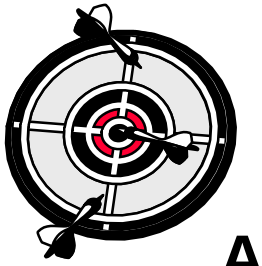
Agenda

DAY 1

1. The Device Under Test (DUT)
2. SystemVerilog Verification Environment
3. SystemVerilog Language Basics - 1
4. SystemVerilog Language Basics - 2

Unit Objectives



After completing this unit, you should be able to:

- Use system functions and controls for randomization of variables
- Define aliases using `typedef`
- Use different types of language operators
- Use flow control constructs to build a SV testbench
- Define and use subroutines in a SV program
- Understand lifetime of a code block

System Functions: Randomization

- **`$urandom`: Return a 32-bit unsigned random number**
  - Run-time switch: **`+ntb_random_seed=`*`seed_value`***
  - **`srandom`**(***`seed`***): set random seed for **`$urandom`**
    - **`$random`** Verilog function gives poor distribution and repeatability
- **`$urandom_range`(*`max`*, [*`min`*]): specify range of unsigned random number**
- **`randcase`: Select a weighted executable statement**

```
randcase
   10 : f1();
   20 : f2();  // f2() is twice as likely to be executed as f1()
   50 : x = 100;
   30 : randcase … endcase; // randcase can be nested
endcase
```

User Defined Types and Type Cast

- Use typedef to create an alias for another type

```
typedef bit [31:0] uint;
typedef bit [5:0] bsix_t; // Define new type
bsix_t my_var;            // Create 6-bit variable
```

- Use `<type>'(<value>|<variable>)` to convert data types (static cast – checks done at compile-time)

```
bit[7:0] load[];
//int is a signed type. Use care when randomizing
int tmp = $urandom; //tmp can be negative
load = new[(tmp % 3) + 2]; //(tmp % 3) can be -2!!!
load = new[(uint'(tmp) % 3) + 2];//tmp cast to uint
load = new[(unsigned'(tmp) % 3) + 2];
```

What are the possible sizes of the array?

Operators

`+ - * /`	arithmetic	`~`	bitwise negation
`%`	modulus division	`&`	bitwise and
`++ --`	increment, decrement	`&~`	bitwise nand
`> >= < <=`	relational	`|~`	bitwise nor
`!`	logical negation	`|`	bitwise inclusive or
`&&`	logical and	`^`	bitwise exclusive or
`||`	logical or	`^~`	bitwise exclusive nor
`==`	logical equality	`{}`	concatenation
`!=`	logical inequality	`&`	unary and
`===`	case equality	`~&`	unary nand
`!==`	case inequality	`|`	unary or
`==?`	wildcard case equality	`~|`	unary nor
`!=?`	wildcard case inequality	`^`	unary exclusive
`<<`	logical shift left	`~^`	unary exclusive nor
`>>`	logical shift right	`?:`	conditional (ternary)
`<<<`	arithmetic shift left	inside	set membership
`>>>`	arithmetic shift right	iff	qualifier

Assignment:

`= += -= *= /= %= <<= >>= <<<= >>>= &= |= ^= ~&= ~|= ~^=`

`inside` Operator

- **Use `inside` operator to find an expression within a set of values**

```
bit[31:0] smpl, r1, r2; int golden[$] = {3,4,5};
if (smpl inside {r1, r2})... //(smpl == r1 || smpl == r2)
if (smpl inside {[r1:r2]})... //(smpl inside range r1 to r2)
if (result inside {1, 2, golden}) .. // same as { 1, 2, 3, 4, 5}
```

- **`inside` operator uses**
  - `==` operator on non-integral expressions
  - `==?` on integral expressions
    - `x` and `z` are ignored in set of values
    - wildcards (?) preferred instead of x and z

```
if (result inside { 3'b1?1, 3'b00? })…
// {3'b101, 3'b111, 3'b000, 3'b001}
```

`iff` Operator

- **Use `iff` operator to qualify**
  - event controls
    - `@(rtr_io.cb iff(rtr_io.cb.frame_n[prt_id] !== 0));`
  - property execution – not covered in this workshop
  - coverage elements – covered later
    - cover points
    - bins of cover points
    - cross coverage
    - cross coverage bins

Know Your Operators!

