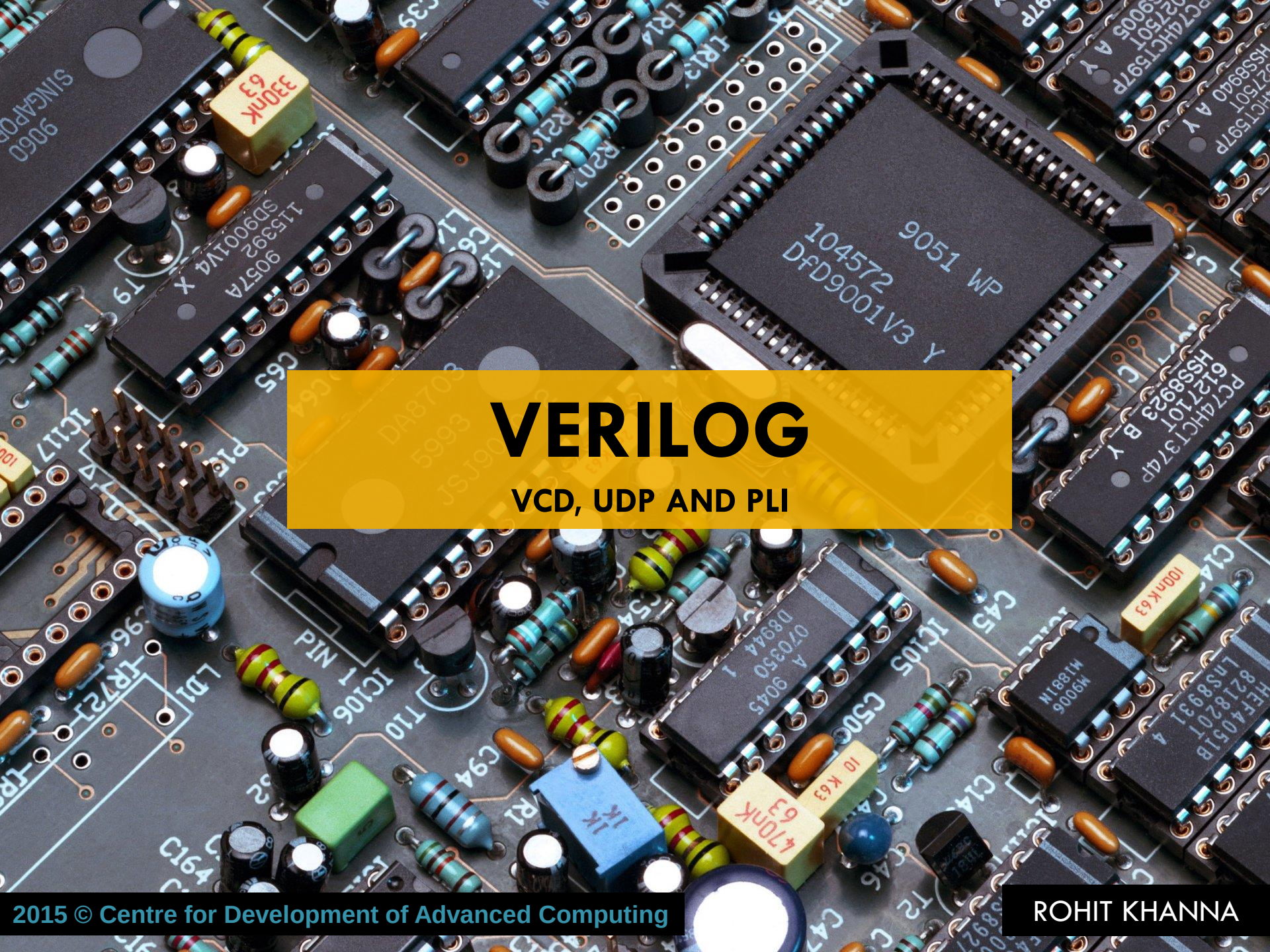


VERILOG

VCD, UDP AND PLI

ROHIT KHANNA

VCD(Value Change Dump) Files

- A VCD file is an ASCII file, which contains header information(time scale, date, software version, etc), variable definitions, and value changes for variables specified in system tasks.
- Any value change on selected variables can be written to a VCD file during simulation.
- Post processing tools can accept VCD file and visually display variable hierarchy, value and strength in terms of waveform.

