Automating Root-Cause Analysis to Reduce Time to Find Bugs by Up to 50%

By Kishore Karnane and Corey Goss, Cadence Design Systems

If you're spending more than 50% of your verification effort in debug, you're not alone. For many design, verification, and embedded software engineers as well as engineers verifying complex standard protocols, debug is the primary bottleneck in verification. Most debug today is completed using the traditional methodology of print statements paired with waveforms. Given that today's designs have grown significantly in complexity, the traditional debug approach is no longer adequate. This paper introduces a new way to debug that uses innovative debug concepts such as automated root-cause analysis (RCA), "Big Data" capture, and a unified apps-based debug analysis platform. Together, these advanced technologies can help you uncover bugs 50% faster than with traditional debug approaches.

Contents

Introduction

Whether you are designing intellectual property (IP) blocks or full systems on chip (SoCs), debug has been an onerous and tedious process in the project cycle because, traditionally, it is manual and time-consuming. Most engineers today are using post-process register-transfer level (RTL) waveform inspection paired with log files for message analysis. Since this methodology requires the engineer to know, up front, where to add print statements and what signals to record, it leaves the engineer vulnerable to running multiple debug iterations if the right information is not present at the end of the run.

On average, it takes engineers three to five cycles through the debug loop (code->simulate->debug) to isolate and fix a bug[1]. There are also several classes of procedural code failures such as infinite loops, null pointers, and scoreboard data mismatches that cannot easily be debugged using waveforms and log files alone.

Re-running simulations multiple times just to obtain more debug information significantly hampers overall debug performance and throughput. As shown in Figure 1, today's designs are causing engineers, on average, to spend 50% of their overall verification effort in debug, which includes triaging failures and analyzing simulation output.

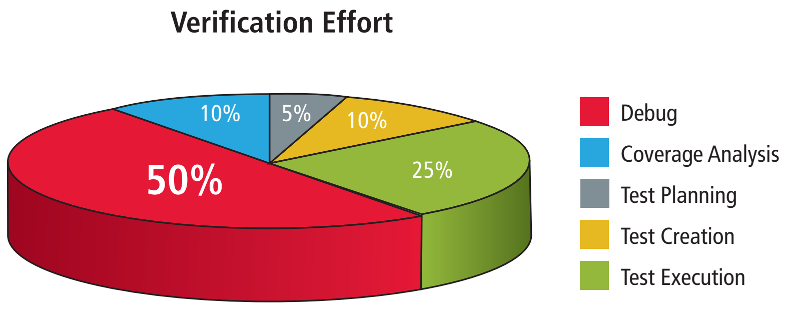


Figure 1: Engineers are spending at least 50% of their verification effort in debug.
Source: Interviews with Cadence® customers

A key reason for the increase in debug time seen over the past decade is that today's designs are orders of magnitude larger, more complex, and involve sophisticated object-oriented programming- (OOP-) based testbenches, third-party IP, and embedded software running on many cores. Today's testbenches are typically created in hardware verification languages (HVLs) like SystemVerilog, ***e***, and SystemC. Test cases are highly randomized with many moving parts. When an issue arises, identifying the origin of the bug can be quite challenging. Is it in the testbench? In the RTL? Or the VIP? Perhaps it is in the analog components or some other aspect of the chip? A substantial amount of analysis time is needed to understand the "big picture" as well as narrow down the area of the bug prior to identification.

Given the complexity of today's designs, you might assume that debug methodology has advanced as well. However, this has not been the case. Interestingly, most verification engineers are still using the same traditional debug flow (as shown in Figure 2 below) that has been used for more than 20 years. Particularly with larger or more complex designs, where debug data comes from many sources, this traditional debug flow can be very inefficient. Making matters worse, much of the data captured from run to run tends to be redundant, as the engineer incrementally changes probes to tease out just the right information from their verification engine. Significant changes and improvements are urgently needed to allow engineers to keep pace with today's shrinking project schedules and improve overall debug performance.

