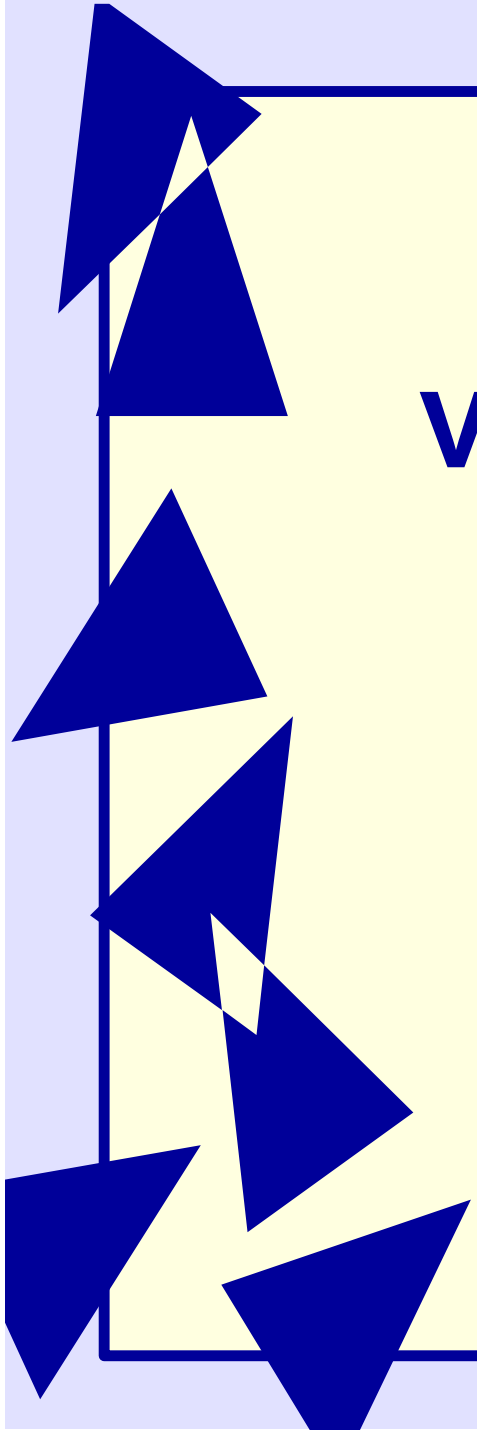


# Verification with VCS Workshop

Synopsys Education and Training Services

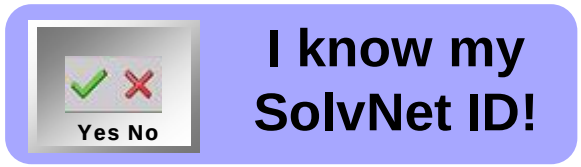
# Synopsys Support Center



- https://solvnet.synopsys.com/EnterACall
- Send e-mails: prchelp@synopsys.com
- Make a call: 800-820-0284

# What is Your SolvNet ID?






If you do not have a SolvNet ID, please speak with the instructor.

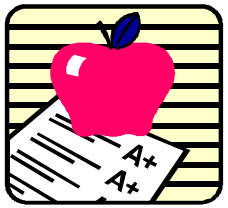
# First Things First



- Instructor Introduction
- Student Guide
- Lab Guide

# Workshop Prerequisites






- Understanding of digital IC design
- Familiarity with UNIX and X-Windows
- Familiarity with a UNIX-based text editor
- Familiarity with Verilog

# Workshop Target Audience



- Digital ASIC design engineers
- Digital ASIC verification engineers
- Limited VCS debugging experience

# Workshop Goal



Acquire the skills to verify and debug Verilog designs using Synopsys VCS

# Workshop Measurable Objectives



**By the end of this workshop you should be able to:**

- **Simulate Verilog designs using VCS**
- **Debug Verilog designs using VCS**
- **Run fast RTL-level regression tests for your Verilog design**
- **Run fast gate-level regression tests for your Verilog design**
- **Acquire the skills and knowledge to successfully implement coverage driven verification methodology using Synopsys tools**

# Agenda: Day 1



DAY 1

| Unit | Topic | Lab |
|---|---|---|
| 1 | VCS Simulation Basics |  |
| 2 | VCS Debugging Basics |  |
| 3 | Debugging with DVE |  |
| 4 | Post-Processing with VCD+ Files | |

# Agenda: Day 2



DAY 2

| Unit | Topic | Lab |
|---|---|---|
| 5 | Debugging Simulation Mismatches |  |
| 6 | Fast RTL Level Verification |  |
| 7 | Fast Gate Level Verification |  |
| 8 | Code Coverage |  |

# Student Introductions



- What is your name?
- What is your job?
- What is your background/work experience?
- What do you want to get out of this workshop?

# Icons Used in this Workshop




Lab Exercise


Caution


Question


Note


Hint, Tip, or Suggestion


Remember


Checklist

# Agenda: Day 1



DAY 1

| Unit | Topic | Lab |
|---|---|---|
| 1 | VCS Simulation Basics |  |
| 2 | VCS Debugging Basics |  |
| 3 | Debugging with DVE |  |
| 4 | Post-Processing with VCD+ Files | |

# Unit Objectives



**After completing this unit, you should be able to:**

- **Compile a Verilog design using VCS**
- **Simulate the Verilog design**

# VCSä is



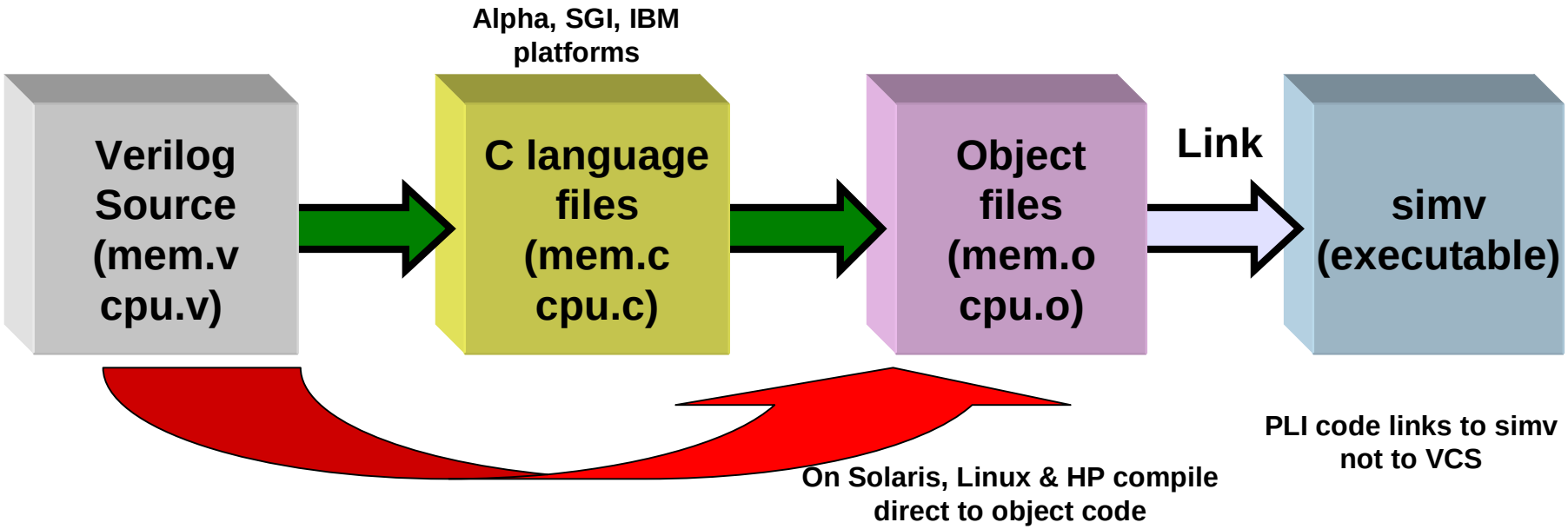

- **A compiled simulator**
  - Verilog Compiled Simulator
  - Digital functional simulator
- **Complies with IEEE-1364**
  - Including PLI 1.0/VPI (PLI 2.0) (Programming Language Interface)
- **Supports simulation at multiple abstraction levels**
  - Behavioral
  - RTL
  - Gate (with SDF support)
  - Sign-off

# How VCS Works



- **Accepts design descriptions in Verilog, C/C++ PLI and models**
- **Two step simulation process:**
  - Step 1: Compile
  - Step 2: Run

**Depending upon platform, VCS first generates C code from the Verilog source, then it compiles and links the object files to the simulation engine to create an executable.**



# VCS Compilation Command Format



```
> vcs sources_files [compile_time_options]
```

- *sources_files*
  - All Verilog source files of the Design Under Test (DUT)
  - Separated multiple source files by spaces
  - Top module should contain testbench for the DUT
- *compile_time_options* (optional)
  - Controls how VCS compiles the source files
  - Critical for optimization for visibility and performance
  - Each unit of this workshop will describe how best to use these *compile_time_options*
- **Generates simulation binary executable `simv` (default)**

# Compile-Time Option Examples



**vcs -help** lists compile options, run-time options, environment variables

**Command line options (commonly used):**

| | |
|---|---|
| **-Mupdate** | **Incremental compilation (only changed files are compiled)** |
| **-R** | **Run after compilation** |
| **-gui** | **Starts the DVE gui at runtime.** |
| **-l <filename>** | **set log file name** |
| **-sverilog** | **Enable SystemVerilog language support** |
| **+v2k** | **Compile with support for Verilog 2001 extensions** |

# Compile-Time Option Examples



## Compile-time options to access Verilog library files

| | |
|---|---|
| `-v` *`lib_file`* | Search for unresolved module references in file *`lib_file`* |
| `-y` *`lib_dir`* | Search for unresolved module references in files residing in directory *`lib_dir`* |
| `+libext+`*`lib_ext`* | Use file extension *`lib_ext`* when searching for files in library directory *`lib_dir`* |
| `+incdir+`*`inc_dir`* | Search *`inc_dir`* directory for `` `include `` files |

## Access Verilog files and options via a file

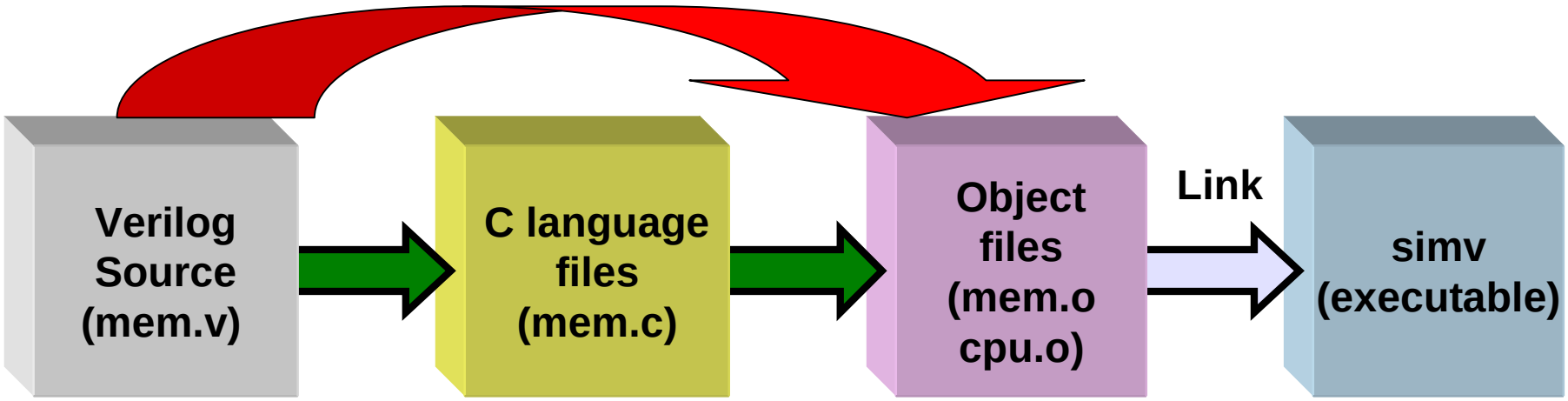
| | |
|---|---|
| `-f` *`file`* | File containing a list of absolute pathnames for the *`sources_files`* and a subset of VCS options |

## User selected simulation binary name

| | |
|---|---|
| `-o` *`foo`* | Creates executable *`foo`* instead of `simv` |

## Faster compilation by compiling only the modules you have changed

If you made a change to module mem in mem.v, then VCS compiles only the module mem. The cpu module object code is used from a previous compilation and linked with the new mem object code in the creation of the executable.



# VCS -Mupdate



## Creates a makefile that is built and maintained by vcs:

**Example**

**vcs file1.v file2.v file3.v -Mupdate:**

- **Compiles Verilog source files, puts c files in directory called csrc**
- **Makefile created in csrc directory**
- **C compilation accomplished via makefile**
- **Object files linked to produce simv**

**Subsequent compilations will be incremental:**

- **Appropriate c files updated**
- **Makefile adjusted**
- **Incremental c compilation as appropriate**
- **Object files re-linked**

**Handy tip:** The –Mupdate switch can be shortened to –M but this will make use of an existing makefile whereas –Mupdate generates a brand new makefile.

# VCS Simulation Command Format



```
> simv [run_time_options]
```

- *run_time_options* (optional)
  - Controls how VCS executes the simulation binary
- **Simulation results reported via**
  - Verilog system task calls
  - User defined PLI routines

# Run-Time Option Examples



- **Stop simulation at time 0**

`-s` Stops simulation at time 0

- **$plusargs() switches**

`+`*`userswitch`* User defined run-time switch

- **Compile-time option status**

`-E echo` Displays compile-time options used for the creation of the current simv executable

- **Log file control**

`-l` *`logfile`* Write output to *`logfile`*

# Sample Simulation Run



Compile all Verilog source files including testbench

VCS generated the simulation binary **simv**

Run simulation with simulation binary

Simulation results reported via Verilog system task calls

```
>ls
adder4bit.v addertb.v
>vcs addertb.v adder4bit.v
...
Parsing design file 'addertb.v' 'adder4bit.v'
Top Level Modules:
     adder_testbench
No TimeScale specified
2 unique modules to generate
2 of 2 modules done
Invoking loader…
simv generation successfully completed
>ls
adder4bit.v addertb.v csrc/ simv
>simv
Chronologic VCS Simulator copyright 1991-2005
Contains Synopsys proprietary information.
Compiler version Y-2006.06; Runtime version Y-2006.06; Dec
11 11:35 2006
0 + 0 = 0
...
F + F = E
*** Verification completed without error ***
     V C S   S i m l a t i o n   R e p o r t
Time: 0
CPU Time:  0.070 seconds; Data Structure size: 0.0Mb
Fri May 12 11:35:31 2000
>
```

# Misc Switches

- **Useful switches for working with Synopsys support**

| | |
|---|---|
| `-ID` | Gets host machine information |
| `-Xman=4` | Combines all source files into a single file “tokens.v” *(Use for submitting test cases to Synopsys)* |

# Using DesignWare Library with VCS



- **Instantiating DesignWare components in Verilog**
  - Format:

```
DWpart #(parameters) u1(.porta(a), .portb(b));
```

  - DesignWare multiplier example:

```
DW02_mult #(inst_A_width, inst_B_width)
u1(.A(inst_A), .B(inst_B),.TC(inst_TC),.PRODUCT(inst_PRODUCT));
```

- **Accessing DesignWare simulation library**

```
-y $SYNOPSYS/dw/sim_ver +libext+.v+
```

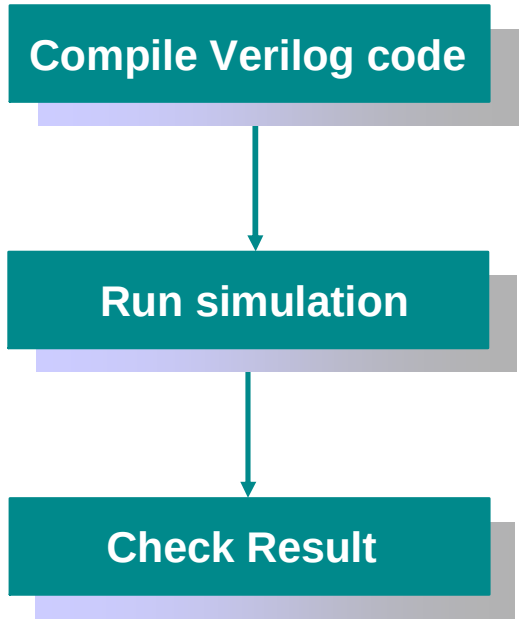
# Summary



- Compile a Verilog design using VCS
- Simulate the Verilog design

15 min

## Simulate a simple Verilog design



# Agenda: Day 1



DAY 1

| Unit | Topic | Lab |
|---|---|---|
| 1 | VCS Simulation Basics |  |
| 2 | VCS Debugging Basics |  |
| 3 | Debugging with DVE |  |
| 4 | Post-Processing with VCD+ Files | |

# Unit Objectives



**After completing this unit, you should be able to:**

- **Describe three methods of debugging Verilog code using VCS**
- **Invoke UCLI debugger**
- **Debug Verilog designs using UCLI**

# VCS Debugging Support



- **Trace and locate causes of errors**
- **Three general methods:**
  - Verilog System Task calls
  - VCS UCLI
  - VCS DVE
- **Four factors to consider:**
  - Simulation speed
  - Signal visibility
  - Signal tractability
  - Usability

# Debugging via Verilog System Tasks 2-4

- **Simulation speed**
  - Fast
- **Signal visibility**
  - Specified by the Verilog system task calls
- **Signal traceability**
  - Mainly pencil and paper
- **Usability**
  - Useful for quick visual feedback
  - May require multiple iterations of inserting Verilog system task calls followed by compile and simulate

# Common Verilog System Task Calls



## n Debug visibility:

| | |
|---|---|
| `$display` | **Prints formatted message to console** |
| `$strobe` | **Like `$display` except printing is delayed until all events in the current time step have executed** |
| `$monitor` | **Monitor signals listed and prints formatted message whenever one of the listed signals changes** |
| `$time` | **Returns current simulation time as a 64-bit integer** |

## n Stopping simulation:

| | |
|---|---|
| `$stop` | **Halts simulation like a breakpoint.** |
| `$finish` | **Halts simulation and terminate the simulation session** |

## n Simulation stimulus and reference:

| | |
|---|---|
| `$readmemh` | **Reads ASCII data from a disk file. Each digit is hexadecimal** |
| `$readmemb` | **Reads ASCII data from a disk file. Each digit is binary** |

# Embedding Verilog System Task Calls 2-6

Instantiated Design Under Test (DUT)

Print adder result whenever input or output of adder changes

Stimulus generation

Verify results

Indicate completion of test

Design Under Test (DUT)

```
module adder_test;
wire [3:0] sum_out;
reg [3:0] a_in, b_in;
in [8:0]
adder u1(a_in, b_in, sum_out);

initial
$monitor($time, "  %h + %h = %h", a_in, b_in, sum_out);

initial
begin
  for (in = 0; in <= 9'h0ff; in = in + 1) begin
    a_in = in [7:4]; b_in = in [3:0];
    if (sum_out !== (a_in + b_in)) begin
      $display("***ERROR at time = %0d ***", $time);
      $display("a=%h, b=%h, sum=%h", a_in, b_in, sum_out);
      $stop;
    end
    #100;
  end
  $display("*** Testbench Successfully completed! ***");
  $finish;
end
endmodule

module adder(a, b, sum);
input [3:0] a, b;
output [3:0] sum;
assign sum = a + b;
endmodule
```

# Debugging via UCLI



- **Simulation speed**
  - Speed depends on the scale of visibility you specify
- **Signal visibility**
  - User specified
- **Signal traceability**
  - User specified breakpoints
  - Some pen and paper
- **Usability**
  - Supports scripting
  - UCLI is compatible with TCL 8.3 and any TCL command can be used with UCLI.