VCD System Tasks and Keywords

- Following are some useful System Tasks for VCD operations:

$dumpfile	$dumpon
$dumpvars	$dumpoff
$dumpall	$dumpflush

- Keyword Inside VCD File:

$comment	$var	$dumpon	$dumpall
$date	$scope	$dumpoff	$version
$timescale	$end	$dumpvars	$upscope

VCD System Tasks

- $dumpfile is used to create a VCD file

  $dumpfile[("file name")]; //Default file name is dump.vcd

- $dumpvars is used to specify variables to be dumped

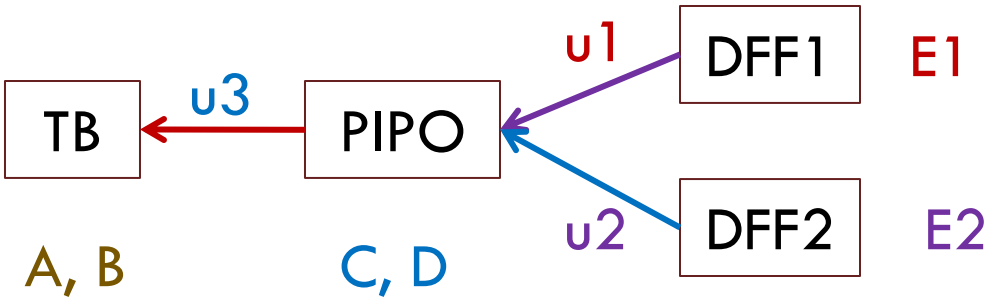
  $dumpvars(level, variables or modules);

level→ how many level of hierarchy of modules has to be dumped to the VCD file.

1→ will dump all variables in current specified module, not modules instantiated inside current specified module.

0→ will dump all variables in current specified module and modules instantiated inside current specified module.

VCD System Tasks




```
$dumpvars(1, TB);                    //Dumps values of A and B
$dumpvars(1, TB.u3);                 //Dumps values of C and D
$dumpvars(2, TB);                    //Dumps values of A, B, C and D
$dumpvars(0, TB);                    //Dumps values of A, B, C, D, E1 and E2
$dumpvars(1, TB.u3, A);              //Dumps values of A, C and  D
$dumpvars(1, TB.u3.u1,TB.u3);        //Dumps value of C, D and E1
```

VCD System Tasks

- $dumpall is used to write current value of all selected variable

```
$dumpall;
```

- $dumpon/$dumpoff is used to resume or stop dumping process

```
$dumpoff;    //stop dumping
$dumpon;     //resume dumping
```

- $dumpflush is used to empty VCD buffer and ensure all data in buffer is written to VCD file. It is used to update the dump file while simulation is running.

```
$dumpflush;
```

VCD Keywords

- $comment represents comment inside a VCD file

  $comment This is the comment $end

  $comment Multi-line Comment1

  Multi-line Comment2 $end

- $timescale specifies time scale which was used during simulation

  $timescale time_number time_unit $end

- $date specifies the date on which VCD file was created

  $date data_in_text_format $end

- $upscope indicates change of scope to next higher level in design hierarchy. $upscope $end

VCD Keywords

- $var section specifies properties and identification code of variable been dumped.

