Class Based SystemVerilog for verification

Introduction

Legal Notice

- This material is provided as part of AEDVICES Consulting Trainings.
- It is the sole ownership of AEDVICES Consulting.
- Duplication and Copy of this material, partial or complete, is not authorized outside the scope of the training, unless agreed in a separate written agreement.

- Cette présentation fait partie intégrante des formations fournies par AEDVICES Consulting EURL
- AEDVICES Consulting EURL reste le seul propriétaire de ce document.
- Aucune copie, partielle ou totale, n'est autorisée au-delà de la formation fournie, sauf accord écrit fourni par AEDVICES Consulting.

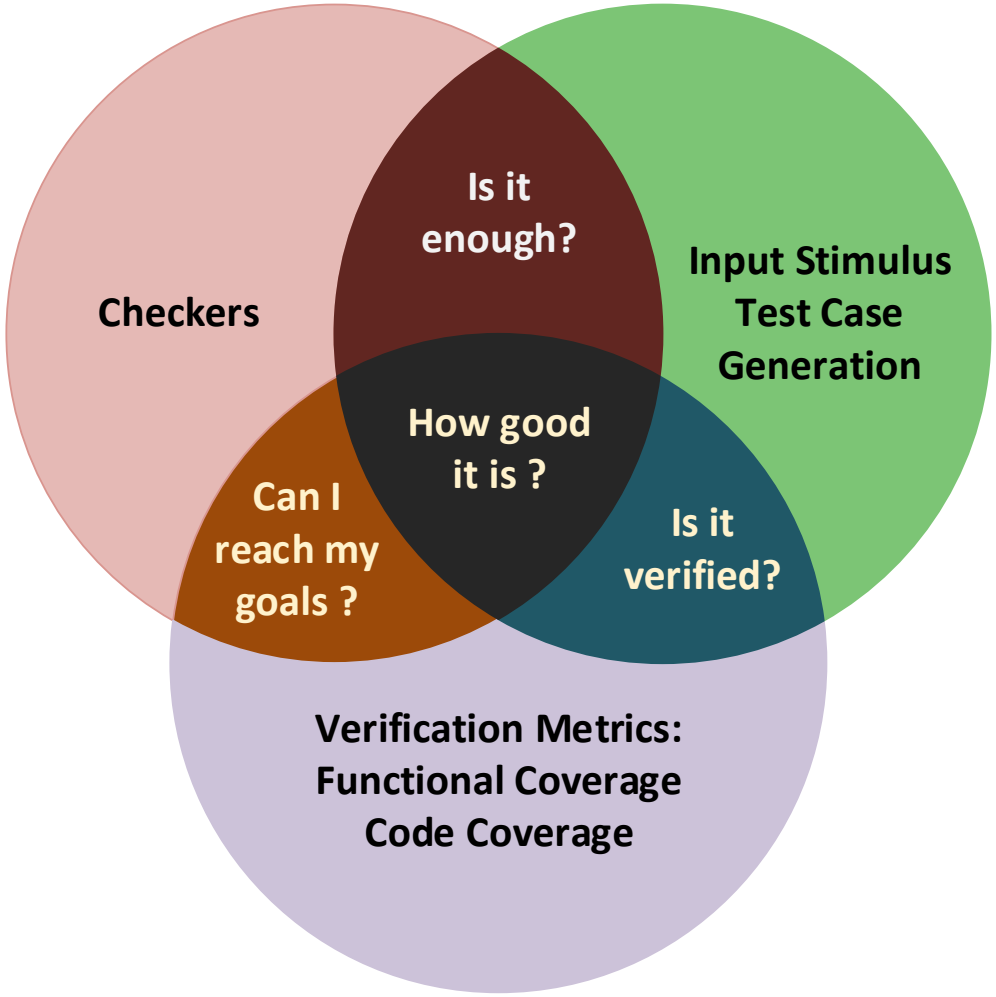
Email contact@aedvices.com for any request.

