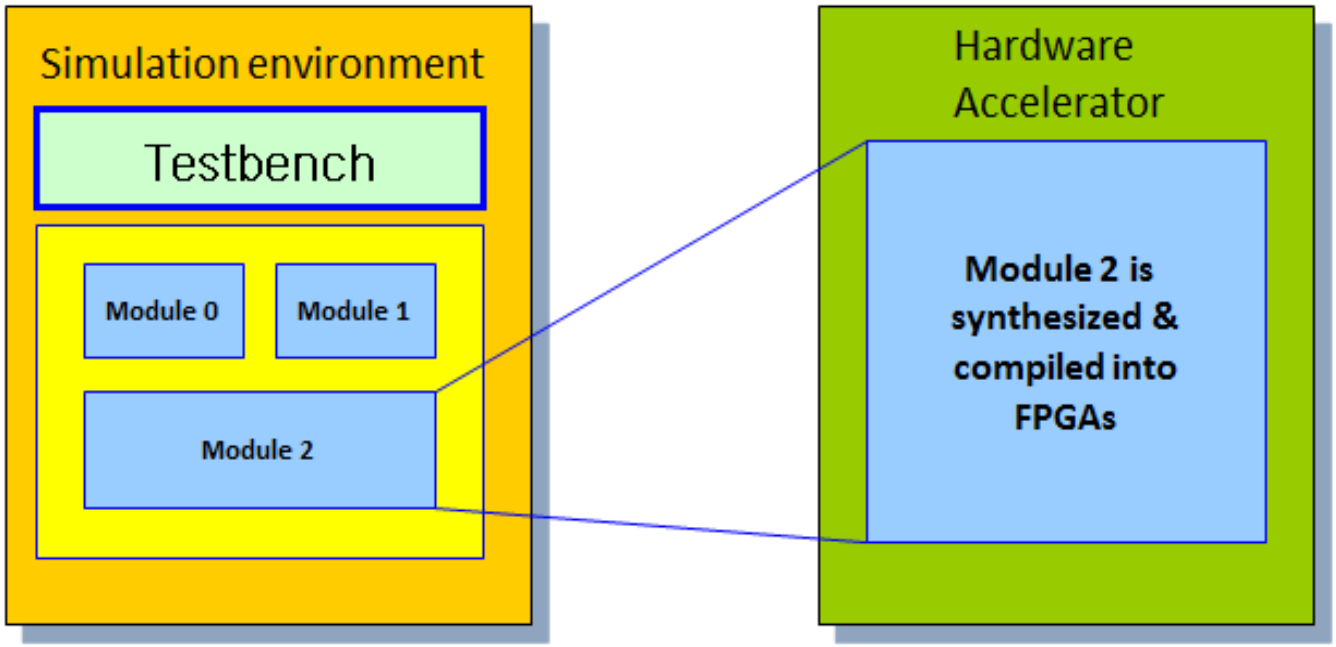


Trends

- Accelerated Simulation/Emulation
- HW+SW Co- Verification

Hardware-Accelerated Simulation

- Simulation performance is improved by moving the time-consuming part of the design to hardware.