  $var var_type size identifier_code var_name $end

- $scope specifies scope of variable been dumped

  $scope module modulename/instance_name $end

- $version indicates version of VCD writer and system task $dumpfile used to create the file

  $version version_text system_task $end

Design Example

```
module comb_logic (a, b, c, d, e);
input a, b, c, d;
output e;

wire x;
wire y;

assign x= a & b;
assign y= c | d;
assign e= x ^ y;

endmodule
```

VCD Test Bench

```
module testbench;
reg a, b, c, d;  wire e;

comb_logic u0 (a, b, c, d, e);
// add VCD initial here

initial
forever begin
a=$random; b=$random;
c=$random; d=$random;
#3; end
endmodule
```

```
//VCD initial block
initial
begin
$dumpfile;
$dumpvars(1, testbench, u0.x, u0.y);
#20 $dumpoff;
#30 $dumpon;
#5 $dumpall;
end
```

Questasim VCD Feature

Creating VCD file without using Verilog System Tasks

```
vsim  –novopt testbench  //Start TB simulation
vcd   file   vcd_filename //Create VCD file
vcd   add  var1 var2     //Add variables to be dumped
add wave *               //Add wave window
run  time_value          //Run Simulation
```

Converting a VCD file to a waveform file

```
vcd2wlf   vcd_filename   wlf_filename.wlf
```

UDP (User Defined Primitive)

- Verilog allows used to define their own primitives, referred to as User Defined Primitives (UDP's).
- UDP definitions are independent of a module. They are at same level as that of module definition in syntax hierarchy.
- UDP does not contain any module or primitive instantiation inside it. They are instantiated the same way other modules and gate level primitives are instantiated.
- UDPs can be categorized into two:
  - Combinational UDPs
  - Sequential UDPs

Parts of UDPs

```
primitive primitive_name (output_port_name,      // One output
                          input_port_names);     //Multiple Inputs

output output_port_name;
input input_port_names;

reg output_port_name;                            //Only for sequential UDP
initial output_port_name=initial value; //Only for sequential UDP

table                                            //State Table
table_entries ;                                  //Defines UDP Functionality
endtable

endprimitive
```

UDP Rules

- It can have multiple inputs but only one output. Inout ports are not allowed.
- Output port must be the first port in the port list.
- All ports have to be scalar. Vector ports are not allowed.
- UDP supports value 0, 1 and X. Value Z is not supported by UDP, Z value passed to a UDP is considered as X.
- In case of sequential UDP output has to be declared as reg and value can be initialized with help of initial block.

Combinational UDPs

- Output has to be first port in the port list.
- Output state is function of current input states. Whenever input state changes, UDP is evaluated and output state is set to a value as indicated by entry in the state table for current input state.
- State Table (similar to Truth Table) is used to define relation between current input state and output state.
- The output is X for a combination of input state not present in the state table.

Example

```
primtive my_or (c, a , b);                    //output to specified first
output c;
input a, b;

//primitive my_or (output c, input a , b);  //ANSI C Style

//State Table to be specified here

endprimitive
```

State Table

- State Table is specified with help table and endtable keyword.
- Each entry in the state table of a combinational UDP has following Syntax

  input1_val input2_val … inputN_val : output_val;

- Input values in the state table should be specified in the same order as they appear in port (terminal) list.
- " : " is used to separate in inputs and output.

State Table

- " ; " is used end a state table entry.
- All possible combinations of inputs where output produces a known value, must be explicitly covered.
- Combinations of the inputs that are not explicitly specified will drive the output to the unknown value X.
- " ? " is used to represent don't care conditions. ? Covers 0, 1, X values
- There should not any conflicting entries in the state table.

Example1

```
//State Table for my_or
table
// a   b  :  c ;
    0   0  :  0;     //Order of input same as in port list
    0   1  :  1;
    1   0  :  1;
    1   1  :  1;
 endtable
```

This state table is incomplete because input combination (1, X) and (X, 1) are not covered which should produce output 1

Example2

```
//State Table for my_or
table
// a   b  :  c ;
    0   0  :  0;     //Order of input same as in port list
    0   1  :  1;
    1   0  :  1;
    1   1  :  1;
    1   X  :  1;
    X   1  :  1;
 endtable
```

Example3

A better alternative to previous example.

```
//State Table for my_or
table
// a   b  :  c ;
    0   0  :  0;     //Order of input same as in port list
    ?   1  :  1;     // ? Expands to 0, X, 1
    1   ?  :  1;     // ? Expands to 0, X, 1
 endtable
```

Any other input combination will give output as X.

