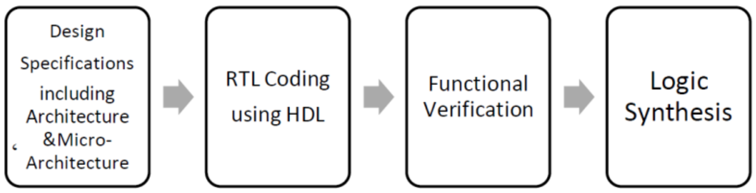
Lab 1: SV Data Types

Objective:

The objective of this lab is to introduce some of the basic verification concepts, and Sysyem Verilog basic data type.

1. Introduction to Digital Design Verification

In a normal chip design flow, we go through a number of pre-defined phases to ensure the best performance and immunity against bugs.


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

1.1 Verification Levels

A typical design phase is divided into multiple blocks as shown.

The design starts with sub-units building up to form units, then core IP blocks forming sub-systems, then into a system on chips till they reach a full board with all chips assembled; accordingly, verification is also divided into multiple levels, each level of verification is suited for a specific objective.

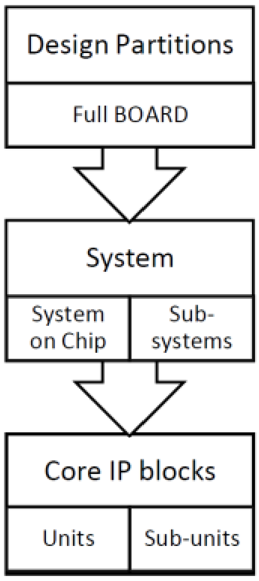
- **Basic verification:** (at Unit/Sub-unit level)

  Where you don't have to make a proper testbench or a verification environment, you can perform ad-hoc basic operations.
- **Functional verification** (at IP blocks level)

  Where verification is done in a parallel, independent method for more efficient flow.
- **System level verification:** (at Systems level)

  Focuses more on interactions and transactions during data flow.
- **Connectivity:** (at Board level)

  To ensure the full connectivity of different components



1.2 Main Verification Methods

- **Simulation-Based Verification:**

  This is the most used verification method where the test vector generator generates a stimulus for the design under test, then a golden reference, checker, or scoreboard is used to generate the expected output, which is then compared to the actual output determining the validity of this design. The test generator's simplicity or complexity depends on the design complexity and randomness; we may also use techniques like assertions and coverage in this verification process.

- **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.

- **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.

1.3 Emulation-Based Verification

Complete hardware mapping into an emulator, an array of FPGAs. It is like a real target system, so speed up is much greater, but debugging would be a great challenge as visibility is limited.
As the DUT is implemented as Hardware on the emulator, and the verification environment is separated into two parts:

1. Synthesizable part runs on the Emulator.
2. Not synthesizable part runs with the Simulation tool on a workstation.

2. SystemVerilog Basic Data Types

SystemVerilog **Data type** is a storage format with a specific range or type. SystemVerilog has different data types to implement efficient models and testbenches. Some of these data types are built in SystemVerilog, and the others are user-defined data types to provide more flexibility and reusability during development.

Type	**Description**	**Example**
Two states data types: have only two possible values 0 or 1		
bit	user-determined range, always unsigned	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;
shortreal	like float in C	shortreal f;
real	like double in C	real g;
realtime	identical to real	realtime now;
Four states data types: have four possible values 0, 1, X, Z		
reg	user-determined size	reg [7:0] a_byte;
logic	identical to reg in every way	logic [7:0] a_byte;
integer	32 bits, signed	integer i, j, k;
time	64-bit unsigned	time now;

Important Notes:

- the default for all data types in SV is logic, but signals with more than one driver needs to be declared as net type such as a **wire** to resolve the final value. **Also when four state data type assigned to two state data type**
- Be careful connecting 2-state variables to the design under test, especially its outputs. If the hardware tries to drive an X or Z, these values are converted to a 2-state value, and your testbench code may never know. Don't try to remember if they are converted to 0 or 1; instead, always check for propagation of unknown values. Use the $isunknown() operator that returns 1 if any bit of the expression is X or Z, as shown in the figure below:

3. String data types

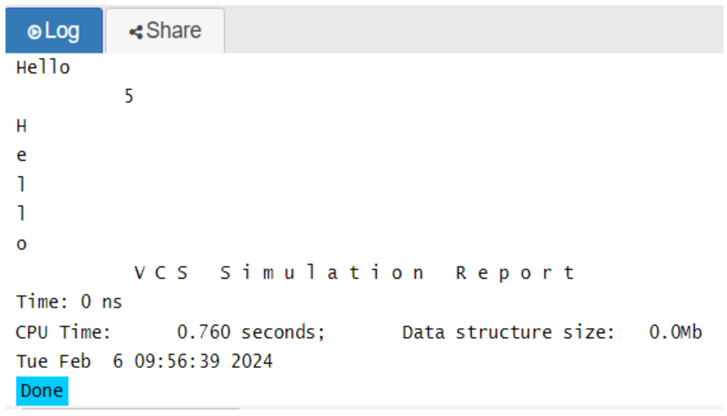
A string is an ordered collection of characters. The length of string equal to it's number of the characters and may vary during simulation.

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

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.

Example (1):

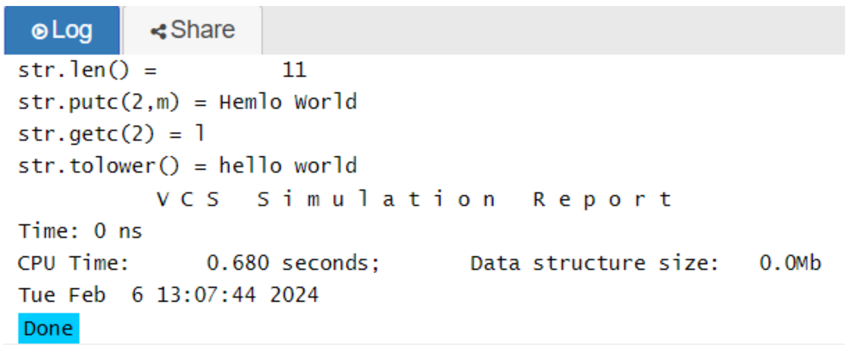
```
module sv();

  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
```

Example (2)

```
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
```

4. Enumeration and TypeDef

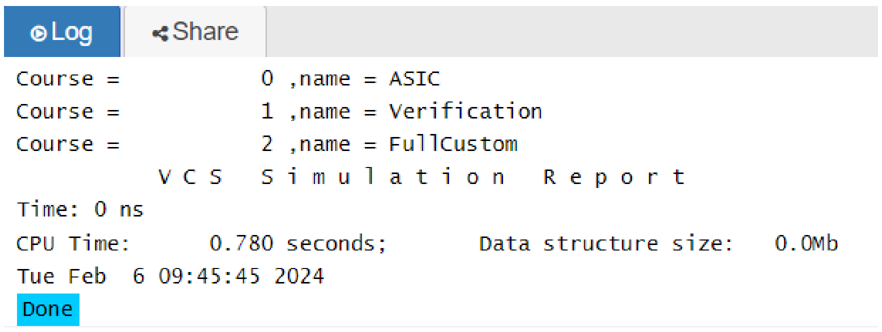
Enumeration allows defining data types whose values have names. It helps represent state values, opcodes, and types. The named values of an enumeration type act like constants. The default type of the eunm is **int**.

SystemVerilog Typedefs gives a user-defined name to an existing data type. As the new data-type can then be used throughout the code and hence avoids the need to edit in multiple places if required. Therefore, allow defining quite complex types.

We can assign any integer value for each enumerated name. If it doesn't have an assigned value, then it automatically takes the incremented value of the previous name.