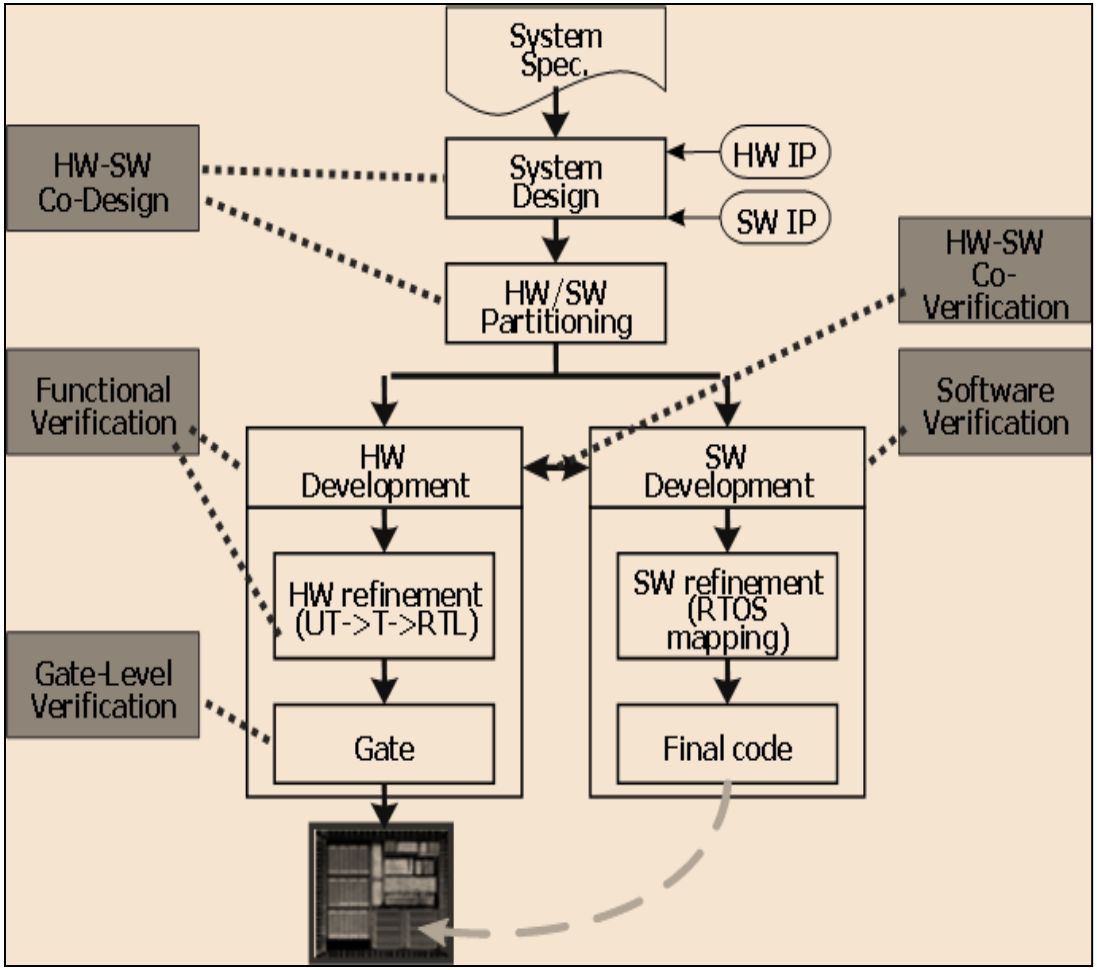
Hardware-Accelerated Simulation

- Challenges
  - Improves speed but degrades on HW-SW communication
  - Abstracting HW-SW communication at transaction level rather than cycle level desired for better speeds

- HW Emulation
  - Full mapping of HW into an emulator (array of FPGAs)
  - More like a real target system. Speed up possible up to 1000X simulation
  - Debug is a challenge with limited visibility
  - Usually used for HW+SW co-verification

HW/SW Co-Verification

- SOC designs involve both HW and SW development
- HW and SW components are verified separately but needs to work together in real product
- Co-Verification will help projects to complete in short time and with higher confidence on both HW and SW quality



Summary

- What We learned ?
  - What/Why - Verification ?
  - Different methods – Simulation/Formal/Assertion
  - Directed vs Random Testing and Coverage
  - Other Trends – Emulation and HW+SW

Structure of a Design or Test bench code

Packages

Parameters/defines, type defines, structs etc useful across files

Module/Program

- Contains declarations
  - functions/tasks, data types, objects etc
- Contains concurrent processes/blocks
  - module instance, continuous assign, always blocks etc

always blocks/initial blocks/processes

- Contains sequential code
  - If then else, case etc -> control constructs
  - Repeat/for loops/do..while -> loops
  - Blocking/non-blocking assignments etc

Data Types

0	Logic state 0 - element/net is at 0 volts
1	Logic state 1 - element/net is at some value > 0.7 volts
x or X	Logic state X - element/net has either 0/1 - we just don't know
z or Z	Logic state Z - net has high impedence - maybe the wire is not connected and is floating

2 valued data type

bit: *1 bit (packed array)*
byte: *8 bit (character)*
shortint: *16 bit*
int: *32 bit*

4 valued data type

logic: *1 bit (packed array)*

integer: *32 bit*
time: *64 bit unsigned*

```
bit [14:0] bus; // unsigned
bit signed [11:0] i,q;

shortint unsigned addrs;
logic [15:0] addrs;
```

```
string myName = "John Smith";
byte c = "A"; // assign to c "A"
bit [1:4][7:0] s = "hello" ; // assigns to s "ello"
```

Integer Data Type

shortint	2-state (1, 0), 16 bit signed
int	2-state, 32 bit signed
longint	2-state, 64 bit signed
byte	2-state, 8 bit signed
bit	2-state, user-defined vector size
logic	4-state (1,0,x,z) user-def
reg	4-state user-defined size
integer	4-state, 32 bit signed
time	4-state, 64 bit unsigned

Signed/Unsigned

- byte, shortint, int, integer and longint defaults to signed
  - Use unsigned to represent unsigned integer value
    - Example: **int unsigned ui;**
- bit, reg and logic defaults to unsigned
- To create vectors, use the following syntax:
  - **logic [1:0] L; // Creates 2 bit logic vector**

