The American University in Cairo

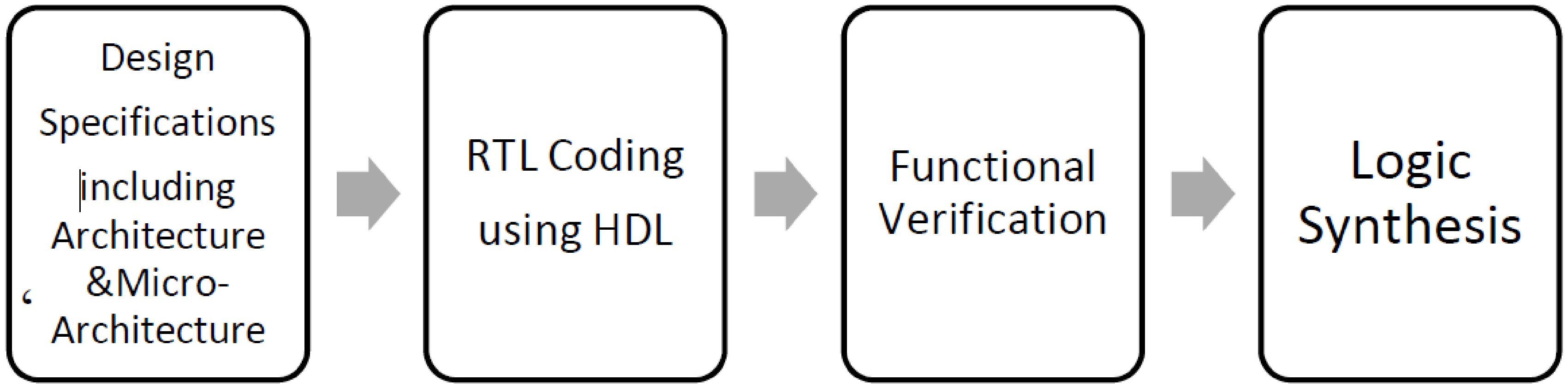
CENTER OF NANOELECTRONICS & DEVICES

CND212: Digital Testing and Verification

Digital IC Verification

Introduction

Normal VLSI/Chip design flow


Design Specifications including Architecture &Micro-Architecture
RTL Coding using HDL
Functional Verification
Logic Synthesis

The Verification Process

- The process of checking that a given design correctly implements the specification and the required functionality.
- A verification engineer accomplishes this by:
  - Reading the hardware specification
  - Creating a verification plan
  - Building tests that show the RTL code correctly implements the features
- Verification also includes error injection and handling.
  - How does the design handle errors?
  - How should it recover from them?
- Verification consumes around 80% of the total product development time.

What is Verified?

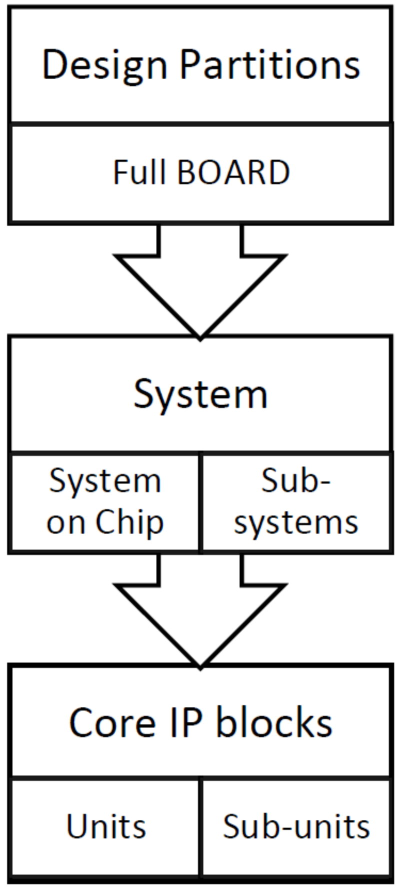
Functional Verification

Performance Verification

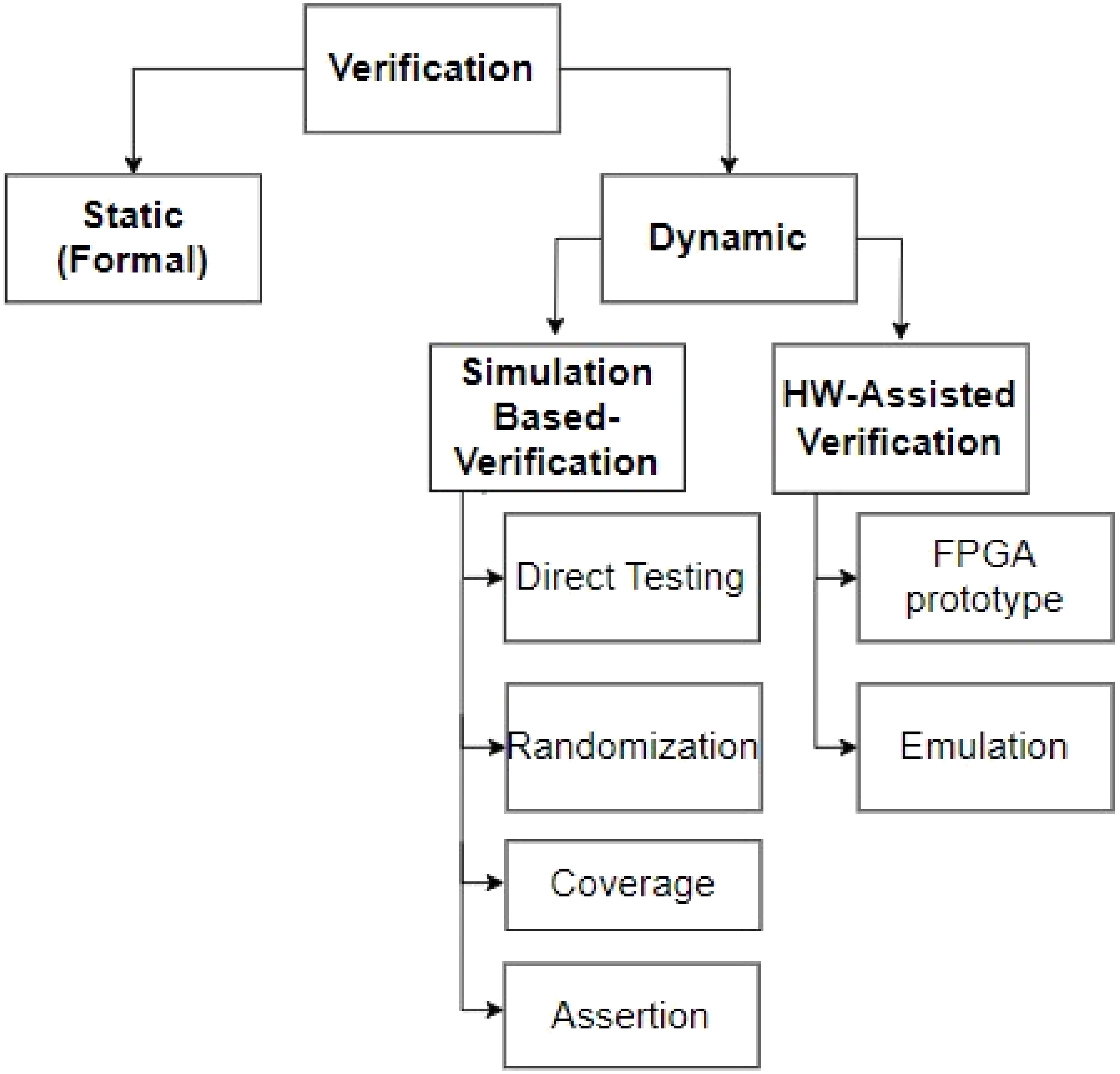
Timing Verification

Verification Levels

- Basic verification: (at Unit/Sub-unit level)
- Functional verification (at IP blocks level)
- System level verification: (at the Systems level)
- Connectivity: (at Board level)