Session Objectives

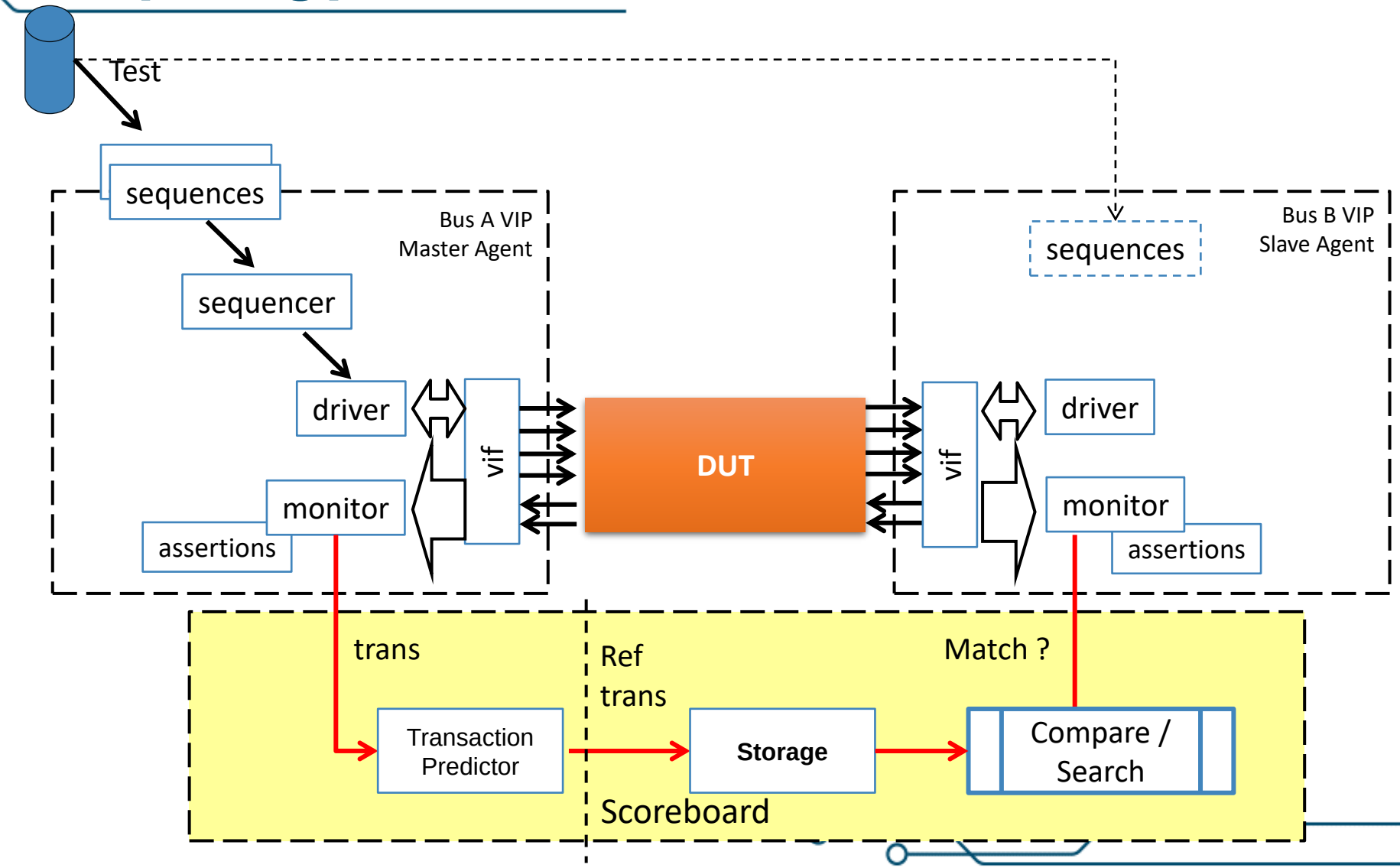
- Learns the basic of SystemVerilog Syntax
- Learns the basic of Class and Objects in SystemVerilog
- Understands the concept of random generation

Coverage Driven Verification

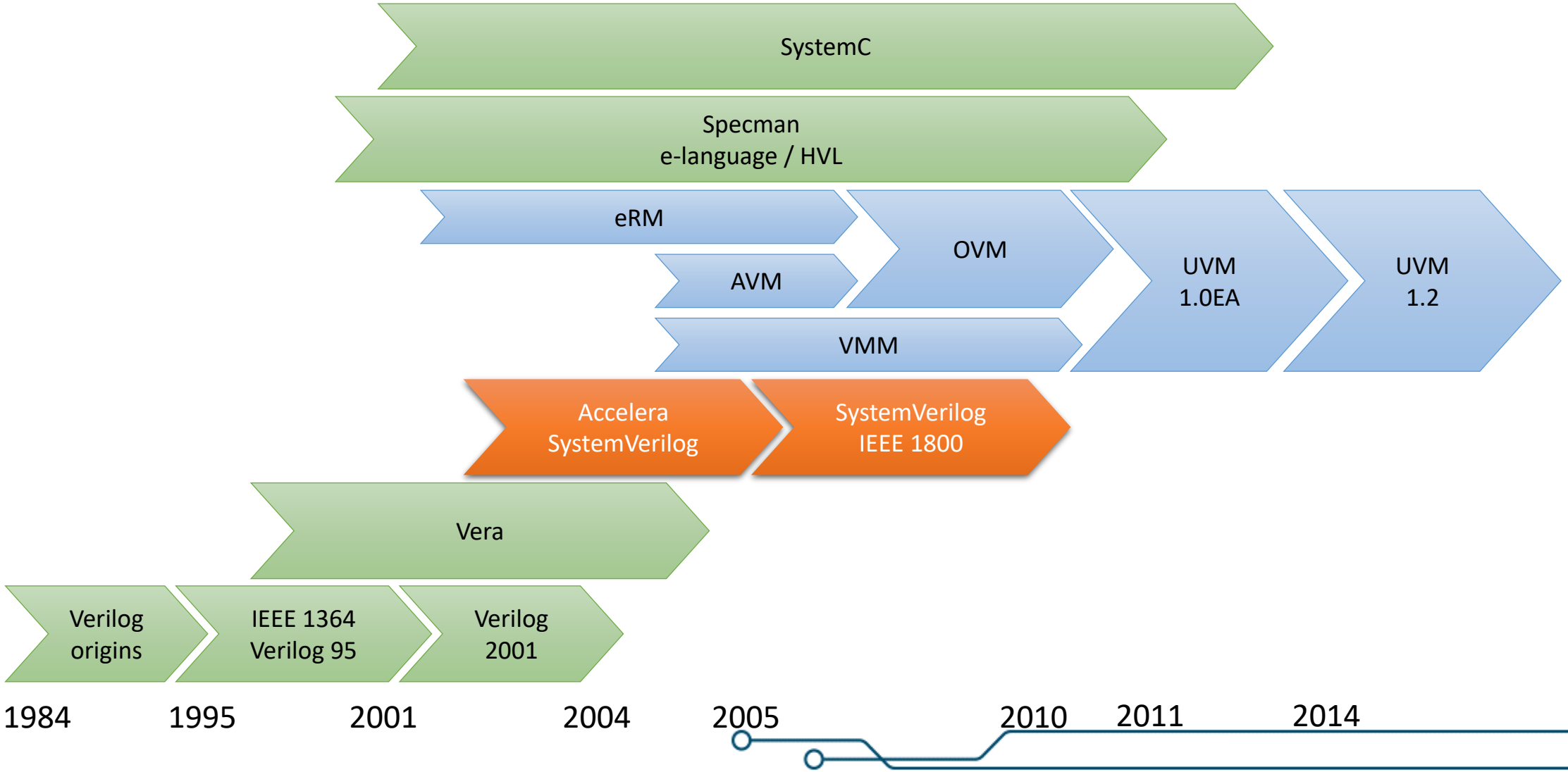
- All system behaviors should be:
  - Exercised (generation)
  - Checked (verification)
  - Monitored (coverage)