Strings

- String – dynamic allocated array of bytes
- SV provides methods for working with strings

Str1 == Str2	Equality
Str1 != Str2	Inequality
<, <=, >, >=	Comparison
{Str1, Str2, … Strn}	Concatenation
Str1[index]	indexing – return 0 if out of range

String Methods

- len
- putc
- getc
- toupper
- tolower
- compare
- icompare
- substr

- atoi, atohex, atoct, atobin
- atoreal
- itoa
- hextoa
- octtoa
- bintoa
- realtoa

Operators

Bitwise Operators		
~	~m	Invert each bit of m
&	m&n	Bitwise AND of m and n
|	m|n	Bitwise OR of m and n
^	m^n	Bitwise EXOR of m and n
~^	m~^n	Bitwise EX NOR of m and n

Unary Reduction Operators		
&	&m	AND all bits of m together (1-bit result)
~&	~&m	NAND all bits of m together (1-bit result)
|	|m	OR all bits of m together (1-bit result)
~|	~|m	NOR all bits of m together (1-bit result)
^	^m	EXOR all bits of m together (1-bit result)
~^	~^m	EXNOR all bits of m together (1-bit result)

Operators

Arithmetic Operators		
+	m+n	Add n to m
-	m-n	Subtract n from m
-	-m	Negate m (2's complement)
*	m*n	Multiply m by n
/	m/n	Divide m by n
%	m%n	Modulus of m/n

Logical Operators		
!	!n	Is m not true?
&&	m&&n	Are both m AND n true?
||	m||n	Are both m OR n true?

Operators

Equality Operators (Compares logic values of 0 and 1)		
==	m==n	Is m equal to n? (1-bit True/False result)
!=	m!=n	Is m not equal to n? (1-bit True/False result)
Identity Operators (Compares logic values of 0,1,X and Z)		
===	m^n	Is m identical to n? (1-bit True/False results)
!==	m~^n	Is m not equal to n? (1-bit True/False result)
Wild Eqality Operators		
=?=	m=?=n	A equals b, X and Z values are wild cards
!?=	m!?=n	A not equals b, X and Z values are wild cards

- The three types of equality (and inequality) operators in SystemVerilog behave differently when their operands contain unknown values (X or Z).
- The == and != operators result in X if any of their operands contains an X or Z.
- The === and !== check the 4-state explicitly, therefore, X and Z values shall either match or mismatch, never resulting in X.
- The =?= and !?= operators treat X or Z as wild cards that match any value, thus, they too never result in X.

Summary

- Supports a wide range of data types just like any programming language
- Need to be careful on where to use 2- valued versus 4 valued data types

Format indication

- %b %B binary
- %c %C character (low 8 bits)
- %d %D decimal %0d for minimum width field
- %e %E E format floating point %15.7E
- %f %F F format floating point %9.7F
- %g %G G general format floating point
- %h %H hexadecimal
- %l %L library binding information
- %m %M hierarchical name, no expression
- %o %O octal
- %s %S string, 8 bits per character, 2´h00 does not print
- %t %T simulation time, expression is $time
- %u %U unformatted two value data 0 and 1
- %v %V net signal strength
- %z %Z unformatted four value data 0, 1, x, z

Loops and Flow Control

- If-else-if
- case/casex/casez
- forever
- repeat
- while
- do..while
- foreach

Examples - Loops

```
for (int i; i < arr.size();
     j+=2, i++) begin
     arr[i] += 200;
        arr[i]--;
  end
```

```
1)x = 0;
  while (x)  begin
     $display("%d", x);
     x--;
  end
```

```
2) do
     begin
     $display("%d", x);
      x--;
     end
     while (x);
```

forever

- Continuous execution, without end
- Used with timing controls
- Usually last statement in some block

```
initial : clock_drive
begin
   clk = 1'b0;
   forever #10 clk = ~clk;
end : clock_drive
```

repeat

- Repeat a block ‘x’ times, no conditional test
  - **repeat (*expr*) *statement***

- Example

```
x = 0;
repeat (16)
  begin
   $display(“%d”, x++);
  end
```

case/casez/casex

- **case:** 4-value exact matching
- **casez:**
  - Handles z as don't care
- **casex:**
  - Handles both x and z as don't care

```
casez(a)
3'b00?: $display("0 or 1"); //LINE -1
3'b0??: $display("2 or 3"); //LINE -2
default: $display("4 to 7");
```

1) If a=010 => displays "2 or 3"
2) If a=00x => displays "0 or 1"
3) If a=0zx => displays "0 or 1" ← z matches 0

User defined types

Syntax:

typedef <base_data_type> <type_identifier>

eg:

typedef int inch ; *// inch becomes a new type*

inch foot = 12, yard = 36; *// 2 new variables of type 'inch'*

Enumerated types

Enumeration is a useful way of defining abstract variables.