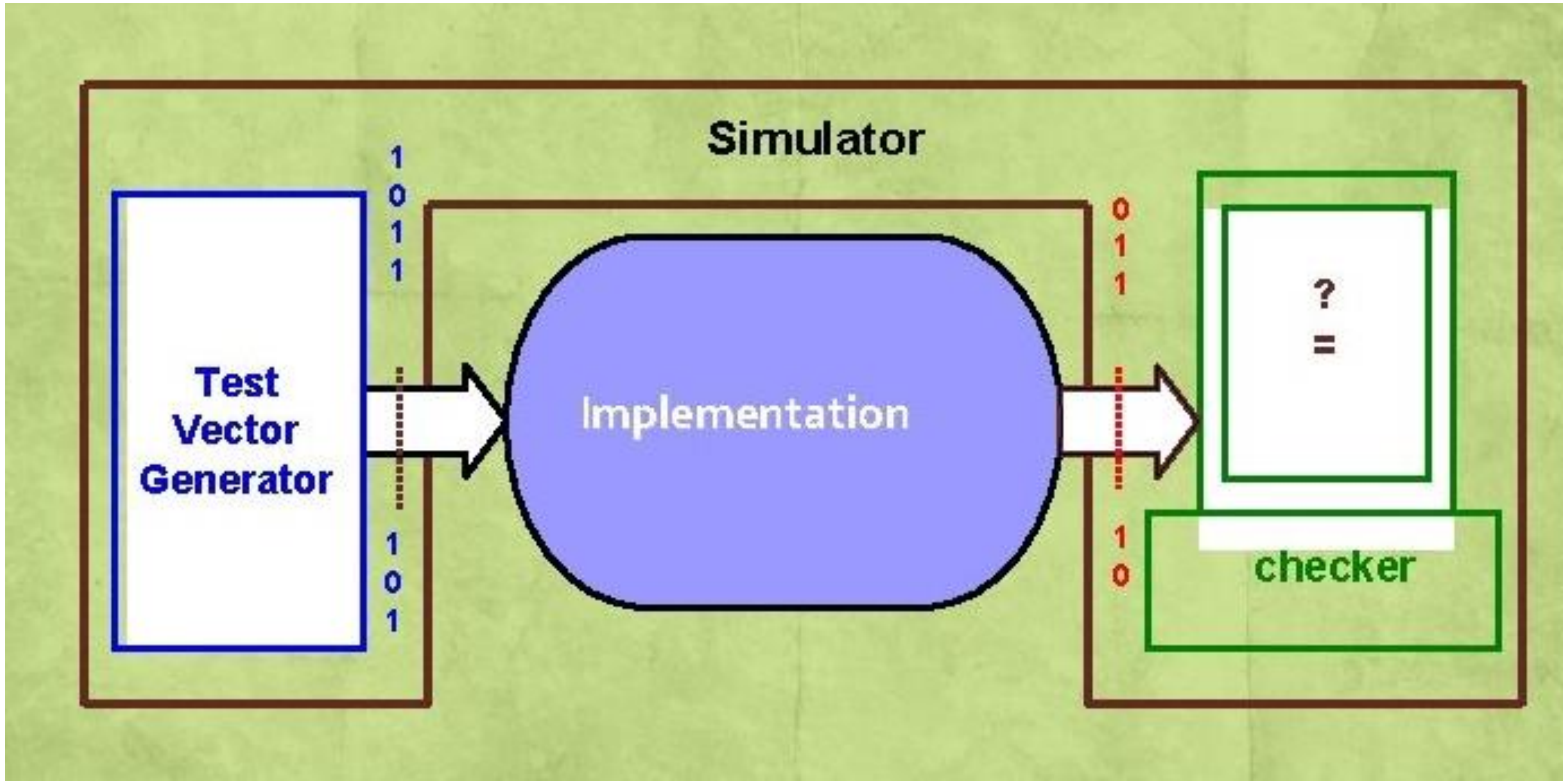


Verification Methods



Verification Methods

Simulation Based Verification



Vivolo, L.. *Transaction-based Verification And Emulation Combine For Multi-megahertz Verification Performance*. [online] Electronicdesign.com. Available at: <https://www.electronicdesign.com/technologies/eda/article/21796417/transactionbased-verification-and-emulation-combine-for-multimegahertz-verification-performance>.

Verification Methods

Formal Verification

- It is a method by which we prove or disprove a design implementation against a formal specification or property by using a mathematical model.
- An algorithm is used to explore all possible input values over time and to exercise all possible states in a design.
- It works well for small designs where the number of inputs, outputs, and states is small.

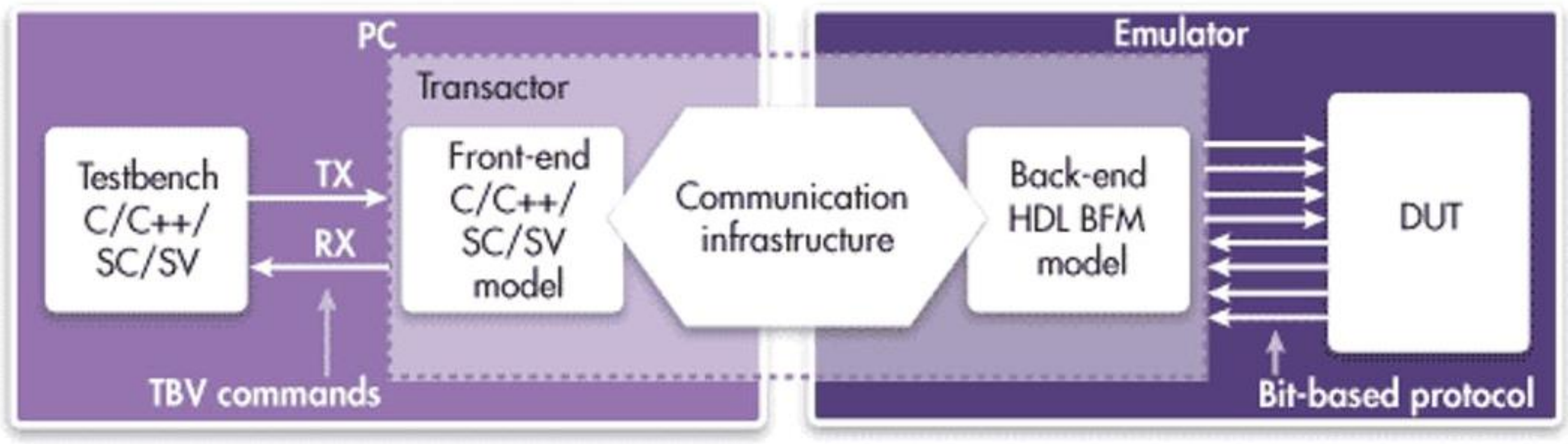
Verification Methods

Assertion-based Verification

- An assertion is a statement about a design's intended behavior, which must be verified.
- It is a way of capturing a design's intention or part of the design's specification in the form of a property; this property can be used along with dynamic simulation and with formal verification to ensure whether this specification is met.
- Assertions Benefits:
  - Improves observability and debugging ability.
  - Improves integration through correct usage checking.
  - Improves verification efficiency.

Verification Methods

Emulation Based Verification

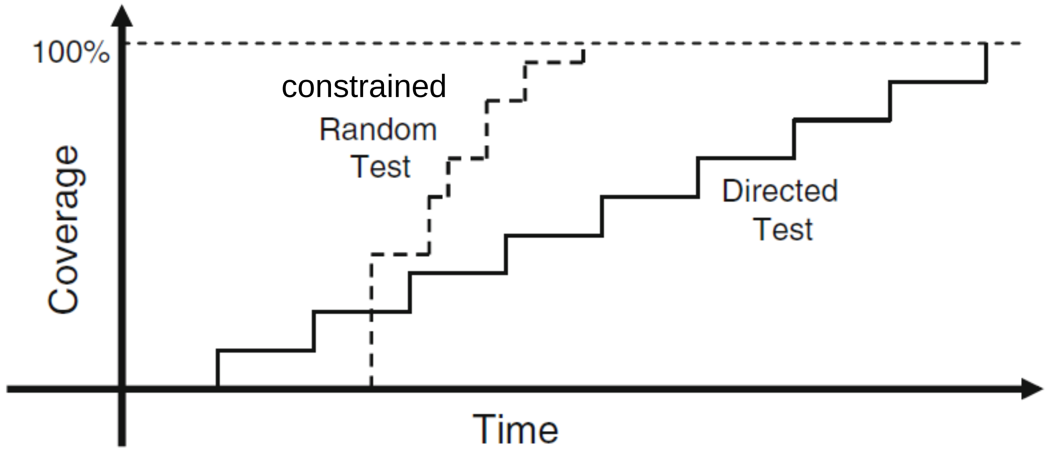


Vivolo, L., 2021. *Transaction-based Verification And Emulation Combine For Multi-megahertz Verification Performance*. [online] Electronicdesign.com. Available at: https://www.electronicdesign.com/technologies/eda/article/21796417/transactionbased-verification-and-emulation-combine-for-multimegahertz-verification-performance.