# Compile and Invoke UCLI Debugger



- **Compile and invoke UCLI in one step**

```
> vcs source.v –debug|debug_all -R -ucli
```

  - `-ulci` invokes UCLI and stop simulation time at time 0

- **Compile and invoke UCLI in two steps**
  - Compile

```
> vcs source.v –debug|debug_all
```

  - Invoke UCLI and stop simulation time at time 0

```
> simv -ucli
```

# UCLI Debugger Command Line Options 2-9

- -debug
  - **Enables command line debugging option. This flag does not enable line stepping.**
- -debug_all
  - **Enables command line debugging option including line stepping.**
- -ucli
  - **Forces runtime to go into UCLI mode by default**
- -gui
  - **Compile time option invokes the DVE gui when issued at runtime.**

# UCLI Debugger Command Line Options 2-10

- -l *logFilename*
  - **Captures simulation output, such as user input UCLI commands and responses to UCLI commands.**
- -i inputFilename
  - **Reads interactive UCLI commands from a file, then switches to reading from standard command line input.**
- -k *keyFilename*
  - **Writes interactive commands entered to *inputFilename*, which can be used by a later simv as -i *inputFilename***

# UCLI Commands



| Command | Purpose | Old VCS command (if any) |
|---|---|---|
| show | List scopes, objects and types | show vars |
| get | Get value in any radix | print |
| stop | Breakpoint w/ time, event, file/line/thread | break |
| change | Assign value | set |
| force | Hold value - can't override from design Clock Gen support | force |
| scope | Scope in design hierarchy | show scopes |
| run | Run the simulation | . , continue |

# UCLI Commands



| Command | Purpose | Old VCS command (if any) |
|---|---|---|
| dump | Dump the objects - post-processing | $vcdpluson |
| save | Save the current simulation state. | $checkpoint |
| restore | Restore | $restore |
| senv | Show environment settings in TcL | - |
| config | Change the base etc. | - |
| step | interactive line stepping | step |
| next | interactive line stepping | - |

# UCLI Commands



| Command | Purpose | Old VCS command (if any) |
|---|---|---|
| listing | Displays source text | - |
| start | Start a new simulation | - |
| loads | Get loads (fan-out) across boundaries | - |
| drivers | Get drivers (fan-in) across the boundaries | - |

# Sample UCLI Command Set



- **Changing scope:**

```
scope [-up [level] | -active] [hierarchicalpath]

Show or set the current scope to the specified
  instance. With no arguments, the current scope is
  returned.

-active (Sets the scope to the active point within the
  tool)

-up (Climbs up the scope hierarchy one or n levels)

show -[scopes | ports | variables| signals]
```

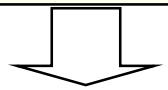
- **Read/write commands:**

```
force  [-deposit | -freeze | -drive ] path value

release path

memory -read|-write nid -file filename [-radix radix]

[-start start_address] [-end end_address]
```

# UCLI Example



```
shell> vcs –debug_all driversget.v
          .
          .
Simv generation successfully completed
shell> simv -ucli
```



```
ucli% scope
top

ucli% config -timebase 1ns
1ns

ucli% dump -add top.t1 -depth 2

VCD+ Writer Y-2006.06 Copyright 2005 Synopsys Inc.
1

ucli% stop -change top.t1.cnt
1

ucli% stop -absolute 3
2
ucli% step
driversget.v, 16 : ra = 0;

ucli% step
driversget.v, 17 : rb = 0;
```

# UCLI Example (cont)



```
ucli% listing
file ./driversget.v, line 17
12: wire wa, wb, wc, wd, we;
13:
14: initial
15: begin
16: ra = 0;
17:=> rb = 0;
18: rc = 0;
19: rd = 0;
20: re = 0;
21: rf = 0;
22: clk = 0;

ucli% run
Stop point #1 @ 00 ps; top.t1.cnt = 0
ucli% stop -disable 1
1
ucli% run
Stop point #2 @ 3000 ps;
ucli% stop -change rc
3
ucli% run
Stop point #3 @ 5000 ps; top.t1.rc = 0
ucli% run
Stop point #3 @ 10000 ps; top.t1.rc = 1
ucli% quit
V C S S i m u l a t i o n R e p o r t
Time: 10000 psCPU Time: 0.010 seconds; Data structure
size: 0.0MbWed Nov 3 08:13:36 2004
```

# Problems in Simulation?



**If you suspect simulation is having problems**

- **Determine whether or not time is advancing:**
  - Halt simulation by hitting CTRL-C
  - Check simulation time
  - Continue simulation by typing in “run” at UCLI prompt
  - Halt simulation again after a short period of time
  - If simulation time has not changed, simulation may be caught in an infinite loop or waiting for a phantom trigger
- **Determine potential location of problem:**
  - Re-compile with `-debug_all` compile-time option
  - Halt simulation
  - Use UCLI command `show` to display variables
  - Trace code execution with UCLI command `next`

# Problems in Simulation? (cont)



- **Use Verilog system tasks to help isolate error**
  - Insert $display statements in the area indicated by source code tracing
  - Display all variables associated with branching statements
- **Re-Compile and monitor $display print out to determine the cause of the problem**
- **Repeat procedure if necessary**

**Example:**

The following is caught in an infinite loop -

```
reg [3:0] i, test;
initial
for (i = 0; i <= 4'hf; i = i + 1)
begin
  test = test + ^i
end
```

```
$display("i = %0d", i);
```

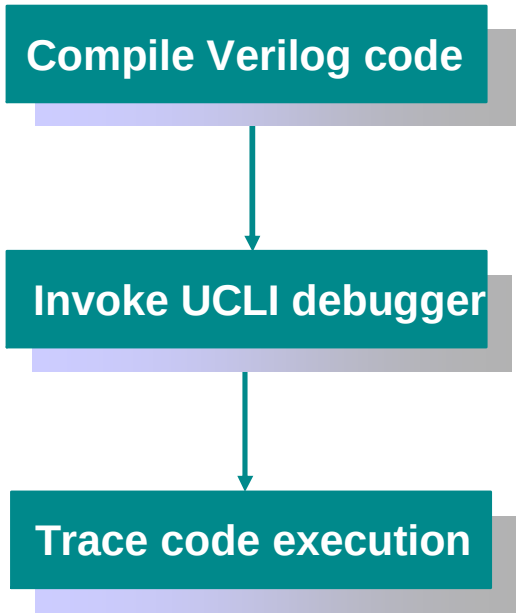
**Inserting this $display statement will clearly show that this loop never ends**

# Lab 2 Introduction



30 min

**Debug a simple Verilog design using UCLI**



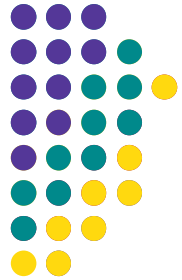
# Agenda: Day 1



DAY 1

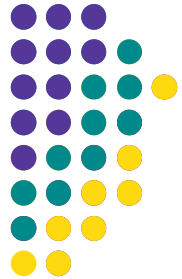
| Unit | Topic | Lab |
|---|---|---|
| 1 | VCS Simulation Basics |  |
| 2 | VCS Debugging Basics |  |
| 3 | Debugging with DVE |  |
| 4 | Post-Processing with VCD+ Files | |

# *Objectives*



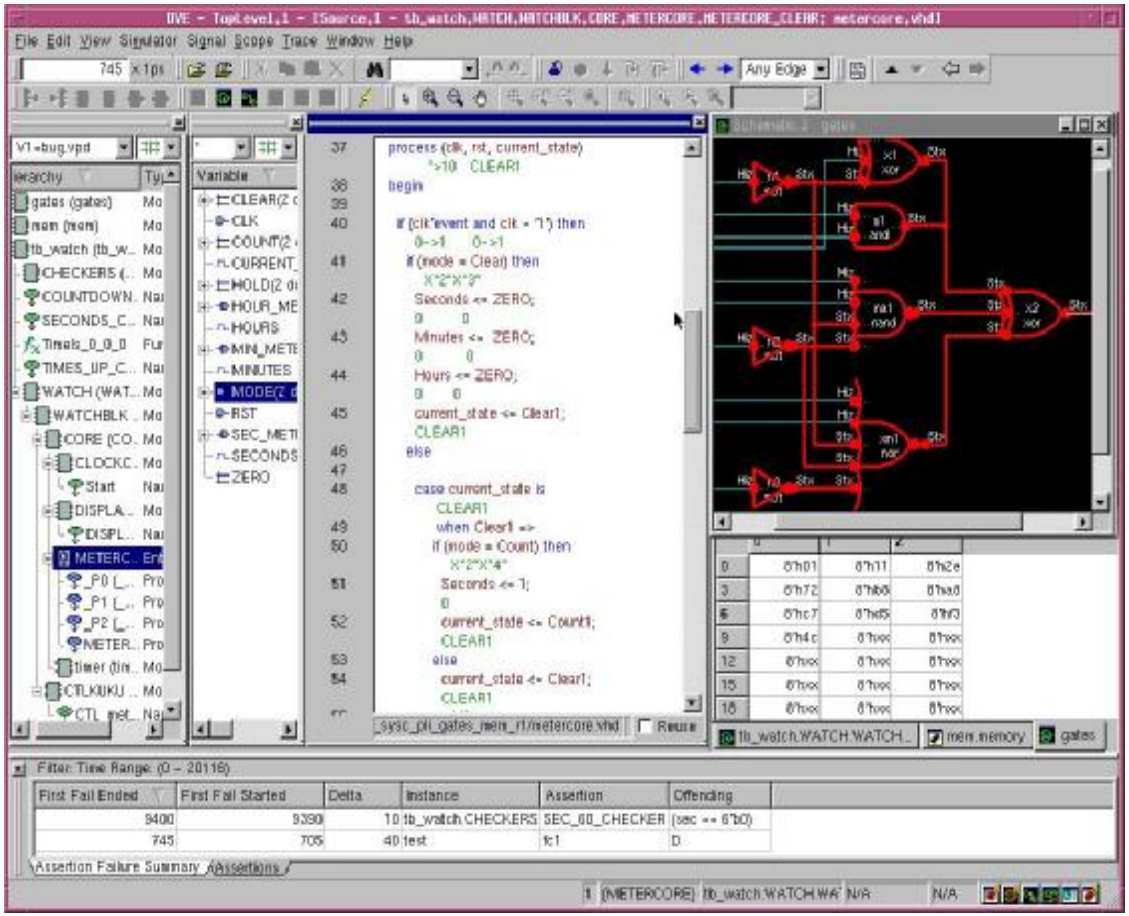
- **Learn to use basic features for debugging RTL**
  - An introduction to the basic features
    - Waveform debugging
    - Source code debugging
    - Listing features
    - Assertions
    - “C/C++” debugger
  - Analyzing design components
    - memories, busses, gates

# *Documentation*



- **User reference manual**
  - $VCS_HOME/doc/UserGuide/dve_ug.pdf
- **Release notes (DVE)**
  - $VCS_HOME/gui/dve/doc/DVEReleaseNotes.txt
- **Quick start example**
  - $VCS_HOME/gui/dve/examples/tutorial/quickstart/quickStart.html
  - Help-> Tutorial  (for Mixed HDL)
- **Example directory**
  - $VCS_HOME/gui/dve/examples
- **dve –help**
  - Gives information about the current DVE command line options

## *Design Debug Productivity*



*Docked windows inside workspace boundaries*

- **An Intuitive and Easy to use GUI**
- **Quickly Find Bugs**
  - RTL or Gate
  - Assertions
  - Testbench
- **Supports**
  - Interactive
  - Post-simulation analysis
- **Multiple Languages**
  - Verilog
  - VHDL
  - C/C++
  - SystemC
  - NTB

# Both Interactive and Post-simulation



- **Full Interactive and Post-simulation Analysis Support**
- **Analyze value change data**
  - value and strength information
  - delta cycle information
  - annotated in Source, Schematic, Path or List views
- **Analyze source execution**
  - Available only in interactive analysis !
  - Ability to select time and instances of interest !
  - Line by line
- **Save and Restore simulation state**
  - Save current state then redisplay it

# *Context Sensitive Menus (CSM)*



- **Point at an object**
  - signals, instances, ports, panes, and assertions.
  - configure main toolbar
- **Click Right Mouse Button (RMB) down**
  - menu appears with relevant options
- **Point to choice**
- **Release button**

# Drag and Drop & Selection



## Drag and Drop

- Point at an object in a pane or window
  - instance, signal, assertion
- Hold LMB down
- Drag object to a new location
- Release button

## Selection

- Use LMB for a simple selection
- Use LMB and Control key (to add or remove an item to selection)
- Or Use LMB and Shift key (to group select)
- Press LMB and drag to select a group of objects

# Invoking DVE
## Interactive Mode



- **Starting from compilation**

```
%vcs source.v –R –gui –debug_all
```

  - -R
    - Starts DVE immediately after compilation (optional)
  - -gui
    - Enables DVE
  - -debug or -debug_all
    - -debug enable command line debugging (no line stepping)
    - -debug _all enables command line debug including line tracing (optional)
  - -ucli
    - Forces runtime to go into UCLI debugger mode (optional)

- **With existing simulation executable**

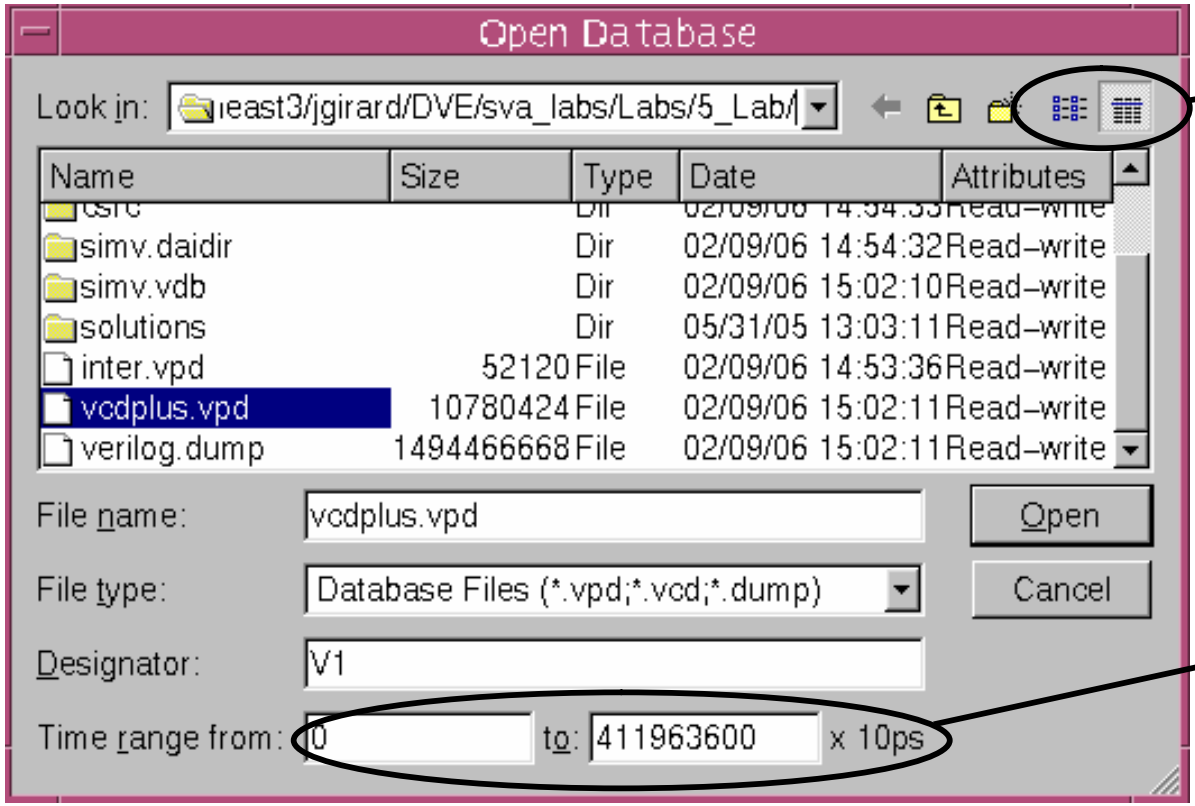
```
%simv -gui
```

  - **-gui**
    - Starts DVE from existing simulation executable (default is simv)

- **Launch DVE GUI**

```
% dve &
```

- **Open Database (vcd,vpd)**
  - click the Open Database icon  open dialog box




**Detailed or List view**

**Note:**
**dve -vpd <filename> brings up dve with file already loaded**

**Optional: Select time slice**

# Windows Icons



- **DVE pop-level window**
  - Frame for displaying current data objects
  - Can contain other windows and panes
    - Source, Schematic, Path, Wave, List, Memory
- **Opening new-top level window**
  - Click the corresponding window icon to remove target symbol  (red circle)

  - Window-> New-> Source
    - New objects will be displayed in new window

# Interactive Simulator Controls



- **Simulation Execution**
  - Click the continue icon  to “start/continue” button
  - Click the stop icon  to stop
  - Enter a ucli command
    - ucli% **run** (run until break point)
    - ucli% **run 100** (run for 100 time units)
    - ucli% **run 100ms** (run for 100 ms)
    - ucli% **run -posedge wb_ack_i** (run until positive of wb_ack_i)
  - Use simulator controls to set a simulation break point and run VCS
    - “Step Time”
    - “Go To Time”

# Interactive Simulator Controls (Cont'd)

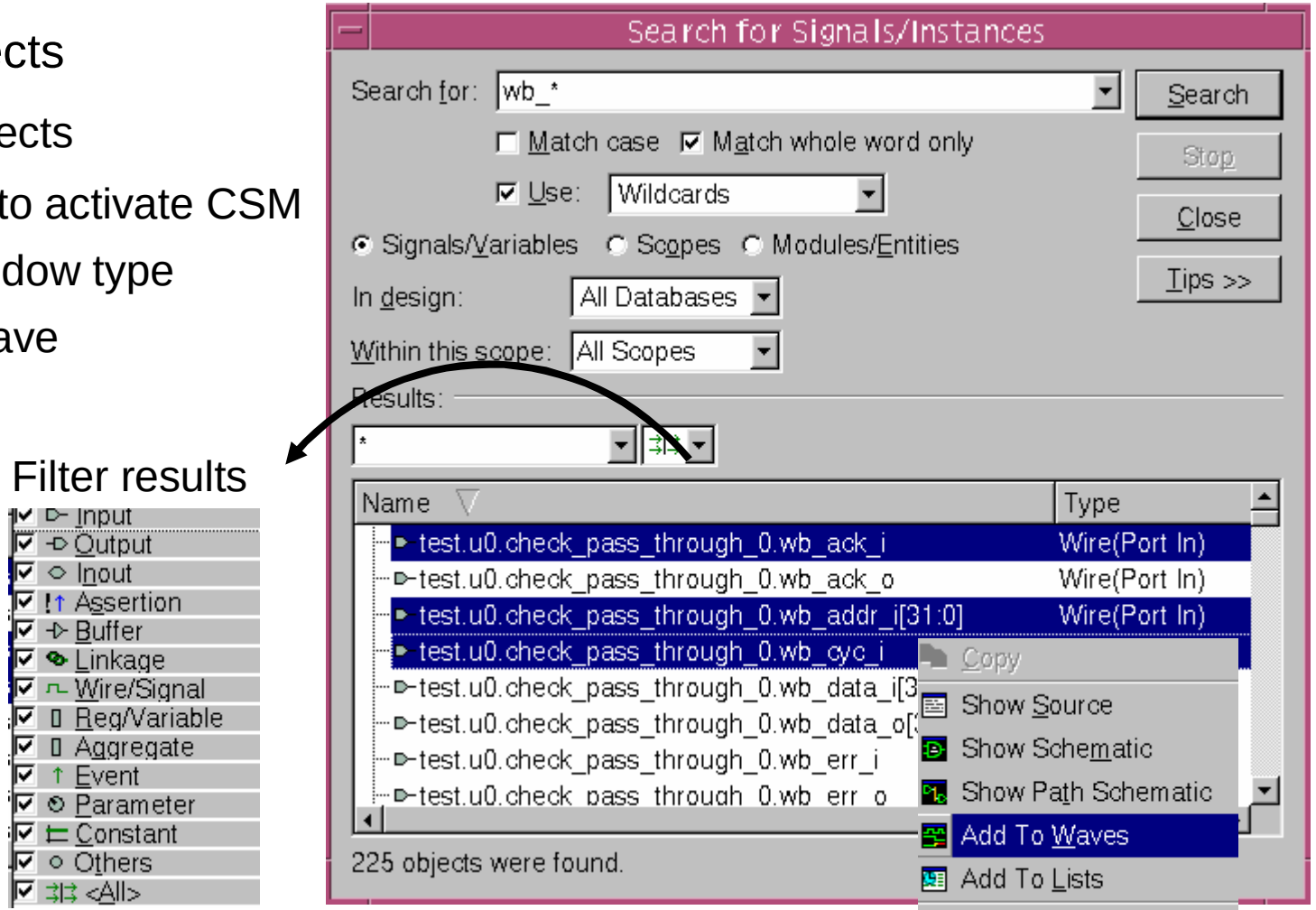


- **Simulation controls**
  - Click step icon  to simulate next executable line
  - Click next icon  to step over tasks and functions
  - Click restart icon  to reset simulation to time zero
  - ucli commands
    - ucli% step
    - ucli% next
    - ucli% restart

# Searching for Objects

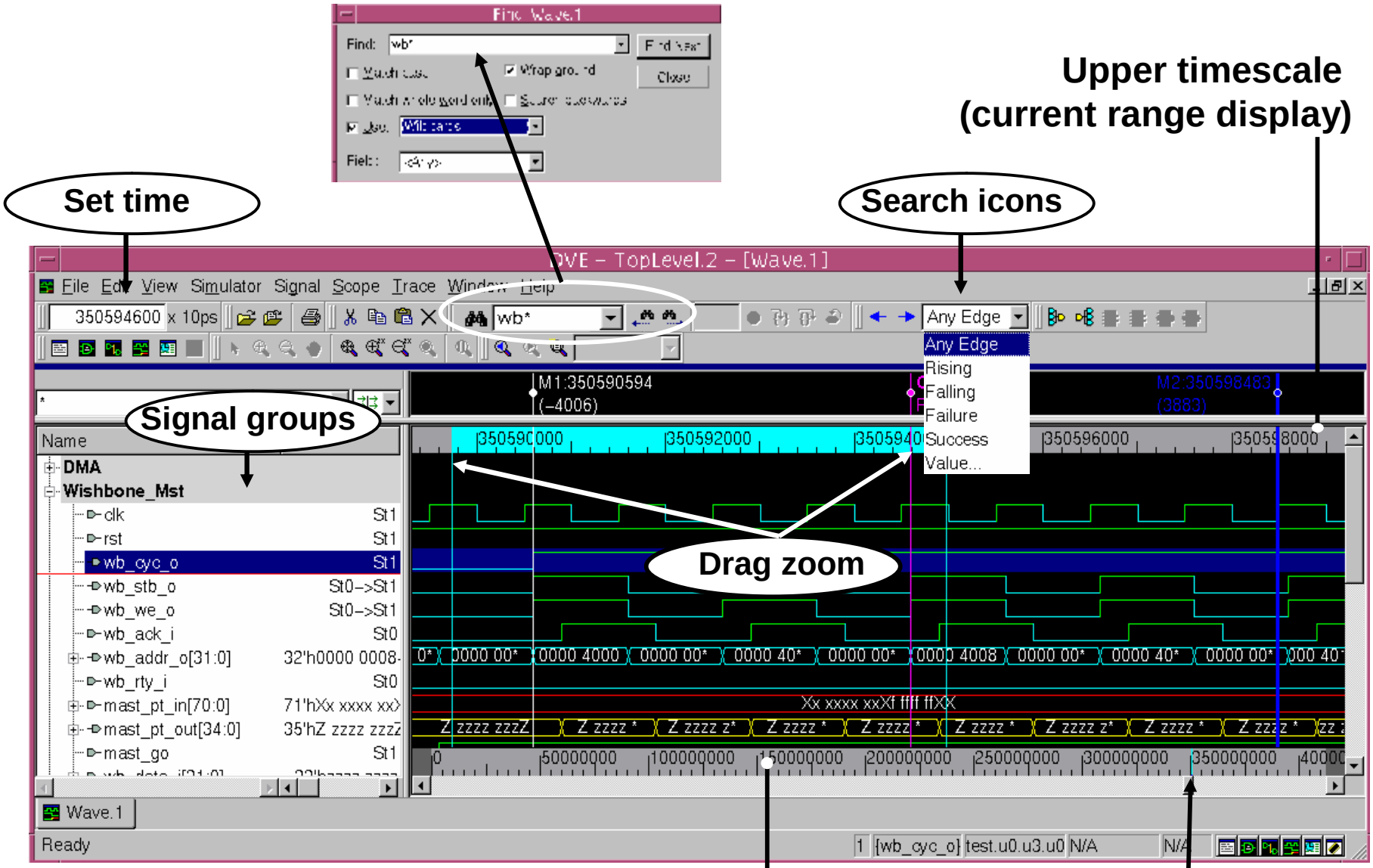


- **Toolbar menu: *Edit -> Search for Signals/ Instances***
  - Viewing objects
    - Select objects
    - right click to activate CSM
    - Select window type
      - e.g. Wave