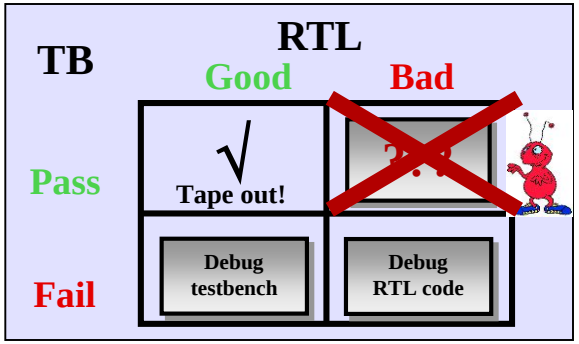
- **What is printed to console with following code?**

```
logic[3:0] sample, ref_data;
sample = dut.cb.dout[3:0];
if (sample != ref_data) $display(" Error! ");
else $display(" Pass! ");
```

  - When sample = 4’b1011 & ref_data = 4’b1010
  - When sample = 4’b101x & ref_data = 4’b1010
  - When sample = 4’b101x & ref_data = 4’b101x

- **Avoid false positives by checking for pass condition!**

```
sample = dut.cb.dout[3:0];
if (sample == ref_data) ;
else $display(" Error! ");
assert(!$isunknown(sample));
assert($onehot(sample));
if(!$onehot0(sample)) ...;
```



Sequential Flow Control

- **Conditional:**
  - `if (x==7) a=7; else a=8;`
  - `a = (x == y) ? 7 : 8;`
  - `assert (true condition);`
  - `case(expr) 0: …; 1: …; default: …; endcase`
- **Loops:**
  - `repeat(expr) begin … end`
  - `for(expr; expr; expr) begin … end`
  - `foreach(array[index]) begin … end`
  - `while(expr) begin … end`
  - `do begin … end while (expr);`
  - `break` to terminate loop
  - `continue` to terminate current loop iteration

Subroutines (`task` and `function`)

- Tasks can block
- Functions can not block

Pass by reference

```
task print_sum(ref int a[], input int start=0);
  automatic int sum = 0;
  for (int j=start; j<a.size(); j++)
    sum += a[j];
  $display("Sum of array is %0d", sum);
endtask
…
print_sum(my_array);
```

Default value

`task` does not return value

Subroutine in `program` block defaults to static can be made automatic. Subroutine in `class` defaults to automatic

Subroutine `variables` default to subroutine scope and lifetime. Can be made automatic or static

```
function automatic int factorial(int n);
  static int shared_value = 0;
  if (n < 2) return(1);
  else return(n * factorial(n-1));
endfunction
…
result = factorial(my_val);
```

Pass by value

`function` returns value

Subroutine Argument Binding and Skipping

- **Arguments can be bound (passed) to the task by**
  - position
  - name
- **Arguments can be skipped**

```
program automatic test;
task tally(ref byte a[], input logic[15:0] b, c = 0, u, v);
…
endtask
  initial begin
    logic[15:0] B = 100, C = , D = 0, E = 0;
    byte A[] = {1,3,5 ,13};
    tally(A, B, , D, E);
    tally(.c(C), .b(B), .a(A), .u(D), .v(E) );
  end
endprogram
```

skipped arguments use default value

arguments passed by position

arguments passed by name

Subroutine Arguments

- **Type and direction are both sticky** See note
  - Any following arguments default to that type and direction

direction	effect
`input`	copy value in at beginning - default
`output`	copy value out at end
`inout`	copy in at beginning and copy out at return
`ref`	pass by reference, makes argument variable the same as the calling variable. Changes to argument variable will change calling variable immediately
`const ref`	pass by reference but read only. Saves time and memory for passing arrays to tasks & functions