Topology of a UVM Environment



A bit of history



SystemVerilog in a nutshell

aedvices CONSULTING

- SystemVerilog is an extension to Verilog
  - Add higher level data structures
  - Add more design constructs
  - Add Object Oriented constructs
  - Add System Level Design constructs
  - Add Verification Language constructs

- Add software aspects to a hardware language
- Add verification constructs to a design language

➔ One language to rule them all !!!

The SystemVerilog 2017 Standard

https://ieeexplore.ieee.org/document/8299595

IEEE STANDARDS ASSOCIATION IEEE

1800-2017 - IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language

Status: Active - Approved

IEEE Std 1800™-2017
(Revision of
IEEE Std 1800-2012)

Front cover (1 sur 1315)

SystemVerilog keywords

stackoverflow

Reserved keywords count by programming language?

Is there a ranking or table of the number of reserved keywords in various programming languages?

C ? **Python ?**

Java ? **SystemVerilog ?**

VHDL ?

SystemVerilog is second on this list !!!

Lists of keywords in ...

- ANSI COBOL 85: 357
- SystemVerilog: 250 + 73 reserved system functions = 323
- VHDL 2008: 115 reserved words
- C#: 79 + 23 contextual = 102
- F#: 64 + 8 from ocaml + 26 future = 98
- C++: 82
- Dart: 54
- Java: 50 (48 without unused keywords `const` and `goto`)
- PHP: 49
- Ruby 42
- JavaScript: 38 reserved words + 8 words reserved in strict mode only
- Python 3.7: 35
- C: 32
- Python 2.7: 31
- Go: 25
- Elm : 25
- CoffeeScript: 19, not necessarily "reserved", plus ~50 to avoid from JS
- Smalltalk: 6 pseudo-variables
- iota: 2

SystemVerilog keywords

aedvices CONSULTING

Table B.1—Reserved keywords

accept_on	default	forkjoin
alias	defparam	function
always	design	generate
always_comb	disable	genvar
always_ff	dist	global
always_latch	do	highz0
and	edge	highz1
assert	else	if
assign	end	iff
assume	endcase	ifnone
automatic	endchecker	ignore_bins
before	endclass	illegal_bins
begin	endclocking	implements
bind	endconfig	implies
bins	endfunction	import
binsof	endgenerate	incdir
bit	endgroup	include
break	endinterface	initial
buf	endmodule	inout
bufif0	endpackage	input
bufif1	endprimitive	inside
byte	endprogram	instance
case	endproperty	int
casex	endspecify	integer
casez	endsequence	interconnect
cell	endtable	interface
chandle	endtask	intersect
checker	enum	join
class	event	join_any
clocking	eventually	join_none
cmos	expect	large
config	export	let
const	extends	liblist
constraint	extern	library
context	final	local
continue	first_match	localparam
cover	for	logic
covergroup	force	longint
coverpoint	foreach	macromodule
cross	forever	matches
deassign	fork	medium

Table B.1—Reserved keywords *(continued)*

modport	reject_on	time
module	release	timeprecision
nand	repeat	timeunit
negedge	restrict	tran
nettype	return	tranif0
new	rnmos	tranif1
nexttime	rpmos	tri
nmos	rtran	tri0
nor	rtranif0	tri1
noshowcancelled	rtranif1	triand
not	s_always	trior
notif0	s_eventually	trireg
notif1	s_nexttime	type
null	s_until	typedef
or	s_until_with	union
output	scalared	unique
package	sequence	unique0
packed	shortint	unsigned
parameter	shortreal	until
pmos	showcancelled	until_with
posedge	signed	untyped
primitive	small	use
priority	soft	uwire
program	solve	var
property	specify	vectored
protected	specparam	virtual
pull0	static	void
pull1	string	wait
pulldown	strong	wait_order
pullup	strong0	wand
pulsestyle_ondetect	strong1	weak
pulsestyle_onevent	struct	weak0
pure	super	weak1
rand	supply0	while
randc	supply1	wildcard
randcase	sync_accept_on	wire
randsequence	sync_reject_on	with
rcmos	table	within
real	tagged	wor
realtime	task	xnor
ref	this	xor
reg	throughout	

Done !

- This is it.
- You know all about SystemVerilog

Session Content

aedvices CONSULTING

SystemVerilog for Verification

- Review of the main Verilog language elements
- Blocks and Control Flow
- Data Types
- Class Based / Object Oriented Programming with SV
- Constrained Random Variables

Verilog Syntax

- Case sensitive
- Keywords all in lower case
- Comments using // or /* */
- Mnemonics
  - starts with letters
  - contain letters, digits or underscore _

    regexpr: [a-zA-Z] [a-zA-Z0-9_]*
- Statements
  - Terminated with semi-colon ;