Problem Statements

Write a UDP for NAND and XNOR

```
primitive my_nand (output c,
                   input a, b);

table
// a   b  :  c;
   0   ?  :  1;
   ?   0  :  1;
   1   1  :  0;
endtable

endprimitive
```

```
primitive my_xnor (output c,
                   input a, b);

table  // a   b  :  c;
   0   0  :  1;
   1   0  :  0;
   0   1  :  0;
   1   1  :  1;
endtable

endprimitive
```

Test Bench

```
`include "my_or.v"          //Including Primitive file
module test;

reg a, b;
wire c;

my_or u0 (c, a , b);        //Primitive Instance

initial
forever begin
a=$random; b=$random; #10;
end

endmodule
```

Example4

```
primitive mux4_1 (output y, input i1, i2, i3, i4, sel0, sel1);
table  //  i1   i2   i3   i4   sel0  sel1  :  y;
           0    ?    ?    ?     0     0    :  0;
           1    ?    ?    ?     0     0    :  1;
           ?    0    ?    ?     0     1    :  0;
           ?    1    ?    ?     0     1    :  1;
           ?    ?    0    ?     1     0    :  0;
           ?    ?    1    ?     1     0    :  1;
           ?    ?    ?    0     1     1    :  0;
           ?    ?    ?    1     1     1    :  1;
 endtable
endprimitive
```

Sequential UDPs

- Output has to be the first port and has to be declared as reg.
- An initial statement can be used to initialize output of sequential UDP.
- The input specification of state table entries can be in terms of input levels or edge transitions.
- All possible combinations of input must be specified to avoid unknown output values.

Example1

```
primtive d_latch (q, d, clear, en);          //output to specified first
output  q;
reg q;
input d, clear, en;

initial
q=0;

//State Table to be specified here

endprimitive
```

Example2

```
primtive d_ff (q, d, clear, clk);          //output to specified first
output  q;
reg q;
input d, clear, clk;

initial
q=0;

//State Table to be specified here

endprimitive
```

State Table

- Each entry in the state table of a sequential UDP has following Syntax

  input1_val input2_val … inputN_val : current_state : next_state;

- current state is the current value of output.

- next state is evaluated based on inputs and current state. This output value is specified in next state.

- " - " is used to represent no change in state.

Example

```
//State Table for d_latch with en and clear
table
// d   clear  en : q  :  q+ ;
    ?     1     ? : ? : 0 ;     // clear=1, op=0
    ?     0     0 : ? : - ;     // clear=0, en=0 op=latch
    0     0     1 : ? : 0 ;     // clear=0, en=1 op=d
    1     0     1 : ? : 1 ;     // clear=0, en=1 op=d
 endtable
```

State Table

- For edge sensitive UDP transition are denoted by writing old value followed by a new value inside ().

  Examples:

  (01)→ 0 to 1 transition

  (10)→ 1 to 0 transition

  (0?)→ 0 to 0, 1or X transition

  (?1)→ 0, 1 or X to 1 transition

- Only one edge specifications are allowed per state entry in a state table.

- Cover all possible combinations of transactions and levels in the state table for which output has known value.