Define an enumeration with “ enum ”

```
enum  {red, green, yellow} traf_lite1, traf_lite2;
```

Values can be cast to integer types and auto-incremented

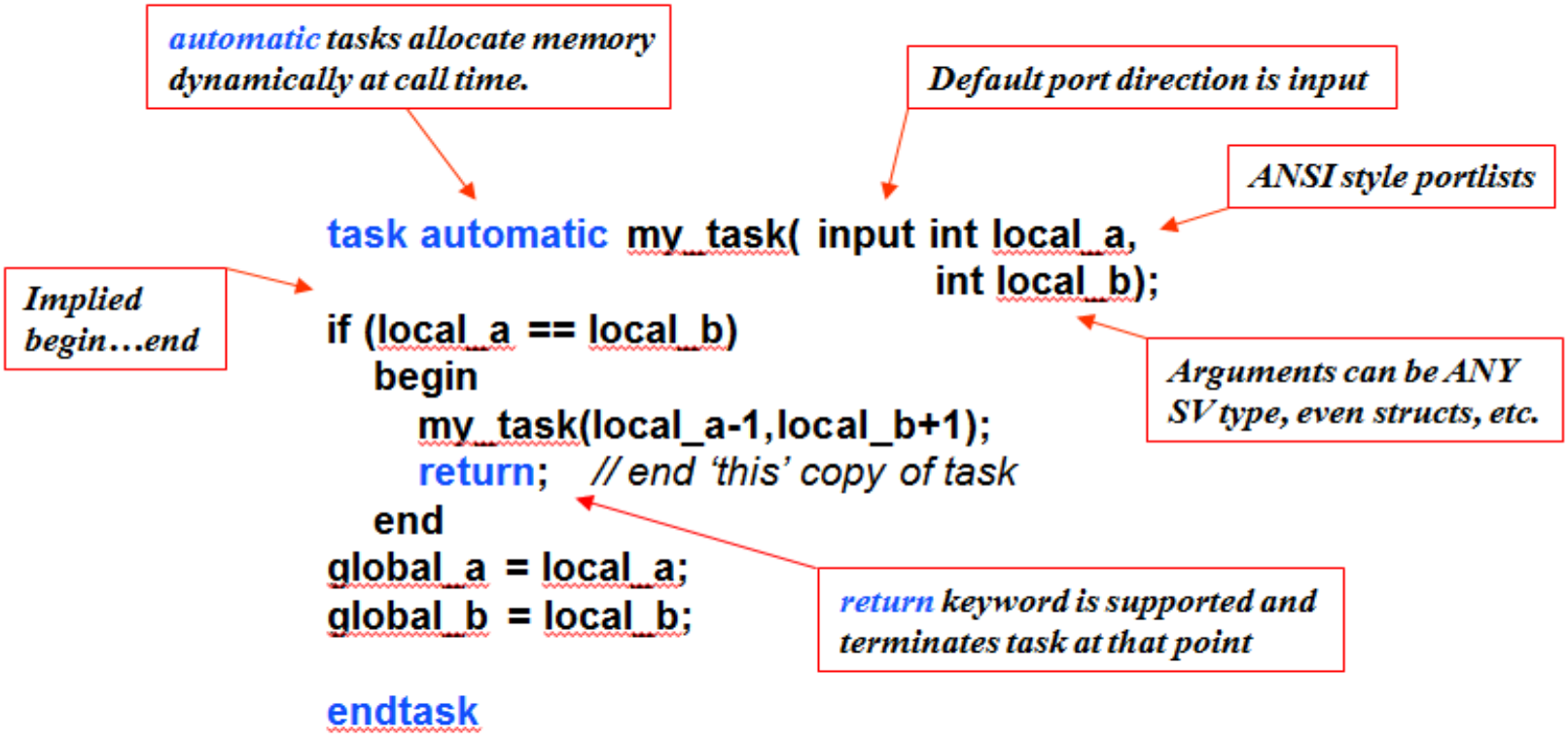
```
enum  { a=5, b, c} vars;                    // b=6, c=7
```

A sized constant can be used to set size of the type

```
enum  bit [3:0] { bronze=4'h3, silver, gold} medal;  // All medal members are 4-bits
```

Tasks and Functions

SystemVerilog makes a number of extensions to basic Verilog syntax.