Keywords

lways	**end**	**initial**	**output**	scalared	triand
and	**endcase**	**inout**	pmos	small	trior
assign	**endfunction**	**input**	**posedge**	specify	vectored
begin	**endmodule**	**integer**	primitive	specparam	**wait**
buf	endprimitive	**join**	**pull0**	strong0	wand
bufif0	endspecify	large	**pull1**	strong1	weak0
bufif1	endtable	macromodule	**pulldown**	supply0	weak1
case	**endtask**	medium	**pullup**	supply1	**while**
casex	**event**	**module**	rcmos	table	**wire**
casez	**for**	nand	**reg**	**task**	wor
cmos	force	**negedge**	release	**time**	xnor
deassign	**forever**	nmos	**repeat**	tran	**xor**
default	**fork**	nor	rnmos	tranif0	
defparam	**function**	**not**	rpmos	tranif1	
disable	highz0	notif0	rtran	tri	
edge	highz1	notif1	rtranif0	tri0	
else	**if**	**or**	rtranif1	tri1	

SV Basics

- SV is Verilog compatible
- Verilog Recap
  - Module
  - Always
  - Initial
  - Blocking assignments
  - Wire, reg, …

  - Statements ends with ;
  - Blocks are defined with
    - begin
    - end

```
// This is a comment
`timescale 10ns/10ps
module uart( clock, reset,
             req , gnt , reg_addr, reg_data_in ,
             reg_data_out, dvalid ,
             tx , rx);
   input clock;
   input reset;
   input reg_addr;
   wire [31:0] reg_addr;
//...
   always @(posedge clock)
      if reset
          reg1 <= 32'h0000;
      else
        begin
          if ( req & reg_address == 'h00001 )
            begin
              gnt <= 1;
              reg1 <= reg_data_in;
            end
          else
            //....

        end
endmodule;
```

Verilog vs VHDL

aedvices CONSULTING

Verilog

- module
- always
- function
- task

VHDL

- entity / architecture
- process
- function
- procedure

Literals / constants

- Literal values can be:
  - Signed or not
  - Of a fixed width or not
  - In hexadecimal, decimal, octal, binary
- Syntax:

  [-] [width]’[s][base][value]
- Example:

  ’sheba

  32’o1234

  4’b0101

Session Content

aedvices CONSULTING

SystemVerilog for Verification

- Review of the main Verilog language elements
- Blocks and Control Flow
- Data Types
- Class Based / Object Oriented Programming with SV
- Constrained Random Variables

Verilog Control structures

- blocks:
  - begin / end

- Conditions

```
if (condition)
   action_A;
 else
   action_B;
```

```
if (condition)
   begin
      action_A;
      action_B;
   end
 else
   action_C;
```

```
case (variable)
   value0: action0; break;
   value1: action1; break;
   default: default_action;
 endcase
```

Additional Verilog Control

- Forever loop

```
initial
   begin
     clk <= 0;
     forever #10 clk <= ~clk;
   end
```

- Repeat loop

```
initial
   repeat (10)
     do_something(data);
```

- While loop

```
initial
   while ( ! req )
      gnt <= 0
```

- Do while

```
do
     action1();
 while ( condition );
```

Operators

Arithmetic: + - * / % **
Relations: < <= > >= == !=
Bit-Wise: ~ & | ^ ~^ ^~
Logical: ! && ||

Reductions: & | ~& ^ ~^
Shift: >> <<
Concatenation: { a , b }
Replication: { n { item } }
Conditional: c ? A : B

```
a = 'b101;
b = 'b110;
Z = 'b111;

c = a & b; // c == ‘b100
D = a && b; // D== ‘b1

E =  & a; // E == 0
F = & z ; // F == 1

G = { a , b } ; // G = ‘b101110
H = { 2 { a } }; // H = ‘b101101
```

Operator Precedence

Highest priority

Operator	Name
[ ]	bit-select or part-select
( )	parenthesis
!, ~	logical and bit-wise NOT
&, |, ~&, ~|, ^, ~^, ^~	reduction AND, OR, NAND, NOR, XOR, XNOR; If X=3'B101 and Y=3'B110, then X&Y=3'B100, X^Y=3'B011;
+, -	unary (sign) plus, minus; +17, -7
{ }	concatenation; {3'B101, 3'B110} = 6'B101110;
{{ }}	replication; {3{3'B110}} = 9'B110110110
*, /, %	multiply, divide, modulus; */ and % not be supported for synthesis*
+, -	binary add, subtract.
<<, >>	shift left, shift right; X<<2 is multiply by 4
<, <=, >, >=	comparisons. Reg and wire variables are taken as positive numbers.
==, !=	logical equality, logical inequality
===, !==	case equality, case inequality; not synthesizable
&	bit-wise AND; AND together all the bits in a word
^, ~^, ^~	bit-wise XOR, bit-wise XNOR
|	bit-wise OR; AND together all the bits in a word
&&	logical AND. Treat all variables as False (zero) or True (nonzero).
||	logical OR. (7||0) is (T||F) = 1, (2||-3) is (T||T) =1, (3&&0) is (T&&F) = 0.
? :	conditional. x=(cond)? T : F;

Lowest priority

```
// common mistake:
// make bit 0 of A
// and compare with 0
  if ( A & 1 == 0 )
    dead_code();
  else
    always_executed();
```

Arithmetic Operators

```
c = a + b;      // add
c = a - b;      // substract
c = a * b;      // multiply
c = a / b;      // divide
c = a % b;      // modulo
c = a ** b;     // exponent : a^b
c = +a;         // positive
c = -a;         // negative
```

Comparison Operators

```
if (a < b)        // smaller than
if ( a > b )      // greater than
if ( a <= b )     // smaller or equal
if ( a >= b )     // greater or equal
if ( a == b)      // equal
if ( a != b )     // different
if ( a === b )    // 4-state equal
if ( a !== b )    // 4-state different
```

Logical / Binary operators

aedvices CONSULTING

Logical

```
a && b          // Logical AND
a || b          // Logical OR
!a              // Logical NOT
```

Binary

```
a & b           // bit-wise and
a | b           // bit-wise or
a ^ b           // bit-wise xor
a ~^ b          // bit-wise xnor
~a              // bit-wise not
```