Example:

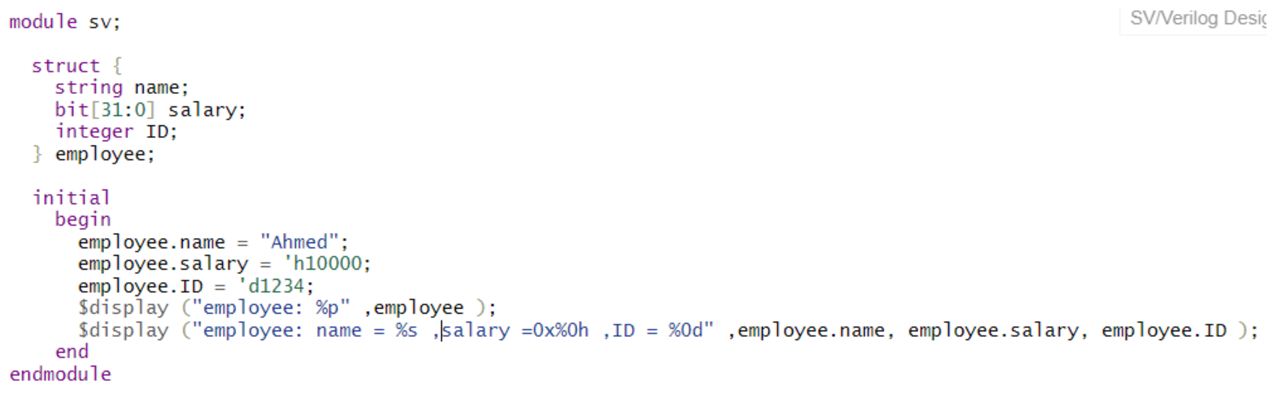
```
module sv();

  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
```



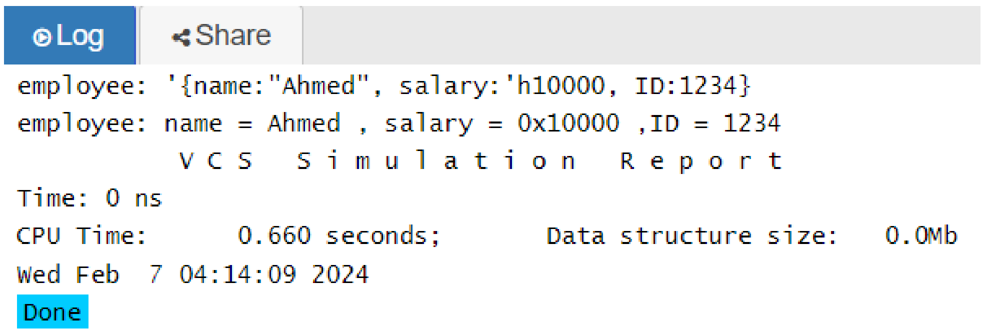
```
Course =          0 ,name = ASIC
Course =          1 ,name = Verification
Course =          2 ,name = FullCustom
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.780 seconds;       Data structure size:   0.0Mb
Tue Feb  6 09:45:45 2024
Done
```

5. Structures/Unions

A SystemVerilog structure contains elements of different data types. The struct is unpacked by default. We can create a user-defined data type of a struct using **typedef** for multiple struct variables with the same contents. For implementing packed structs, a struct is defined using the **Packed** keyword. It **contains only** packed data types (string is not allowed). The example below shows how to declare a structure with elements with different data types.



```
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
```

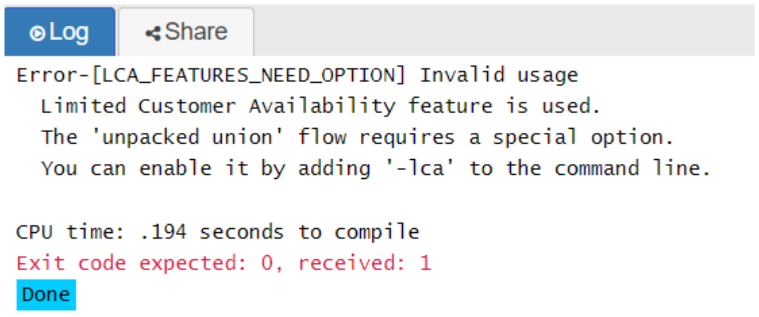


```
employee: '{name:"Ahmed", salary:'h10000, ID:1234}
employee: name = Ahmed , salary = 0x10000 ,ID = 1234
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.660 seconds;       Data structure size:   0.0Mb
Wed Feb  7 04:14:09 2024
Done
```