Full recursion is supported (automatic variables/arguments stored on stack)

- Can do concurrent calls
- Can do recursive calls

Functions

automatic functions allocate memory dynamically at call time (full recursion).

Default port direction is input (also supports output)

ANSI style portlists

```
function automatic int factorial (int n);
  if (n==0) return(1); // factorial 0 is 1
  else return(factorial(n-1)*n);
endfunction
```

Implied begin…end

Arguments and return type can be ANY SV type, even complex structs, etc.

return(value) is supported and terminates function at that point

```
function void inverta();
   a = !a
endfunction

reg a;

initial
  inverta(); // function called like a task
```

Return type of void means no return value!

Recommended style (instead of writing a task) to guarantee a task executes with 0 delay.

Tasks and Functions - Usage

- Tasks
  - Tasks can enable other tasks and functions
  - Tasks may execute in non-zero simulation time.
  - Tasks may have zero or more arguments of type input, output and inout.

Tasks and Functions - Usage

- Functions
  - Function can enable other functions only. Task cannot be called from functions.
  - Functions should execute in zero simulation time.
  - Functions have only one return value but System Verilog also allows functions to have input, output or inout types.

Tasks and Functions - Usage

- Functions
  - Function can enable other functions only. Task cannot be called from functions.
  - Functions should execute in zero simulation time.
  - Functions have only one return value but System Verilog also allows functions to have input, output or inout types.
-

Argument passing

- Pass by value
  - Each argument is copied to the subroutine area

```
function int crc( byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
        crc ^= packet[j];
    end
    endfunction
```

- Pass by reference
  - A reference to the original argument is passed instead of copying

```
function int crc( ref byte packet [1000:1] );
    for( int j= 1; j <= 1000; j++ ) begin
        crc ^= packet[j];
    end
endfunction
```