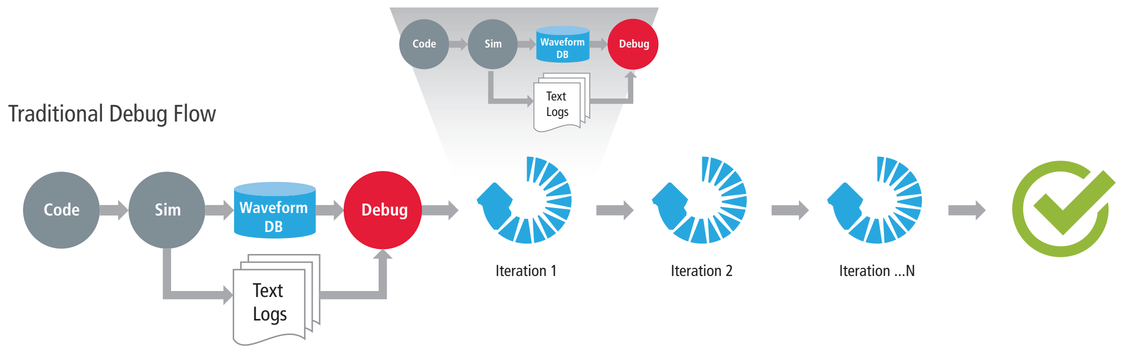


Figure 2: A traditional debug flow involves several debug iterations to collect more data

What is Debug Performance?

Before we discuss how we can improve debug performance, let's first define what debug performance is. Since debugging is a highly manual task involving analysis, code modification, and simulation, the time required to identify and fix a bug is not as easy to measure as the performance of a simulation run. We must consider a more holistic approach, involving two key aspects, when measuring debug performance:

- Verification engine cycles: Simulation throughput on a given server
- Engineer cycles: Amount of time an engineer spends analyzing debug data to isolate the bug

Verification engine cycles are always an important aspect of any project. However, as design complexity continues to increase, verification engine performance becomes a smaller part of the overall debug performance equation, since simulations can be executed in the background of an engineer's other day-to-day tasks. Analysis, on the other hand, requires focused attention and, as a result, is increasingly becoming the bottleneck that must be addressed to increase debug performance. The number one goal in increasing debug performance is to use engineer cycles as efficiently as possible. Engineers are, after all, much more expensive to any company than purchasing a faster server for running simulations.

In the next two sections, let's take a look at some of the debug techniques used today, along with their limitations when it comes to debugging complex SoCs.

Limitations in Root-Cause Analysis (RCA) Today

Typically, an engineer runs a simulation and an erroneous output gets detected by a checker within the verification environment. Using a traditional debug approach, the source of the bug could potentially be identified using debug techniques available today, such as:

- Print statements / probing
- Signal tracing
- Breakpoints /single stepping

In order to use the above techniques, the engineer reruns the simulation to collect debug information and/or set breakpoints to allow for interactive debug. The user may also modify the design to add more print statements or, perhaps, probe all of the relevant signals. After this step, the engineer would have to re-run the simulation yet again. This cycle of iterations might be repeated several times until the engineer actually records enough information to find the bug. In a normal scenario, each simulation iteration can last minutes to hours. For example, consider the diagram shown below in Figure 3. This environment has digital, analog, VIP and embedded software all simulating together. Today's debug methodologies typically have the engineer recording a subset of detailed debug information for the digital portions of the design only, leaving the remaining portions of the design to be debugged using limited information available within log files. What if we had a packet with an integer for the security code and a security code calculator with an integer as well? The actual code must be 16 bits. In our testing, we inject random values into the digital portion of the design and happen to get a few unexpected values at the output, which causes failures. During debug, we trace the failure back to what we think was the bug within the digital code that generated the erroneous values and we make a fix to the code and re-run. Surprisingly, the errors are still appearing. What we failed to recognize is that the underlying bug is, in fact, due to the complex interaction between the software, the VIP, and the digital portions of the design. It just so happens that, with today's debug methodologies, the most debug information is available for the digital portions of the design. Without recording enough debug information from all portions of a verification environment, engineers are, essentially, debugging in the dark, which leads to many unnecessary debug iterations and very manual and tedious debug efforts.