Verification Testbench

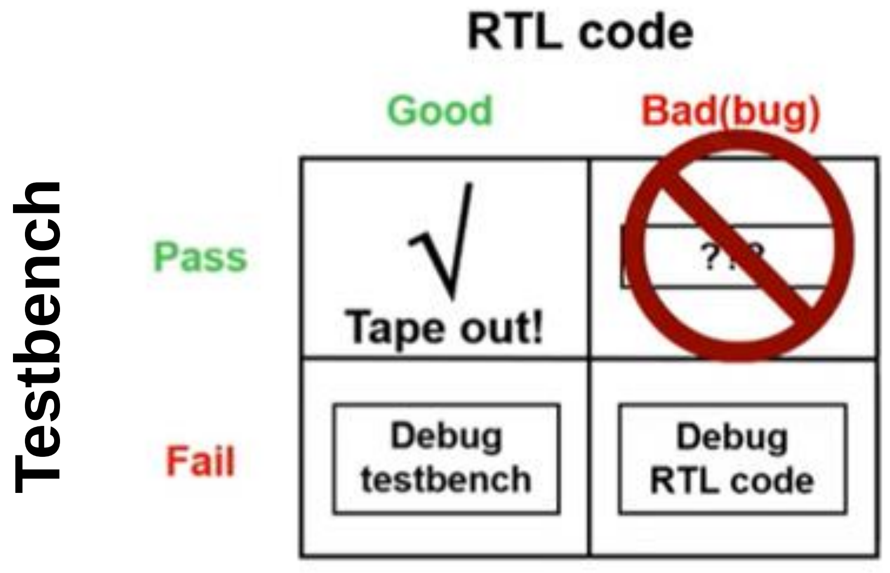
- A Testbench is implemented to check the functional correctness of the DUT. This is achieved by doing the following:
  - Generate stimulus
  - Apply stimulus to the DUT
  - Capture the response
  - Check for correctness
  - Measure progress against the overall verification goals

- There are different approaches for testing:
  - Directed Testing
  - Pure Random Testing
  - Constrained random stimulus



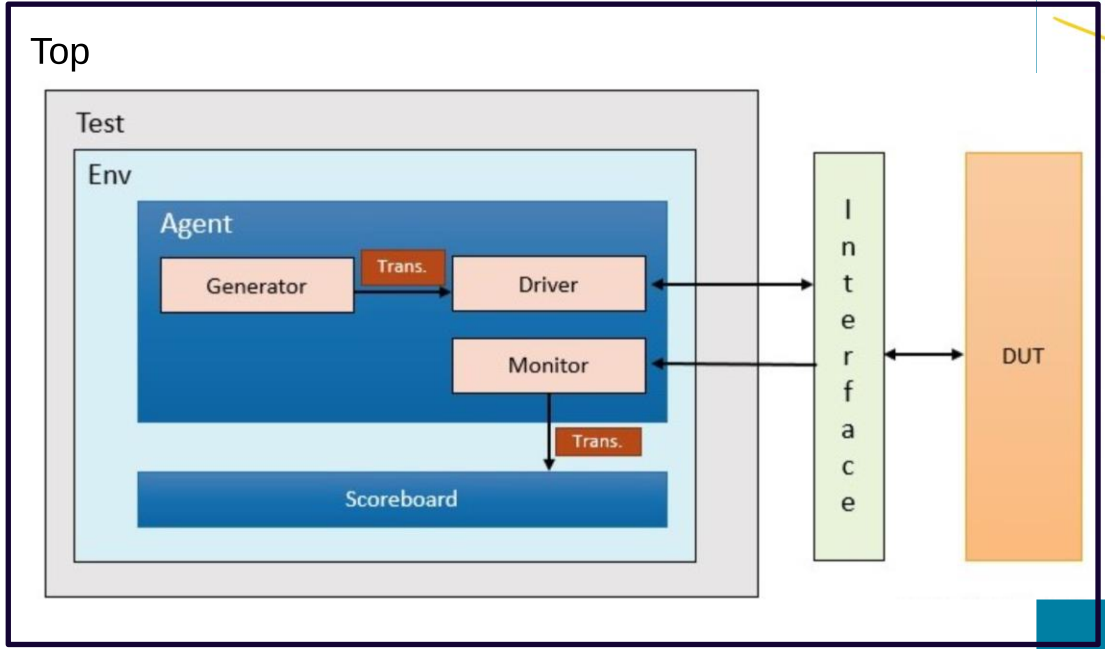
Well Designed Verification Environment

- Test Environment must:
  - Be structured for debug
  - Avoid false positives
- Tests must:
  - Achieve Functional Coverage
    - Prevent untested regions
  - Reach Corner Cases
    - Anticipated Cases
    - Error Injection
      - Environment Error
      - DUT Error
    - Unanticipated Cases
      - Random tests
    - Be Robust, Reusable, Scalable



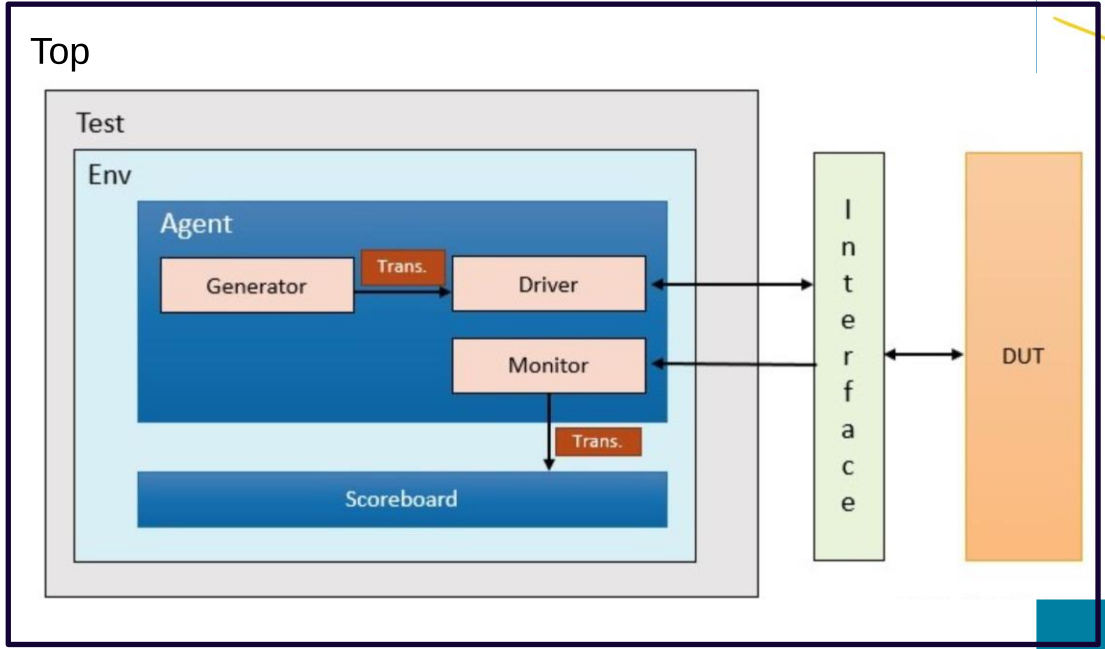
Verification Testbench

- Transaction
  - Class that holds a structure used to communicate with DUT.
  - Then converted to pin-level data to be driven to the DUT or monitored.
- Generator
  - Creates or generates randomized transactions or stimuli and passes them to the driver.
- Driver
  - Interacts with DUT directly.
  - Receives randomized transactions from the generator and drives them to the DUT as a pin-level activity.
- Monitor
  - Observe pin-level activity on the connected interface at the input and output of the design.
  - Then, convert the pin-level activity into a transaction packet and send it to the scoreboard for checking purposes.