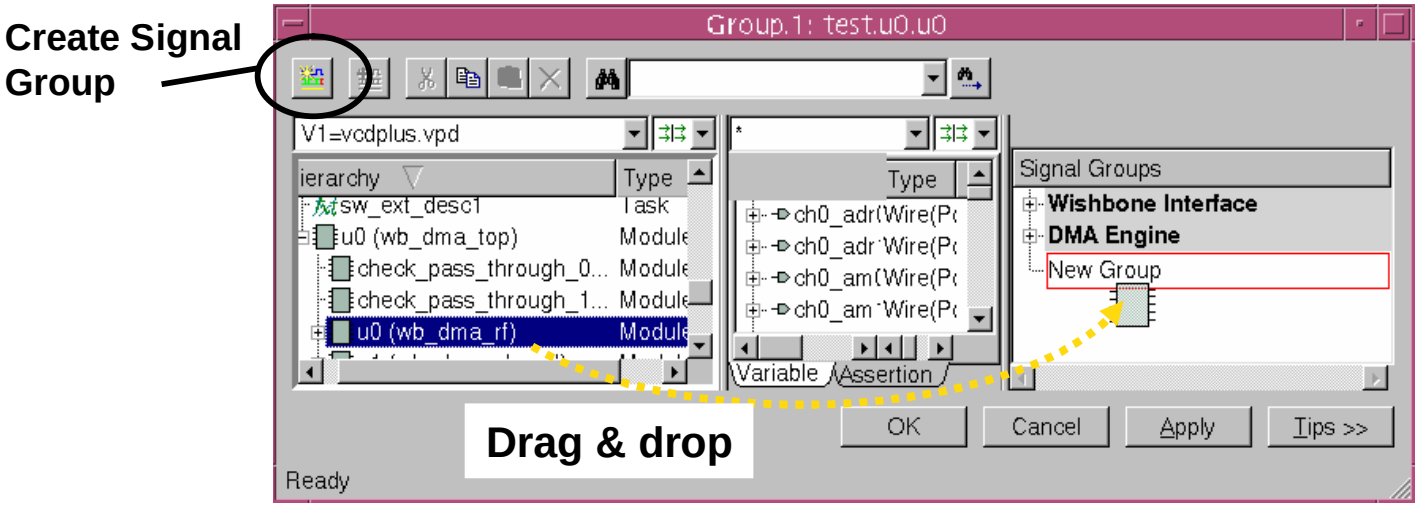
# Wave Window
## Overview





- **Viewing signals**
  - Select object (signals, scopes or assertions)
    - Click the wave icon  to add objects to a wave window
    - Use CSM and select 

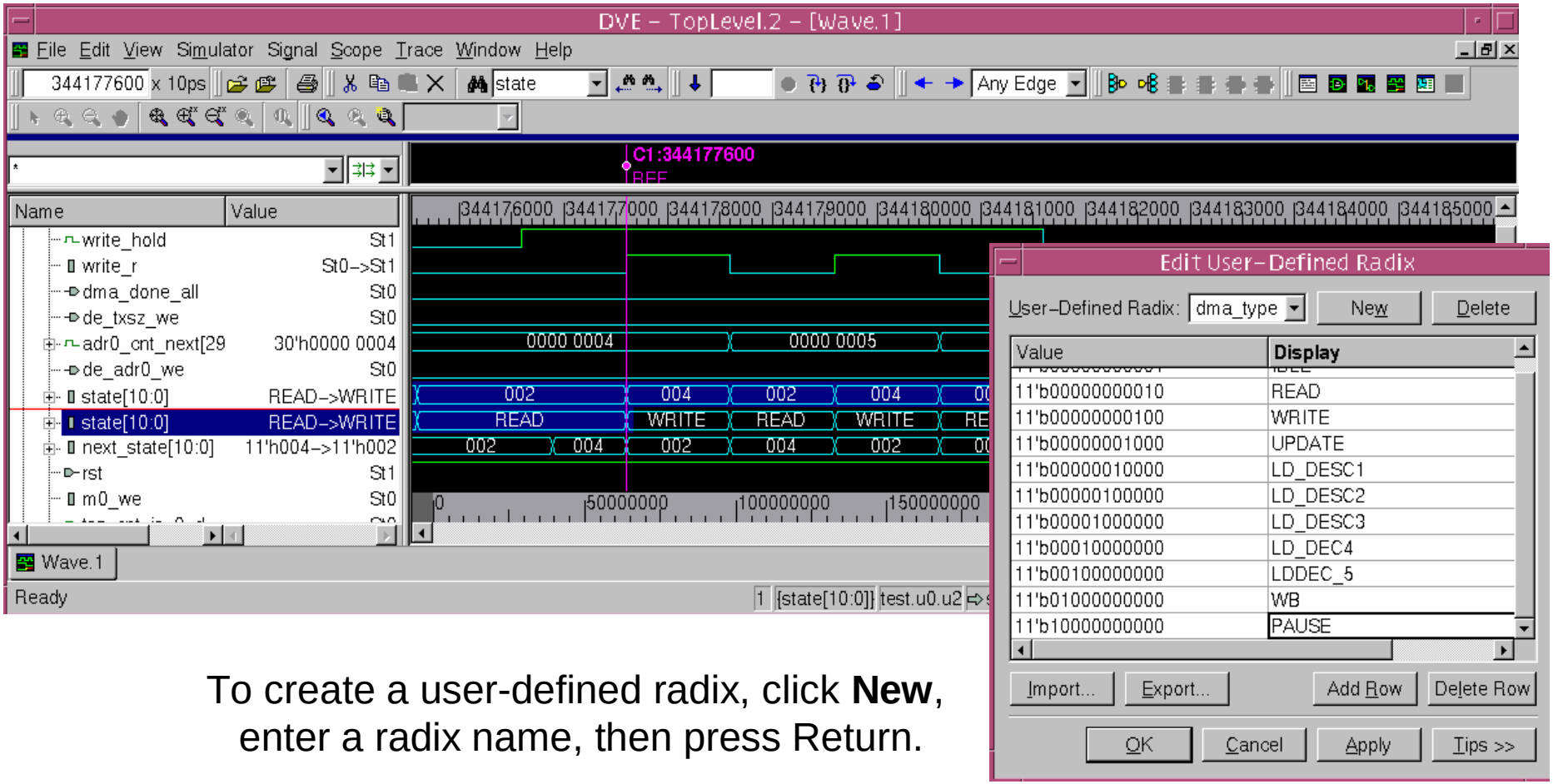
item
    - Double click on a failing assertion summary tab
    - Or drag and drop object to open wave window
- **Grouping signals**
  - **Toolbar menu**: *Signal -> Signal Groups …*



# User Defined Radixes



- **User Define Radixes**
  - **Toolbar menu:** *Signal -> Set Radix->User-Defined->Edit*
    - Import or Export user types: **IDLE 11'b00000000001** -- file format

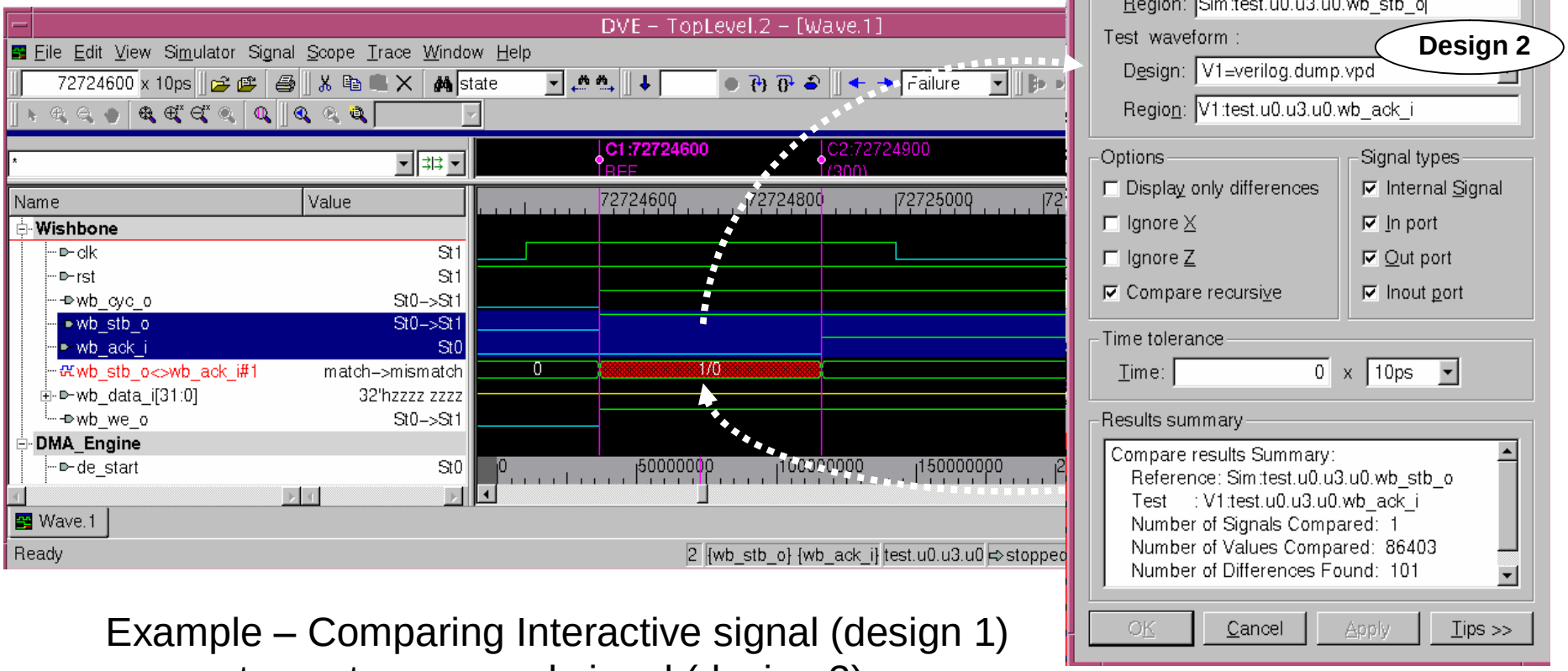



To create a user-defined radix, click **New**, enter a radix name, then press Return.

# Signal Compare Tool



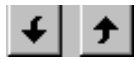
- **Compare function**
  - Select two signals, scopes or designs
  - **Toolbar menu**: *Signal -> Compare*
    - A new signal is created for each compare point




Example – Comparing Interactive signal (design 1) to post processed signal (design 2)

# Bus Builder Tool

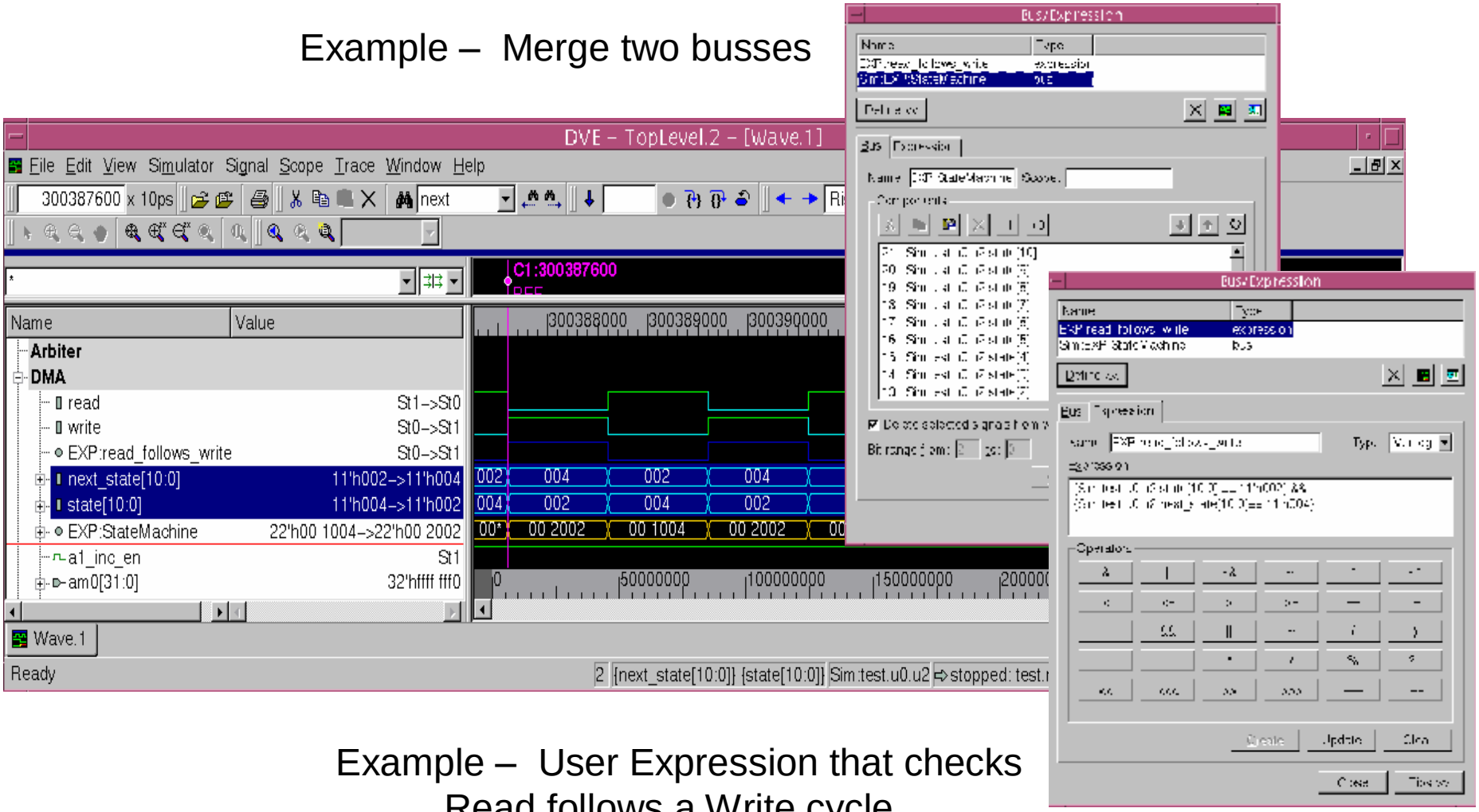


- **Creating a bus**
  - Select the signals (List / Wave)
  - **Toolbar menu**: *Signal -> Set Bus …*
  - Specify Name
  - Use  up/down selection button to arrange order
  - Bit reverse order, select bus and click  the reverse order button
  - Bus padding, use  add one's and  to add zero's
- **Search for events**
  - Event searching in either time direction (List / Wave)
  - Select the signals
  - **Toolbar menu**: *Signal -> Set Expressions …*
  - Specify Name

# Bus Builder Example



Example – Merge two busses



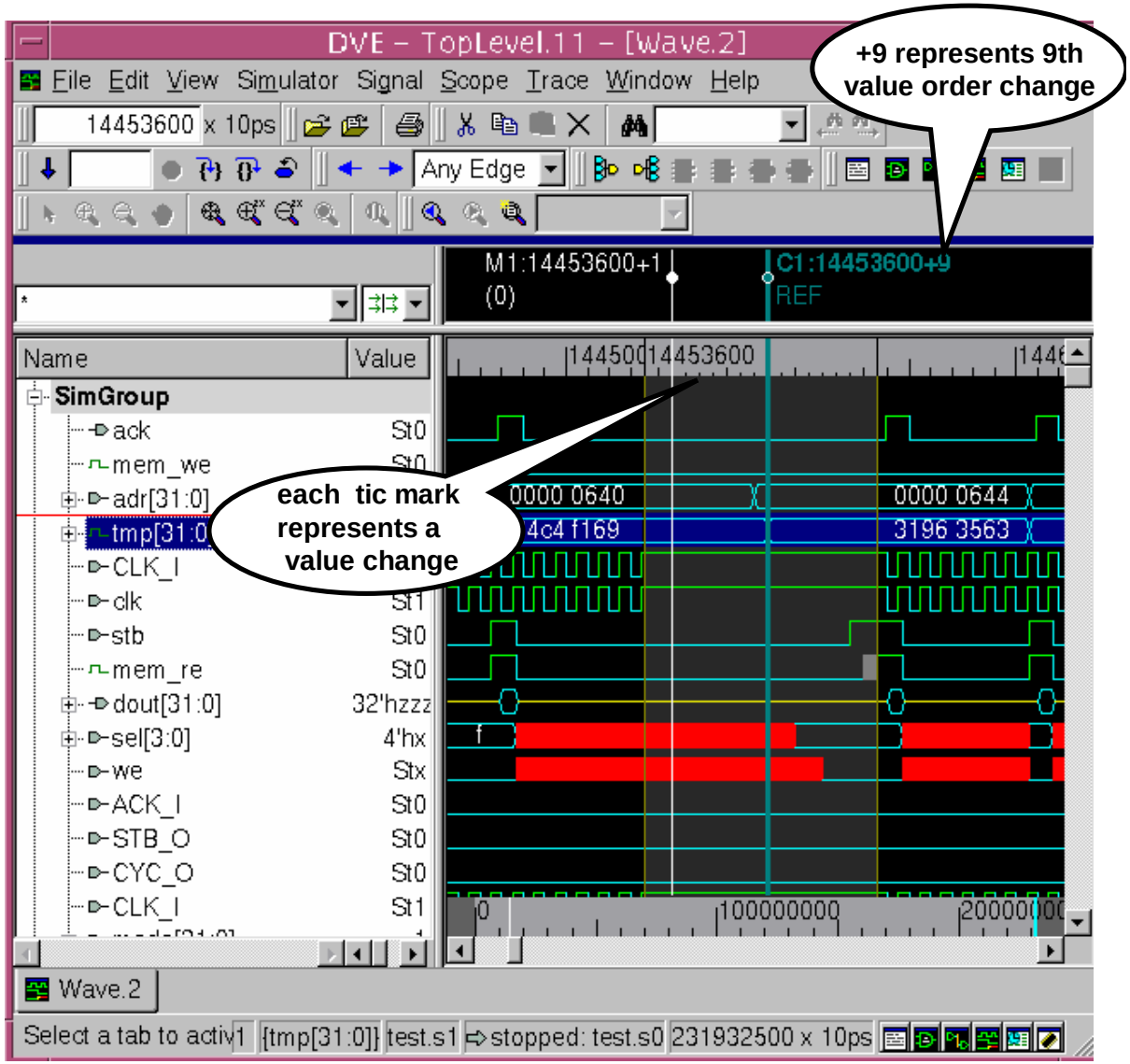

Example – User Expression that checks
Read follows a Write cycle

# Displaying Delta Cycle



- **Value changes that occurred within the same time step**
  - Toggle recording of delta cycle
    - *Simulator => Capture Delta Cycle Values*
  - Record delta cycle data in the VCD+ file add
    - $vcdplusdeltacycleon
  - To view event queue
    - right click to activate CSM
    - Select *Expand Time*

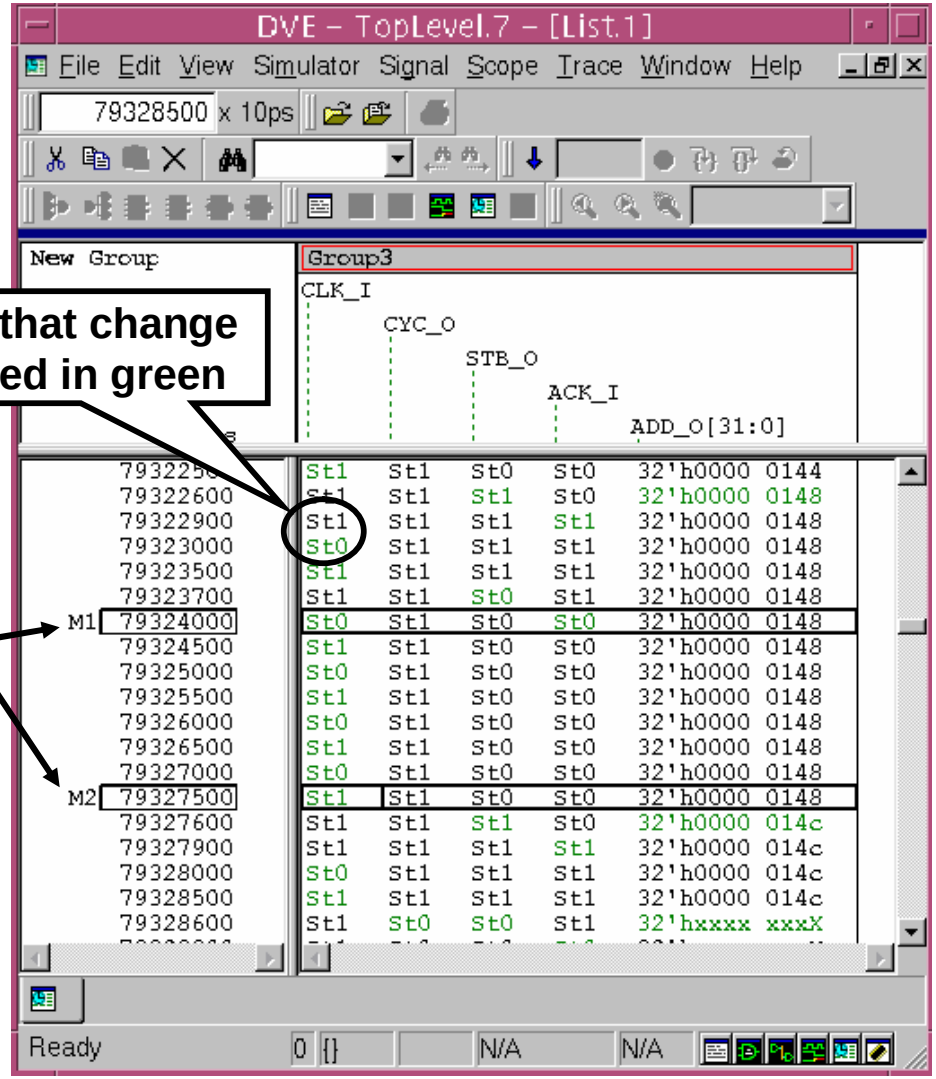
**Warning Perf Impacted**



# List Window
## Overview



- **Views simulation results in ASCII format**
  - Updates when displayed signal changes
  - Locks to C1 cursor
  - Set markers
  - Compare signals
  - View database
  - Print tabular output to a textfile