Default dir is input, default type is logic

a, b: input logic
u, v: output bit [15:0]

Read-only pass via reference

```
task T3(a, b, output bit [15:0] u, v, const ref byte c[]);
```

Test For Understanding

- **What's the direction and data type of each argument?**

```
task T3(ref byte a[], logic[15:0] b, c, output u, v);
  b = c;
  foreach(a[i])
    a[i] = i;
endtask
```

dir is ?, type is ?

dir is ? type is ?

dir is ? type is ?

```
initial begin
    logic[15:0] B = 100, C = 0, D = 0, E = 0;
    byte A[] = {1,3,5,8,13};
    T3(A, B, C, D, E);
    foreach(A[i])
      $display(A[i]);
    $display(B, C, D, E);
end
```

What will be displayed?

Recommendation: declare all directions

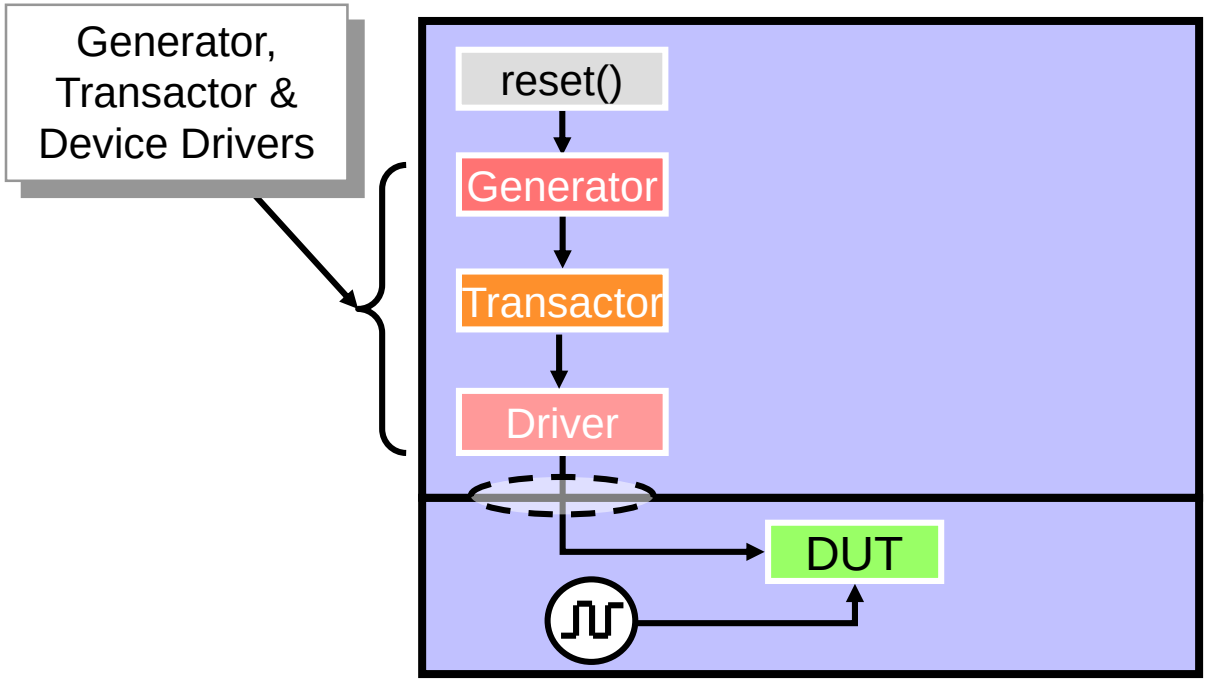
Code Block Lifetime Controls

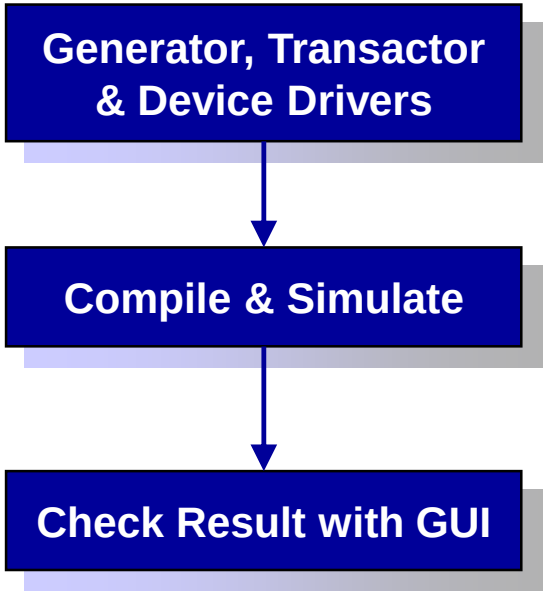
- **Simulation ends when all programs end**
  - Execution of a program ends when
    - All **`initial`** blocks in program reach end of code block
    - Or, **`$finish`** is executed
- **Execution of a subroutine ends when**
  - **`endtask`**, **`endfunction`** is encountered
  - Or, **`return`** is executed
- **Execution of a loop ends when**
  - **`end`** (of loop **`begin`**) is encountered
  - Or, when **`break`** is executed
- **Execution of loop immediately advances to next iteration**
  - When **`continue`** is executed

Lab 2 Introduction

Develop Generator, Transactor & Device Drivers to drive one packet through the router

60 min





Unit Objectives Review

Having completed this unit, you should be able to:

- **Use system functions and controls for randomization of variables**
- **Define aliases using `typedef`**
- **Use different types of language operators**
- **Use flow control constructs to build a SV testbench**
- **Define and use subroutines in a SV program**
- **Understand lifetime of a code block**

Appendix

Import and Export Verilog subroutines

Import and Export C/C++ subroutines (DPI)

Import and Export Verilog subroutines

Import and Export Verilog Subroutines

```
task root_task();                          // Root level subroutine
  $display("I'm Root Task"); endtask
module bfm(top_io.BFM bfm_io);             // BFM's that implement I/O via SystemVerilog interface