Verification Testbench

- Test
  - The test is at the top of the hierarchy, initiating the construction of the environment components and the connection between them.
  - Responsible for the testbench configuration and stimulus generation process.
- Testbench top
  - The top-level component that includes interface and DUT instances.
  - The design is connected to the testbench in the Top model.
- Efficient Verification Testbench
  - Layered.
  - Reusable.
  - Scalable.
  - Easier to maintain.



SystemVerilog Basics

SV Lexical Conventions

- Same as Verilog
- Case sensitive identifiers (names)
  - Any sequence of letters, digits, $, and _
    - First character cannot be a digit or $
  - White spaces are ignored except within strings
- Comments:
  - Single line // ….
  - Multiple lines /* …. */

SV Number Format

- Number format now supports *sign/unsigned* declaration
  - <size>’[s|S]b (binary)
  - <size>’[s|S]d (decimal)
  - <size>’[s|S]h (hexadecimal)
  - <size>’[s|S]o (octal)
  - Can be padded with ‘_’ for readability
- Sign Conversion
  - *signed’(myvar);*
  - *unsigned’(myvar);*

```
logic [7:0] my value; //defaults to unsigned

my_value = 8’b1;     // 8’b0000_0001  (unsigned)
my_value = 8’sb1;    // 8’b0000_0001  (signed)

//unsized padding of bits

my_value = ‘1;       //8’b1111_1111
```

SV Data Types

- SystemVerilog introduces new data types with the following benefits.
  - **Two-state**: better performance, reduced memory usage
  - **Queues**, **dynamic** and **associative arrays** and **automatic storage**: reduced memory usage, built-in support for searching and sorting
  - **Unions** and **packed structures**: allows multiple views of the same data
  - **Classes** and **structures**: support for abstract data structures
  - **Strings**: built-in string support
  - **Enumerated** types: code is easier to write and understand

SV Data Types

- Data type is a storage format having a specific range or type.
  - Two states data type, each bit is
    - Logic 0
    - Logic 1
  - Four states data type, each bit is
    - Logic 0
    - Logic 1
    - Logic X; unknown (either 0/1)
    - Logic Z; high impedance (floating)

SV Data Types: Four State (0|1|X|Z)

reg | logic [msb:lsb] variable_name [=initial_value];

- 4-state data type: logic | reg
  - reg and logic are synonyms
  - SystemVerilog improves the classic reg to have the following features:
    - It can be driven by continuous assignments, gates, and modules, in addition to being a variable.
    - It is given the new name logic so that it does not look like a register declaration.
- Initialized to ‘x if initial value is not specified
- Defaults to unsigned

SV Data Types: Four State (0|1|X|Z)

- Sized 4-state variable: integer, time

integer variable_name [=initial_value];

  – 32-bit signed

time variable_name [=initial_value];

  – 64-bit signed

```
integer  a = - b;
time current_time;
b = -a;
current_time = $time;
If  (current_time >= 100ms)  ….. ;
```

SV Data Types: Two State (0|1)

bit [msb:lsb] variable_name [=initial_value];

- Better compiler optimizations for better performance
- Variable initialized to ‘0 if initial_value is not specified
- Assigned 0 for x or z value assignments
  - Sized as specified
  - Defaults to unsigned

SV Data Types: Two State (0|1)

2-state-type variable_name [=initial_value];

Type	Description	Example
bit	user-determined range	bit [3:0] a_var;
byte	8 bits, signed	byte a, b;
shortint	16 bits, signed	shortint c, d;
int	32 bits, signed	int i,j;
longint	64 bits, signed	longint lword;

SV Data Types: Two State (0|1)

2-state-type variable_name [=initial_value];

Type	Description	Example
shortreal	like float in C	shortreal f;
real	like double in C	real g;
realtime	identical to real	realtime now;

SV Data Types: String Data Type

```
string variable_name [=initial_value];
```

- Defaults to empty String “”
- Can be created with $sformatf() system function
- Built-in operators and methods:
  - ==, !=, compare () and icompare()
  - Itoa (), atoi, atohex () , toupper (), tolower (), etc.
  - len (), getc (), putc (), substr ()

SV Data Types: String Data Type

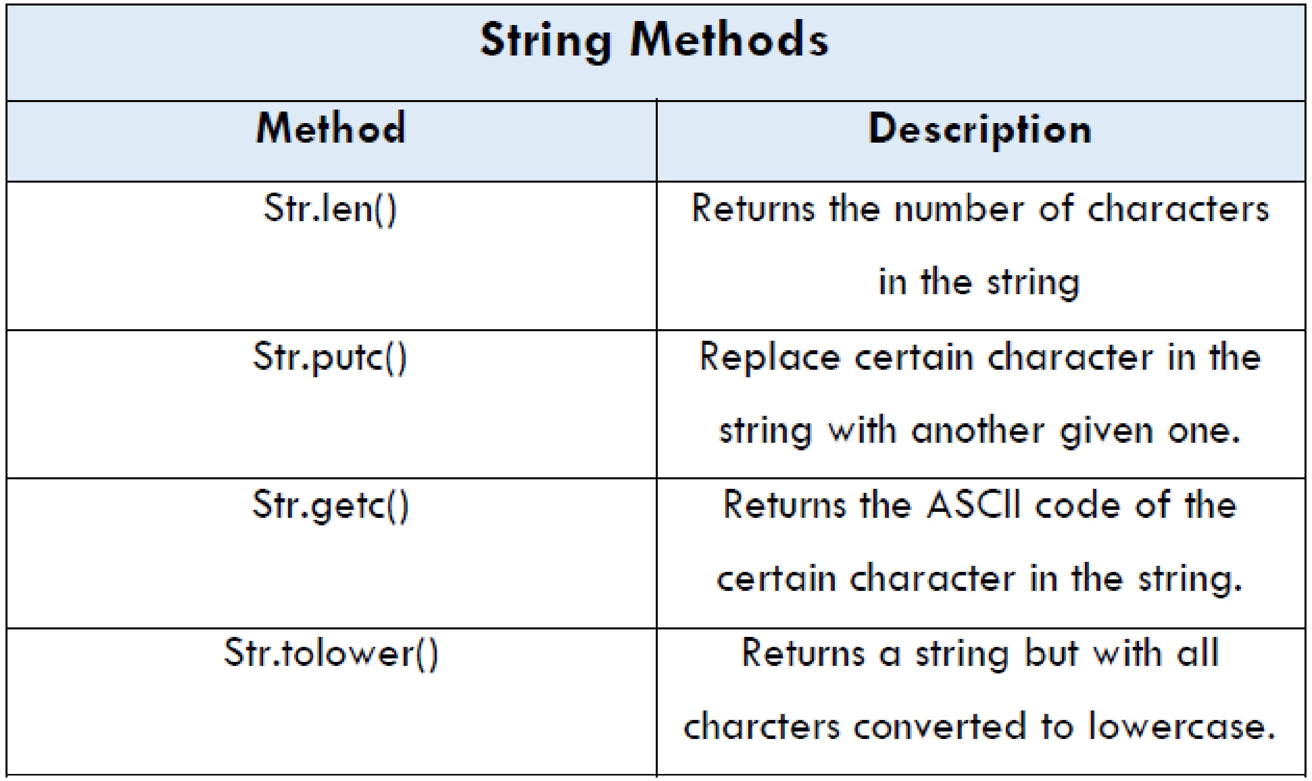
string variable_name [=initial_value];

String Operators

Operator	Syntax	Description
Equality	$Str1 == $Str2	Returns 1 if the two strings are equal, 0 otherwise.
Inequality	$Str1!= $Str2	Returns 1 if the two strings are not equal, 0 otherwise.
Comparison	$Str1 < $Str2 $Str1 <= $Str2 $Str1 > $Str2 $Str1 >= $Str2	Returns 1 if the correspondig condition is true and 0 if false
Concatenation	{ Str1, Str2, ...}	All strings will be concatenated into one string
Replication	{N{ Str }}	Replicates the string N number of times
Indexing	Str[index]	Returns a byte or the equivelant ASCII code at the given index. If given index is out of range, it returns 0