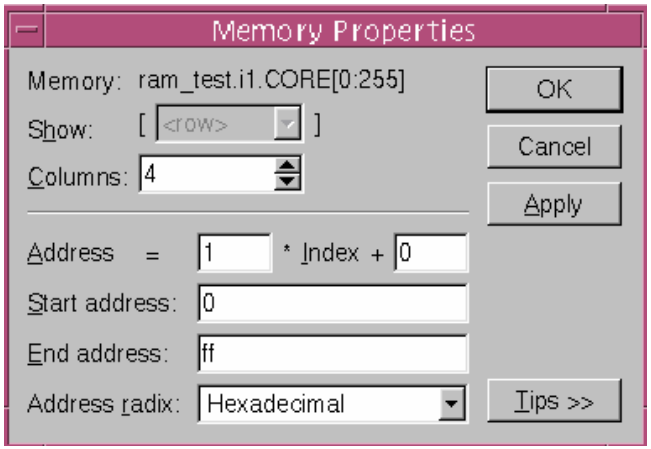

To Add signals click  or drag & drop them

# Memory Viewer

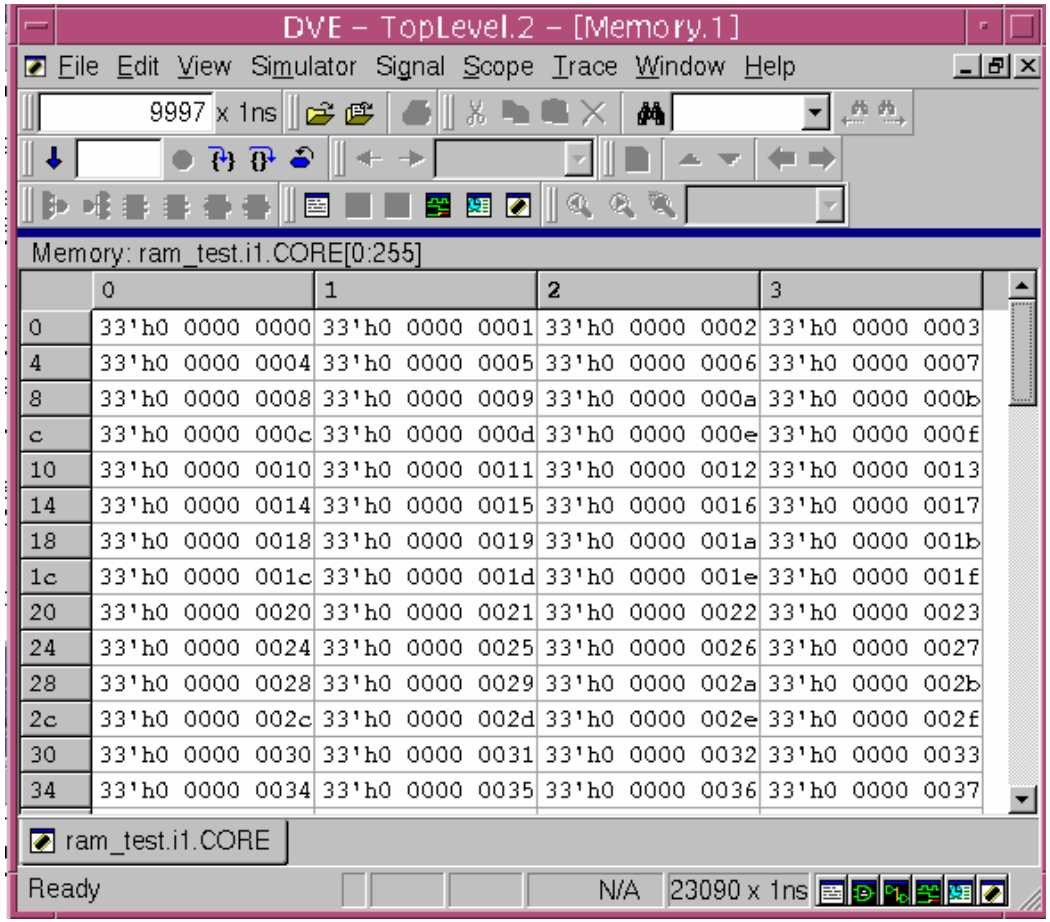


Currently requires:
$vcdplusmemon or
*Simulator->Add Dump ..*
(Check Aggregates box)

- **To dump memories**
  - Select a memory in the “Data” pane
    - right click to activate CSM
    - Select *Show Memory*
    - right click to activate memory properties




Example – Memory Properties dialog



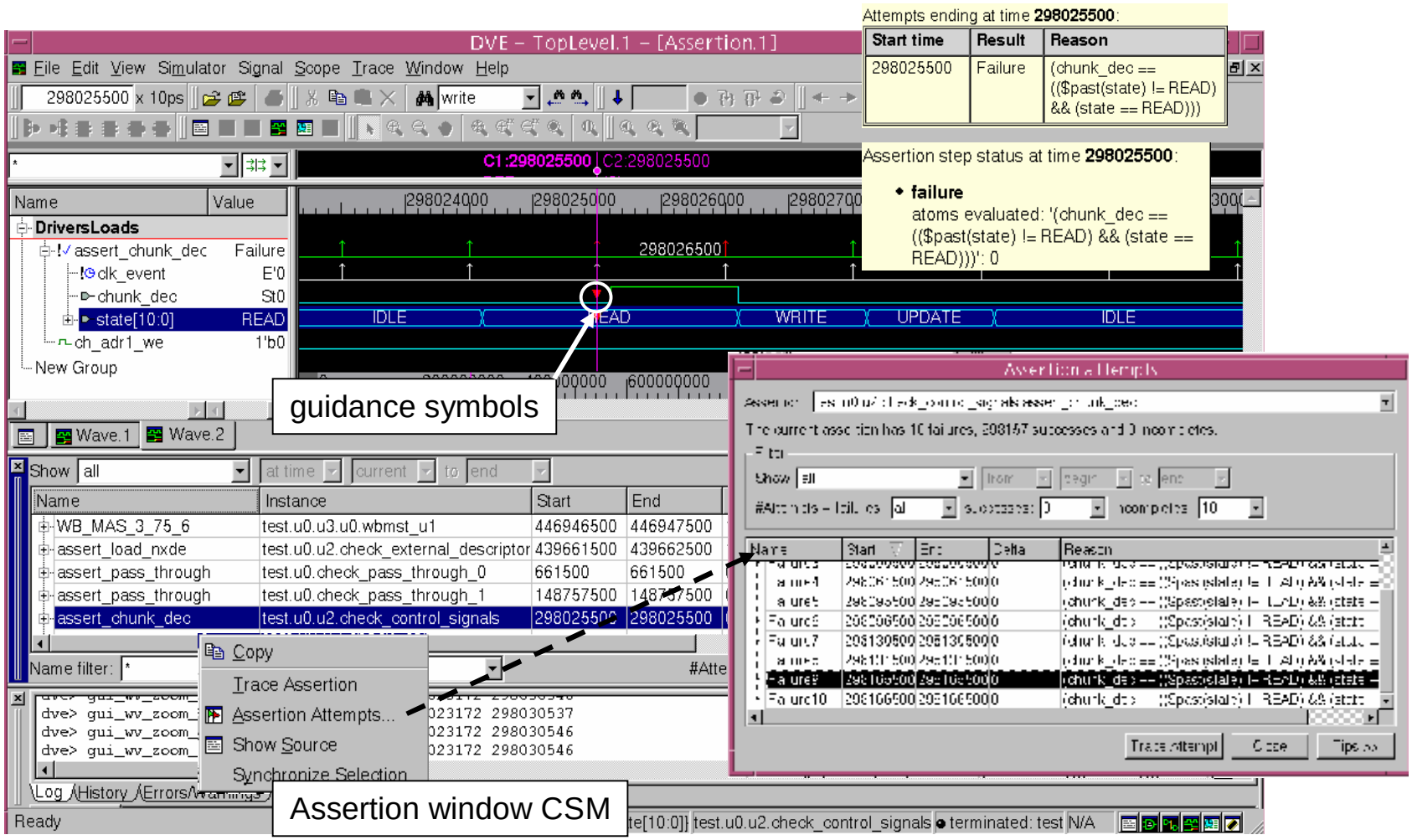

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 33'h0 0000 0000 | 33'h0 0000 0001 | 33'h0 0000 0002 | 33'h0 0000 0003 |
| 4 | 33'h0 0000 0004 | 33'h0 0000 0005 | 33'h0 0000 0006 | 33'h0 0000 0007 |
| 8 | 33'h0 0000 0008 | 33'h0 0000 0009 | 33'h0 0000 000a | 33'h0 0000 000b |
| c | 33'h0 0000 000c | 33'h0 0000 000d | 33'h0 0000 000e | 33'h0 0000 000f |
| 10 | 33'h0 0000 0010 | 33'h0 0000 0011 | 33'h0 0000 0012 | 33'h0 0000 0013 |
| 14 | 33'h0 0000 0014 | 33'h0 0000 0015 | 33'h0 0000 0016 | 33'h0 0000 0017 |
| 18 | 33'h0 0000 0018 | 33'h0 0000 0019 | 33'h0 0000 001a | 33'h0 0000 001b |
| 1c | 33'h0 0000 001c | 33'h0 0000 001d | 33'h0 0000 001e | 33'h0 0000 001f |
| 20 | 33'h0 0000 0020 | 33'h0 0000 0021 | 33'h0 0000 0022 | 33'h0 0000 0023 |
| 24 | 33'h0 0000 0024 | 33'h0 0000 0025 | 33'h0 0000 0026 | 33'h0 0000 0027 |
| 28 | 33'h0 0000 0028 | 33'h0 0000 0029 | 33'h0 0000 002a | 33'h0 0000 002b |
| 2c | 33'h0 0000 002c | 33'h0 0000 002d | 33'h0 0000 002e | 33'h0 0000 002f |
| 30 | 33'h0 0000 0030 | 33'h0 0000 0031 | 33'h0 0000 0032 | 33'h0 0000 0033 |
| 34 | 33'h0 0000 0034 | 33'h0 0000 0035 | 33'h0 0000 0036 | 33'h0 0000 0037 |

Example – Memory dump @ 23090ns

# Assertion Summary Window

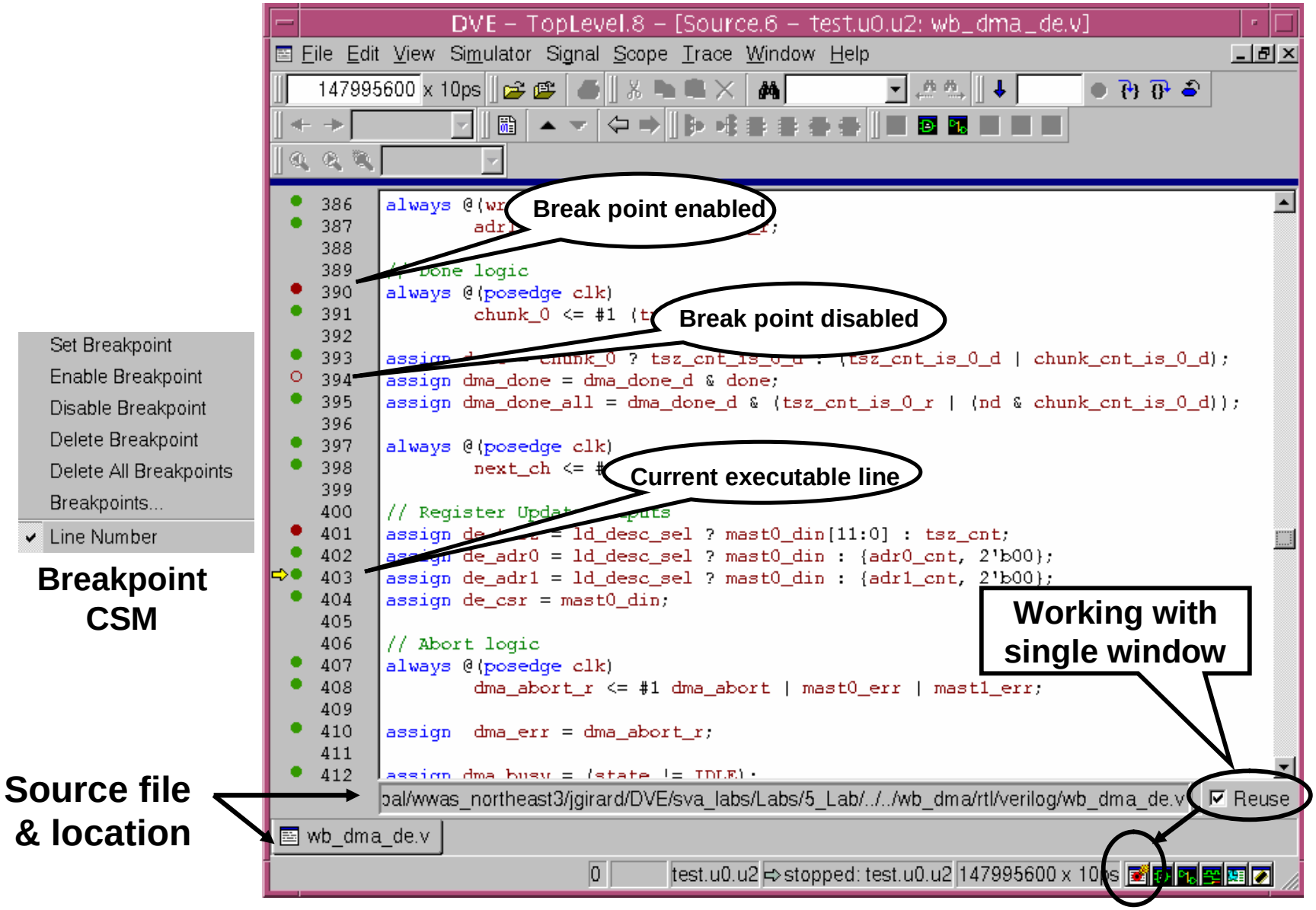


- Compile VCS with ***-debug -assert dve***
  - Double click assertion failure to populate wave window
    - Place cursor over guidance symbols to obtain details status



# Source Window
## Overview

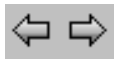




# Debugging Source



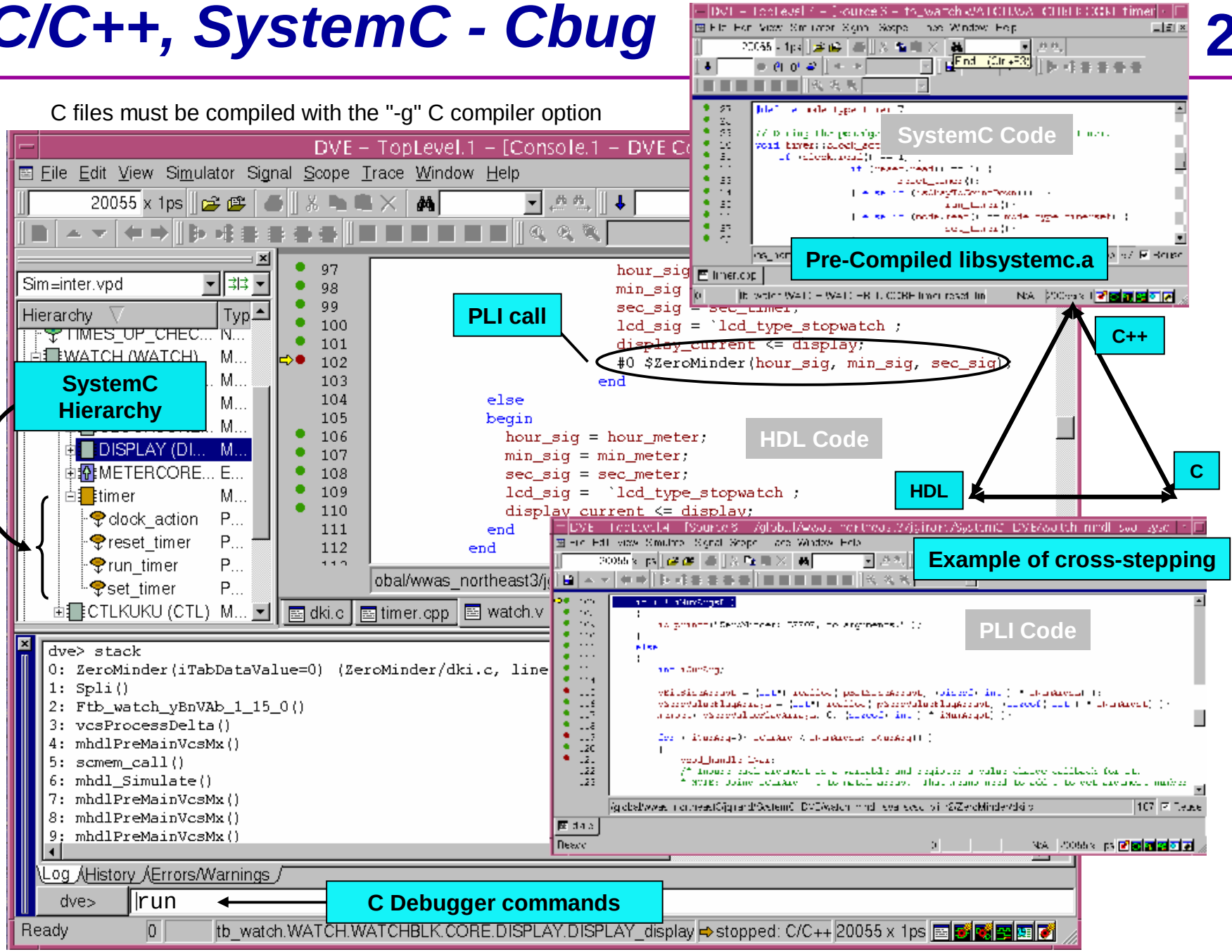
- **Stepping source** ***(-debug_all)***
  - Select object (signals, scopes or assertions)
    - Click the source icon  to add objects to a source window
    - Use CSM and select 

item
    - Double click on a scope icon
    - Drag and drop object to open source window
  - Click step icon  to simulate next executable line
  - Click next icon  to step over tasks and function
  - Click annotate values icon  for backannotation
- **Navigating hierarchy**
  - Place cursor on module instance and click right
    - select *Move Down to Definition* (Source CSM) or click 
    - select *Move Up to Parent_* or click 
    - To move backwards or forwards in a list of scopes click 

# C/C++, SystemC - Cbug



C files must be compiled with the "-g" C compiler option



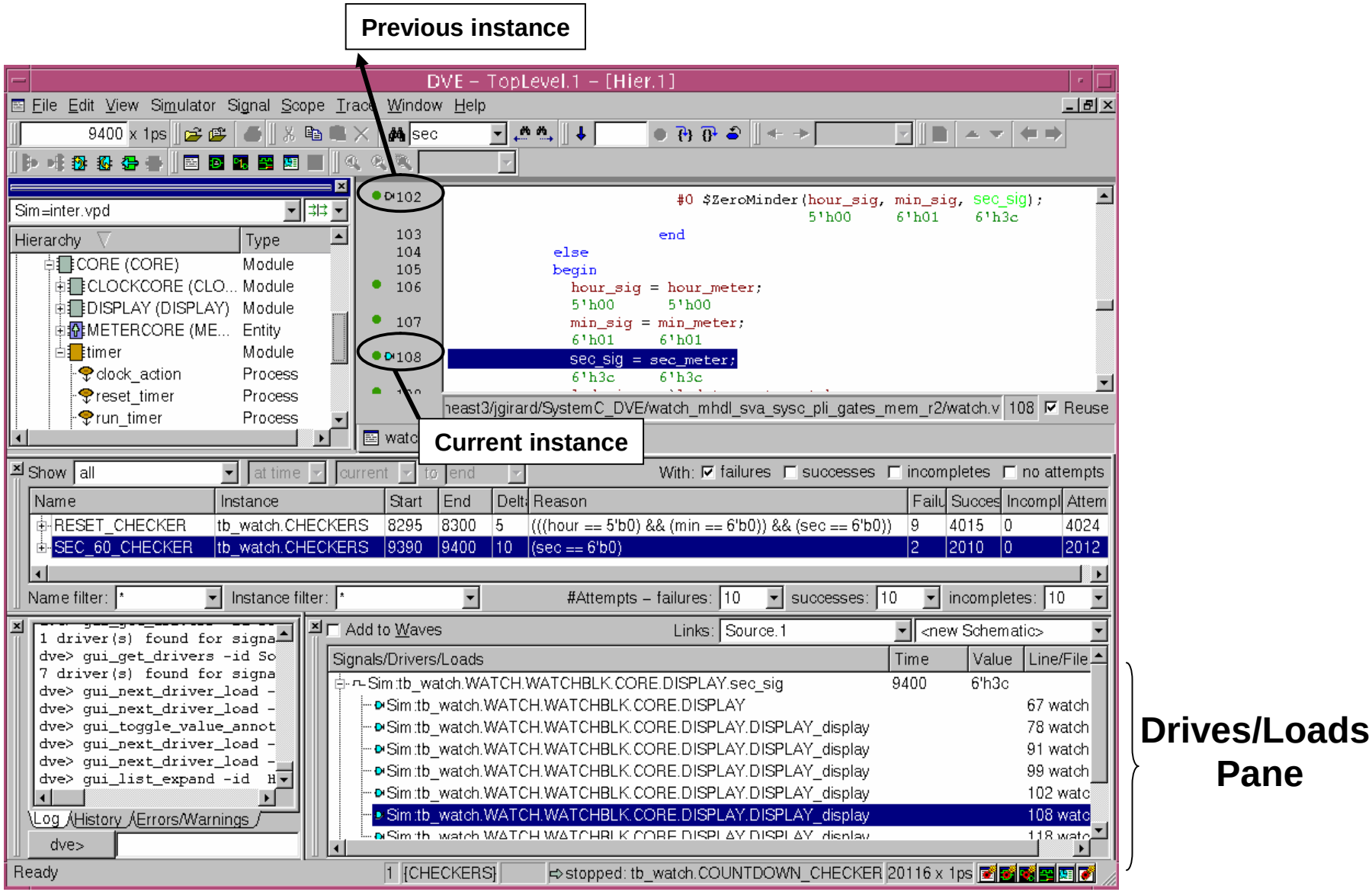
# Backtrace



- **Problem**
  - A number of signals exhibiting less desirable values
- **Solution**
  - Perform a "*backtrace*"
  - Displays a list of active drivers/loads at specified time
  - Trace back to the earliest unwanted signal transition or value
  - Identify signal responsible for the erring behavior
  - Reapply procedure, and eventually locate source of misbehavior
- **Displaying Drivers/Loads**
  - In any DVE analysis window highlight or select a signal
  - Then simply click either the  driver or  load icon
  - Or in a Wave or list window double click on signal
  - Then use next  and previous  instances icons