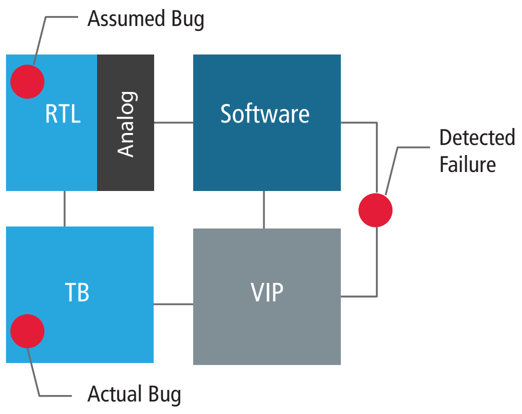


Figure 3: RCA as defined today. A bug could erroneously be attributed, masking the real problem.

Big Data Capture and Analysis

The term "Big Data" is being used all around us. Generally it refers to a fairly passive collection of a complete set of data points that allows for powerful and detailed analysis at a very low cost. This is in contrast to a traditional sampling-based approach, which focuses on a small, tightly constrained, and carefully selected set of data points for analysis. A sampling-based approach requires upfront thought, while offering limited analysis. By contrast, a Big Data approach requires little upfront effort with much more opportunities for advanced analysis after collection.

When it comes to verification, without a Big Data approach, engineers must rely on the traditional sampling-based approach of making educated guesses at where to sample debug data (print statements + probing of waveform signals). After sampling, they can only debug with the data points that they had the foresight to sample. To ask further debug questions, they must rerun to collect more samples, which likely will cost the user another re-compile, elaboration, and simulation run penalty. A Big Data approach to debug samples the entire debug data set once (messages, waveforms, source execution order, call stack, active threads, etc.), which allows engineers (and debug tools) to analyze and play back the simulation time and again. The engineers can then:

- Ask deeper questions about what really happened during the run
- Highlight causality
- And, possibly, point out correlations that would otherwise go unnoticed through a sampling-based approach

Additionally, traditional approaches of print statement plus waveform often cause the engineer to do a lot of context switching from the log to the wave. If not enough information is available, the engineer also may have to edit to the source to add print statements, re-run, and re-ramp. Imagine the time savings of having a complete set of recorded information available for analysis from a single GUI, as illustrated in Figure 4. You would no longer need to deal with context switching. Instead, you could seamlessly move from debugging testbench issues to RTL issues with equivalent details recorded for each. With this approach, you could potentially find bugs in minutes, instead of hours or even days. By decreasing the number of debug iterations, you could also take better advantage of your existing set of simulation licenses. Rather than running many debug iterations, you can use those licenses to hunt for new bugs through increasing regressions.

Now that we've examined RCA and Big Data, let's discuss how new technology integrates these capabilities into a unified, apps-driven platform that guides engineers in finding the root cause of the bug with a single iteration.

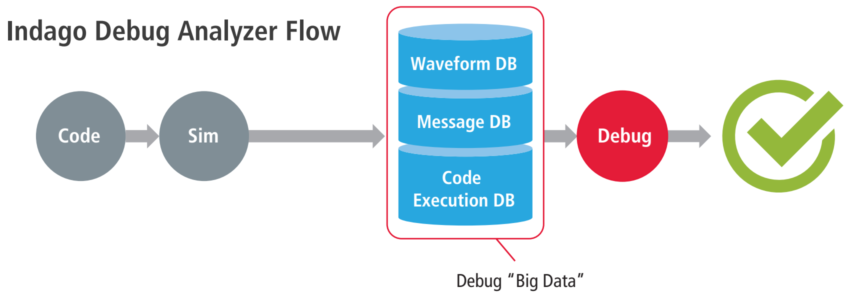


Figure 4: Big Data flow records complete data set for entire simulation run, not just waves

Integrated, Unified Platform for Uncovering Underlying Bugs

Cadence's new Indago™ Debug Platform, developed with patented, automated RCA technology and Big Data capture techniques, raises the level of debug abstraction to reduce the time to identify bugs by up to 50% vs. traditional debug methods. The platform supports IP-level to SoC-level debug. It works with Cadence and third-party verification engines to provide a unified, consistent debug platform for RTL, testbench, verification IP, and hardware/software debug. Because the platform is unified and synchronized, ramp-up is minimized when moving from debugging one aspect of your environment to another. The platform comes with three built-in apps:

- Indago Debug Analyzer App, which extends RCA from RTL to the ***e*** testbench (IEEE 1647) to SystemVerilog (IEEE 1800) and provides interactive post-process capabilities, along with synchronized debug of RTL and the testbench.
- Indago Embedded Software Debug App, which synchronizes software and hardware source code debug to resolve bugs associated with embedded software applications. This app is optimized for the Cadence Incisive® simulation platform and the Cadence Palladium® XP emulation platform.
- Indago Protocol Debug App, which provides intuitive debugging by using Cadence verification IP (VIP) to visualize advanced protocols such as DDR4, ARM® AMBA® AXI, and ARM AMBA ACE

Most engineers find that the most natural way to debug an issue is to start from the failure point and work backwards from there. For example, when debugging why an RTL signal transitioned to a particular value, engineers typically trace the drivers of the signals back through time and space to find the first occurrence of that value. Unlike most debug solutions today that are limited to RTL for RCA, the Indago Debug Platform applies RCA to both RTL and testbench, so you can debug a failure that originates in your SystemVerilog or ***e*** testbench just as easily as an RTL failure.

For example, if you encounter a UVM error message during simulation, you might ask what conditions led to this error message or why the variables within the failure message took on their values. Through Big Data capture and analysis, the Indago Debug Platform uses powerful algorithms to analyze each debug point as you navigate. At each point, the platform identifies causal relationships for the most typical debug questions, which are presented to you in an intuitive RCA GUI component. Using the RCA component, you can intuitively click your way through the complete failure scenario, seamlessly diving into the testbench or RTL code to quickly diagnose the root cause of the failure. To ensure that you can retrace your debug steps, each debug decision taken in the RCA component is saved in an investigation tree.

As Indago Debug Platform records the complete execution order of your source code, you can examine, at each point in the RCA process, the complete call stack, active threads, local variable values, as well as single step forward or backward to replay the simulation result. With the platform's set of RCA tools, you're equipped to better understand the complete failure scenario and diagnose bugs much more quickly than with traditional approaches.

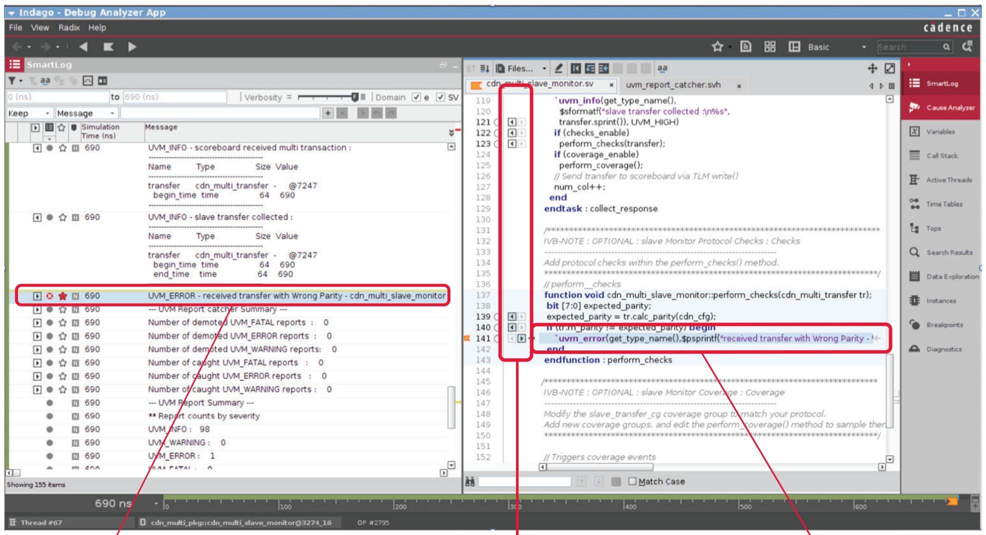
Key features of the Indago Debug Platform include:

- Root-Cause Analyzer, an intuitive, flow-oriented debug component that presents you with options as to where you should look to debug
- SmartLog, a dynamic, integrated message viewer with fast filtering of messages from many sources, including HDL, testbench, C/C++, SystemC, and assertions
- Reverse debugging, which allows single stepping forward or backward in time and the ability to add print statements to the SmartLog directly from the source
- Multi-engine data analysis, which allows debug data to be read from the testbench, RTL, embedded software, third-party simulations, or verification IP
- Debug handoff, which allows you to highlight values, set bookmarks, write notes, and save for a quick handoff to another engineer to continue the debug process
- Time tables, which allow you to select any line or variable and query for all- access throughout the entire simulation. Results are presented as a table.
- Powerful searching, with hyperlinked search results organized by source, value, type, and log
- Variables, whereby variables, type, and value are presented to you as you are debugging
- Call stack analysis, which enables you to understand the full calling tree at any point during debug
- Memory viewer, which lets you view RTL as well as the software memory view

In the next sections, we'll take a closer look at each of the platform's built-in apps.

Synchronized Testbench and RTL Debug

The Indago Debug Analyzer App is used for synchronized debug of the testbench and RTL. As shown in Figure 5, when you initially launch the App, the GUI opens to display SmartLog, an interactive log analysis tool that is hyperlinked back to the source code. If errors are present within the log, they are identified via a unique icon and automatically bookmarked. To save time, the GUI will open with the first error already highlighted, so you can begin debug immediately. As discussed in our previous section, the source browser displays the error message and allows you to step forward/backward through the recorded data to "play back" the bug scenario from any interesting point. Direct access buttons allow you to immediately jump the debugger to points of interest, such as the last time a line was executed or the last time a value was assigned to a variable.



Simulation Time (ns)	Message
690	UVM_INFO - scoreboard received multi transaction :
690	UVM_INFO - slave transfer collected :
690	UVM_ERROR - received transfer with Wrong Parity - cdn_multi_slave_monitor
690	--- UVM Report catcher Summary ---
690	Number of demoted UVM_FATAL reports : 0
690	Number of demoted UVM_ERROR reports : 0
690	Number of demoted UVM_WARNING reports: 0
690	Number of caught UVM_FATAL reports : 0
690	Number of caught UVM_ERROR reports : 0
690	Number of caught UVM_WARNING reports : 0
690	--- UVM Report Summary ---
690	** Report counts by severity
690	UVM_INFO : 98
690	UVM_WARNING : 0
690	UVM_ERROR : 1

```
-----------------------------------------
Name         Type              Size  Value
-----------------------------------------
transfer     cdn_multi_transfer  -    @7247
  begin_time  time              64    690
-----------------------------------------

-----------------------------------------
Name         Type              Size  Value
-----------------------------------------
transfer     cdn_multi_transfer  -    @7247
  begin_time  time              64    690
  end_time    time              64    690
```