Example

```
//State Table for positive edge triggered d_ff with clear
table // d   clear   clk : q  : q+ ;
    ?     1      ? : ? : 0 ;    //Clear output
    ?   (10)     ? : ? : - ;    //Hold data at negedge clear
    0     0    (01) : ? : 0 ;   //Capture data at posedge clk
    1     0    (01) : ? : 1 ;   //Capture data at posedge clk
    0     0    (X1) : ? : 0 ;   //Capture data at posedge clk
    1     0    (X1) : ? : 1 ;   //Capture data at posedge clk
    ?     0    (1?) : ? : - ;   //Ignore negedge of clk
    ?     0    (X0) : ? : - ;   //Ignore negedge of clk
   (??)   0     ?   : ? : - ;   //Ignore data when clk steady
    ?     0    (0x) : ? : - ;   //Ignore clk going to unknown state
endtable
```

Shorthand Symbols

Symbols	Expansion	Meaning
?	0,1, X	All possible values
b	0,1	Binary values
-	No Change	Maintain Last value
r	(01)	True Rising Edge
f	(10)	True Falling Edge
p	(01), (0X) or (X1)	Potential Rising Edge
n	(10), (1X) or (X0)	Potential Falling Edge
*	(? ?)	Any change in value

Programming Language Interface (PLI)

- Designers frequency need to customize capability of verilog language by defining own system tasks and functions.
- To do this designers need to interact with internal representation of design and the simulation environment in Verilog Simulator.
- PLI provides a set of interface routines to read, write internal data representation and extract information from simulator environment.

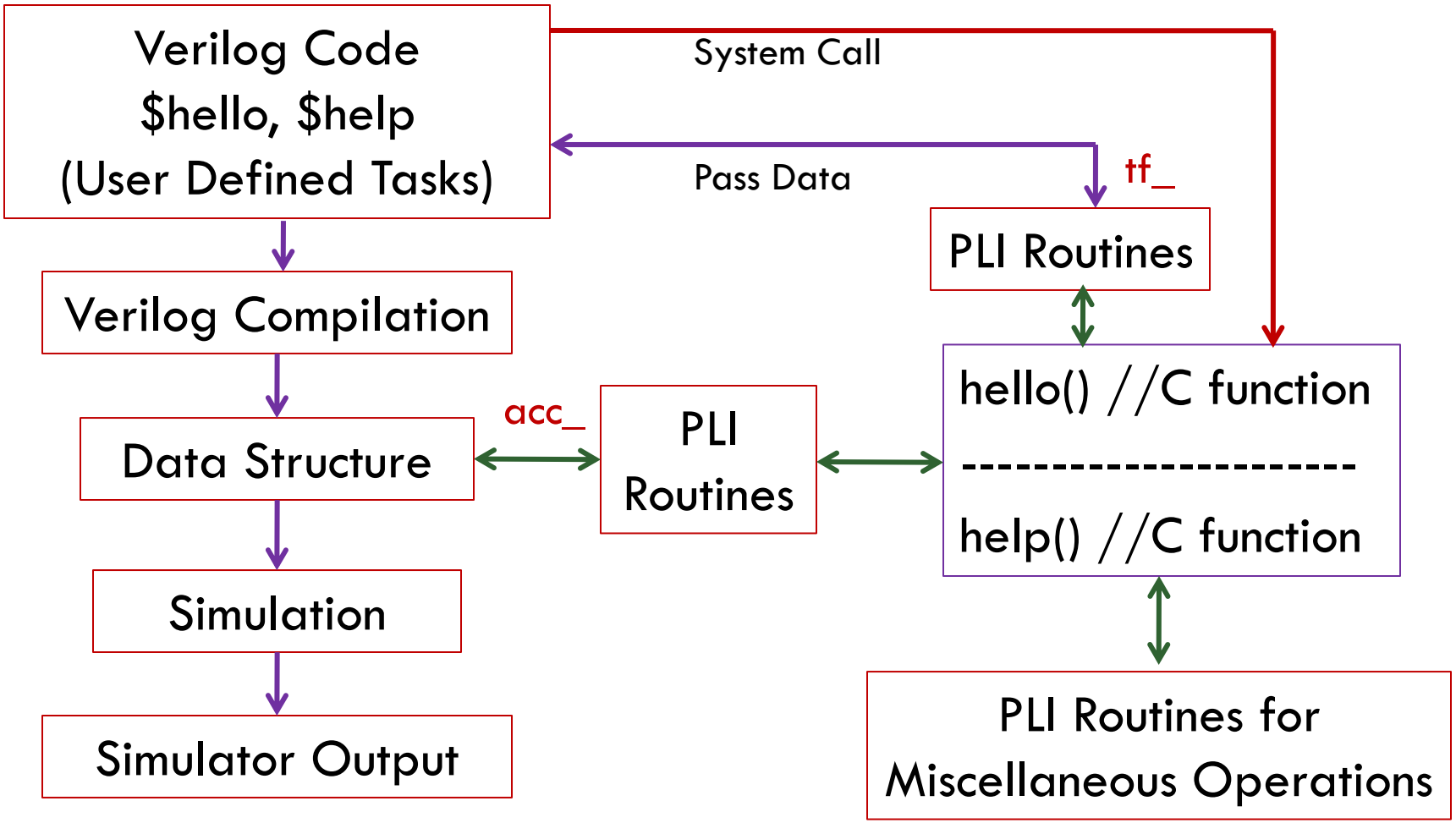
Programming Language Interface (PLI)