Other operators

aedvices CONSULTING

- Reduction

```
&a  // bit-wise and of all bits of 'a'
|a        // bit-wise or of all bits of 'a'
^a        // bit-wise xor of all bits of 'a'
```

- Shift

```
a << n    // a shifted to left by n bits
a >> n    // a shifted to right by n bits
```

- Conditional

```
C ? A : B          // A if C is true, B otherwise
```

Concatenation & repetition

```
{ A , B }                          // vector composed of A and B
{ 3'b101 , b[2:0] , a[3] }         // vector of 7 bits
{ 6 { a } }                        // vector composed of 6 times a,
                                   // equivalent to { a , a , a , a , a, a, }
{ 4{a} , b[3:0] , 3'b101 }
```

Note: there is no restriction in using the concatenation on the left hand side of the assignment.

```
assign { MSB , LSB } = a + b;
```

Task / Function

aedvices CONSULTING

- A task can
  - belong to a module
  - have inputs
  - have outputs
  - use its own variables
  - return value
  - have time consuming actions (waits)
- A task must not
  - Use wires

- A function can
  - belong to a module
  - have input
  - have output
  - use its own variables
  - return value
- A function must not
  - Use time consuming actions

Task example

```
task getMinimum;
   input  [15:0] a,b, clk;
   output [15:0] retval;

   begin
     @(posedge clk);
     retval <= ( a < b ) ? a : b;
   end

 endtask

 reg val1,val2,clk,result;
 initial
   begin
     getMinimum(val1,val2,clk,result);
   end
```

A function example

```
function [7:0] getMax;
   input [7:0] a,b;
   begin
     getMax = ( a < b ) ? b : a ;
   end
 endfunction

 assign c = getMax( m , n );
```

Session Content

aedvices CONSULTING

SystemVerilog for Verification

- Review of the main Verilog language elements
- Blocks and Control Flow
- Data Types
- Class Based / Object Oriented Programming with SV
- Constrained Random Variables

SV Data Types

- Verilog has 4-state integer values (bit values in 0, 1, Z, X)
- SystemVerilog adds 2-state integer values (bit values in 0 , 1)

2-state	4-state
bit	logic
int, shortint, longint, byte	integer
	reg, wire, time

- "Unsigned"
- Real and Short Real (same as C-double and C-float)
- Char Strings
- Void !
- Chandles (class handles through DPI)
- Event
- User defined types
- Structs and Unions

SV Data Types example

```
shortint unsigned data;      // 16 bits unsigned data
bit [23:0] address;          // 24 bits address
string myName = "Francois"; // this is me
byte c = "F";                // a byte which takes the
                             // char value "F"
event done;
typedef enum {READ,WRITE,NOP} direction_t;
typedef bit [23:0] address_t;
typedef struct packed unsigned {
    address_t addr;
    shortint data;
} transaction_s;
union { int phy_val; shortreal real_val; } a;
```

Agregate Data Types

aedvices CONSULTING

SystemVerilog Agregate Data types:

- Structures
- Unions
- Arrays
- Dynamic arrays
- Associative Arrays
- Queues

structures

- A structure is a collection of fields, each of their own type

```
// Structure
program foo1;
  struct {
    bit[31:0] addr;
    bit[31:0] data;
  } A;

  A.data = 12;
endprogram
```

```
program foo2;
  typedef struct {
    bit[31:0] addr;
    bit[31:0] data;
  } trans_t;
  trans_t A;

  A.data = 12;
endprogram
```

Others

- Packed Structures
- Arrays
- Dynamic Arrays
- Keyed List / Dict / Hash
- Queues

- Will be seen next time

Session Content

aedvices CONSULTING

SystemVerilog for Verification

- Review of the main Verilog language elements
- Blocks and Control Flow
- Data Types
- Class Based / Object Oriented Programming with SV
- Constrained Random Variables

Classes

aedvices CONSULTING

- *Looks like* C++ or Java classes:
  - Encapsulation of data/properties
  - Encapsulation of methods
    - Functions
    - Tasks
  - Inheritance and polymorphism
  - Possible dynamic allocation (Wow !!! like in software!!!)
  - Parameterization
  - Virtual methods
  - Static properties and methods
  - Class constructor
  - No Destructor → Memory Garbage Collection (rely on the tool)

Definitions

- A class is a type container which includes:
  - Data
  - Subroutines (tasks and functions) that operate on these data

- Class Properties: The class data
- Class Methods: the class tasks and functions
- Object: an instance of a class

Class Syntax

IEEE SystemVerilog

```
class_declaration ::=                                          // from A.1.2
    [virtual ] class [ lifetime ] class_identifier [ parameter_port_list ]
        [ extends class_type [ ( list_of_arguments ) ] ]
        [ implements interface_class_type { , interface_class_type } ] ;
        { class_item }
    endclass [ : class_identifier]
```

Simple Class Example

```
class cBaseTransfer;
   bit [31:0] address;
   bit [64:0] data;
   tOpcode    opc;

   // create an Transfer for a register
   function RegWrite(int regID, tData regData);
      opc = WRITE4;
      address = 0'h38000000 + regID * 4;
      data    = regDATA & 0'bFFFF;
   endfuncion;
endclass;
```

Constructor

```
class cBaseTransfer;
   bit [31:0] address;
   bit [64:0] data;
   tOpcode    opc;

   // constructor
   function new();
       address = 0;
       data    = 0;
       opc     = IDLE;
   endfunction;

  // …
endclass;
```

```
CBaseTransfer tr = new;
```

Note: the constructor new does not return any type, not even void

Class inheritance

```
class cMyTransfer extends cBaseTransfer;
   bit secure;
   bit cacheable;
   bit [10:0] transactionID;

   // constructor
   function new();
      secure = 0;
      cacheable = 0;
      transactionID = 0;
   endfunction;

   // create an Transfer for a register
   function RegWrite(int regID, tData regData);
        super.RegWrite(regID, regData);
        secure = 1; cacheable = 0;
   endfuncion;
endclass;
```

this, super and $unit

aedvices CONSULTING

this

refers to the current object instance of the object class

super

refers to the parent class, allowing direct access to the parent class public and protected methods and properties.