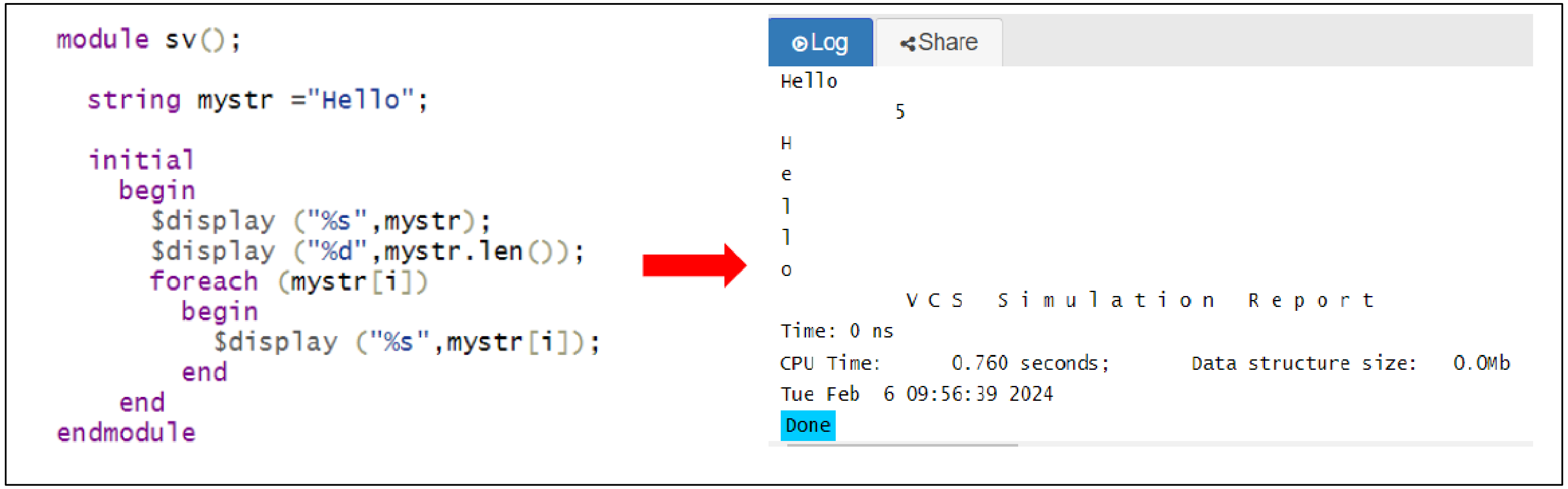
SV Data Types: String Data Type

string variable_name [=initial_value];



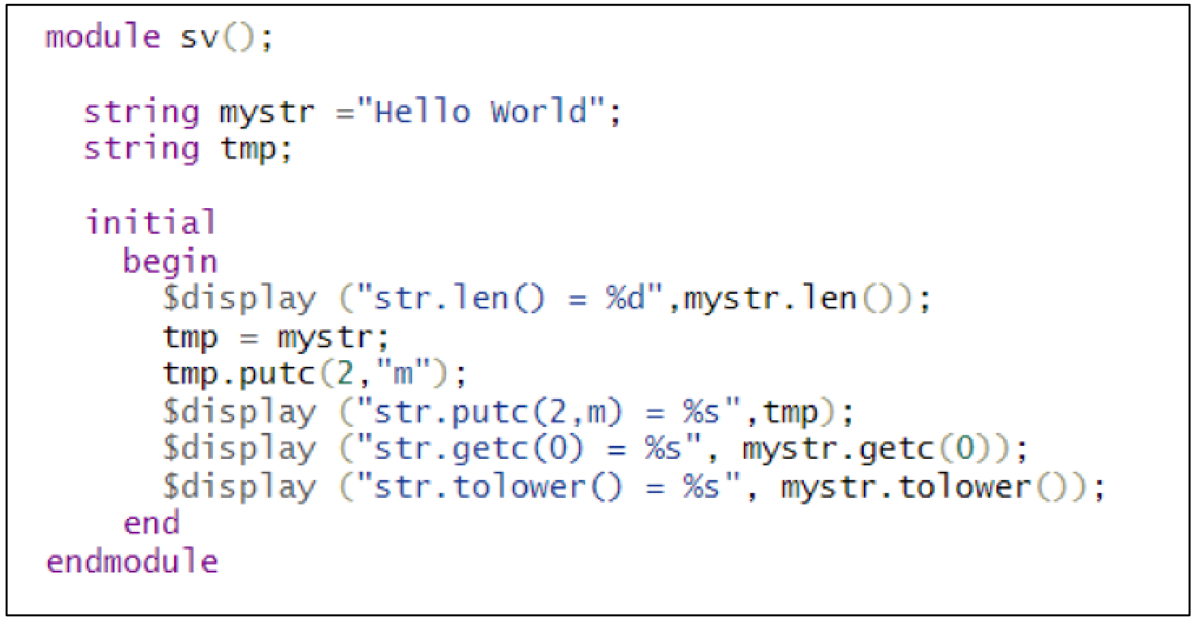
String Methods	
Method	**Description**
Str.len()	Returns the number of characters in the string
Str.putc()	Replace certain character in the string with another given one.
Str.getc()	Returns the ASCII code of the certain character in the string.
Str.tolower()	Returns a string but with all charcters converted to lowercase.

SV Data Types: String Data Type

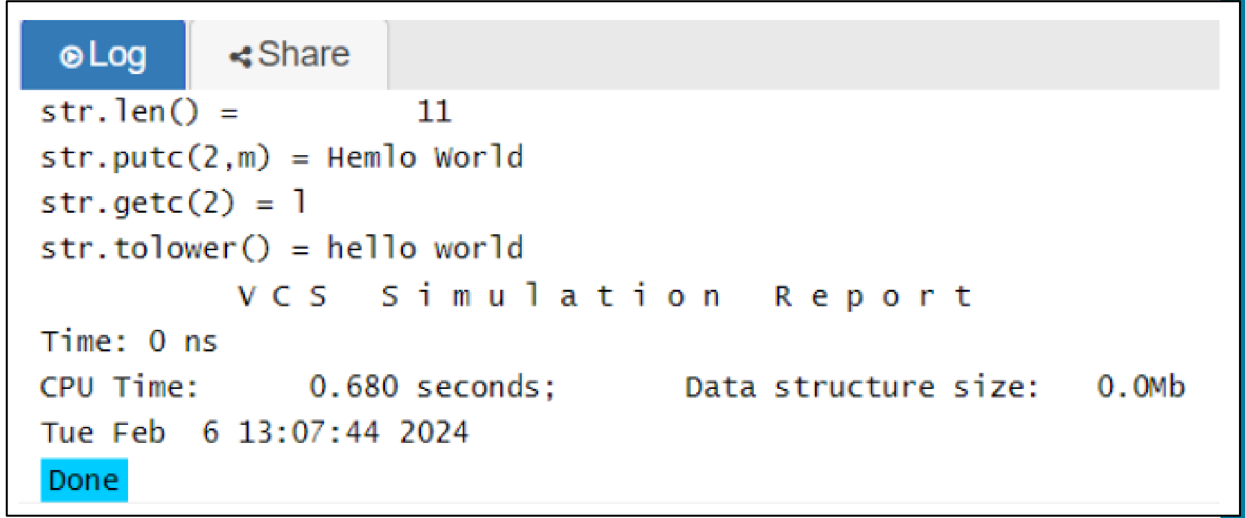


```
module sv();

  string mystr ="Hello";

  initial
    begin
      $display ("%s",mystr);
      $display ("%d",mystr.len());
      foreach (mystr[i])
        begin
          $display ("%s",mystr[i]);
        end
    end
endmodule
```

```
Hello
          5
H
e
l
l
o
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.760 seconds;      Data structure size:   0.0Mb
Tue Feb  6 09:56:39 2024
Done
```



```
module sv();

  string mystr ="Hello World";
  string tmp;

  initial
    begin
      $display ("str.len() = %d",mystr.len());
      tmp = mystr;
      tmp.putc(2,"m");
      $display ("str.putc(2,m) = %s",tmp);
      $display ("str.getc(0) = %s", mystr.getc(0));
      $display ("str.tolower() = %s", mystr.tolower());
    end
endmodule
```



```
str.len() =          11
str.putc(2,m) = Hemlo World
str.getc(2) = l
str.tolower() = hello world
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.680 seconds;      Data structure size:   0.0Mb
Tue Feb  6 13:07:44 2024
Done
```