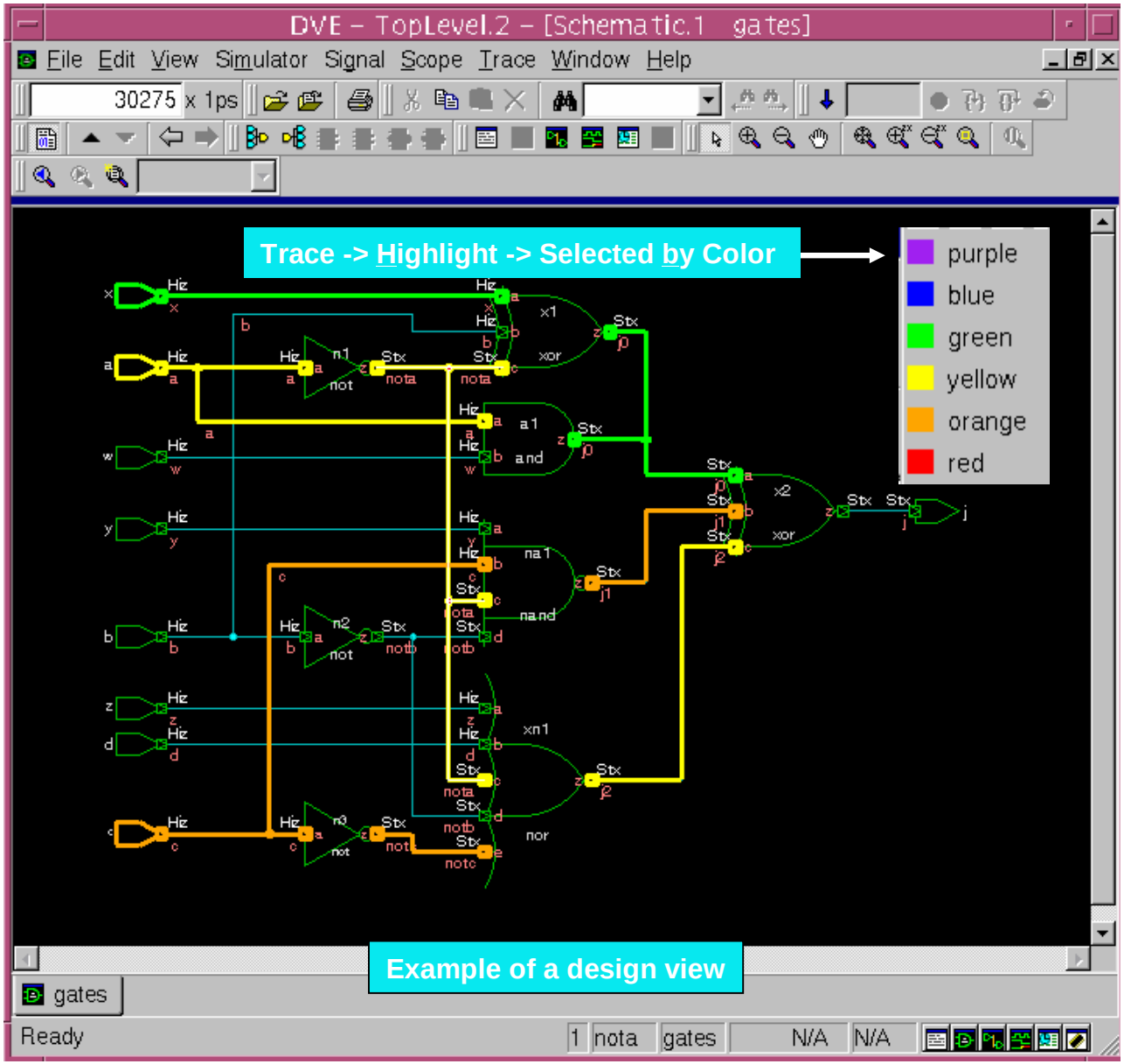
# Driver / Load Pane





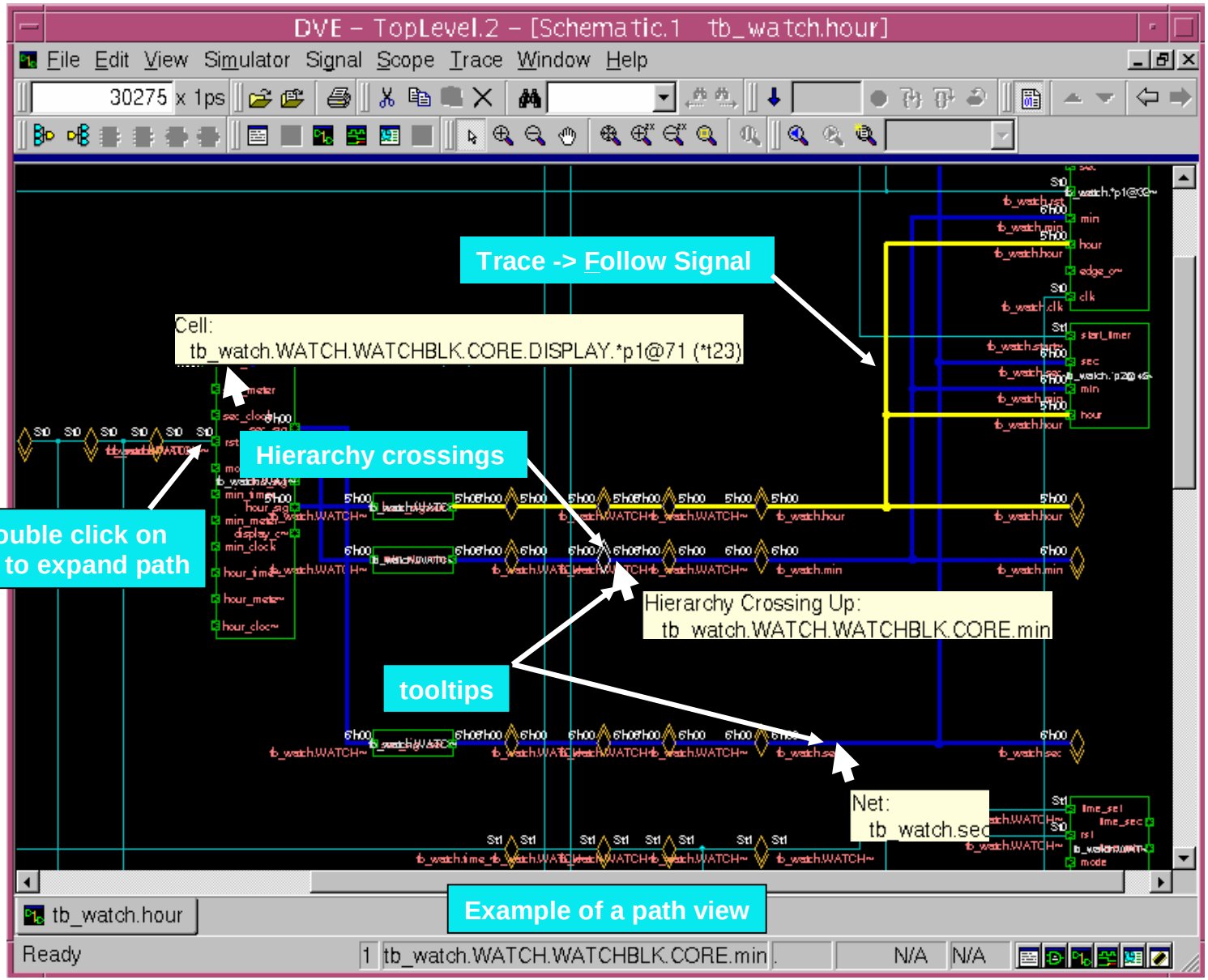
# Schematic Window





# Path Tracing



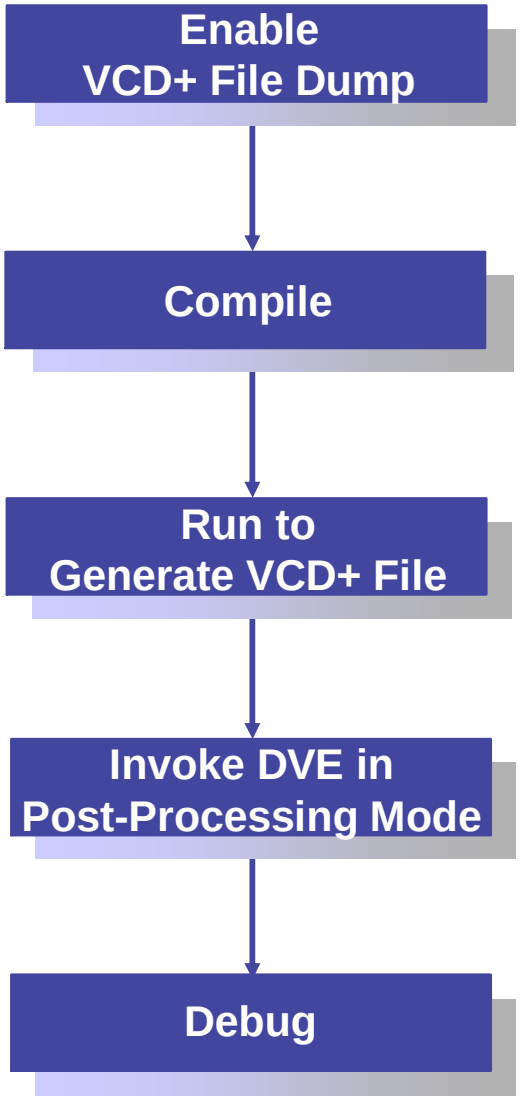


# Lab 3 Introduction



45 min

**Locating sources of error in Verilog code using DVE in Post-Processing mode**



# Agenda: Day 1



DAY 1

| Unit | Topic | Lab |
|---|---|---|
| 1 | VCS Simulation Basics |  |
| 2 | VCS Debugging Basics |  |
| 3 | Debugging with DVE |  |
| 4 | Post-Processing with VCD+ Files | |

# Unit Objectives



- Embed VCD+ system tasks in source code
- Compile and run simulation to generate VCD+ file
- Invoke DVE in post-processing mode
- Read VCD+ file into debugger memory
- Debug

# DVE Post-Processing Mode



- **Simulation speed**
  - Simulation speed depends on data dump commands
  - Debugging speed is fastest
- **Signal visibility**
  - User specified
- **Signal tractability**
  - Signal traced via waveform, schematic or source
- **Usability**
  - Graphical interface is the most user friendly
  - Can be used at all levels of complexity

# When to Use Post-Processing Mode



- **Use post-processing mode when:**
  - Debugging a mature design
  - Simulation analysis needed by multiple engineers
  - Run simulation in script
- **Multiple users can debug in parallel**
  - The VCD+ file, once generated, can be read by multiple users to debug different problems in parallel

# What are VCD+ files?



- **Binary simulation history files**
  - Similar to VCD (Verilog Change Dump) ASCII files
  - Stores transition times, values of nets and registers and design hierarchy
- **Differs from VCD files in the following ways:**
  - Compressed binary format requires less disk space
  - Compressed binary format loads faster
  - Supports recording of order of source code execution
  - Built-in VCD+ system tasks provided for controlling contents of VCD+ file and size
- **DVE only process VCD+ files**
  - VCD files can be converted into VCD+ files

# Post-processing Debugging Procedure 4-6

- Modifying the Verilog source code to include VCD+ file dump system task call $vcdpluson
- Compiling the Verilog source code with VCS
- Generating VCD+ file by executing the simulation binary created by VCS
- Invoking DVE in post-processing mode
- Reading VCD+ dump file in DVE
- Examine simulation results in Waveform window, Schematic window and Assertion window to locate error in source code displayed on Source window

# VCD+ System Tasks



- **VCD+ system tasks may be inserted in source files or entered at the simulation interactive prompt**

- `$vcdpluson(`*`level_number`*`,`*`module_instance`*`,…|`*`net_or_reg`*`,…)`

  **Start recording nets and registers (require `-debug` compile switch):**

  - *`level_numbers`*

    **specifies levels of hierarchy to record**

    `0` - **record the entire hierarchy of specified module**
    `1` - **record the top-level hierarchy of specified module**
    `n` - **record through n-levels of hierarchy of specified module**

  - *`module_instance`*

    **specifies module to record**

  - *`net_or_reg`*

    **specifies individual net or register to record**

  - *Omitting all arguments records all nets and registers of entire design*

- `$vcdplusoff (`*`module_instance`*`,…|`*`net_or_reg`*`,…)`

  **Stops recording in a module instance or individual net or register**

# VCD+ System Tasks (cont)



- `$vcdplusautoflushon`

  **Instructs VCS to write results from simulation memory to VCD+ file whenever there is an interrupt such as $stop system task or a ucli stop command or a DVE Stop button activation**

- `$vcdplusautoflushoff`

  **Turns off automatic flushing of data on an interrupt**

- `$vcdplusflush`

  **Instructs VCS to write results from memory to VCD+ file**

# VCD+ System Tasks



- `$vcdplusdeltacycleon`

  **Turns on delta cycle recording for post-processing**

- `$vcdplusdeltacycleoff`

  **Turns off delta cycle recording**

- `$vcdplusglitchon`

  **Turns on zero delay glitches for post-processing**

- `$vcdplusglitchoff`

  **Turns off zero delay glitches recording**

# VCD+ System Tasks Example



- Sample Verilog source code

```
module adder_testbench;
reg        test_cin;
wire       adder_cout;
reg [3:0]  test_a, test_b;
wire [3:0] adder_sum;
reg [15:0] addr, stimulus, ref_result;
reg [15:0] testmem [0:1023];

adder u1(test_a, test_b, test_cin, adder_cout, adder_sum);

initial
begin
  $vcdpluson;
  $vcdplusdeltacycleon;
  $vcdplusglitchon;
  $readmemb("adder_ref.vec", testmem);
  for (addr=0; addr <= 16'h03ff; addr=addr+2)
    begin
      #100 stimulus = testmem[addr];
           test_cin = stimulus[8];
           test_a = stimulus[7:4];
           test_b = stimulus[3:0];
    end
  #100 $display("\n----- Simulation Completed Without Error -----\n");
end
```

# Compiling for VCD+ file



```
>vcs files vcdplus_switches Other_switches
```

- ***files***
  - Source files (including Verilog, C/C++ PLI) as defined in Unit 1
- ***vcdplus_switches***
  - Instructs VCS compiler to recognize VCD+ system tasks
  - Controls VCD+ file generation
- ***other_switches***
  - Compile-time options (e.g. `-Mupdate`, `-R`, etc… )
- Sample VCD+ file compilation command:

```
vcs source.v –debug_all
```

- Sample command for invoking DVE in post-processing mode:

```
> dve -vpd vcdplus.vpd
```

# VCD+ file Compilation Switches



| | |
|---|---|
| `-debug` | **Required compile-time option** |
| `+vpdfile+`*`filename`* | **Specifies writing to an alternative VCD+ filename rather than the default** `vcdplus.vpd` |
| `+vpdupdate` | **Allows simultaneous writing and reading of the VCD+ file** |
| `+vpdbufsize+`*`MB`* | **Specifies the size of temporary buffer to store VCD+ values before writing to disk (*default is 5MB or room for 15 value changes*)** |
| `+vpdfilesize+`*`MB`* | **Specifies maximum size of VCD+ file (*when limit is reached, new event replaces oldest*)** |

# VCD+ Tips and Suggestions



- **Use target based dumping:**
  - Capture time slices
    - Use $vcdpluson in conjunction with $vcdplusoff and #delay

```
module source;
  moduleA u1 (a,b,c);
  moduleB u2 (d,e,f,g);
  moduleC u3 (siga,sigb,sigc);
  ....
// save all signal data in module u1 from time 100 to 300,
// save all the variables in module u2 along with 5 levels
// of hierarchy from time 200 to 500, save two variables
// in module u3 starting at 600
  fork
    #100 $vcdpluson(source.u1);
    #200 $vcdpluson(5,source.u2);
    #300 $vcdplusoff(source.u1);
    #500 $vcdplusoff(5,source.u2);
    #600 $vcdpluson(source.u3.siga,source.u3.sigb);
  join
  ....
```

    - stop and resume recording anytime during simulation
  - Start by dumping only the first few levels and work down until the problem is isolated

# VCD+ Tips and Suggestions (cont)



- **Avoid recording Verilog statements execution**
- **Selectively dump only small modules**
- **Use +vcdbufsize+nn  to control memory buffer size**
  - rule-of-thumb 1M for every 5K gates
  - bigger the buffer the faster simulation runs

# VCD+ Tips and Suggestions (cont) 4-15

- **Use compiler directives *`ifdef* and *`endif***

```
`ifdef dumpme
  $vcdpluson();
`endif
```

- **Dumping controlled by compile time switch +define+dumpme**
- **It is not recommended to use $test$plusargs**

Example:

```
initial begin:enable_dumping
  if ($test$plusargs("dumpall")) $vcdpluson();
  else if ($test$plusargs("dump+moduleA"))
    $vcdpluson(1,moduleA);
end
```

- Even if dumping is disabled at run-time, the fact that $vcdpluson is enable at compile time means slower simulation

# Agenda: Day 2



DAY 2

| Unit | Topic | Lab |
|---|---|---|
| 5 | Debugging Simulation Mismatches |  |
| 6 | Fast RTL Level Verification |  |
| 7 | Fast Gate Level Verification |  |
| 8 | Code Coverage |  |

# Unit Objectives



**After completing this unit, you should be able to:**

- **Use +race utility to locate race condition code**
- **Use $vcdplusdeltacycleon to locate race condition code**
- **Use vcdiff & vcat to locate race condition code**

# Causes of Simulation Mismatches



- **Functional simulation mismatches:**
  - Different simulator vendors
    - Race condition in source code
    - Vendor implementation
  - Different version of simulator from same vendor
    - Race condition in source code
- **RTL-Gate mismatches:**
  - Same simulator
    - Race condition in source code
    - Poor coding style

# Race Conditions



- The most common causes of simulation mismatches are *race conditions*
- A race condition is a coding style for which there could be several correct results; i.e. the code is ambiguous
- All race conditions are some variation of using (read) or setting (write) a data value at the same time it is changing
- Race conditions may result in logic that behaves unexpectedly, and should be fixed before synthesis

## Organization of events in Verilog simulation time step:

- **Current time step:**
  - Slot 1:
    - Evaluate right-hand side of non-blocking assignments
    - Evaluate right-hand side of & change left-hand side of blocking assignments
    - Evaluate right-hand side of & change left-hand side of continuous assignments
    - Evaluate inputs and change outputs of primitives
    - Print output from $display and $write
    - Call PLI calltf routines for system tasks and system functions
    - Note: All actions in Slot 1 are intermixed in any order!!!
  - Slot 2:
    - Call misctf routines which were scheduled using tf_synchronize()
  - Slot 3:
    - Change left-hand side of non-blocking assignments evaluated in Slot 1
  - Slot 4:
    - print output from $monitor and $strobe
    - Call misctf routines which were scheduled using tf_rosynchronize()
- **Next time step**

# Race Conditions: Read-Write



- Race: Using and setting a value at the same time

```
module race;
    reg a;
    initial begin
        a = 0;
        #10 a = 1;
    end

    initial begin
        #10 if (a) $display("May not print");
    end
endmodule
```

*There is no guaranteed ordering of the two initial blocks, so the `$display` may never execute*

# Race Conditions: Read-Write



- Race: Using and setting a value at the same time

```
module race;
    reg a;
    initial begin
        a = 0;
        #10 a = 1;
    end

    initial begin
        #11 if (a) $display("Will print");
    end
endmodule
```

4 Solution: Delay the `if` statement to another time-step

# Race Conditions: Write-Write



- **Race: Setting a signal with different values at the same time**

```
module race;
    reg a;
    initial #10 a = 0;
    initial #10 a = 1;
    initial
       #20 if (a) $display("May not print");
endmodule
```

***A race occurs at time 10 because there is no guaranteed ordering between the two initial blocks***

# Race Conditions: Write-Write



- Race: Setting a signal with different values at the same time

```
module race;
    reg a;
    initial #10 a = 0;
    initial #11 a = 1;
    initial
       #20 if (a) $display("Will print");
endmodule
```

4 Solution: Stagger the assignments to reg a by adding delay

# Other Types of Races



- **Continuous Assignment Evaluation**

```
assign p = q;
always @(posedge clk)
begin
   q = 0;
   if (p) $display("May not display");
end
```

***Continuous assignments with zero delays may propagate earlier than in V-XL, and hence $display may not print***

# Other Types of Races (cont)



- Continuous Assignment Evaluation

```
assign p = q;
always @(posedge clk)
begin
   q <= 0;
   if (p) $display("Will display");
end
```

- Solution: Use a non-blocking assignment to q.
  p will be updated in the next time-step

# Other Types of Races (cont)



- **Time Zero Races**

```
initial begin
   reset = 0;
   clock = 0;
   forever #50 clock=~clock;
end
always @(negedge reset)
    $display("May or may not display at time
zero");
```

***Transition of reset to 0 may happen before or after event trigger (always @(negedge reset))***

# Other Types of Races (cont)



- Time Zero Races

```
initial begin
  reset = 1;
  #10 reset = 0;
  clock = 0;
  forever #50 clock=~clock;
end
always @(negedge reset)
  $display("Will not display at time zero");
```

- Solution 1: Delay the negedge to after time 0.
- Solution 2: Initialize reset to 1 to ensure no x→0 transition.

# Time Zero Mismatches



- **The *+alwaystrigger* (5.1 and later, *+vcs+arm* in 5.0) compile-time switch resolves some time-zero races**
- **Ensures that always blocks with an initialized signal in its event control list are triggered at time zero**
- ***+alwaystrigger became default compile option after vcs 5.2***

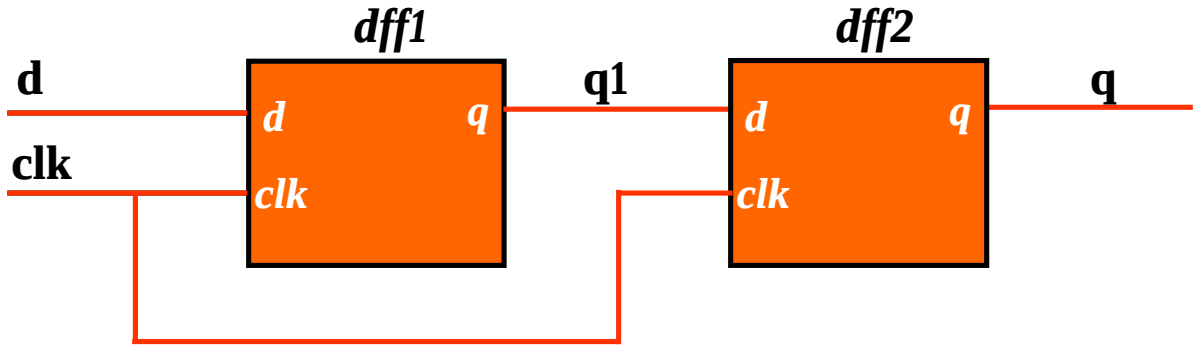
```
Module top;
   reg rst;
   wire val;
   bot b1(rst,val);
   initial rst=1'b1;
endmodule
module bot(rst, val);
   input rst;
   output val;
   reg val;
   always @(rst);
      val=1'b1;
endmodule
```

Without *+alwaystrigger*, the **`initial`** block races with the **`always`** block. Output signal **`bot.val`** may be x or 1 at time zero.