$unit

refers to the unit name space

This & Super Example

```
int a = 1;

class parent;
  int a = 2;
  function new(int a=3);
    this.a = a + 5;
  endfunction // new
endclass // base

class child extends parent;
  int a = 7;
  function new(int a);
    this.a = a + super.a + $unit::a + 13;
  endfunction
endclass // child
```

```
program foo;
  parent p;
  child c;

  initial
  begin
    c = new(19);
    p = c;
```

```
# Loading work.foo(fast)
VSIM 4> run
# a =                1
# c.a =             41
# p.a =              8
```

Encapsulation: global/local/protected

- Class methods and properties can be:
- Public/Global:
  - default (when not specified)
  - visible from any scope, other objects
  - are inherited
- Local/Private
  - Keyword modifier: **local**
  - Only visible within the class method scopes
- Protected
  - Keyword modifier: **protected**
  - Only visible within the class scope and all subclasses
  - Can be inherited

Encapsulation: local and protected

```
class cBaseTransfer;
   local shortint ID;
   protected function init();
       this.ID = 0;
   endfunction;
   bit [31:0] address; // public
endclass;

class cMyTransfer extends cBaseTransfer;
   local shortint ID;  // not the same as in parent class
   protected function init();
       this.ID = 1;
   endfunction;

   function do_something();
      address = REG_A_ADDR;
   endfunction;
endclass;
```

Virtual Methods

- Pure virtual are not defined by the parent class, they provide a prototype.
- Defined in the subclass
- Calling a virtual method of an object calls the method of the subclass object instance

Virtual Methods example

```
class BasePacket;
  // no default implementation, just provide the prototype
  virtual function integer send(bit[31:0] data);
    // default implementation
    // ...
  endfunction
endclass

class EtherPacket extends BasePacket;
  virtual function integer send(bit[31:0] data);
    // EthernetPacket Send Implementation
    //...
  endfunction
endclass
```

```
initial
    begin
      BasePacket packets[3];
      EtherPacket ep = new; // extends BasePacket
      TokenPacket tp = new; // extends BasePacket
      GPSSPacket gp = new; // extends EtherPacket
      packets[0] = ep;
      packets[1] = tp;
      packets[2] = gp;
      packets[0].send(); /// same as ep.send()
    end
```

Virtual Class example

```
virtual class BasePacket;
   // no default implementation, just provide the prototype
   pure virtual function integer send(bit[31:0] data);
 endclass

 class EtherPacket extends BasePacket;
   virtual function integer send(bit[31:0] data);
     // body of the function
     //...
   endfunction
 endclass
 initial
   begin
     EtherPacket ep = new; // extends BasePacket
     TokenPacket tp = new; // extends BasePacket
     GPSSPacket gp = new; // extends EtherPacket
     packets[0] = ep;
     packets[1] = tp;
     packets[2] = gp;
   end
```

Static properties and methods

- **Static** properties are shared between all instances of a class.
- **Static** properties do not necessitate a object instance to be accessed

- **Static** methods can be called with no object instance.
- **Static** methods can access static properties

Static properties and methods

```
class BasePacket;
   static shortint ID = 0;
   function new();
     ID = ID + 1;
   endfunction

   static function shoftint get_nr_instances();
     return ID;
   endfunction
 endclass

 initial
 begin
   BasePacket bp;
   int a = BasePacket::get_nr_instances(); // returns 0
   bp = new();
   a = BasePacket::get_nr_instances(); // returns 1
   a = bp.get_nr_instances();          // returns 1
   bp = new();
   a = BasePacket::get_nr_instances(); // returns 2
 end
```

Useful for implementing the singleton pattern, or unique ID

Out of Block Declaration (like C++)

- Class prototype and implementation can be split

```
/// File: packet.svh
class Packet;
  // ..
  extern protected virtual function int send(int data);
endclass
```

```
/// File: packet.sv
function int Packet::send(int data);
  //...
endfunction
```

Parameterized classes

- Identical to C++ template

```
// scalar parameter
class vector #(int size = 1);
  bit [size-1:0] a;
endclass

// type parameter
class stack #(type T=int);
  local T items[];
  extern task push ( T a );
  extern task pop ( ref T a );
endclass
```

Functions / Tasks

- Same as Verilog Tasks
- Equivalent to VHDL process
- Can be in encapsulated in a class
  - Monitors
  - Drivers
  - Processs that consumes time

Taks and Methods example as part of a class

```
class CTransaction ;
  int addr;
  int data;

  task drive();
    tb.req <= 1;
    tb.addr_o <= addr;
    @(posedge tb.clock);
    tb.addr_o <= data;
    @(posedge tb.clock);
    tb.req <= 0;
  endtask

  function int get_value();
    return tb.data_i;
  endfunction

  function void start_trans();
    fork
      this.drive();
    join_none
  endfunction
endclass
```

```
initial
  begin
    CTransaction t = new();
    t.drive();          // consumes time
    t.start_trans();    // returns immediatly
  end
```

Type Casting

aedvices CONSULTING

Scalar Type Casting

```
DEST_VAR = TYPE'(SRC_VAR);
```

Class Casting

```
success = $cast ( DST_OBJ_PTR , SRC_OBJ_PTR )
```

Scalar Type Casting Example

```
typedef enum { RED=0, BLUE=1, YELLOW=2 ) color_t;
typedef enum { TRIANGLE=0, RECTANGLE=1, CIRCLE=2 } shape_t;

 initial
   begin
     color_t c = BLUE;
     shape_t s;
     s = shape_t'(c); // s == RECTANGLE !!!
   end
```