- Three generations of PLI:
  - Task/Function(tf_) routines are used for operations such as involving user defined tasks and functions, utility functions, callback mechanism and write data to output devices.
  - Access routines(ac_) are used to access and modify variety of objects in Verilog HDL Description.
  - Verilog Procedure Interface(vpi_) routines are superset of tf_ and ac_ routines

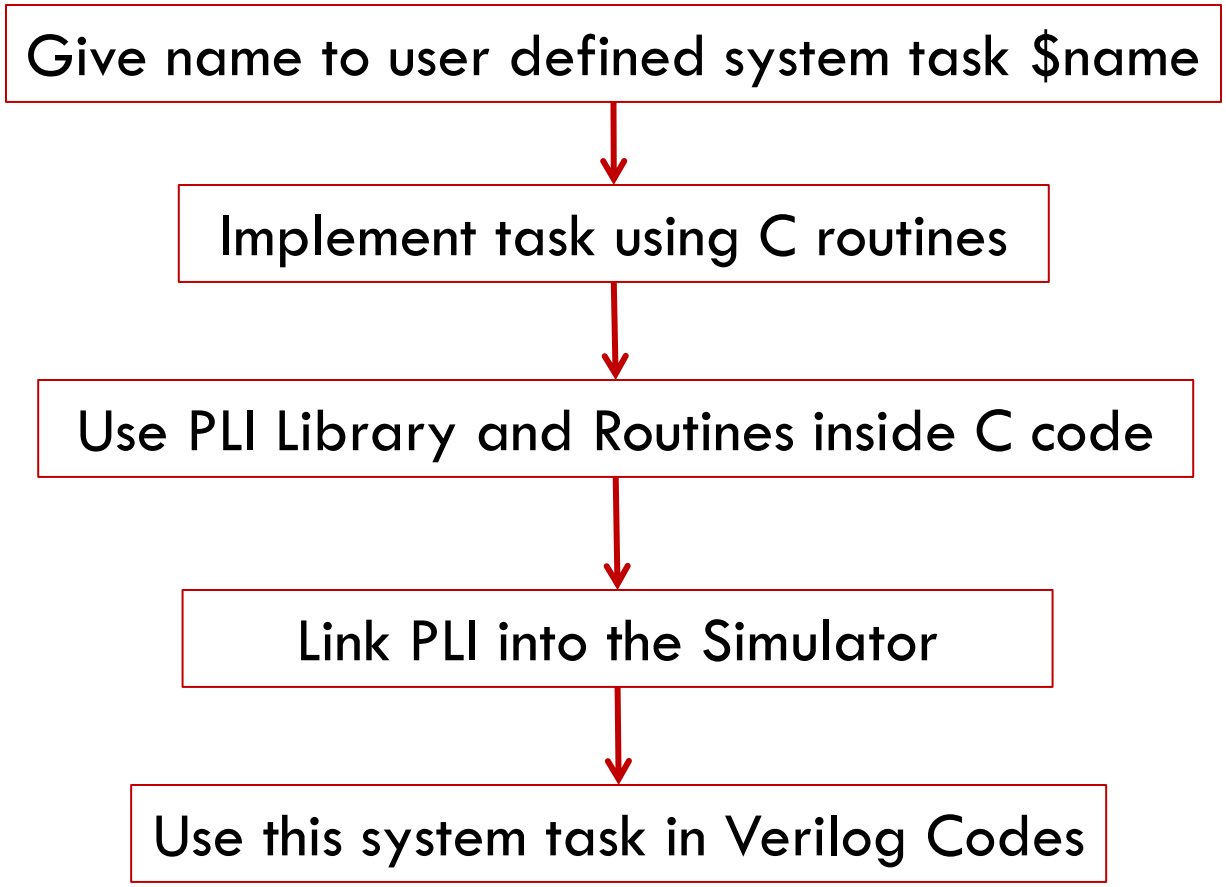
Programming Language Interface (PLI)