With *+alwaystrigger*, the **`always`** block is triggered at time zero, because **`rst`** is initialized. **`Bot.val`** == 1

# Flip-Flop Race: Read-Write

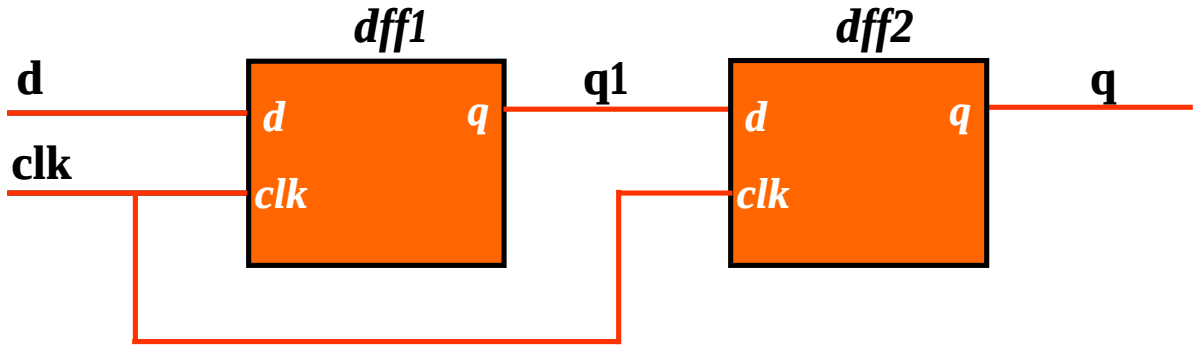





```
module dff(q, d, clk);
output q;
reg q;
input d, clk;
always @ (posedge clk)
  q1 = d;
always @ (posedge clk)
    q = q1;

endmodule
```

# Flip-Flop Race: Read-Write (cont)






```
module dff(q, d, clk);
output q;
reg q;
input d, clk;
always @ (posedge clk)
  q1 <= d;
always @ (posedge clk)
  q <= q1;
endmodule
```

## 4 Solution: Use Non Blocking Assignment

# Coding Rules of Thumb



- Synchronous blocks drive only with Non-Blocking Assignments
- Combinatorial and initial blocks drive only with Blocking Assignments
- Don’t drive regs from multiple blocks
- Be careful with the interaction of continuous assignments and procedural blocks

# Debugging Race Conditions



- **Use Delta Cycle Display feature in DVE to view event ordering in waveforms**
- **Use VCS provides race condition checker:**
  - Enabled with compile switch `+race`
  - Produces report file `race.out` that shows most races
- **Back-track from visible mismatch to the origin:**
  - Use `$dumpvars` to dump VCD files
  - Use `vcdiff` to show differences in simulation
  - Use `vcat` to display VCD file values in readable format

# VCS Race Tool +race



- **A dynamic tool sitting on top of VCS event-scheduler and “observes” events on variables**
- **Enabled with:**
  - `+race` for entire design
  - `+racecd` for part of design enclosed in `` `race `` and `` `endrace ``
- **Outputs `race.out` file with reports on races exposed by simulation run**

# Race output example



```
// File test.v
module test;
reg a;
initial a=1; // write at time 0
initial $display(a); // read at time 0
endmodule
```

```
% vcs -R +race test.v
...
...
0 "a": read test (test.v: 5) && write test (test.v: 4)
```

# Post-Process race.out File



- **Use post-processing perl scripts to prune the verbose `race.out` output:**
  - PostRace.pl
  - In $VCS_HOME/bin directory
- **Options in the PostRace.pl:**
  - -hier <hierarchy-name> (ex:-hier top)
  - -sig <sig-name> (ex: -sig databus1)
  - -minmax <min> <max> (ex: -minmax 12 16)
  - -nozero (ex: -nozero)
  - -uniq (ex: -uniq)
- **Modifying the PostRace.pl Script:**
  - The first line of the PostRace.pl Perl script is as follows:
    - #! /usr/local/bin/perl

# Back Tracking Mismatches



- **Some races will not be found by `+race`**
- **Use `$dumpvars` to generate VCD dump files from the two simulator runs:**
  - VCD file shows all signal changes in the simulation
  - VCS comes with two utilities to examine a VCD file.
  - Located in VCS install dir: $VCS_HOME/<arch>/util/
- **Use `vcat` to filter the contents of a VCD file**
- **Use `vcdiff` to compare two VCD files**
- **Add `$display` or `$monitor` can provide additional help to show important signals**

# vcat Usage



**vcat dump1.vcd -scope top.dut.moda**

```
dump1.vcd: scopes:8 signals:496 value-
changes:23582

--- top.dut.moda.A ---
  0 0
  10 1
  100 0
  130 1
  160 0
  240 1
  440 1
--- top.dut.moda.enable ---
  0 0
  240 1
  600 0
--- top.cla_0. top.dut.moda.write_bus ---
  0 001
```

**vcat broken.vcd -scope top.dut.moda -raw**

```
--- 0 --- top.dut.moda.A --- 0 ---
--- 0 --- top.dut.moda.enable --- 0 ---
--- 0 --- top.dut.moda .B --- 0 ---
--- 0 --- top.dut.moda.sel --- 1 ---
--- 0 --- top.dut.moda.H --- 10100 ---
--- 0 --- top.dut.moda.L --- 0 ---
--- 0 --- top.dut.moda.write_bus --- 001 ---
--- 0 --- top.dut.moda.z --- 1 ---
--- 0 --- top.dut.moda.data_bus --- 010000000 ---
--- 10 --- top.dut.moda.A --- 1 ---
--- 10 --- top.dut.moda.sel --- 1 ---
--- 10 --- top.dut.moda .B --- 1 ---
--- 10 --- top.dut.moda.H --- 11100 ---
--- 20 --- top.dut.moda.btmp --- 0 ---
--- 20 --- top.dut.moda.ctmp --- 0 ---
--- 20 --- top.dut.moda.write_bus --- 101 ---
--- 20 --- top.dut.moda.z --- 0 ---
--- 30 --- top.dut.moda.data_bus --- 011110000 ---
```

# vcdiff Usage



**vcdiff dump1.vcd dump2.vcd**

```
< dump1.vcd: scopes:58 signals:1294
> dump2.vcd: scopes:58 signals:1294

--- top.dut.modb.a --- 112360 ---
<   112360  0100100000000000001000
---
>   33120    0001000000000000000000

--- top.dut.modb.dbus --- 112360 ---
<   112360  010010000
---
>   33120    000100000
```

Time that a difference occurred

Previously driven at this time

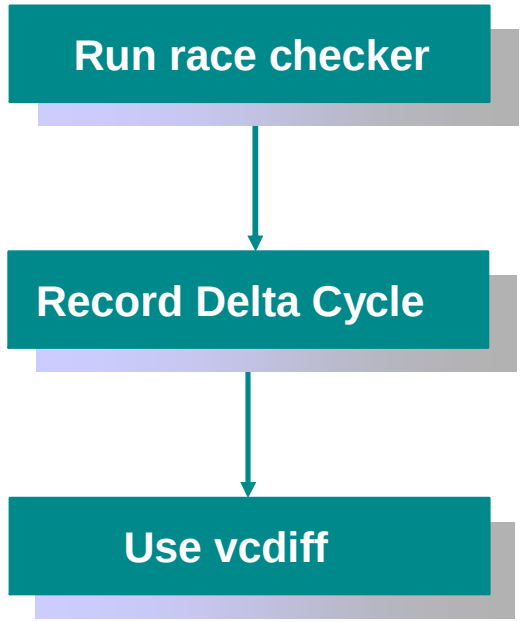
Current value from dump1

# Lab 5 Introduction



30 min

## Locating Verilog Race Conditions in Verilog code



# Revisiting the Unit Objectives



**Having completed this unit, can you:**

- **Use +race utility to locate race condition code**
- **Use $vcdplusdeltacycleon to locate race condition code**
- **Use vcdiff & vcat to locate race condition code**

# Agenda: Day 2



DAY 2

| Unit | Topic | Lab |
|---|---|---|
| 5 | Debugging Simulation Mismatches |  |
| 6 | Fast RTL Level Verification |  |
| 7 | Fast Gate Level Verification |  |
| 8 | Code Coverage |  |

# Unit Objectives



**After completing this unit you should be able to:**

- **Improve RTL simulation performance with good coding styles**
- **Improve RTL simulation performance by using the +rad compile time switch**

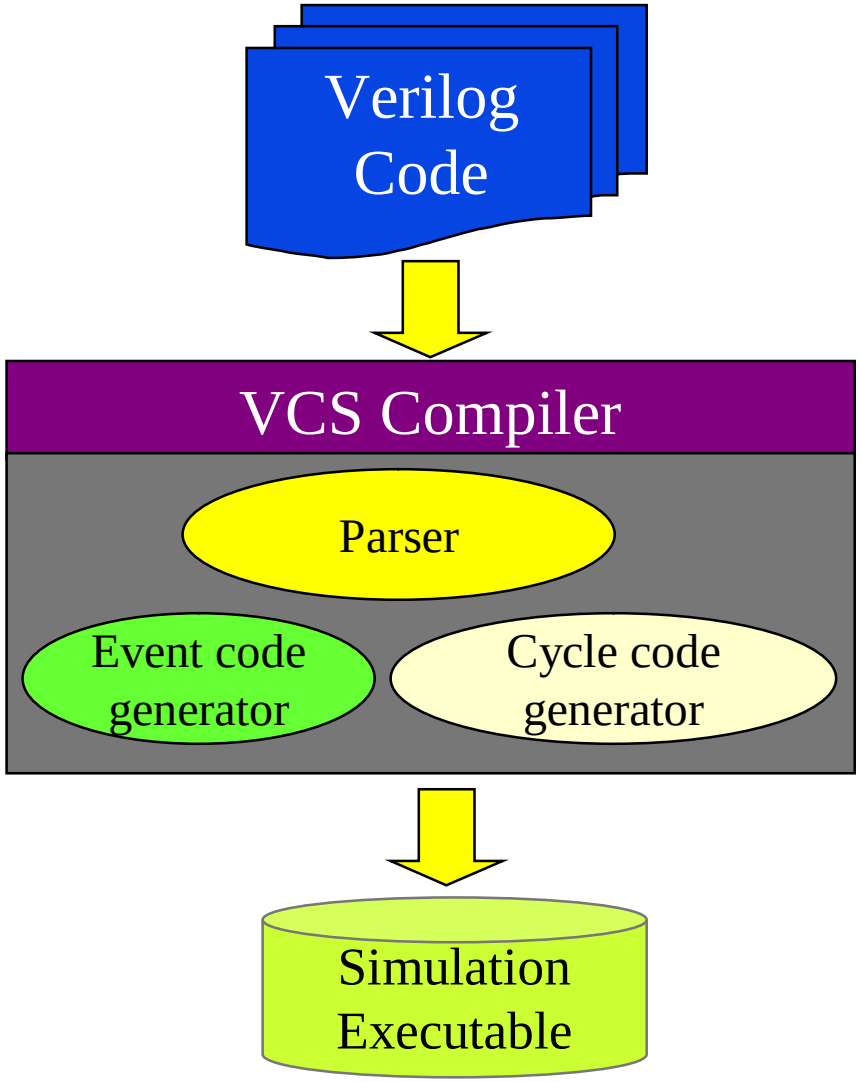
# Simulation Performance Depends On



- Good coding practices
- Good use of tool optimization features
- Good control in use of debugging switches
- Good control of need for re-compile

# VCS Architecture






- **Three major components in VCS to improve performance:**
- **Parser**
  - Parser accelerate-able code to code generators
- **Event code generator**
  - Accelerate random logic simulation
- **Cycle code generator**
  - Accelerate sequential block simulation
- **Performance starts at the parser**

# General Coding Guidelines



- **Use synthesizable subset of Verilog language**
  - Give VCS better chance of performing code optimization
- **Raise your level of abstraction**
  - Give simulator less work to do
- **Avoid inefficient constructs**
  - Switch level primitives (trans) and bidirectional
  - Strength modeling
- **Use small stimulus blocks**
  - Avoid large initial blocks (<10,000 lines of code)
  - Use file based stimulus (e.g. $readmemh)

# Things to Avoid for Sequential Logic



- **Avoid these constructs in sequential logic:**
  - repeat
  - wait
  - fork - join
  - assign - deassign
  - force - release
  - disable
  - case

# General Things to Avoid



- **Gate-level constructs:**
  - nmos, pmos, cmos, rnmos, rpmos, rcmos, pullup, pulldown, tranif0, tranif1, rtran, rtranif0 and rtranif1
- **Unaccelerated data types:**
  - time, realtime, real, named event, trireg net and integer array
- **Cross module referencing:**
  - Cross module reference is not optimized
  - Writing hierarchical XMRs is not a good idea

  e.g. **top.x = y**

# Use VCS Optimization Features



- **+rad optimization:**
  - Compile time switch
  - Attempts to optimize design by:
    - Raising the level of abstraction
    - Parsing code for fast event and cycle-based simulation
  - Often referred to as Radiant Technology

# +rad Optimization



- **Performs semantically-preserved optimization for both RTL and Gate level simulation**
- **Optimizes complex logic to simpler form through logic expression abstraction:**

```
assign x[0] = (a==0);
assign x[1] = (a==1);
assign x[2] = (a==2);
```

```
assign x = (1 << a);
```

- **Performs global optimizations across hierarchy**
- **Optimized results more “event efficient” leading to faster simulations**
- **May change hierarchy and signal of a design**

# Another +rad Example: Vectorization 6-10

- **Input Verilog**

```
module up;
…..
Dff d0(in[0],out[0],clk);
Dff d1(in[1],out[1],clk);
..
Dff dn(in[n],out[n],clk);
…..
endmodule

module Dff(in,out,clk);
input in, clk;
output out;
reg out;
always @(posedge clk)
    out <= in;
endmodule
```

- **Optimized Verilog**

```
module up;
…..
Dff_veci_vec i1(in,out,clk)
…..
endmodule

module Dff_veci_vec(in,out,clk);
input clk;
input [n:0] in;
output [n:0] out;
reg [n:0] out;

always @(posedge clk)
    out <= in;
endmodule
```

# Using +rad Optimization



- **Ideal designs for +rad optimization:**
  - No debug capabilities enabled
  - No timing checks or sdf back-annotation
  - Simulation times dominate the compile times
- **Compile with +rad compile-time switch:**

```
>vcs source.v +rad +optconfigfile+filename
```

  - `+optconfigfile+`*`filename`* (optional)
    - **Localize scope of +rad optimizations**

# Configuration File Line Format



```
module|instance|tree [(depth)]{identifier} {attribute};
```

- `module`
  - Apply attribute to all instances of module named “identifier”
- `instance`
  - Apply attribute to all instances of module named “identifier”
  - Apply attribute to all module instances in path specified by “identifier”
  - Apply attribute to individual signal specified by “identifier”
- `tree`
  - Apply attribute to all instances of module named “identifier”
- `depth`
  - Specifies number of lower level hierarchy to apply attribute
- `attributes` = noOpt, noPortOpt, RadLight, Opt, PortOpt

# Minimize Compile Time



- **Use Local Disk:**
  - Avoid over-the-network disk space for code generation
- **Minimize debug flags:**
  - +acc Use PLI Table file with minimum ACC enabled
  - -debug Use only if doing interactive debugging
  - -I Use only if doing debugging
  - -debug_all Use only if doing line tracing during debug

# VCS +prof: A Verilog Profiler



- **Determine simulation bottleneck**
  - Key to improving simulation performance
- **Use VCS +prof utility**
  - Breaks simulation CPU time and memory consumption down by percentage for each module and Verilog construct

# Using the Profiler



- **Compile and simulation design with `+prof`**

```
>vcs -f my_design.f -R +prof
```

- **VCS generates vcs.prof file with the following view on CPU time & memory based simulation profile report:**
- **For CPU time**
  - Top level view
  - Module view
  - Instance view
  - Module to Construct mapping view
  - Top level construct view
  - Construct view across design
- **For memory**
  - Top Level View
  - Module View

# Top Level View



- **Display CPU time used by:**
  - PLI applications that executed along with VCS
  - VCS for writing VCD and VCD+ files
  - VCS for internal operation overhead
  - The constructs and statements in your design

```
================================================================================
                                TOP LEVEL VIEW
================================================================================
                          TYPE          Time         %Totaltime
--------------------------------------------------------------------------------
                           DPI          1024           0.06
                           PLI         39309           2.48
                           VCD             0           0.00
                        KERNEL       1544664          97.35
                       MODULES             0           0.00
                      PROGRAMS          1728           0.11
--------------------------------------------------------------------------------
```

# Module View



- Display modules (all instances) using most CPU time

```
================================================================================
                                MODULE VIEW
================================================================================
Module(index)                %Totaltime    No of Instances   Definition
--------------------------------------------------------------------------------
fifo32X8tb             (1)     20.30          1             fifo32X8tb.v:7.
fifo_mem               (2)     12.90          2             fifo_mem.v:1.
fifo_cntrl             (3)     10.57          1             fifo_cntrl.v:1.
ram16X8                (4)      7.82          2             ram16X8.v:1.
fifo32X8               (5)      5.92          1             fifo32X8.v:1.
--------------------------------------------------------------------------------
```

# Instance View



- **Displays individual module instances using most CPU time**

```
================================================================================
                                INSTANCE VIEW
================================================================================
Instance                                                      %Totaltime
--------------------------------------------------------------------------------
fifo32X8tb                                          (1)       20.30

fifo32X8tb.fifo.mem_even                            (2)       10.57

fifo32X8tb.fifo.cntrl                               (3)       10.57

fifo32X8tb.fifo                                     (5)       5.92

fifo32X8tb.fifo.mem_odd.ram                         (4)       4.44

fifo32X8tb.fifo.mem_even.ram                        (4)       3.38

fifo32X8tb.fifo.mem_odd                             (2)       2.33

--------------------------------------------------------------------------------
```

# Top Level Construct View



- **Display memory for each type of construct in design**

```
================================================================================
                          TOP-LEVEL CONSTRUCT VIEW
--------------------------------------------------------------------------------
                  Verilog Construct          %Totaltime
--------------------------------------------------------------------------------
                       Combinational            23.89
                              Always            13.32
                                Task            13.11
                             Initial             7.19
                        Timing Check             0.00
                            Function             0.00
                         Module Path             0.00
                                Port             0.00
                                 Udp             0.00
                           Protected             0.00
________________________________________________________________________________
```

# Module to Construct Mapping View



- Displays CPU time used by Verilog construct

```
=============================================================================
                        MODULE TO CONSTRUCT MAPPING
=============================================================================

_____________________________________________________________________________
                                1. fifo32X8tb
-----------------------------------------------------------------------------
Construct type    %Totaltime    %Moduletime     LineNo
-----------------------------------------------------------------------------
Initial              7.19          35.42        fifo32X8tb.v : 29-36.
Task                 6.55          32.29        fifo32X8tb.v : 83-114.
Task                 4.02          19.79        fifo32X8tb.v : 50-76.
_____________________________________________________________________________

_____________________________________________________________________________
                                2. fifo_mem
-----------------------------------------------------------------------------
Construct type    %Totaltime    %Moduletime     LineNo
-----------------------------------------------------------------------------
Combinational        7.61          59.02        fifo_mem.v : 21, 23, 25.
Always               4.86          37.70        fifo_mem.v : 13-15.
Always               0.42          3.28         fifo_mem.v : 17-19.
_____________________________________________________________________________
```

# Top Level View



- This view shows you how much memory was used by:
  - Any PLI or DPI application that executes along with VCS
  - VCS for writing VCD and VPD files
  - VCS for internal operations (known as the kernel) that can't be attributed to any part of your design.
  - The Verilog modules in your design
  - A SystemVerilog testbench program block, if used

```
================================================================================
//        Simulation memory:      729588 bytes

================================================================================
                                  TOP LEVEL VIEW
================================================================================
                             TYPE          Memory    %Totalmemory
--------------------------------------------------------------------------------
                              DPI               0            0.00
                              PLI            4721            0.65
                              VCD               0            0.00
                           KERNEL          716395           98.19
                          MODULES            8472            1.16
                         PROGRAMS               0            0.00
--------------------------------------------------------------------------------
```

# Module View



- The module view shows the amount of memory used, and the percentage of memory used, by each module definition.

```
================================================================================
                                MODULE VIEW
================================================================================
Module(index)               Memory    %Totalmemory No of Instances   Definition
--------------------------------------------------------------------------------
codectb               (1)      8472            1.16          1               codectb.v:7.
--------------------------------------------------------------------------------
```

# Profiler Tips



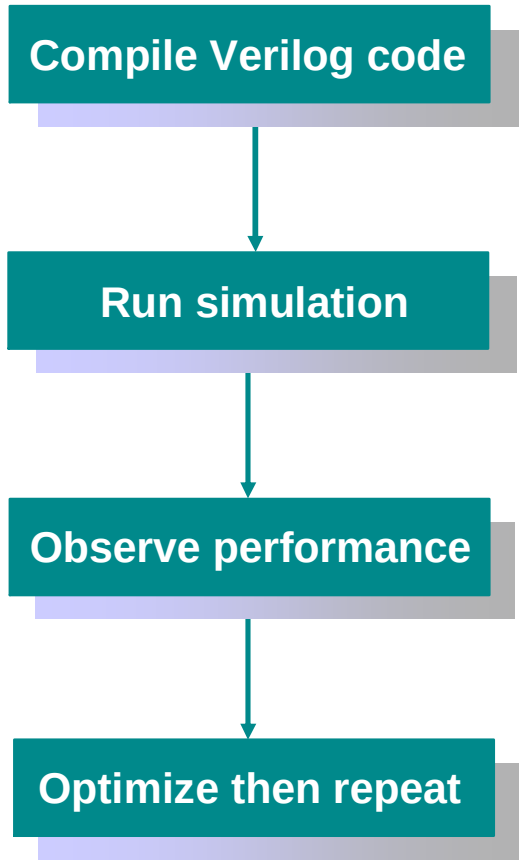
- **Use VCS profiler on a regular basis to catch potential simulation bottlenecks**
- **Resolving simulation bottlenecks:**
  - Provides better simulation performance
  - May expose real design issues

# Lab 6 Introduction



60 min

**Improve simulation performance for an existing Verilog design**



# Agenda: Day 2



DAY 2

| Unit | Topic | Lab |
|---|---|---|
| 5 | Debugging Simulation Mismatches |  |
| 6 | Fast RTL Level Verification |  |
| 7 | Fast Gate Level Verification |  |
| 8 | Code Coverage |  |

# Unit Objectives

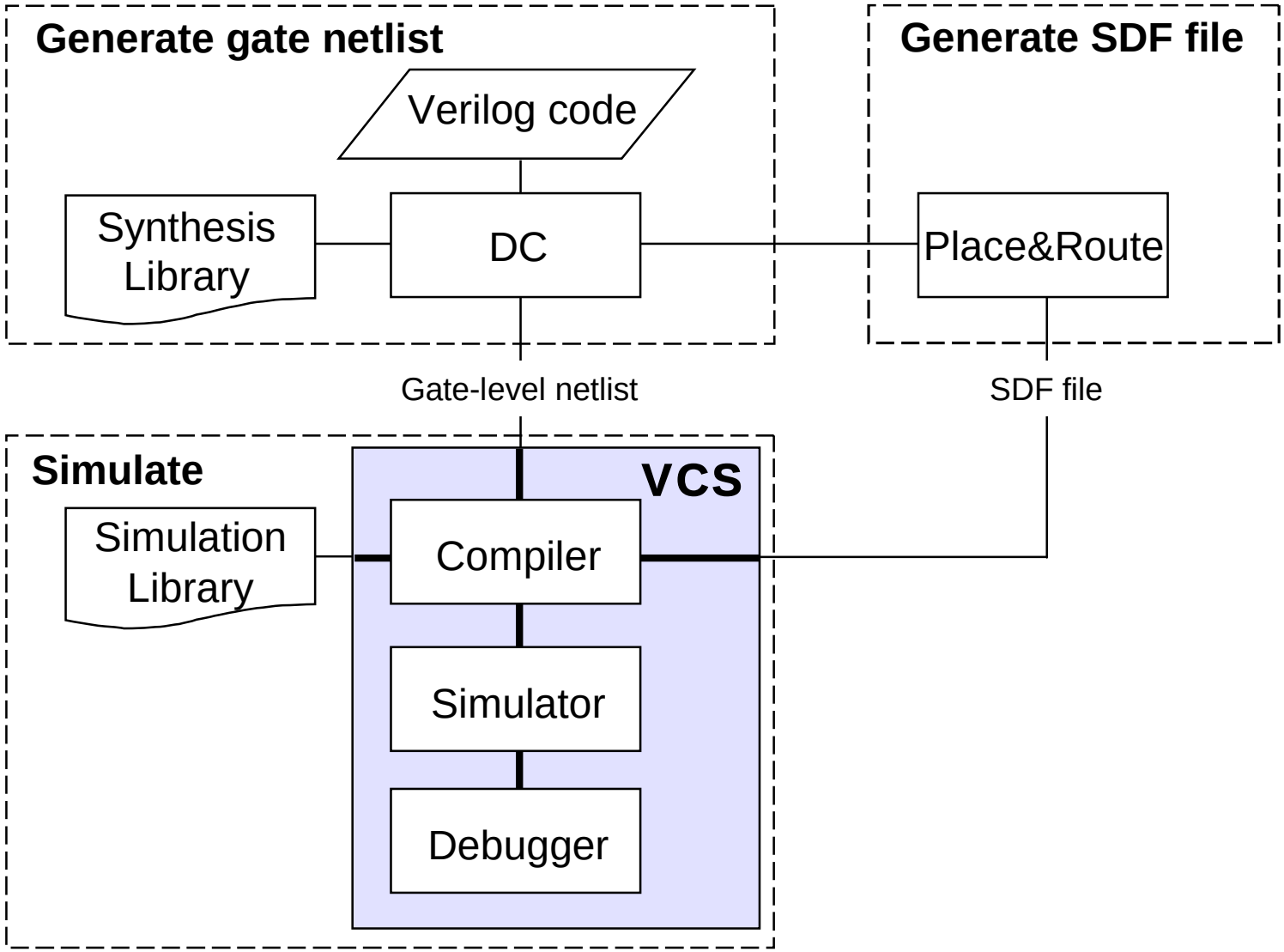


**After completing this unit you should be able to:**

- **Verify the Verilog Gate-Level netlist matches the RTL-Level simulation using VCS**
- **Demonstrate the same Verilog Gate-Level netlist simulates faster with the +rad**
- **Compile, back-annotate SDF and verify functionality of an existing Verilog design**