Fixed Size Arrays

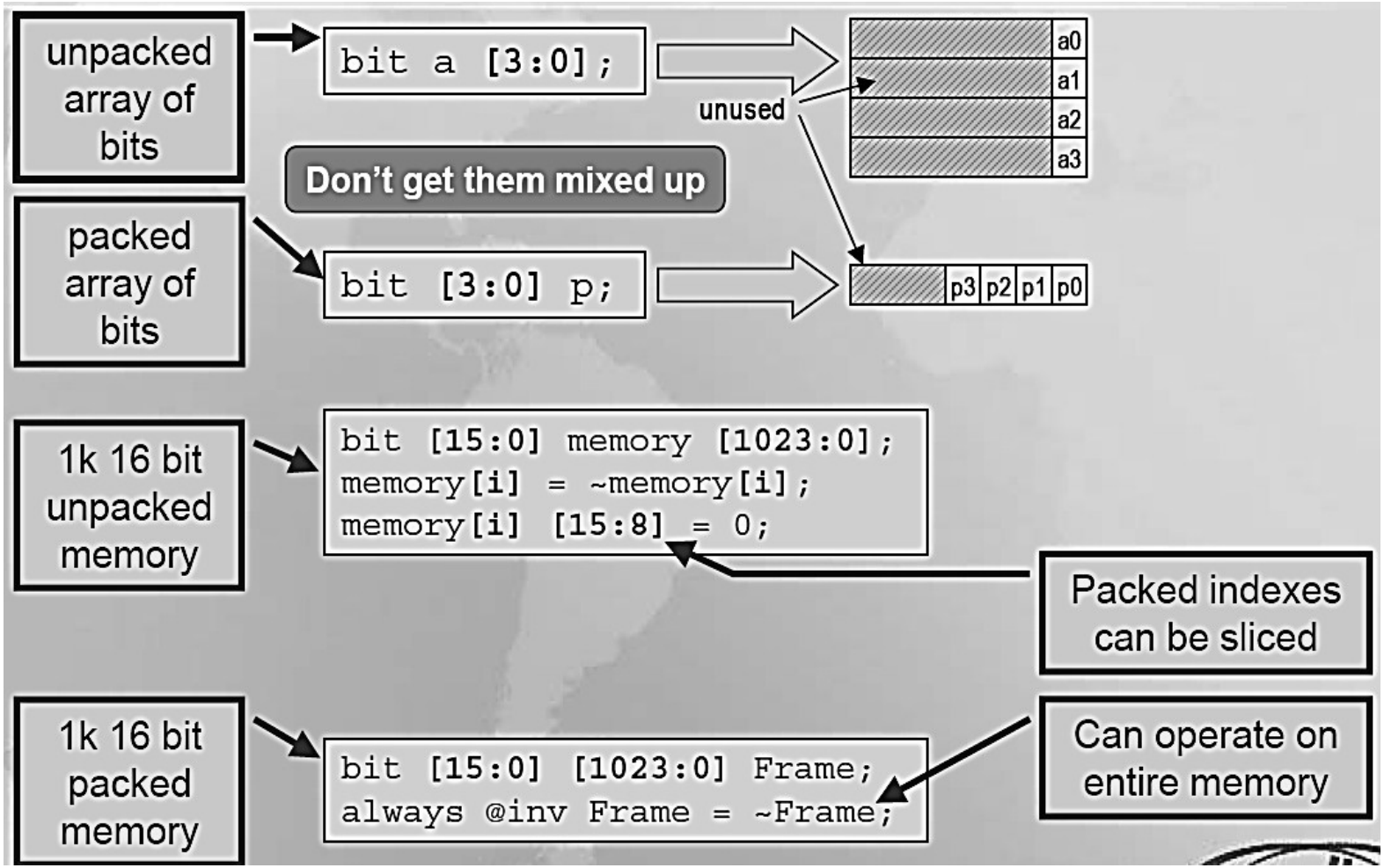
Declaring fixed-size arrays

- **int lo_hi[0:15]; // 16 ints [0]..[15]**
- **int c_style[16]; // 16 ints [0]..[15]**

Multi Dimensional

- **int array2 [0:7][0:3]; // Verbose declaration**
- **int array3 [8][4]; // Compact declaration**
- **array2[7][3] = 1; // Set last array element**

Packed and Unpacked Arrays



Dynamic Arrays

Dynamic declaration of one index of an unpacked array

Syntax:

data_type array_name[] ;

Declares a dynamic array *array_name* of type *data_type*

data_type array_name[] = new[array_size] [(array)] ;

Allocates a new array *array_name* of type *data_type* and size *array_size*
Optionally assigns values of *array* to *array_name*
If no value is assigned then element has default value of *data_type*

Examples:

```
bit [3:0] nibble[ ];          // Dynamic array of 4-bit vectors
integer mem[ ];               // Dynamic array of integers

int data[ ];                  // Declare a dynamic array
data = new[256];              // Create a 256-element array
int addr = new[100];          // Create a 100-element array
addr = new[200](addr);        // Create a 200-element array
                              // preserving previous values in lower 100 addresses
```

Dynamic Array - Methods

- **Resizing**

  <array> = new[<size>](<src_array>);

  dyn= new[j * 2](fix);

function int size() Returns the current size of the array

```
int addr[ ] = new[256];
int j = addr.size();          // j =  256
```

function void delete() Empties array contents and zero-sizes it

```
int addr[ ] = new[256];
addr.delete();
```

Associative Arrays

- Associative arrays are used when the size of the array is not known or the data is sparse.

- ***Syntax:***

  data_type array_name [index_type];

  In other words

  value_type array_name [key_type];

Associative Array Methods

Function	Use
num()	Returns number of entries
delete(<index>)	Index for delete optional. When specified used to delete given index else whole array.
exists (<index>)	Returns 1 if element exists at index else 0
first (<index>), last (<index>)	assigns to the given index variable the value of the first/last (smallest/largest) index in the associative array. It returns 0 if the array is empty, and 1 otherwise.
next (<index>), prev (<index>)	finds the entry whose index is greater/smaller than the given index.

Dynamic vs Associative Arrays

- A dynamic array gets created with a variable size and stays that size in a contiguous block of memory. Its elements are indexed starting with integer 0. This is most efficient way of accessing a block of memory, especially when you need to access to the entire array.

- Each element of an associative array gets allocated as you access them. So there is a lot more overhead for the creation of an associative array versus the same size dynamic array. And since the elements of an associative array are not always in a contiguous block of memory, there is overhead in accessing each element. (i.e. have to check if the element is allocated, and then where is it located)

- The benefit of an associative array is since each element gets allocated individually, you don't need to allocate a contiguous set of array elements. This is useful when you only plan to access a relatively few (sparse) elements of an memory address space. Another benefit is that you don't have to use an integer as an index to each element. You can associate any type of key, like a string to access each element.