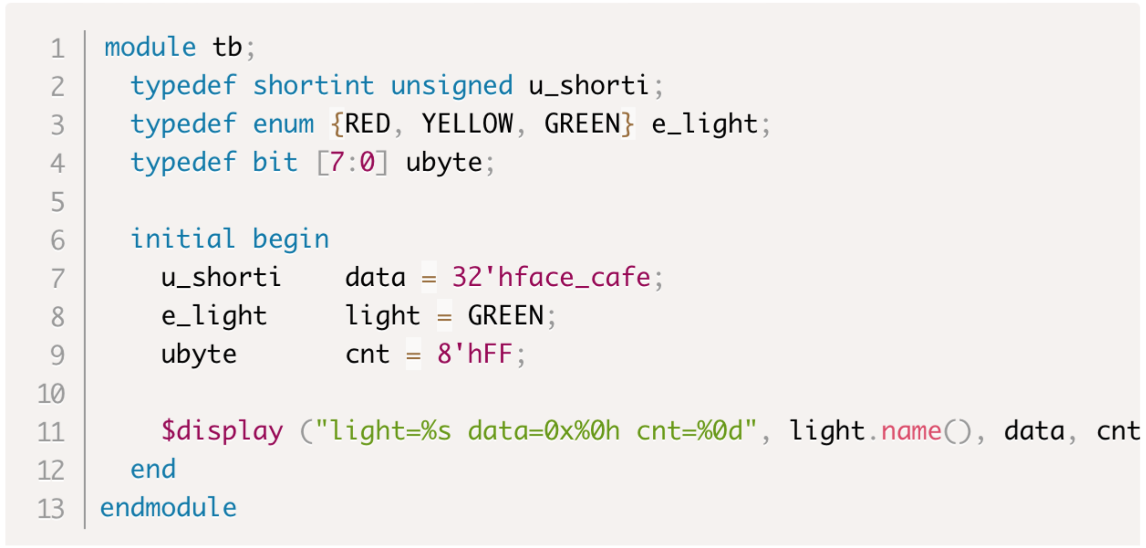
SV Data Types: Enumerated Data Type

```
enum           {RED, YELLOW, GREEN}        light_1;        // int type; RED = 0, YELLOW = 1, GREEN = 2
enum bit[1:0]  {RED, YELLOW, GREEN}        light_2;        // bit type; RED = 0, YELLOW = 1, GREEN = 2
```

- Define enum variables

enumtype variable_name [=initial_value];

  - Date type defaults to int
  - Variable initialized to ‘0 if initial_value is not specified (x for a 4-state data-type)
  - enum can be displayed as ASCII or value



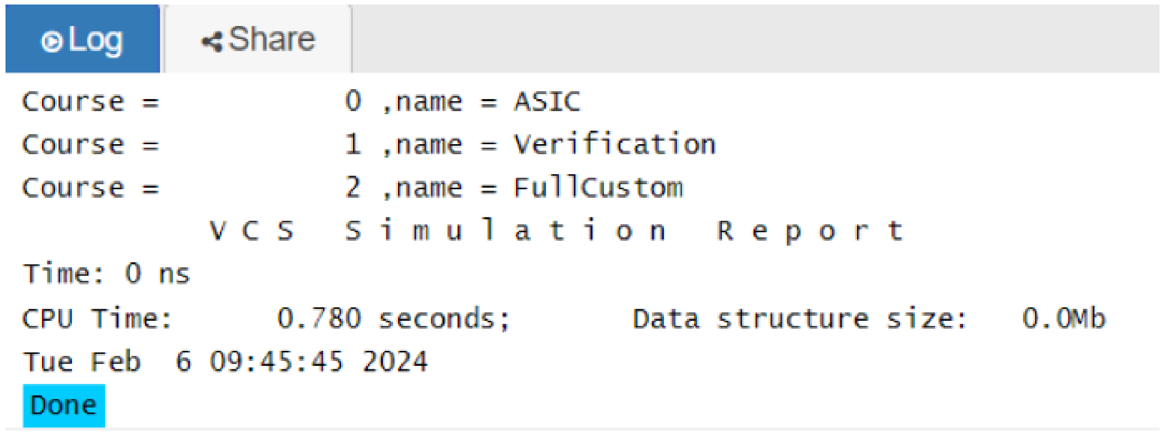
```
module tb;
  typedef shortint unsigned u_shorti;
  typedef enum {RED, YELLOW, GREEN} e_light;
  typedef bit [7:0] ubyte;

  initial begin
    u_shorti    data = 32'hface_cafe;
    e_light     light = GREEN;
    ubyte       cnt = 8'hFF;

    $display ("light=%s data=0x%0h cnt=%0d", light.name(), data, cnt
  end
endmodule
```

- Define enumerated type

typedef enum [data_type] {named constants} enumtype;

SV Data Types: Enumerated Data Type

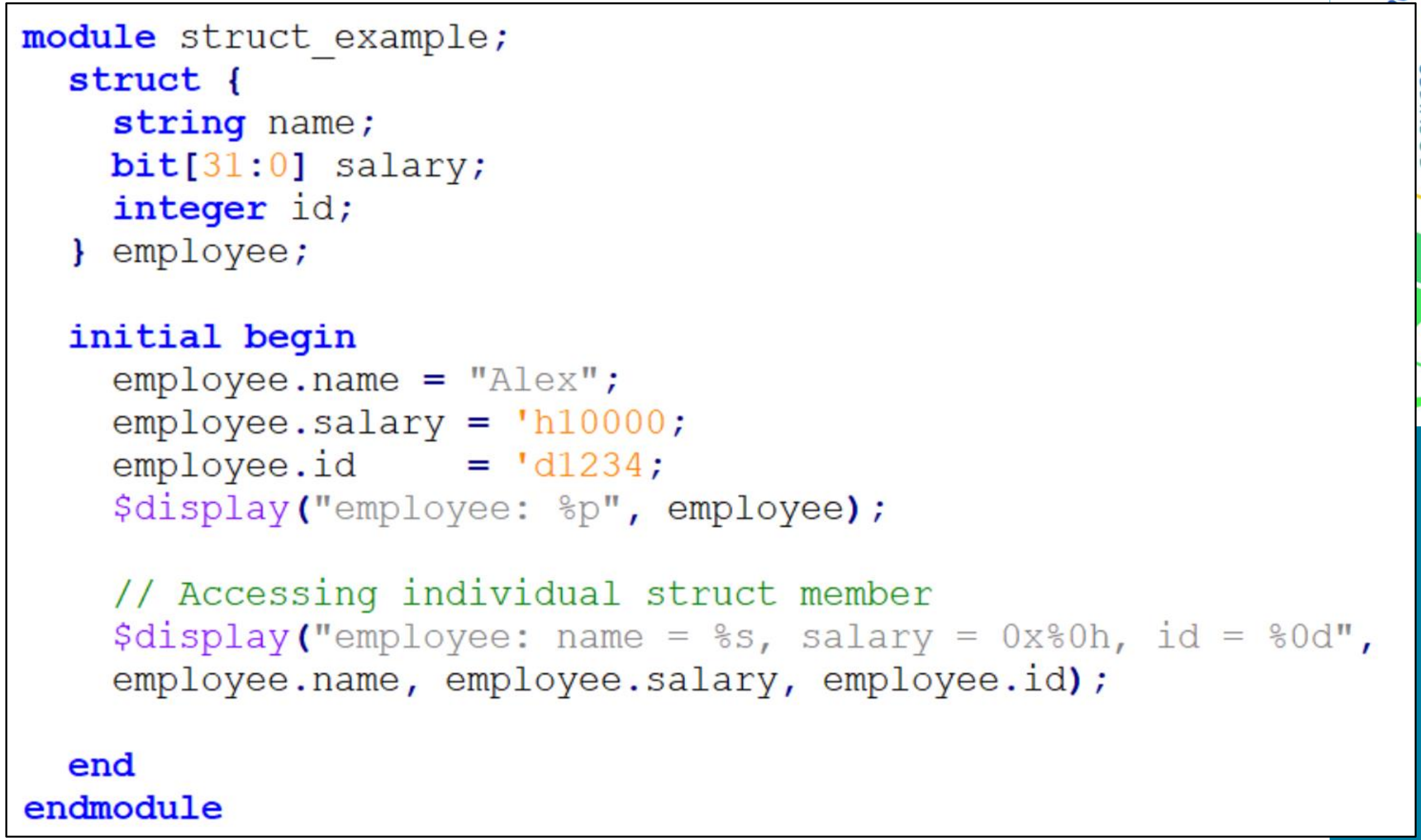
```
module sv();

  typedef enum {ASIC, Verification ,FullCustom}  Courses;
  Courses course;

  initial
    begin
      course = ASIC;
      $display ("Course = %d ,name = %s",course , course.name());
      course = Verification;
      $display ("Course = %d ,name = %s",course , course.name());
      course = FullCustom;
      $display ("Course = %d ,name = %s",course , course.name());
    end
endmodule
```



```
Course =           0 ,name = ASIC
Course =           1 ,name = Verification
Course =           2 ,name = FullCustom
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.780 seconds;       Data structure size:   0.0Mb
Tue Feb  6 09:45:45 2024
Done
```