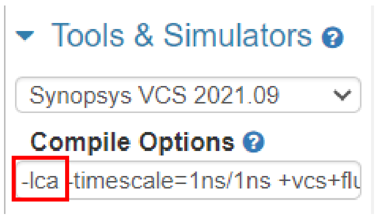
SystemVerilog Union is like struct, as it can contain different data types members, but they share the same memory location. Therefore, the Union data type is an efficient data structure and restricts the use of one member at a time as in the below example.

Example (1):

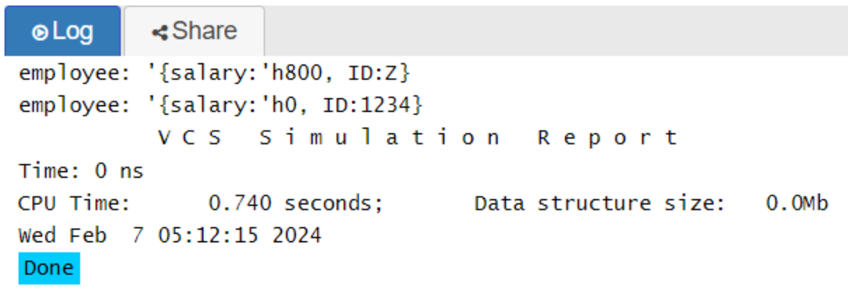
```
module sv;

  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
```



```
-lca -timescale=1ns/1ns +vcs+flu
```



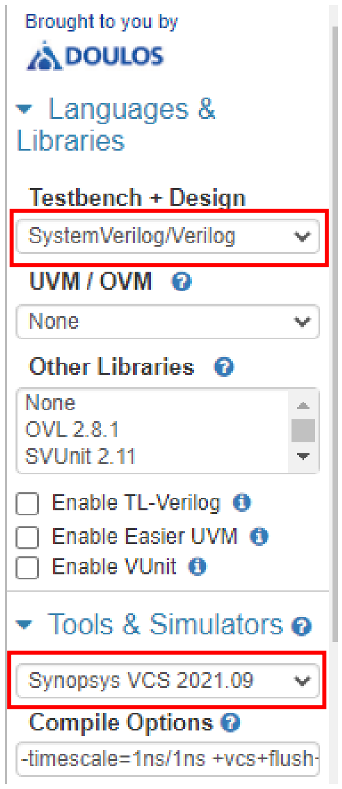
```
employee: '{salary:'h800, ID:Z}
employee: '{salary:'h0, ID:1234}
           V C S   S i m u l a t i o n   R e p o r t
Time: 0 ns
CPU Time:      0.740 seconds;       Data structure size:   0.0Mb
Wed Feb  7 05:12:15 2024
Done
```

6. Practical Part

In this lab, we will use EDA playground to simulate and practice SV. [Click here]

How to setup…

1. **Choose your language**
2. **Choose your simulator**



Testbench + Design: SystemVerilog/Verilog

UVM / OVM: None

Other Libraries: None, OVL 2.8.1, SVUnit 2.11

Enable TL-Verilog ☐

Enable Easier UVM ☐

Enable VUnit ☐

Synopsys VCS 2021.09

Compile Options: -timescale=1ns/1ns +vcs+flush-

Try yourself

In this lab, a model of communication protocol packet construction can be designed using the system Verilog data types. The model constructs and displays from 10 to 20 packets of random data.

Each packet contains the following:

1. Packet ID: 0,1,2…
2. Packet sent time.
3. Packet data that is 32-bit width.
4. Packet Type: Data, Command, Control.

The packets are constructed and displayed such that:

1. The First packet type is Control.
2. The second packet type is Command.
3. The rest packets type is data