Queues and Lists

- SV has a built-in list mechanism which is ideal for queues, stacks, etc.
- A list is basically a variable size array of any SV data type.

```
int q1[$];          // $ represents the 'upper' array boundary
int n, item;
q1 = '{ n, q1 };    // uses concatenate syntax to write n to the left end of q1
q1 = '{ q1, n };    // uses concatenate syntax to write n to the right end of q1
```

Queues and Lists

```
item = q1[0];          // read leftmost ( first ) item from list

item = q1[$];          // read rightmost ( last ) item from list

n = q1.size;           // determine number of items on q1

q1 = q1[1:$];          // delete leftmost ( first ) item of q1

q1 = q1[0:$-1];        // delete rightmost ( last ) item of q1

for (int i=0; i < q1.size; i++)     // step through a list using integers
begin … end

q1 = { };              // clear the q1 list
```

Queue Methods

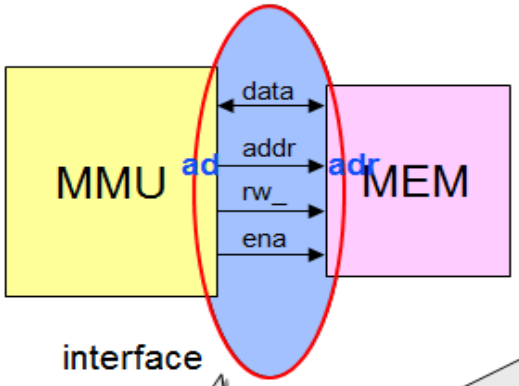
size()	Returns the number of items in the queue. If the queue is empty, it returns 0.
insert()	Inserts the given item at the specified index position. Q.insert (i, e) => Q = '{Q[0:i-1], e, Q[i,$]}
delete()	Deletes the item at the specified index position. Q.delete (i) => Q = '{Q[0:i-1], Q[i+1,$]}

Queue Methods

pop_front()	Removes and returns the first element of the queue. e = Q.pop_front() => e = Q[0]; Q = Q[1,$]
pop_back()	Removes and returns the last element of the queue. e = Q.pop_back() => e = Q[$]; Q = Q[0,$-1]
push_front()	Inserts the given element at the front of the queue. Q.push_front(e) => Q = '{e, Q}
push_back()	Inserts the given element at the end of the queue. Q.push_back(e) => Q = '{Q, e}

Interfaces

Great coding efficiency can be achieved by modeling the blocks of a system at different levels of abstraction, behavioral, rtl, gate, etc.
In Verilog, the I/O between these blocks has always remained at the lowest "wire" level. High-performance system-level simulation requires the abstraction of inter-block communication.



At it's simplest an interface is like a module for ports/wires

```
module mmu(d, a, rw_, en);
  output [15:0] a;
  output rw_, en;
  inout [7:0] d;
  . . .
endmodule

module mem(d, a, rw_, en);
  input [15:0] a;
  input rw_, en;
  inout [7:0] d;
  . . .
endmodule

module system;
  wire [7:0] data;
  wire [15:0] addr;
  wire ena, rw_;

  mmu U1 (data, addr, rw_, ena);
  mem U2 (data, addr, rw_, ena);
endmodule
```

Traditional Verilog

```
interface interf;
  logic [7:0] data;
  logic [15:0] addr;
  logic ena, rw_;
endinterface

module mmu(interf io);
  io.addr <= ad;
  . . .
endmodule

module mem(interf io);
   adr = io.addr;
 . . .
endmodule

module system;
  interf i1;
  mmu U1 (i1);
  mem U2 (i1);
endmodule
```

SystemVerilog

Interfaces – Characteristics

- Brings abstraction-level enhancements to ports and internals
- Interface may contain any legal System Verilog code except module definitions and/or instances.
  - tasks, functions, initial/always blocks, parameters, etc.
- Bus timing, pipelining, etc. may be captured in an interface rather than the connecting modules.
- Interfaces are defined once and used widely, so it simplifies design.
- Interfaces are synthesizable.

ModPorts

- Different users of interface need different views
  - Master/Slave
  - Monitor/driver
- Modport construct allows you to group signals and specify directions

Mod port example

Example 5-9 Interface with modports

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic reset;

  modport TEST (output request, reset,
                input  grant, clk);

  modport DUT (input request, reset, clk,
               output grant);

  modport MONITOR (input request, grant, reset, clk);

endinterface
```

Clocking Blocks

- Used to specify timing of synchronous signals with respect to clock
- Mainly used in test benches and inside interfaces
- It ensures that all signals are driven or sampled with same clock delays
- Interfaces can contain multiple clocking blocks (1 per clock)

Clocking block syntax

```
clocking block_name clocking_event;
    item list;
endclocking : block_name

clocking bus @(posedge clock1);
  default input #10ns output #2ns;
  input data, ready,
          enable=top.mem1.enable;
  output negedge ack;
  input #1 addr;
endclocking
```

Clocking block -example

Example 5-13 Interface with a clocking block