# Gate-Level Validation Flow





# Fast Gate-Level Functional Validation 7-4

```
> vcs -f gate.f +rad +nospecify +notimingcheck +nocelldefinepli+2
```

- **Use +rad**
- **+nospecify**
  - Ignores specify blocks (allows +rad to work, since +rad does not optimize modules with specify blocks)
- **+notimingcheck**
  - Disables timing check system tasks
- **+nocelldefinepli+[1|2]:**
  - +1 disable dumping of internal information of a library element defined by `celldefine compiler directive
  - +2 also disables dumping of information in library or directory specified by -v or -y compile-time switch

# Performance & Debug Considerations 7-5

- **Synthesis:**
  - Preserve design hierarchy
    - Flat designs simulate slower
- **Compilation**
  - Limit excessive use of debug switches
- **Debugging:**
  - Use post-processing debugging techniques
    - Dump VCD+ files
    - Limit amount of dump data
  - Use the race utilities to resolve race issues

# Gate-Level Verification with Timing

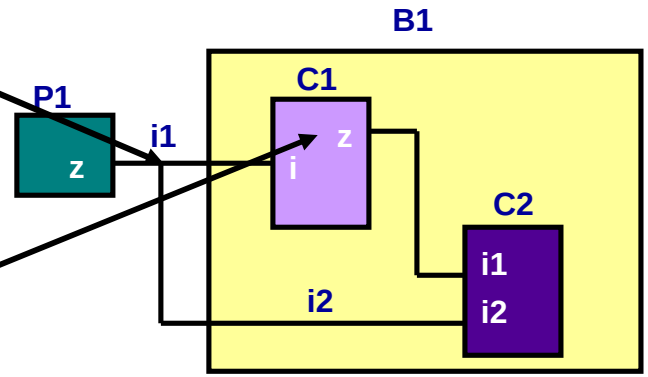


- **Gate-level timing simulation may be needed for:**
  - Asynchronous logic
  - ATPG vector verification
  - Initialization conditions
- **Timing information is embedded in SDF file:**
  - Delays (module path, device, interconnect, port)
  - Timing checks (setup, hold, setuphold, recovery, removal, recrem, skew, width, period, nochange)
  - Timing constraints (pathconstraint, skewconstraint, periodconstraint, sum, diff)
  - Timing environment (arrival, departure, slack, waveform)

# SDF - Example



```
(CELL
 (CELLTYPE "block")
 (DELAY
  (ABSOLUTE
    (INTERCONNECT P1/z B1/C1/i
   (.15:.15:.15)  (.12:.12:.12))
       …
)))
(CELL
  (CELLTYPE "INV")
  (INSTANCE B1/C1)
  (DELAY
    (ABSOLUTE
     (IOPATH i z
     (.34:.34:.34) (.32:.32:.32))
    …
)))
```



# SDF Back-Annotation in VCS



## Compiled back-annotation

- **Insert `$sdf_annotate` in Verilog code**

```
$sdf_annotate("sdf_file" [,module_instance]
[,"sdf_configfile"] [,"sdf_logfile"] [,"mtm_spec"]
[,"scale_factors"] [,"scale_type"]);
```

- **Compile:**

```
> vcs dut_gate.v -v sim_lib.v
```

  - Requires vendor supplied Verilog simulation library

- **SDF configuration file is not supported**

# Alternative Back-Annotation Method 7-9

## Run-time back-annotation

- **Insert `$sdf_annotate` in Verilog code**
- **Create a PLI .tab file**
  - Map `$sdf_annotate` to `sdf_annotate_call`
- **Compile**

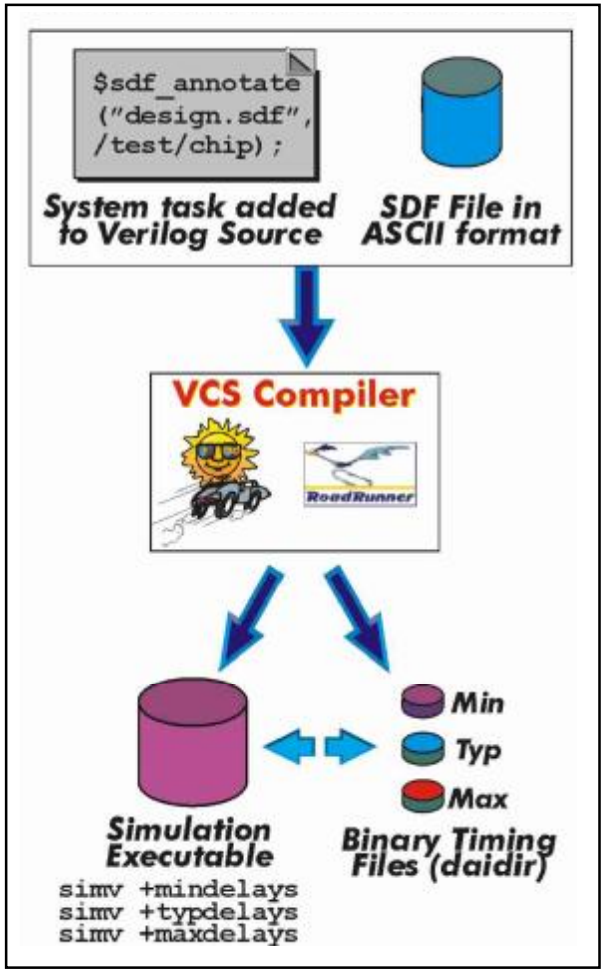
```
> vcs -R -P sdf.tab dut_gate.v -v sim_lib.v
```

- **Use only if the following is true**
  - Included `sdf_configfile` or `scale_type` in `$sdf_annotate` **task call**

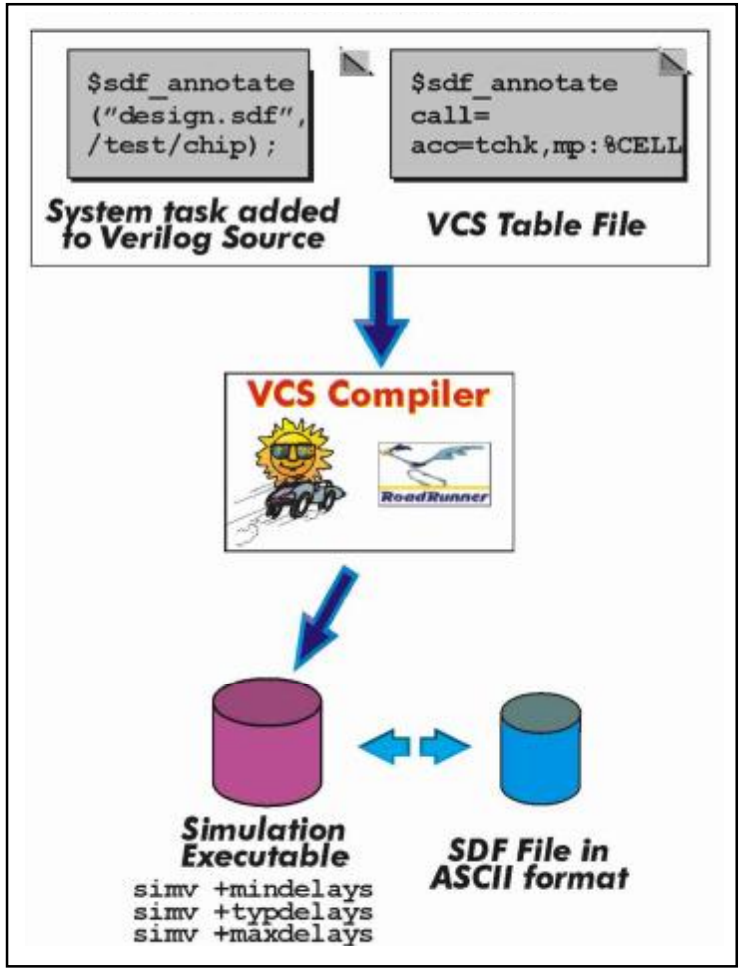
# Compiled SDF Versus Runtime SDF



## Compiled SDF Flow




## Run-time SDF Flow



# Handling Large SDF Files



- **VCS will either parse ASCII SDF file or a precompiled version of the ASCII SDF file**
- **Parsing of the precompiled SDF file is faster**
- **Creating precompiled version of ASCII SDF file:**
  - Use `+csdf+precompile` compile-time switch
  - VCS creates a precompiled version of the SDF file by appending “_c” to the ASCII SDF file’s extension

  Example:

  VCS creates `dut.sdf_c` from `dut.sdf` file
- **Once created, VCS will read the precompiled SDF file during compilation**
  - No compile-time switch is required

# Selecting min/typ/max Delays



- **Selecting min/typ/max for timing:**
  - Compile-time switch:
    - +mindelays
    - +typdelays
    - +maxdelays
  - Or, run-time switch:
    - Compile with +allmtm compile-time switch
    - Specify delay at run time with run-time switch:
      +mindelays
      +typdelays
      +maxdelays
    - Can NOT be used with compile-time switch

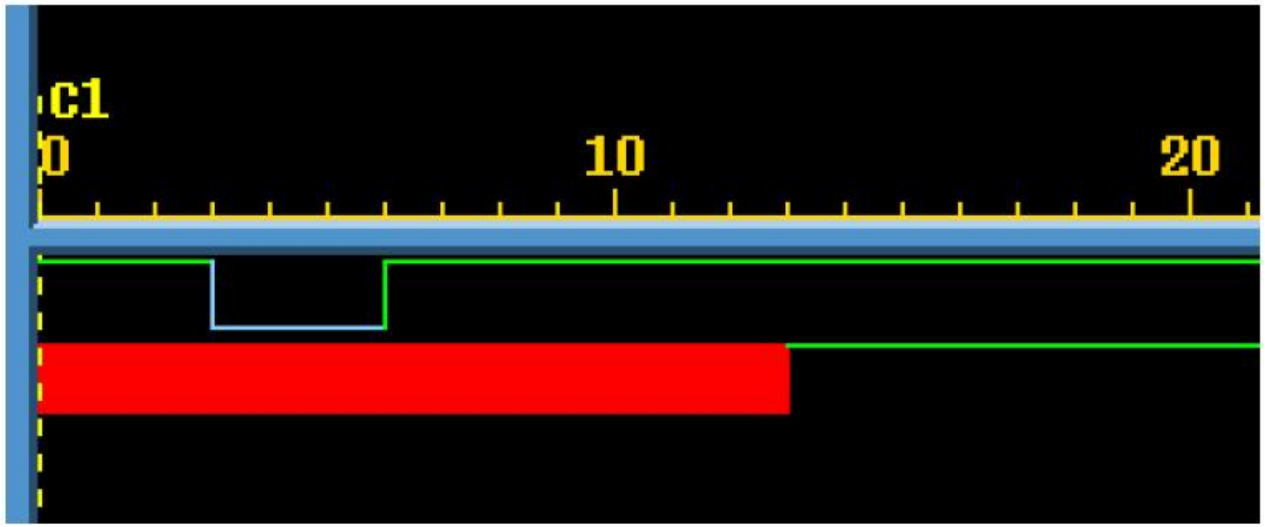
# Delay Filtering

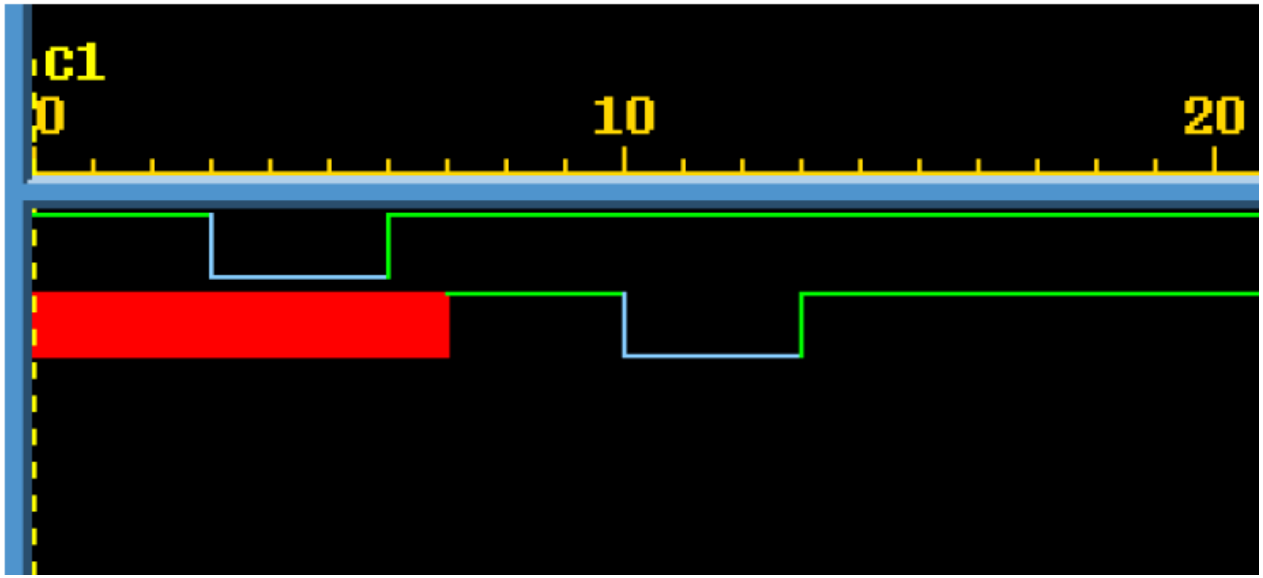


- **Two types of delay filtering: inertial or transport**
- **Inertial delay:**
  - Default VCS delay process
  - Pulses shorter than device delay are filtered out
- **Transport delay:**
  - All pulses are propagated through (no filter)

# Inertial delay VS Transport delay





# Enable transport delay



With compile-time switches:

- `+transport_path_delays`
  Turns on transport delay mode for path delays
- `+transport_int_delays`
  Turns on transport delay mode for interconnects

# Modifying Delay Filtering Criteria



- **For module path delays:**
  - `+pulse_e/`*`number`* (error limit in %)
  - `+pulse_r/`*`number`* (reject limit in %)

  Example: `+pulse_e/70` and `+pulse_r/50`
    - Rejects pulses less than 50% of the delay
    - Outputs X for pulses less than 70% but greater than 50% of the delay and display an error message
- **For INTERCONNECT delays:**
  - `+pulse_int_e/`*`number`* (error limit in %)
  - `+pulse_int_r/`*`number`* (reject limit in %)
  - Same usage as shown in above example

# Additional Features



- Support for Negative Timing Checks
- Support for Multisource Interconnect Delays
- Support for on-event and on-detect pulse filtering
- Support for Delay Mode Selection
- Refer to VCS User Guide for more information

# Performance & Debug Considerations 7-18

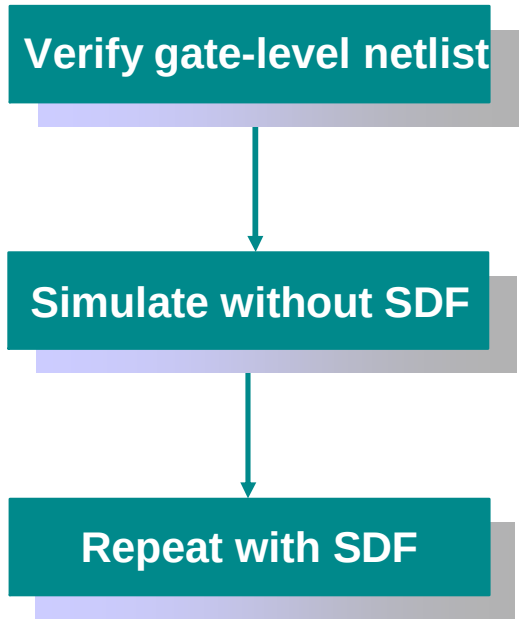
- **Use compiled SDF**
- **Useful guideline**
  - Limit excessive use of debug switches
  - Preserve design hierarchy
  - Use post-processing debugging techniques
  - Use the race utilities to resolve any race issues
- **Things to bear in mind**
  - +rad is disabled for entire design when compiled SDF methodology is used

# Lab 7 Introduction



45 min

**Run a Verilog gate-level simulation with and without timing**



# Agenda: Day 2



DAY 2

| Unit | Topic | Lab |
|---|---|---|
| 5 | Debugging Simulation Mismatches |  |
| 6 | Fast RTL Level Verification |  |
| 7 | Fast Gate Level Verification |  |
| 8 | Code Coverage |  |

# What is Code Coverage?



- **Code Coverage answers questions such as…**
  - Have all the lines of the RTL been stimulated?
  - Have all the states of a FSM been exercised?
  - Have all the conditions of an 'if' statement in the RTL simulated?
  - Have all the blocks of a 'case' statement been exercised?

# What Code Coverage is NOT!



- **Functional Coverage, which answers questions such as…**
  - Have all possible combinations of instructions been verified on a processor ?
  - Have all the 'corner-cases' been tested for a design ?
  - Did an asynchronous interrupt occur when a cache miss was being handled by the processor ?
- **Synopsys has other tools & methodologies to address Functional Coverage**

# Types of Coverage in VCS



- **Statement or line coverage**
  - Has the line been executed?
- **Toggle coverage**
  - What type of switching activity is there?
- **Conditional coverage**
  - Have various permutations of conditions been exercised?
- **FSM coverage**
  - Have I reached all possible states?
- **Path Coverage**
  - Did all paths in an initial or always block get executed?

# Defining Code Coverage



- **Traditional code coverage includes**
  - Line or Statement coverage
    - Least powerful but easiest to understand
    - Checks every assignment has been executed
    - Aim for 100% before using more powerful coverage types
  - Condition
    - Checks all combinations in complex branches

      If (a==1) or (b==1) or (c==1)
    - Various formats available to identify values of multiple conditions
    - Start with basic (sensitized) condition coverage before applying advance condition coverage

# Defining Code Coverage (Cont'd)



- Finite State Machine Coverage
  - Checks states and state transitions
  - Automatically identifies individual state machines, but not cross product states, communication between FSM's
- Toggle Coverage
  - Ensures every node has transitioned from 0->1 and 1->0
  - Used mostly for gate level code coverage
  - Also used for system level connectivity testing

# Line Coverage



- **Reports : which lines, statements, and blocks for any instance/module of design were exercised during simulation**
- **Verilog :**
  - procedural assignment statement
  - system task
  - case
  - while
  - if
  - for
  - continuous assignment statement
  - initial block
  - always block
  - missing else
- **Verilog: assignment statement - which assignment statement causes a bit of a signal to toggle 0->1, 1->0 ( Verilog only)**

# Line Coverage - Verilog Example



## Verilog source code

```
   always @(rst or enable)
 begin
  case (rst && enable)
   0: if (cond1) cas = 1'b0;
   1: if (cond2 || cond3) cas= 1'b1;
      else  $display ( " No
condition true") ;
  endcase
  if (rst && enable) cas1 = cas;
  else cas1 = 1'bz;
  if (rst)  cas2 = 1'b0;
 end
```

## Line coverage annotated data

```
17              always @(rst or enable)
18              begin
19        ==>    case (rst && enable)
20        ==>     0: if (cond1) cas = 1'b0;
20        ==>  if (cond1)
20.1      ==>      cas = 1'b0;
20.2      ==>  MISSING_ELSE
21        ==>  1:if(cond2 || cond3)cas =1'b1;
21        ==>  if (cond2 || cond3)
21.1      ==>      cas = 1'b1;
22        ==>        else  $display ( "No …);
22.1      ==>  MISSING_DEFAULT
23   endcase
```

**Annotated files help to understand how VCS extracts constructs for different metrics ( line, statement, block) and what constructs are covered by a given test**

# Line Coverage - Verilog Example (cont.) 8-9

## Line coverage report file details

| Line No | Coverage | Block Type |
|---|---|---|
| 8 | 1 | INITIAL |
| 19 | 0 | ALWAYS |
| 20 | 0 | CASEITEM |
| 20.1 | 0 | IF |
| 20.2 | 0 | MISSING_ELSE |
| 21 | 0 | CASEITEM |
| 21.1 | 0 | IF |
| 22 | 0 | ELSE |
| 22.1 | 0 | MISSING_DEFAULT |
| 24 | 0 | |
| 24.1 | 0 | IF |
| 25 | 0 | ELSE |
| 26 | 0 | |
| 26.1 | 0 | IF |
| 26.2 | 0 | MISSING_ELSE |

// Module Coverage Summary

| | TOTAL | COVERED | PERCENT |
|---|---|---|---|
| lines | 8 | 1 | 12.50 |
| statements | 12 | 1 | 8.33 |
| blocks | 10 | 1 | 10.00 |
| | | | |
| ALWAYS | 1 | 0 | 0.00 |
| CASEITEM | 2 | 0 | 0.00 |
| IF | 4 | 0 | 0.00 |
| ELSE | 2 | 0 | 0.00 |
| MISSING_ELSE | 2 | 0 | 0.00 |
| INITIAL | 1 | 1 | 100.00 |
| MISSING_DEFAULT | 1 | 0 | 0.00 |

# Condition Coverage



- **Monitors values taken on by Boolean and bitwise expressions**
  - Conditional expressions in conditional operator ( ?: )
  - if statement
  - Expressions in continuous assignment statement ( assign c = a && b; )

# Condition Coverage - Example 1



## Verilog source code

```
always @(posedge clk)
 if (((a[3] && a[2]) && a[1]) && a[0])
  begin
   #1 $display("&& if triggered");
  end
```