- Verilog PLI is mechanism to invoke C functions from a verilog code.
- The function invoked by verilog code is called system call. $write, $display, $random are some build in system task.
- User can define his system task, call it from verilog code. This system call will invoke C function.
- This called function can access(read and write) to Verilog Data Structure with help of PLI Routines which helps in achieving functionality as desired by user.

PLI Interface



Step to work with PLI



Writing C code

hello.c

Step1→ include veriuser.h file

```
#include "veriuser.h"          //contains PLI Routines
```

Step2→ Write C function

```
static hello()
{
io_printf("Hello World");   //PLI routine for writing to
                            //Standard Output
}
```

Step3→ Access PLI Routines

//Explained on next page

PLI Routine

- Each PLI application must register its system tasks and functions with the simulator.
- Structure s_tfcell defined in "veriuser.h" is used for this purpose.

```
struct s_tfcell {
    short type;       /* usertask, userfunction */
    short data;       /* passed as data argument of callback function */
    p_tffn checktf;    /* argument checking callback function */
    p_tffn sizetf;     /* function return size callback function */
    p_tffn calltf;     /* task or function call callback function */
    p_tffn misctf;     /* miscellaneous reason callback function */
    char *tfname;     /* name of system task or function */  }
```

PLI Routine

- Usage in our PLI Example

```
//Creating Array of System Tasks

s_tfcell veriusertfs[] =        //array keep track of user tasks
{
 {usertask, 0, 0, 0, hello, 0, "$hello"},
 //type, 0, 0, 0, function_call, 0, System Task name

 {usertask, 0, 0, 0, help, 0, "$help"},
 // Second System Task Declaration

 {0} /* last entry must be 0 */ };
```

PLI Task/Function Routines

Some Task/Function Routine Example

```
tf_gettime()        //get current simulation time.
tf_getp()           //get integer value of parameter of
                    //task or function.
tf_putp()           //pass integer value to a parameter of
                    //task or function.
io_printf()          //Print on standard console.
tf_warning()        //Report a warning.
tf_error()          // Report an error.
tf_message()        //Report user generated error message.
tf_dofinish()       //Finish simulation
```

Complete C Code (hello.c)

```
#include “veriuser.h”

static hello ()
{ io_printf(“Welcome to PLI”);
 };

static help ()
{tf_putp(0, tf_getp(1));    //accept 1st parameter and return it.
}; //0 means return to LHS target, 1 means 1st argument.

s_tfcell veriusertfs[] ={{usertask, 0, 0, 0, hello, 0, “$hello”},
                         {userfunction, 0, 0, 0, help, 0, “$help"},
                         {0} /* last entry must be 0 */ };
```