  task bfm_io.bfm_task();                  // Subroutine to be accessed by test program
    $display("I'm BFM task"); endtask
endmodule: bfm
module vip();                              // BFM's that do not implement I/O via SystemVerilog interface
  task vip_task();                         // Subroutine to be accessed by test program
    $display("I'm VIP task"); endtask
endmodule: vip
interface top_io();                        // SystemVerilog Interface to be used by test program
  task interface_task();                   // Subroutine to be accessed by test program
    $display("I'm Interface task"); endtask
  task vip_task();                         // Wrapper for non-SystemVerilog interface BFM's
    test_top.VIP.vip_task();               // XMR reference via top-level instance
  endtask
  modport TB(import task interface_task(), import task bfm_task(), import task vip_task());
  modport BFM(export task bfm_task());
endinterface:  top_io
program automatic test(top_io.TB test);
initial begin
  $root.root_task();                       // direct $root access
  test_top.VIP.vip_task();                 // VIP XMR access via top module
  test_top.BFM.bfm_io.bfm_task();          // BFM XMR access via top module
  test.vip_task();                         // interface VIP access
  test.bfm_task();                         // interface BFM access
  test.interface_task();                   // interface access
end
endprogram: test
```

```
module test_top;
  top_io IO();
  test TEST(IO);
  bfm BFM(IO);
  vip VIP();
endmodule
```

Import and Export C/C++ subroutines (DPI)

SV Direct Programming Interface

- **Direct Programming Interface (DPI-C)**
  - SystemVerilog calls C/C++ functions
  - C/C++ calls SystemVerilog functions & blocking tasks
- **Simple interface to C models**
  - Allows SystemVerilog to call a C function just like any other native SystemVerilog function/task
  - Testbench variables passed directly to/from C/C++
  - NO need to write PLI-like applications/wrappers
- **DPI-C cannot be used to**
  - attach callbacks to a signal
  - traverse hierarchy, get handles to instances or objects
    - Instead use PLI/VPI for these tasks