Class Casting example

```
typedef class TransferBase;
typedef class TransferEthernet; // extends TransferBase
```

```
TransferBase base;
TransferEthernet src, dst;

src = new();
base = src;              // Object of parent class can be assigned to a child
if ( $cast(dst,base) )   // Child cannot be directly assigned to a parent object.
  return 1;              // Cast is successful if the base pointer is actually an
                         // instance of the child class
else
  return 0;              // Failed otherwise
```

Session Content

aedvices CONSULTING

SystemVerilog for Verification

- Review of the main Verilog language elements
- Blocks and Control Flow
- Data Types
- Class Based / Object Oriented Programming with SV
- Constrained Random Variables

Random Variables

- SystemVerilog provides support for constrained random generation

- Keyword « rand » makes a properties randomizable
- Method « randomize() » generate all randomizable properties of a class

Random Variable Generation

```
class rndPacket;
   rand bit [15:0] addr;
   rand bit [31:0] data;
   rand tDirection dir;
endclass

function bit do_random_packet ( rndPacket p);
   bit success;
   success = p.randomize();
   addr = p.addr;
   data = p.data;
   return sucess;
endfunction
```

Constraints

- Constraints limits the range of random values
- If constraints cannot be satisfied, randomize() returns FALSE

Constraint Example

aedvices CONSULTING

- Constraints:
  - A in [3..14]
  - C in [12..15]
  - A<B
  - B<C

```
class tempo_config_c;
   rand uint a;
   rand uint b;

   constraint a_less_than_b { a < b; }
   constraint a_range { a >= 3 && a <= 14; }
endclass
```

- Other constraint examples:
  - Register accesses should be in a certain address range
  - Register accesses are 32 bits wide
  - Read only registers should not be written to
  - Invalid opcodes should not be generated
  - A branch should be followed by a NOP
  - Time between two transactions should be at least two clock cycles
  - …

Constraint Solver Problem

- Following the constraints
  - A in [3..14]
  - C in [12..15]
  - A<B
  - B<C
- What if A is generated first and take value 14 ?

- Constraint Solver are complex algorithms
- Solving constraints is a NP-Complete Problem
  - Involve graph theories, mathematical reduction, backward search, …
  - Hopefully, EDA vendors have developed tools for us

Constrained Generation

```
class rndPacket;
   rand bit [15:0] addr;
   rand bit [31:0] data;
   rand tDirection dir;

   constraint addr0 { addr == 0 -> dir == READ };
endclass

function bit do_random_packet ( rndPacket p);
   bit success;
   success = p.randomize() with { addr == 0; };
   addr = p.addr;
   data = p.data;
   return sucess;
endfunction
```

Constraint Example

```
/// \brief constrains the address depending on cfg
constraint address_c {
  if ( cfg != null ) {
    if ( direction == WRITE ) address <= ( ( 1 << cfg.write_address_bus_width ) - 1);
    if ( direction == READ  ) address <= ( ( 1 << cfg.read_address_bus_width ) - 1);
  }
}

constraint data_c {
  if ( cfg != null ) {
    foreach ( transfers[ii] ) {
      if ( direction == WRITE ) transfers[ii].data <= ( ( 1 << cfg.write_data_bus_width ) - 1);
      if ( direction == READ  ) transfers[ii].data <= ( ( 1 << cfg.read_data_bus_width ) - 1);
    }
  }
}
```

Constraint additional example

```
constraint response_delay_c {
    response_delay >= 0;

    if ( cfg == null ) soft response_delay inside { [0:50] };

    if ( cfg != null ) {
      if ( direction == WRITE ) {
       response_delay >= cfg.delay_wready2bvalid_min;
       response_delay <= cfg.delay_wready2bvalid_max;
      }
      if ( direction == READ ) {
       response_delay >= cfg.delay_arready2rvalid_min;
       response_delay <= cfg.delay_arready2rvalid_max;
      }

      address inside { [ 'h00000000:'h50000000] } -> data == 0;
    }
  };
```

Constraint Types

aedvices CONSULTING

Boolean / Relational

```
constraint relation_c { (address < max_address ) || ( data == 0 ) ; }
```

Range

```
constraint range_c    { address inside { [0:100] , 200 , [300:400] } ; }
```

Implication

```
constraint implication_c { access == REG -> address inside { [250:300] } ; }
```

Conditional

```
constraint conditional_c { if ( access == REG ) { address inside { [ 250:300] } ; } }
```

Constraints are boolean expressions

Array Randomization

```
class CBaseTrans;
  rand int my_array[];

  constraint my_constraint {
    my_array.size() == 24;
    my_array[12] == 15;

    foreach ( my_array[ii] )
    {
      ii != 12 -> my_array[ii] == ii;
    }

  }
endclass
```

Randomization Issue

- Random Variable Distribution depends on tools.
  - Most of the time distribution is linear
- Corner cases requires to target certain areas more than others
- Random Distribution is a key factor in order to:
  - Find nasty bugs
  - Hit the functional coverage target faster

Distribution Function

- A distribution function is a constraint
  - Which limits the range of values
  - Add weight to the possible values

Value Distribution

- Make interesting cases more probable, using non uniform distribution

```
constraint address_distributino_c {
   address dist {
     0                 := 1,                  // 1 value, 1 chance out of 14 to be set to this value
     'hFFFFFFFF        := 1,                  // 1 value, 1 chance out of 14 to be set to this value
     [1:10]            := 1,                  // 1 value, 1 chance out of 14 to be set to this value
     [hFFFFFFFF-10:'hFFFFFFFF-1] :/ 1,        // 10/10 value, 1 chance out of 14 to be in the range
     [11:'hFFFFFFFF-10] := 1                  // 10 values, 1 chance out of 14 to be any of
                                              //    these value.
   }
 }
```

Soft Constraint

- A soft constraint is a constraint that may not be satisfied.
- A soft constraint which is not satisfied is not a contradiction.
- A soft constraint is taken in account only if there is no contradiction.