Compiling and Linking

Linux Machine:

Compile Command:

gcc -c -I/<install_dir>/modeltech/include hello.c

Linking Command:

ld -shared -E -o hello.s1 hello.o

Modelsim/Questasim Run

Create Verilog Code (hello.v)

```
module hello;
initial
begin
$hello;
$display($help(5));
end
endmodule
```

Transcript:

```
vlib work
vlog hello.v
vsim -c -pli hello.sl hello
run 10
```

PLI Access Routines

Some PLI Access Routine Example

```
acc_fetch_size()          //returns bit size of net, reg or port.
acc_fetch_range()        // return MSB and LSB range values.
acc_fetch_direction()              //returns direction of port.
acc_next_port()             //returns next port in port list.
acc_next_parameter()        //returns next parameter list.
acc_next_input()           //returns next input port in port list.
acc_next_output()          //returns next output port in port list.
acc_fetch_paramval()       //returns value of a parameter
acc_fetch_fullname()        //pointer to full hierarchal name
acc_handle_tfarg()      //Get handle to function/task argument
acc_initialize()           //Initialize access
acc_close()               //Close access
```

Access Code Example1 (size.c)

```
#include “veriuser.h”          //PLI TF_routines
#include “acc_user.h”          //PLI acc_routines

handle data_size;              //Handler for handling verilog
                               // objects


//Functions in Next Slide

s_tfcell veriusertfs[] ={
{userfunction, 0, 0, 0, size, 0, "$size"},      //System Function
{0} /* last entry must be 0 */ };
```

Access Code Example1 (size.c)

```
//Size user function
static size()
{
acc_initialize();                          //Initialize access
data_size=acc_handle_tfarg(1);   //Handler to 1st argument
tf_putp(0, acc_fetch_size(data_size));
//Return size of 1st argument in the Function call

acc_close();                               //Close access
};
```

Compiling and Linking

Linux Machine:

Compile Command:

gcc -c -I/<install_dir>/modeltech/include size.c

Linking Command:

ld -shared -E -o size.a size.o

Modelsim/Questasim Run

Create Verilog Code (size.v)

```
module exam1(input [2:0] a, output [22:3]b);
initial
begin
$display("size of a is ", $size(a));
$display("size of b is ", $size(b));
end
endmodule
```

Transcript:

```
vlib work
vlog size.v
vsim -c –pli size.a  exam1
run 10
```

Access Code Example1 (dir.c)

```
#include “veriuser.h”                //PLI TF_routines
#include “acc_user.h”                //PLI acc_routines

handle module;                      //Handle for module
handle port;                        //Handle for port


//Functions in Next Slide

s_tfcell veriusertfs[] ={
{usertask, 0, 0, 0, dir, 0, "$dir"},        //System Task
{0} /* last entry must be 0 */ };
```

Access Code Example2 (dir.c)

```
//Task to print direction
static dir (){
acc_initialize();
module=acc_handle_tfarg(1);    //1st argument is module
port=acc_handle_port(module, tf_getp(2));
//(module handle, port number) 0→ 1st port
switch(acc_fetch_direction(port)){
case acclnput:     io_printf("Input Port\n");  break;
case accOutput:  io_printf("Output Port\n");  break;
case acclnout:     io_printf("Inout Port\n");  break;
default:              io_printf("Undefined Port\n");}
acc_close();
};
```

Compiling and Linking

Linux Machine:

Compile Command:

gcc -c -I/<install_dir>/modeltech/include dir.c

Linking Command:

ld -shared -E -o dir.a1 dir.o

Modelsim/Questasim Run

Create Verilog Code (dir.v)

```
module exam2(input [2:0] a, output b, inout c);
initial
begin
$dir(exam2, 0);
//Module name, port number
$dir(exam2, 1);
$dir(exam2, 2);
end
endmodule
```

Transcript:

```
vlib work
vlog dir.v
vsim -c –pli dir.a1   exam2
run 10
```