DPI-C: Import

SystemVerilog calling C/C++ functions

```
import "DPI-C" [cname =][pure] function type name (args);
import "DPI-C" [cname =][pure][context] task name (args);
```

- **cname**: maps C name to SystemVerilog prototype name
- **pure**: value returned only via call (no output/inout argument)
- **context**: required to call SystemVerilog subroutines in C

```
program automatic top;
  import "DPI-C" context task c_test(input int addr);
  initial begin
    c_test(1000);
    c_test(2000);
  end
endprogram: top
```

```
>vcs -sverilog top.sv c_test.c
```

```
#include <stdio.h>
#include <svdpi.h>

void c_test(int addr) {
   ...
}
```

DPI-C: Export

C calling SystemVerilog functions

```
export "DPI-C" [cname =] function name;
```

- ***cname***: maps C name to SystemVerilog prototype name

C calling SystemVerilog (blocking) tasks

```
export "DPI-C" [cname =] task name;
```

- ***cname***: maps C name to SystemVerilog prototype name

```
import "DPI-C" context task c_test(int addr);
export "DPI-C" task apb_write;

task apb_write(input int addr, data);
  ... @(posedge ready); ...
endtask

initial c_test(1000);
```

```
#include <stdio.h>
#include <svdpi.h>
extern void apb_write(int, int);

void c_test(int base_addr) {
  ...
  apb_write(addr, data);
 ...
}
```

```
>vcs -sverilog top.sv c_test.c
```

Declaration of Imported Functions and Tasks

- **An imported task or function can be declared anywhere a native SV function or task can be declared**

```
import "DPI-C" pure function real sin(input real r); // math.h
```

- **Imported tasks and functions can have zero or more input, output and inout arguments. ref is not allowed.**

```
import "DPI-C" function void myInit();
```

  - Imported functions can return a result or void
    - The return result type are restricted to “small” value
      - void, byte, shortint, int, longint, real, shortreal, chandle, and string
      - Scalar values of type bit and logic

```
import "DPI-C" function int getStim(input string fname);
```

- **Map method name if it conflicts with existing name**

```
import "DPI-C" test = function int my_test();
```

DPI-C: Supported Data Types

SystemVerilog	C (input)	C (out/inout)
byte (1)	char	char*
shortint	short int	short int*
int	int	int*
longint (1)	long int	long int*
shortreal	float	float*
real	double	double*
string	const char*	char**
string[n]	const char**	char**

SystemVerilog	C (input)	C (out/inout)
bit	svBit	svBit*
logic, reg	svLogic	svLogic*
bit[N:0]	const svBitVecVal*	svBitVecVal*
reg[N:0] logic[N:0]	const svLogicVecVal*	svLogicVecVal*
array[size]	type[ ]	type[ ]
array[M][N]	type[ ][ ]	type[ ][ ]
array[ ] (import only)	const svOpenArrayHandle	svOpenArrayHandle
chandle	const void*	void*

(1) input/output/inout only, not for function return value

DPI-C: Supported Data Types

- **Arguments must match types between SystemVerilog and C**
  - User responsibility
  - DPI does not check for type compatibility
- **VCS produces vc_hdrs.h**
  - Use it as a guide to see how types are mapped
- **Argument directions**
  - `input` Input to C code
  - `output` Output from C code (initial value undefined)
  - `inout` Input and Output from C code
  - `ref` Not supported by LRM. Use `inout` instead
- **Return types (<32b)**
  - `(unsigned) int, char*`
- **Protection**
  - It's up to the C code to not modify input parameters
  - Use `const` to double check your C code

DPI-C Example: Integer and Strings

SystemVerilog	C Data Type	Description
`int`	`int`	Integer passed by value
`string`	`char*`	String passed by value

```
program automatic top;
  import "DPI-C" function void display_int(int i);
  import "DPI-C" function void display_str(string s);
  initial display_int(10);
  initial display_str("hello");
endprogram: top
```

```
#include <stdio.h>
#include <svdpi.h>
void   display_int(int   i) { io_printf ("C : int = %d\n" , i); }
void   display_str(char* s) { io_printf ("C : str = %s\n" , s); }
```