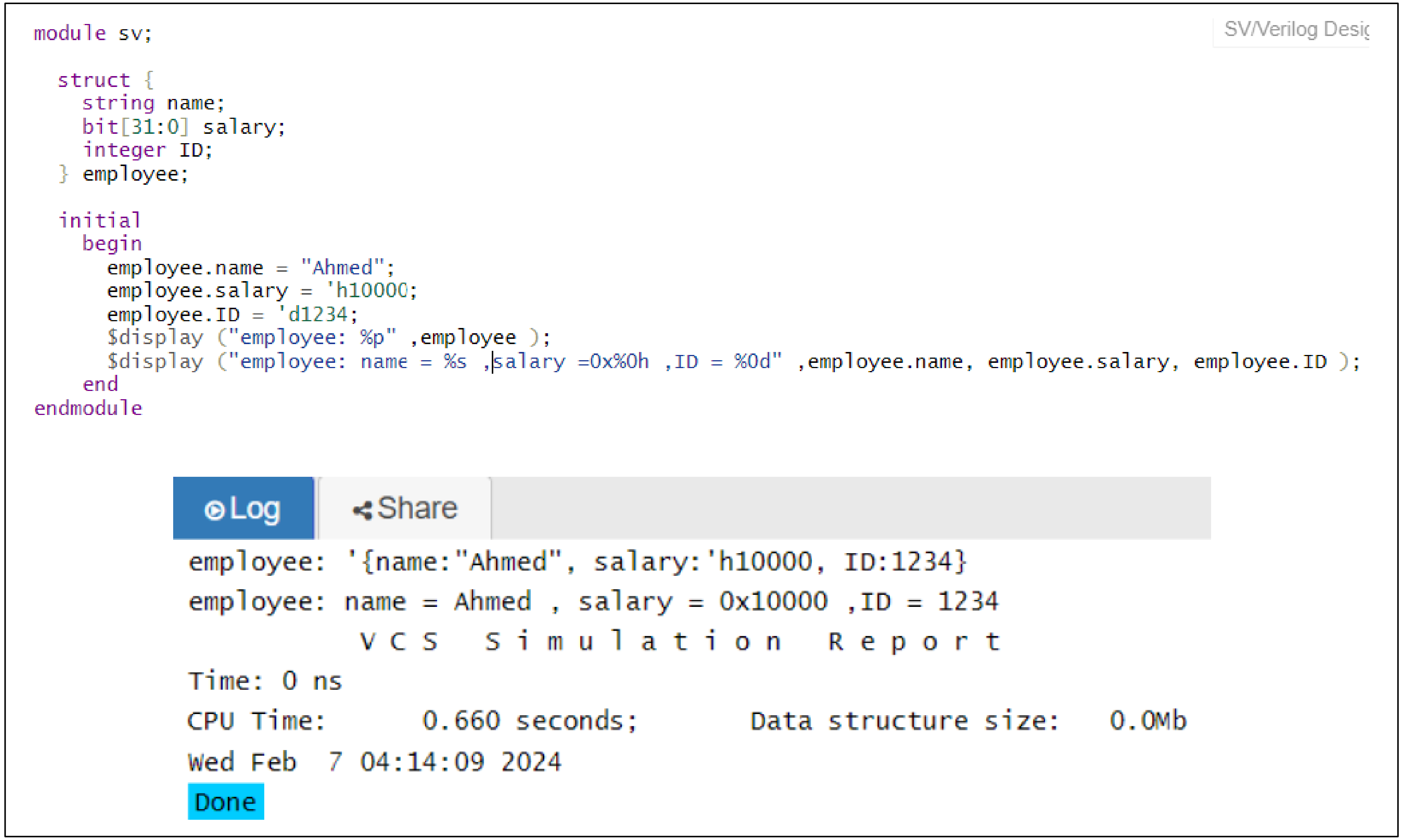
struct – Data Structure

- Contains elements of different data types.
- Struct is unpacked by default.
- The new data type can be used throughout the code, avoiding editing in multiple places if required.
- We can create a user-defined data type of a struct using typedef for multiple struct variables with the same contents



```
module struct_example;
  struct {
    string name;
    bit[31:0] salary;
    integer id;
  } employee;

  initial begin
    employee.name = "Alex";
    employee.salary = 'h10000;
    employee.id     = 'd1234;
    $display("employee: %p", employee);

    // Accessing individual struct member
    $display("employee: name = %s, salary = 0x%0h, id = %0d",
    employee.name, employee.salary, employee.id);

  end
endmodule
```

struct – Data Structure



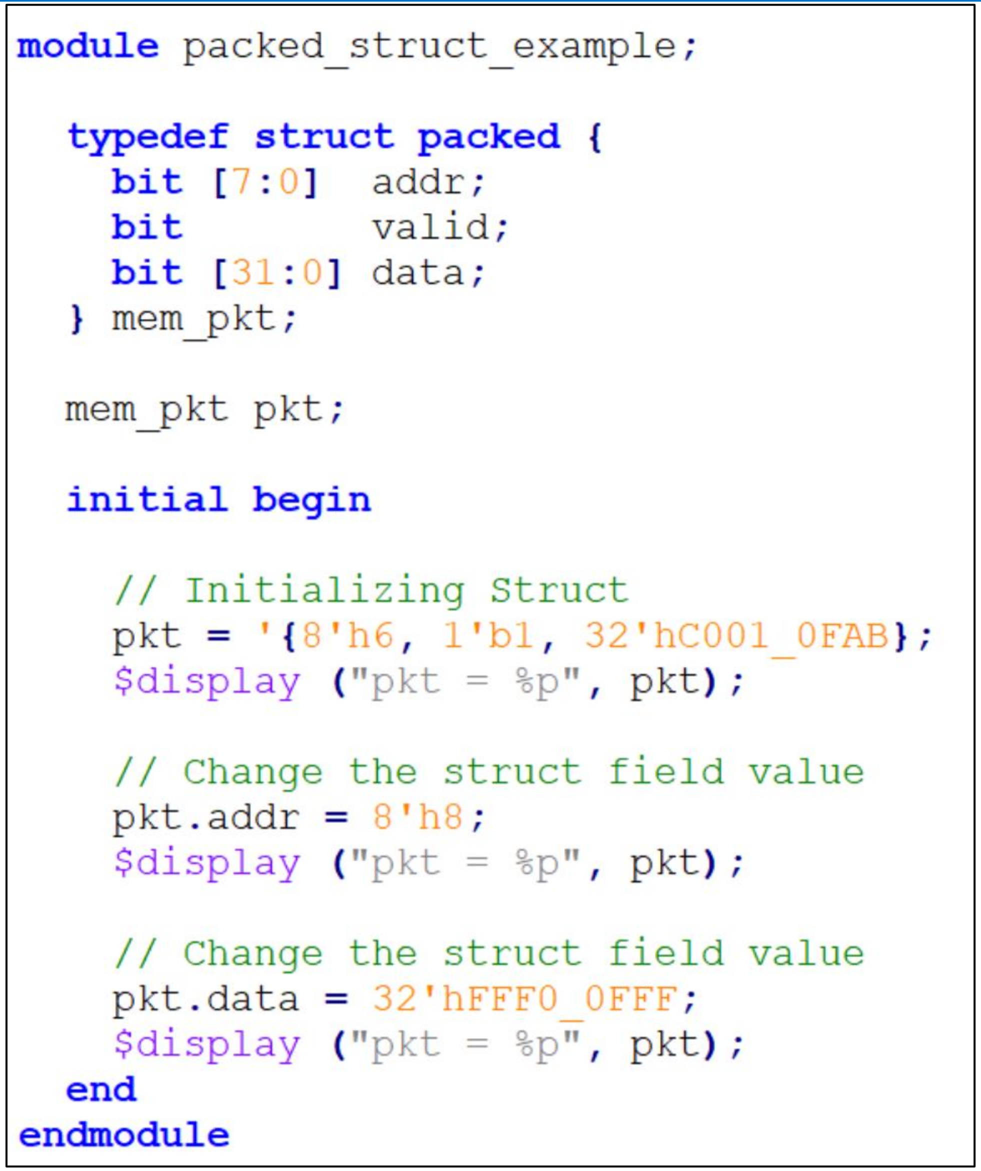
SV/Verilog Desig

```
module sv;

  struct {
    string name;
    bit[31:0] salary;
    integer ID;
  } employee;

  initial
    begin
      employee.name = "Ahmed";
      employee.salary = 'h10000;
      employee.ID = 'd1234;
      $display ("employee: %p" ,employee );
      $display ("employee: name = %s ,salary =0x%0h ,ID = %0d" ,employee.name, employee.salary, employee.ID );
    end
endmodule
```

```
employee: '{name:"Ahmed", salary:'h10000, ID:1234}
employee: name = Ahmed , salary = 0x10000 ,ID = 1234
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.660 seconds;       Data structure size:   0.0Mb
Wed Feb  7 04:14:09 2024
Done
```