## Condition coverage report

```
LINE   39
STATEMENT   if ((a[3] && a[2] && a[1] && a[0]))
                 -1--    -2--    -3--    -4--
EXPRESSION     -1-   -2-   -3-   -4-
               0     1     1     1   | Not Covered
               1     0     1     1   | Not Covered
               1     1     0     1   | Not Covered
               1     1     1     0   | Not Covered
               1     1     1     1   | Not Covered
```

# Condition Coverage - Example 2



## Verilog source code

```
assign d = a1 || b | c;
```

## Condition coverage report

```
LINE   33
STATEMENT   d = (a1 || (b | c))
                 1-    ---2---
EXPRESSION     -1-   -2-
                0     0  | Covered
                0     1  | Not Covered
                1     0  | Not Covered

LINE   33
STATEMENT   d = (a1 || (b | c))
                        1   2
EXPRESSION     -1-   -2-
                0     0  | Covered
                0     1  | Covered
                1     0  | Not Covered
```

**cmView groups expressions to make considered sub-expressions smaller**

# Condition Coverage - Example 3



## Verilog source code

```
assign g = (( f == 1) | ( e == 0)) ?
           (a || b | c) : 0;
```

## Condition coverage report

```
LINE   35
STATEMENT   g = (((f == 1'b1) | (e == 1'b0)) ? ((a
     || (b | c))) : 0)
                -------------1-------------
EXPRESSION     -1-
                0  | Not Covered
                1  | Covered


LINE   35
STATEMENT   g = (((f == 1'b1) | (e == 1'b0)) ? ((a
     || (b | c))) : 0)
                 -----1-----   -----2-----
EXPRESSION     -1-   -2-
                0     0  | Not Covered
                0     1  | Covered
                1     0  | Not Covered
```

```
LINE   35
 STATEMENT   g = (((f == 1'b1) | (e == 1'b0)) ? ((a
      || (b | c))) : 0)
                   -----1-----
 EXPRESSION     -1-
                 0  | Covered
                 1  | Covered


 LINE   35
 STATEMENT   g = (((f == 1'b1) | (e == 1'b0)) ? ((a
      || (b | c))) : 0)
                                  -----1-----
 EXPRESSION     -1-
                 0  | Not Covered
                 1  | Covered
//------------------------------------------

//            Module Coverage Summary

//                      TOTAL        COVERED
      PERCENT
//    conditions        24           9
      37.50
//    logical           24           9
      37.50
```

# Toggle Coverage



- **Reports whether signals and signal bits had 0->1 and 1->0 transitions**
- **A signal is considered fully covered if and only if it toggled in both directions: 0->1 and 1->0**
  - x->1 and x->0 transitions are not counted

# Toggle Coverage – Verilog and VHDL



- **Verilog**
  - Registers
  - Wires
  - Memories (with the +memcbk compile-time option)
- **VHDL - ports and signals of types**
  - bit
  - bit_vector
  - std_logic
  - std_ulogic
  - std_logic_vector
  - std_ulogic_vector
  - signed
  - unsigned

# Toggle Coverage - Example



## Verilog source code

```
input clk, rst;
input [7:0] d;
reg [7:0] ff_out;


  always @( posedge clk or posedge
rst)
    if (rst) ff_out <= 0;
    else     ff_out <= d;
```

## Toggle coverage report ( no cm_count)

```
 //                                      Net Coverage

// Name                                Toggled   1->0    0->1
   clk                                 Yes
   rst                                 No        No      No
   d[2:0]                              Yes
   d[3]                                No        No      Yes
   d[7:4]                              No        No      No

//                                   Register Coverage

// Name                                Toggled   1->0    0->1
   ff_out[2:0]                         Yes
   ff_out[3]                           No        No      Yes
   ff_out[7:4]                         No        No      No
```

# Toggle Coverage - Example (cont.)



## Verilog source code

```
input clk, rst;
input [7:0] d;
reg [7:0] ff_out;

always @( posedge clk or posedge rst)
    if (rst) ff_out <= 0;
    else     ff_out <= d;
```

## Toggle coverage report ( with cm_count )

```
 //                                  Net Coverage
// Name                          Toggled   1->0    0->1    ToggleCount
   clk                           Yes       Yes     Yes     8
   rst                           No        No      No      -
   d[0]                          Yes       Yes     Yes     4
   d[1]                          Yes       Yes     Yes     2
   d[2]                          Yes       Yes     Yes     1
   d[3]                          No        No      Yes     -
   d[4]                          No        No      No      -
   d[5]                          No        No      No      -
   d[6]                          No        No      No      -
   d[7]                          No        No      No      -

//                               Register Coverage
// Name                          Toggled   1->0    0->1    ToggleCount
   ff_out[0]                     Yes       Yes     Yes     3
   ff_out[1]                     Yes       Yes     Yes     2
   ff_out[2]                     Yes       Yes     Yes     1
   ff_out[3]                     No        No      Yes     -
   ff_out[4]                     No        No      No      -
   ff_out[5]                     No        No      No      -
   ff_out[6]                     No        No      No      -
   ff_out[7]                     No        No      No      -
```

# Finite State Machine Coverage



- Recognizes some portion of sequential logic as an FSM and reports which FSM states and which state transitions (among all possible) were executed
- FSM coverage can tell which parts of the design are implemented as FSMs and gives specific information, which other kinds of coverage do not provide, on all possible sequences of state transitions

# FSM Coverage - Example 1



## Verilog source code

```
 parameter idle = 2'b00,
           first = 2'b01,
           second = 2'b10,
           third = 2'b11;

  always @ (posedge clk or posedge rst)
  if (rst) state= idle;
  else     state=next;

 always @(in )
 begin
    next = state; // by default hold case
(state)
    case (state)
     idle :   if (in) next = first;
     first :  if (in) next = second;
     second : if (in) next = third;
     third :  if (in) next = idle;
     default: next = idle;
    endcase
 end
```

## FSM coverage report ( no cm_count )

```
  FSM        state
// state coverage results
             idle                              | Covered
             first                             | Covered
             second                            | Covered
             third                             | Covered
// state transition coverage results
             idle->first                       | Covered
             first->idle                       | Not Covered
         . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
// sequence coverage results
             idle->first                       | Covered
             first->idle                       | Not Covered
             idle->first->second               | Covered
             first->second->idle               | Not Covered
             first->second->third              | Covered
             third->idle->first->second        | Covered
             idle->first->idle                 | Not Covered Loop
             idle->first->second->third->idle  | Covered Loop


         . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
```

# FSM Coverage - Example 1 (cont.)



## Verilog source code

```
parameter idle = 2'b00,
          first = 2'b01,
          second = 2'b10,
          third = 2'b11;


  always @ (posedge clk or posedge
rst)
  if (rst) state= idle;
  else     state=next;


 always @(in )
 begin
    next = state; // by default hold
case (state)
    case (state)
     idle :   if (in) next = first;
     first :  if (in) next = second;
     second : if (in) next = third;
     third :  if (in) next = idle;
     default: next = idle;
    endcase
 end
```

## FSM coverage report ( with cm_count )

```
  FSM       state
// state coverage results
            idle                          | 2
            first                         | 2
            second                        | 2
            third                         | 2
// state transition coverage results
            idle->first                   | 2
            first->idle                   | 0
            first->second                 | 2
            second->idle                  | 0
            second->third                 | 2
            third->idle                   | 1
// sequence coverage results
            idle->first                   | Covered
            first->second->third->idle    | Covered
            second->third->idle->first    | Covered
            third->idle->first->second    | Covered
            idle->first->idle             | Not Covered Loop
//------------------------------------------------------------------
//             Single FSM Coverage Summmary
                  TOTAL       COVERED        PERCENT
States            4           4              100.00
Transitions       6           4              66.67
Sequences         25          16             64.00
```

# Path Coverage



- **Tracks which paths in an initial or always block were executed**
  - Checks consecutive branches through the RTL
  - Branching statements can be “if” or “case” statements

```
module dev (out,clk,c1,c2,c3,c4,i1,i2);
input clk,c1,c2,c3,c4,i1,i2;
output out;
reg out,c,d,b;
always @(posedge clk)
begin
    out = 1'b0;
    if (c1)
        begin
            out = i1 && i1;
            if (c2)
                b = i1 || i2;
        end
    if (c3)
        c = ~i1;
    else
        c = ~i2;
    case (c4)
        1'b0 : d = 1'b0;
        1'b1 : d = 1'b1;
     endcase
end
endmodule
```

# How Can Coverage Be Used?



- **Behavioral code**
  - Line
  - Condition
  - Path
  - FSM
- **RTL code**
  - Line
  - Condition
  - Path
  - Toggle (not recommended)
  - FSM
- **Gate-level code**
  - Toggle

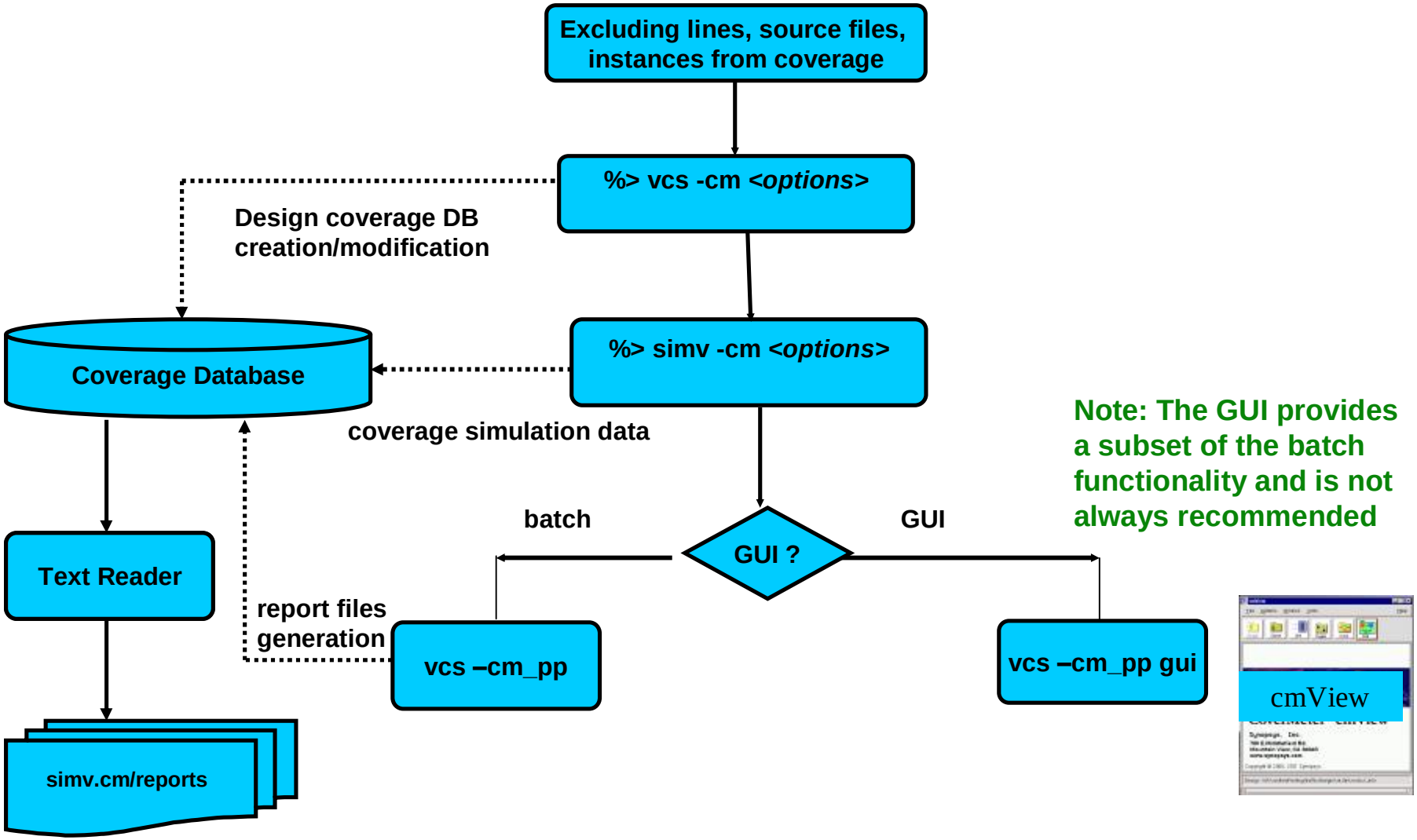
# VCS Coverage Metrics



- **VCM functionality is tightly integrated into VCS**
- **At compilation/elaboration time, it instruments the design for collection of coverage data and creates a coverage database**
- **At simulation time, it collects coverage data and fills the coverage database**
  - Usual simulation time overhead is no more than 20%-30%
  - Toggle coverage is expensive, up to 3 times overhead
- **At post-simulation time there are two modes:**
  - Batch mode: to generate coverage report file and grade test cases
  - GUI mode: to show and manipulate coverage data in graphical form

# Code Coverage Flow






Note: The GUI provides a subset of the batch functionality and is not always recommended

# Excluding Lines, Source Files, and Module Instances from Coverage



- **Use pragmas (meta-comments) in VHDL/Verilog source code**
  - Pragmas for Verilog:

    //VCS coverage on

    //VCS coverage off
  - Pragmas for VHDL:

    -- VCS coverage on

    -- VCS coverage off
- **Design Compiler pragmas act the same for coverage:**

  //synopsys translate_off

  //synopsys translate_on
- **Compile-time/report-time configuration file ( -cm_hier <name_of_file> ) allows inclusion/exclusion of any instance, module/entity, or sub-hierarchy**

# Compiling with Coverage



- **%> vcs *sourcefiles* -cm <coverage_type> <other coverage options>**
- **-cm <coverage_type> specifies the type of coverage to collect**
- **The options are:**
  - line Enables statement (line) coverage
  - tgl Enables toggle coverage
  - cond Enables condition coverage
  - fsm Enables FSM coverage
  - path Enables path coverage
- **Any combination of coverage types can be enabled simultaneously**
  - -cm cond+tgl+line+fsm+path

# Compiling with Coverage (cont.)



- **Default location for coverage data:**
  - ./simv.cm directory - for Verilog and Verilog top designs
- **Use -cm_dir to specify an alternate location and/or name of the coverage database**
- **Renaming simv with -o (-exe) will also rename simv.cm**
- **Compiling with**

```
%> vcs sourcefiles -o mysimv -cm line
```

- **Will create a mysimv.cm coverage directory**
- **-cm_dir option takes precedence over -o (-exe)**

# Compile-Time Coverage Options



- -cm_ cond <arguments> - different features of condition coverage
- -cm_count - enables counting how many times constructs were executed ( for line, condition, toggle, and FSM coverage)
- -cm_noconst - excludes constructs that cannot be covered because some operands are constants (not supported across module boundaries)
- -cm_fsmcfg - specifies FSM coverage configuration file
- -cm_hier - configuration file, which includes/excludes parts of design for different kinds of coverage

**Important note: do not use +rad option; it will change coverage results**

# Simulation with Coverage

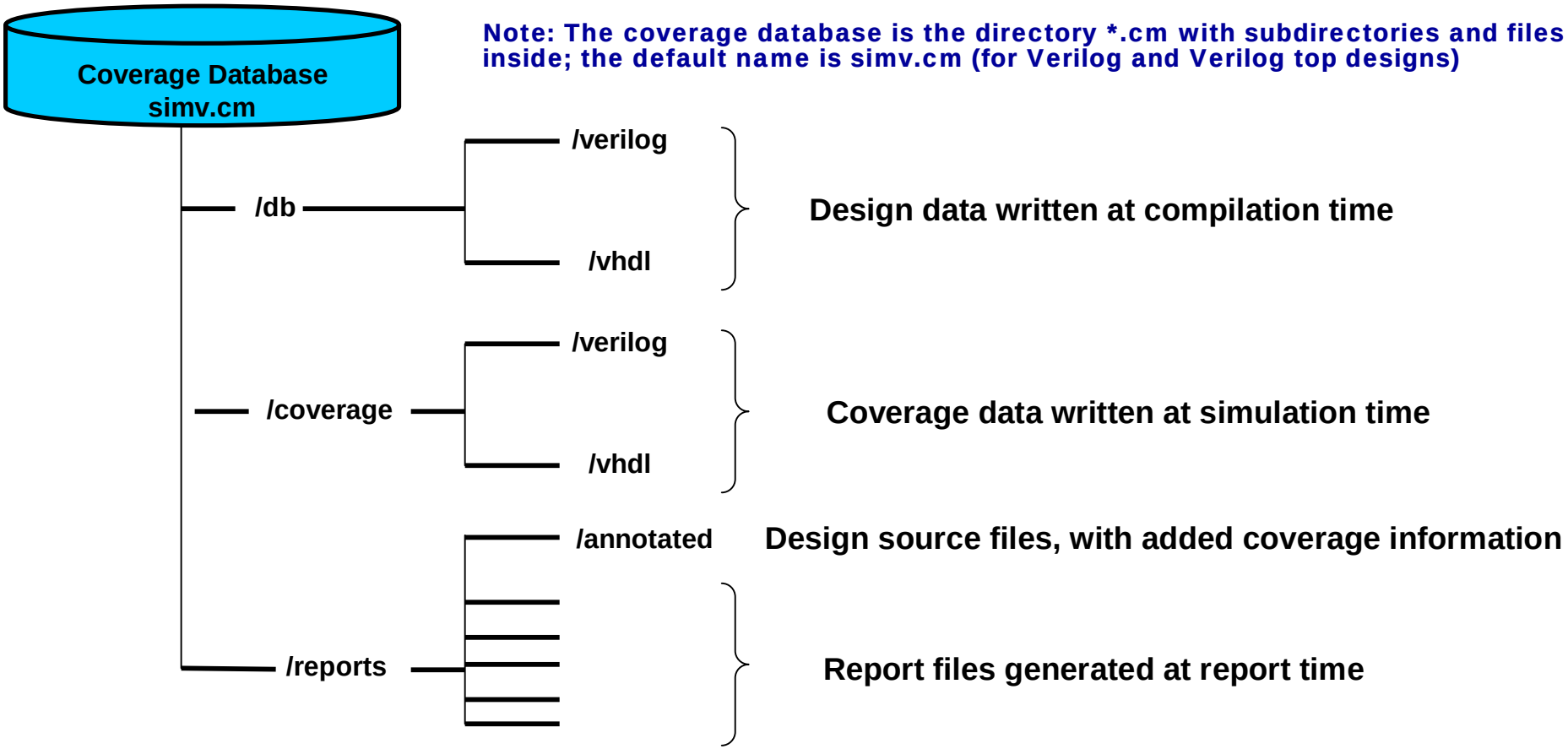


**%> simv -cm <coverage_type> <other coverage options>**

- **Simulation-Time Coverage Options**
- **-cm_name filename - specifies the name of the intermediate data files (highly recommended)**
- **-cm_glitch period - specifies a glitch period during which VCS does not monitor for coverage caused by value changes (recommended to use with period = 0)**
- **-cm_dir directory_path_name - specifies an alternative name and location for the coverage database**
- **-cm_log filename - specifies a log file for monitoring for coverage during simulation**

# Coverage Database





# Report Files for Each Type of Coverage 8-31

- cmView.short_l - A short report file containing only sections for instances in which all coverable objects were not covered. In these sections are only listed the uncovered objects. The report ends with summary information.
- cmView.short_ld - Another short report file, for module definitions instead of module instances
- cmView.hier_l - coverage of sub-hierarchies in the design
- cmView.mod_l - coverage of instances in the design
- cmView.mod_ld - coverage of each module in the design (summary of all module instances)
- cmView.long_l - detailed coverage of each instance in the design
- cmView.long_ld - detailed coverage of each module in the design

**Note: file names shown are for line coverage**

# Report-Time Coverage Options



- -cm_tests <file_name> - defines names of tests cmView reads
- -cm_nocasedef - excludes default choice of case statement from coverage
- -cm_autograde - generates report file with absolute and relative coverage estimation of each test
- -cm_hier - configuration file, which includes/excludes parts of design for different kinds of coverage
- -cm_name - specifies the name of report files (instead of default cmView ) - usually defines test case name for coverage database
- -cm_report - to change the position of summary in report files, ascending or descending order of covered instances
- -cm_verbose - reports coverage summary in terms of tests and type of coverage

# Accumulating Coverage Results



- **To get the total coverage for the design:**
  - Merge different test-case results for the same design (possibly with different test environments)
  - Import module/block-level coverage results to the chip-level design
- **Methods to Merge Coverage Results**
  - Method 1: Build the simv executable once, then simulate several times sequentially using the same testbench but different inputs
  - Method 2: Build several simv executables and simulate sequentially or in parallel

# Method 1 Example



**Build the simv executable once, then simulate three times sequentially using the same testbench but different inputs**

```
set COV =( line+cond+tgl+path+fsm )

#compilation for Coverage

vcs tst.v -cm $COV

#simulation of all test cases

simv -cm $COV  +TEST1 -cm_name TEST1

simv -cm $COV  +TEST2 -cm_name TEST2

simv -cm $COV  +TEST3 -cm_name TEST3

# merging Coverage for all test cases, and generation of report files

vcs -cm_pp -cm $COV -cm_nocasedef -cm_name TOTAL
```

# Autograding



- **Determines how much each test case contributes uniquely to the total coverage**
- **Autograding is coverage-type dependent**
  - For example, a particular test case can be valuable for line coverage, but not for toggle or other types of coverage
- **Implemented for line, condition, toggle, and FSM coverage**
- **cmView can generate an autograding report for one type of coverage per run**
- **Autograding report provides the list of all test cases and their metrics:**
  - Covered - coverage for given test
  - Accumulated - coverage summary of given test and previous ones
  - Difference - additional coverage of given test over the previous accumulated +additional coverage of previous accumulated and missed in given test
  - Incremental - additional coverage of given test over the previous test

# Autograding - Example



**Commands:**

**vcs -cm_pp -b -cm line -cm_autograding 100**
**vcs -cm_pp -b -cm cond -cm_autograding 100**

**Line coverage**

| Test No. | Incremental | Difference | Covered | Accumulated | Test Name |
|---|---|---|---|---|---|
| 0 | 82.35 | 82.35 | 82.35 | 82.35 | TEST1 |
| 1 | 11.76 | 55.88 | 50.00 | 94.12 | TEST2 |
| 2 | 5.88 | 70.59 | 35.29 | 100.00 | TEST3 |

**Conditional coverage**

| Test No. | Incremental | Difference | Covered | Accumulated | Test Name |
|---|---|---|---|---|---|
| 0 | 66.67 | 66.67 | 66.67 | 66.67 | TEST1 |
| 1 | 16.67 | 83.33 | 16.67 | 83.33 | TEST2 |
| 2 | 0.00 | 66.67 | 16.67 | 83.33 | TEST3 |

# Coverage Metrics Summary



- Integration with VCS
- Supports line, condition, toggle, FSM and Path
- Graphical and text based results
- Automatic and/or custom coverage
  - Now extended with Tcl
- Merge coverage for a design using different tests
- Autograding helps create more efficient testbenches

# Lab 8 Introduction



30 min