DPI-C: 4-State Data Types

SystemVerilog	C Data Type	Description
`reg/logic`	`svLogic`	Reg/logic passed by value
`reg/logic[n:0]`	`svLogicVecVal*`	Reg/logic vec passed by value

- **`reg/logic`**
  - 4 State values in SystemVerilog
  - Represented using **`svLogic`** in C

System Verilog	C : svLogic Data
0	0
1	1
Z	2
X	3

- **`reg/logic` arrays**
  - Represented using array of **`svLogicVecVal`** struct in C

```
/* (a chunk of) packed bit array */
typedef uint32_t svBitVecVal;
/* (a chunk of) packed logic array */
typedef struct vpi_vecval {
uint32_t aval; //Data (Value)
uint32_t bval; //Control
} s_vpi_vecval, *p_vpi_vecval;
typedef s_vpi_vecval svLogicVecVal;
```

System Verilog	C : svLogicVecVal	
	Data	Control
0	0	0
1	1	0
Z	0	1
X	1	1

DPI-C Example: `reg/logic`

```
program automatic top;
  import "DPI-C" function void display_reg(logic r);
  import "DPI-C" function void display_vec(logic [31:0] v);
  initial display_reg(1'bx);
  initial display_vec(32'h12xz);
endprogram: top
```

```
#include <stdio.h>
#include <svdpi.h>
void display_reg(svLogic  r)
   { io_printf ("c=%d, d=%d\n", (r>>1)&1,  r&1 ); }
void display_vec(svLogicVecVal* v)
   { io_printf ("c=%x, d=%x\n", v->aval, v->bval); }
```

DPI-C: `chandle`

- **`chandle`**
  - Allows C to allocate memory, pass to SystemVerilog
    - SystemVerilog can only access memory address
  - SystemVerilog can then pass the handle back to C
  - Allocate and de-allocate in the same language

```
//standard C functions
import "DPI-C" function chandle malloc(int size);
import "DPI-C" function void free(chandle ptr);
// abstract data structure: queue
import "DPI-C" function chandle newQ (input string name_of_queue);
// Note the following import uses the same foreign function for implementation as the
// prior import, but has different SystemVerilog name
// and provides a default value for the argument.
import "DPI-C" newQ=function chandle AnonQ(input string s=null);
import "DPI-C" function chandle newElem(bit [15:0]);
```

DPI-C: Array Access

```
logic [31:0] array8[8];
```

- **C Structure contains array details**
  - struct **`svOpenArrayHandle`**

```
import "DPI-C" function int my_func(input int data[ ]);
```

```
int my_func(const svOpenArrayHandle handle) {
  int* data;
  data = (int*) svGetArrayPtr(handle);
  ...
}
```

- **Data / Array**
  - Get a handle to the data type using **`svGetArrayPtr()`**
  - Type cast to the correct type to match SystemVerilog
  - **`char*, int*, svLogicVecVal*`**, etc.

DPI-C: Array Access

using `logic [31:0] array8[8];` for the functions below

- **Access Functions**
  - **`void *svGetArrayPtr(arg)`**
    - Returns pointer to representation of the whole data array
    - Type cast this to the correct type, based on SV array type
  - **`int svDimensions(arb)`**
    - # of dimensions. array8: 1 dimension
  - **`int svSizeOfArray(ary)`**
    - # of bytes to store array, including 4-state. array8: 64
  - **`int svLow(ary, dim)`**
    - Low index of array. array8: 0 for dimension 1
  - **`int svHigh(ary, dim)`**
    - High index of array. array8: 7 for dimension 1