Soft Constraint Example

```
class CTransaction extends base_transaction_c;
  rand bit[31:0]  address;
  rand bit[31:0]  size;
  constraint size_c {
    soft size inside { [0:10]}; // default values are between 0 and 10
  }
endclass

program foo17;
  initial
    begin
      CTransaction a;
      a.randomize() with { size inside {[5:100]} ;}  ; // no contradiction --> [5:10]
      a.randomize() with { size inside {[11:100]} ;} ; // contradictions --> [11:100]
      a.randomize() with {
        disable soft size;          // soft constraint is disabled.
        size inside {[5:100]}   ;   // no contradiction --> [5:100]
      } ;

    end
endprogram
```

The Soft Constraint side effect

- Soft constraints are useful to define default behaviors
- If a soft constraint can be satisfied, it will be satisfied

```
/// assertion settings
constraint axi_config::axi_config_assertion_enable_c {
  soft has_checks   == 1;
  soft assert_deassert_reset_on_clock_rising_edge_enable    == has_checks;
  soft assert_valid_low_during_reset_enable                 == has_checks;
  soft assert_awtrans_stable_until_issued_enable            == has_checks;
  soft assert_wtrans_stable_until_issued_enable             == has_checks;
  soft assert_btrans_stable_until_issued_enable             == has_checks;
}
```

```
task my_test::run_phase(uvm_phase phase);
  axi_config cfg = new();
  cfg.randomize() with {
    assert_valid_low_during_reset_enable == 0;
  };
endtask
```

Cannot be satisfied ?
Really ?
What if it is satisfied ?
Has_checks == 0…
No more checker are activated !
How to know ?
Actually you don't…

Resolving the soft constraint issue

- Soft constraints are useful to define default behaviors
- If a soft constraint can be satisfied, it will be satisfied

```
/// assertion settings
constraint axi_config::axi_config_assertion_enable_c {
  soft has_checks   == 1;
  soft assert_deassert_reset_on_clock_rising_edge_enable    == get_bit_value(has_checks);
  soft assert_valid_low_during_reset_enable                 == get_bit_value(has_checks);
  soft assert_awtrans_stable_until_issued_enable            == get_bit_value(has_checks);
  soft assert_wtrans_stable_until_issued_enable             == get_bit_value(has_checks);
  soft assert_btrans_stable_until_issued_enable             == get_bit_value(has_checks);
}
```

```
task my_test::run_phase(uvm_phase phase);
  axi_config cfg = new();
  cfg.randomize() with {
    assert_valid_low_during_reset_enable == 0;
  };
endtask
```

has_checks need to be known before solving the constraint has_checks == 1

So this constraint cannot be satisfied

Numbers versus Actions

aedvices CONSULTING

- Constraints applieds to random numbers
  - Random values
  - Random valid configurations
  - Random valid transaction payloads

- How to generate random actions ?
  - Select random actions while controlling probability?
  - Go randomly through a statemachine ?

Solution 1: Model probability and states using random variables

Solution 2: randcase / randsequence

Randcase statement

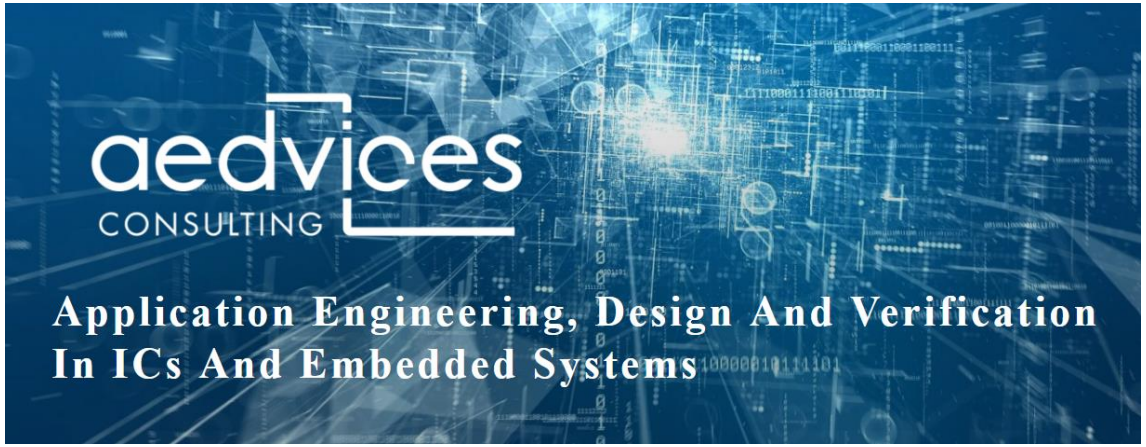
- Randcase creates weighted random actions

```
repeat (50)
   begin
     randcase
       10 : begin
         // do something with a probability of 10%
       end
       80 : begin
         // do something with a probability of 80%
       end
       5  : begin
         // do something with a probability of 5%
       end
       5  : begin
         // do something with a probability of 5%
       end
     endcase
   end
```

SV Class Summary

- Verilog / SystemVerilog : just another language
- Syntax is different
- … not only
- Classes are like in other Java or C++ languages
- Random Variables can be constrained
- We can generate random actions

Contacts



www.aedvices.com
contact@aedvices.com