```
interface arb_if(input bit clk);
  logic [1:0] grant, request;
  logic reset;

  clocking cb @(posedge clk);     // Declare cb
    output request;
    input grant;
  endclocking

  modport TEST (clocking cb,      // Use cb
                output reset);

  modport DUT (input request, reset, output grant);
endinterface
```

Input and output skews

- Input and output signals are sampled at a clocking event.
- For an input skew, the signal is sampled at skew units before the clock event.
- For an output or inout, the signal is driven simulation time units after the corresponding clock.

Input and output skews

- A skew value is a constant expression and can be specified by a parameter.
- If a number is used for the time then skew is interpreted based on timescale.
- An input skew of 1step indicates that the signal is sampled at the end of previous time step.

Clocking block example with skew and parameters

```
clocking cb @(posedge clk);
    parameter INPUT_SKEW = 1;
    parameter OUTPUT_SKEW = 2;
    default input #INPUT_SKEW output #OUTPUT_SKEW;
    input #3step req1;
    input req2;
    output #4ns grant;
endclocking
```

Program Blocks

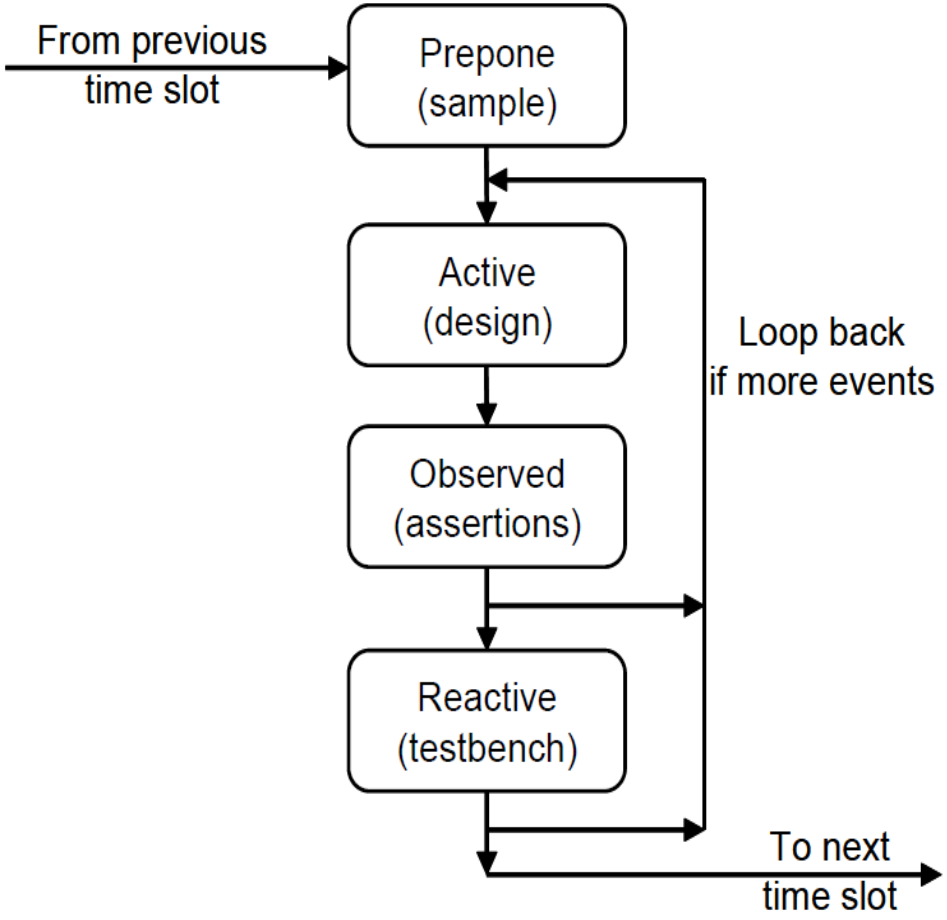
- Race conditions can occur if module is used as top level for both design and Test bench
- A program block is similar to a module. It is used for testbench code.

```
program
  helloWorld();
  initial
  begin: hello
    $display("Hello
  World");
  end

  initial
  begin: there
    $display(“Hello
  There”);
  end
endprogram:
  helloWorld
```

System Verilog Timeslot Evaluation

- Operations of a program block happen in the reactive region of the simulation.



Program Blocks

- Programs can be instantiated inside modules, but not the other way around.
- Program blocks may contain one or more initial blocks, but may not contain always, UDPs, modules, interfaces, or other programs.
  - Usage is normally an initial block and use a forever loop for the whole test to evaluate

Program Block – Example

Example 5-15 Testbench using interface with clocking block

```
program automatic test (arb_if.TEST arbif);
...
  initial begin
    arbif.cb.request <= 2'b01;
    $display("@%0d: Drove req=01", $time);
    repeat (2) @arbif.cb;
    if (arbif.cb.grant != 2'b01)
      $display("@%0d: a1: grant != 2'b01", $time);
    end

endprogram : test
```