DPI-C: Import Examples

SystemVerilog	C Data Type	Description
`int array[]`	`svOpenArrayHandle`	Dynamic array

```
program automatic top;
  import "DPI-C" function void display_array_val(int arr[]);
  int arr[] = new[4];
  initial display_array_val(arr);
endprogram: top
```

```
#include <stdio.h>
#include <svdpi.h>
void display_array_val(svOpenArrayHandle handle) {
  int* data = (int*) svGetArrayPtr(handle); int i;
  io_printf("Dim=%d, Size=%d\n",
            svDimensions(handle), svSizeOfArray(handle));
  io_printf("Low=%d, High=%d\n", svLow(handle,1), svHigh(handle,1));
  for (i=svLow(handle,1); i<=svHigh(handle,1); i++)
    io_printf("data[%0d] = %d\n", i, data[i]);
}
```

DPI-C: Compile and Debug

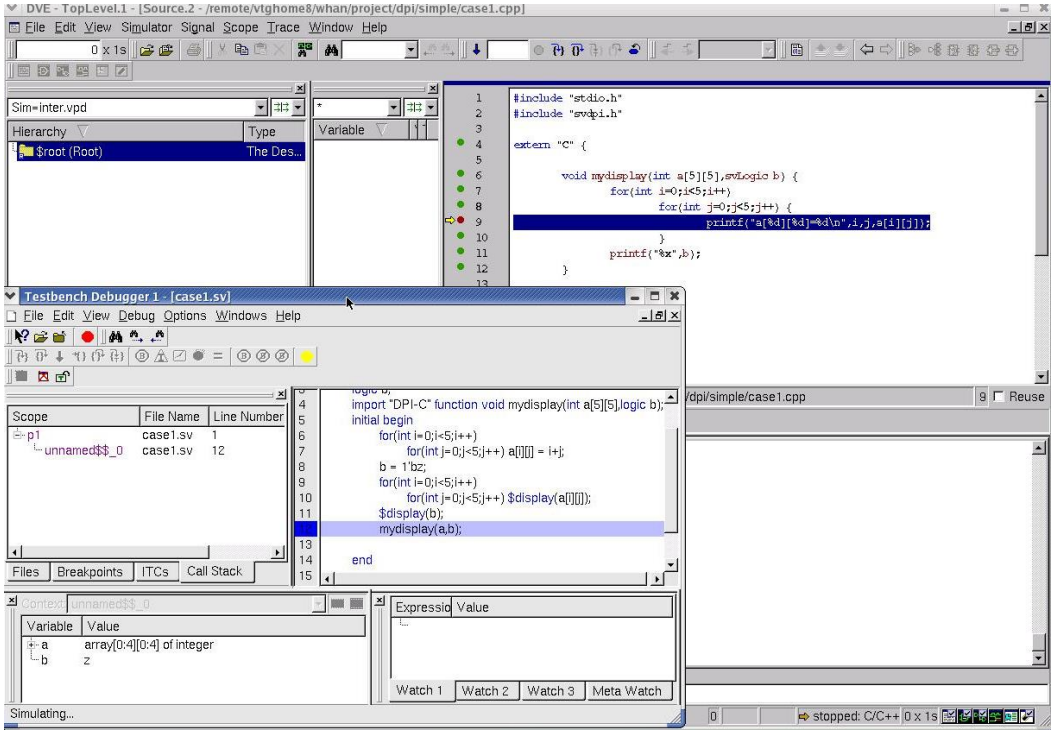
- **Compilation**

```
% vcs -sverilog test.sv test.c
```

- **Enable the C/C++ source code debug with –g options.**

```
% vcs -sverilog case1.sv case1.cpp -debug_all -CFLAGS -g
```

- **DVE supports SV design/testbench and C/C++ integrated debug**



DPI-C: Header Files & Examples

- **DPI header files**
  - $VCS_HOME/include

- **SystemVerilog examples:**
  - $VCS_HOME/doc/examples/sv/dpi

  Directory contains following two subdirectories:

  export_fun – DPI export function for SV

  import_fun – DPI import function for SV

- **NativeTestbench-OpenVera examples:**
  - $VCS_HOME/doc/examples/nativetestbench/openvera/dpi

    Contains example of DPI import function for NTB-OV