struct – Data Structure

- Packed structs
  - struct defined with Packed keyword.
  - Contains only packed data types.
  - Variables are packed together in memory without gabs.



```
module packed_struct_example;

  typedef struct packed {
    bit [7:0]  addr;
    bit        valid;
    bit [31:0] data;
  } mem_pkt;

  mem_pkt pkt;

  initial begin

    // Initializing Struct
    pkt = '{8'h6, 1'b1, 32'hC001_0FAB};
    $display ("pkt = %p", pkt);

    // Change the struct field value
    pkt.addr = 8'h8;
    $display ("pkt = %p", pkt);

    // Change the struct field value
    pkt.data = 32'hFFF0_0FFF;
    $display ("pkt = %p", pkt);
  end
endmodule
```

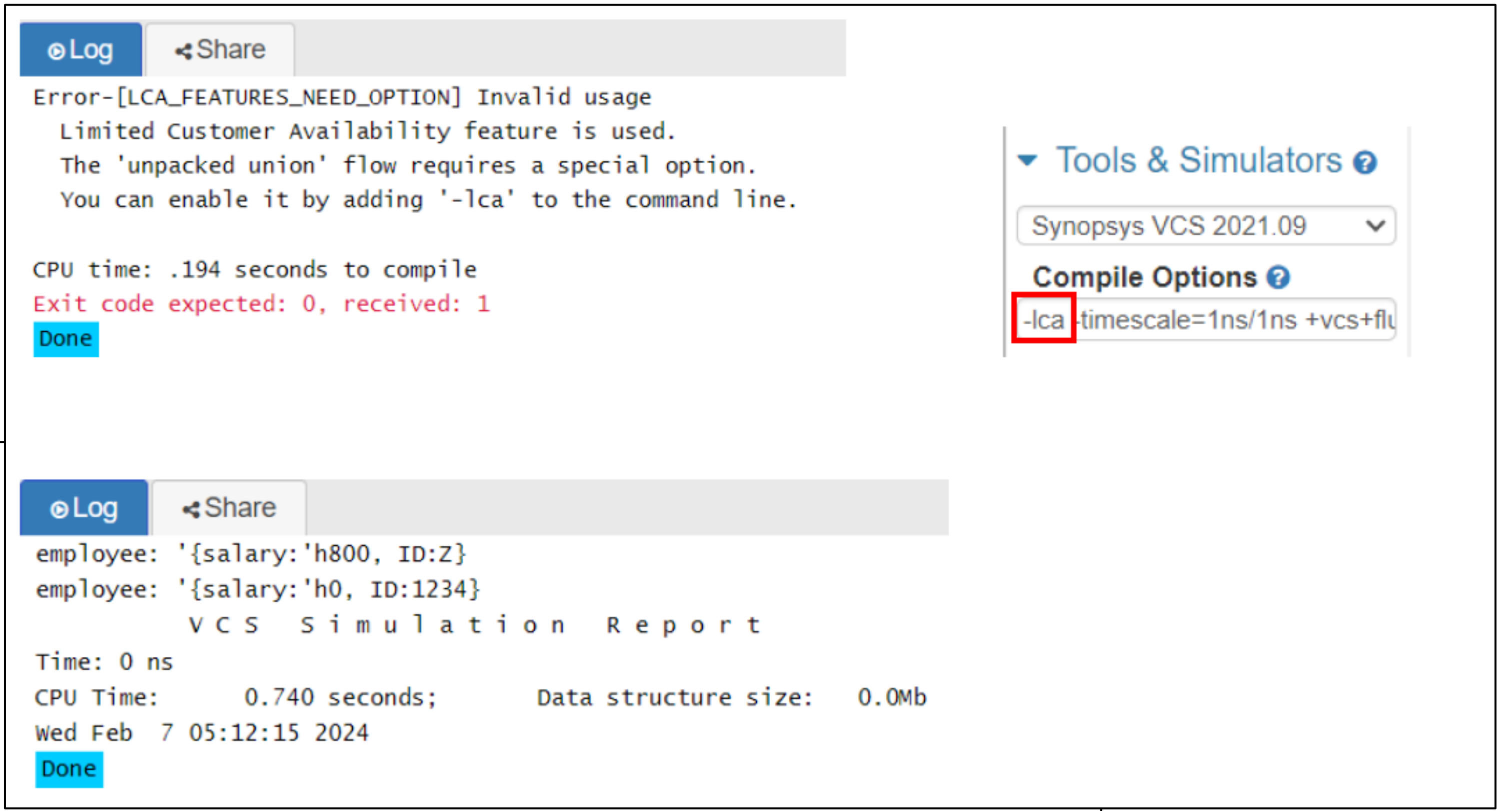
union – Data Union

- Like struct, contain different data types members,
- But they share the same memory location.
- Therefore,
  - Memory efficient data structure.
  - Restricts using one member at a time.

```
module union_example;
  typedef union {
    bit[15:0] salary;
    integer id;
  } employee;

  initial begin
    employee emp;
    emp.salary = 'h800;
    $display("salary updated for EMP: %p", emp);
    emp.id     = 'd1234;
    $display("ID updated for EMP: %p", emp); //Note: Salary information will be lost
  end
endmodule
```

union – Data Union

```
module sv;

  typedef union  {
    bit[15:0] salary;
    integer ID;
  } employee;

  initial
    begin
      employee emp;
      emp.salary = 'h800;
      $display ("employee: %p" ,emp );
      emp.ID = 'd1234;
      $display ("employee: %p" ,emp );         //Note: salary information will be lost
    end
endmodule
```



```
Error-[LCA_FEATURES_NEED_OPTION] Invalid usage
  Limited Customer Availability feature is used.
  The 'unpacked union' flow requires a special option.
  You can enable it by adding '-lca' to the command line.

CPU time: .194 seconds to compile
Exit code expected: 0, received: 1
Done
```

Synopsys VCS 2021.09

Compile Options: -lca -timescale=1ns/1ns +vcs+flu

```
employee: '{salary:'h800, ID:Z}
employee: '{salary:'h0, ID:1234}
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.740 seconds;       Data structure size:   0.0Mb
Wed Feb  7 05:12:15 2024
Done
```

Lab 1: (~ 60 min)

Lab 1: design specs

- In this lab, a model of communication protocol packet construction can be designed using the SystemVerilog data types. The model constructs and displays from 10 to 20 packets of random data.
- Each packet contains the following:
  - Packet ID: 0,1,2…
  - Packet sent time.
  - Packet data that is 32-bit width.
  - Packet Type: Data, Command, Control.
- The packets are constructed and displayed such that:
  - The First packet type is Control.
  - The second packet type is Command.
  - The rest of the packets type is data