cdn_multi_slave_monitor.sv   uvm_report_catcher.svh

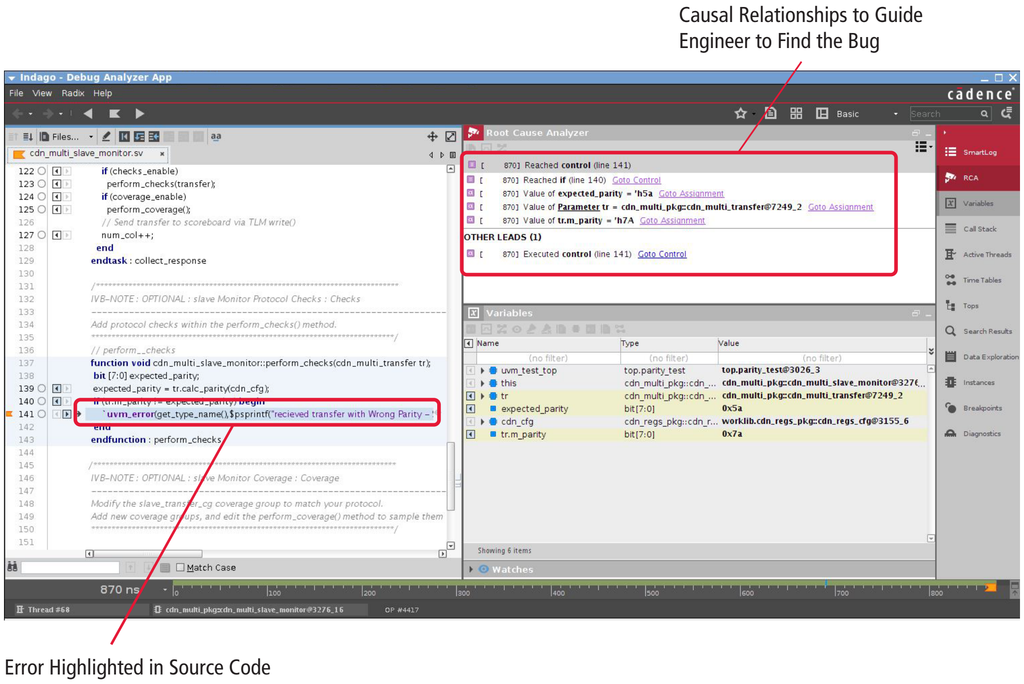
```
      `uvm_info(get_type_name(),
       $sformatf("slave transfer collected :\n%s",
       transfer.sprint()), UVM_HIGH)
      if (checks_enable)
       perform_checks(transfer);
      if (coverage_enable)
       perform_coverage();
      // Send transfer to scoreboard via TLM write()
      num_col++;
    end
  endtask : collect_response

  /*****************************************************************************
  IVB-NOTE : OPTIONAL : slave Monitor Protocol Checks : Checks
  -------------------------------------------------------------------------
  Add protocol checks within the perform_checks() method.
  *****************************************************************************/
  // perform__checks
  function void cdn_multi_slave_monitor::perform_checks(cdn_multi_transfer tr);
   bit [7:0] expected_parity;
   expected_parity = tr.calc_parity(cdn_cfg);
   if (tr.m_parity != expected_parity) begin
     `uvm_error(get_type_name(),$psprintf("received transfer with Wrong Parity -
   end
  endfunction : perform_checks

  /*****************************************************************************
  IVB-NOTE : OPTIONAL : slave Monitor Coverage : Coverage
  -------------------------------------------------------------------------
  Modify the slave_transfer_cg coverage group to match your protocol.
  Add new coverage groups, and edit the perform_coverage() method to sample them
  *****************************************************************************/

  // Triggers coverage events
```

Figure 5: Indago Debug Analyzer App opens with error already highlighted for the user to begin debug immediately

As you debug, there are always questions to be asked for every line of code or variable being examined. For example:

- Why was this line executed?
- Why did a variable on this line have a specific value?

By analyzing the recorded debug data collected, the Indago Debug Analyzer App can provide a list of causal relationships that answer the most common debug questions, as shown in Figure 6. The relationships are presented in a list format that allows you to click your way through the results, seamlessly traversing from testbench code to RTL, to very quickly arrive at the root cause of the investigated value.



cdn_multi_slave_monitor.sv

```
      if (checks_enable)
       perform_checks(transfer);
      if (coverage_enable)
       perform_coverage();
      // Send transfer to scoreboard via TLM write()
      num_col++;
    end
  endtask : collect_response

  /*****************************************************************************
  IVB-NOTE : OPTIONAL : slave Monitor Protocol Checks : Checks
  -------------------------------------------------------------------------
  Add protocol checks within the perform_checks() method.
  *****************************************************************************/
  // perform__checks
  function void cdn_multi_slave_monitor::perform_checks(cdn_multi_transfer tr);
   bit [7:0] expected_parity;
   expected_parity = tr.calc_parity(cdn_cfg);
   if (tr.m_parity != expected_parity) begin
     `uvm_error(get_type_name(),$psprintf("recieved transfer with Wrong Parity -
   end
  endfunction : perform_checks

  /*****************************************************************************
  IVB-NOTE : OPTIONAL : slave Monitor Coverage : Coverage
  -------------------------------------------------------------------------
  Modify the slave_transfer_cg coverage group to match your protocol.
  Add new coverage groups, and edit the perform_coverage() method to sample them
  *****************************************************************************/
```

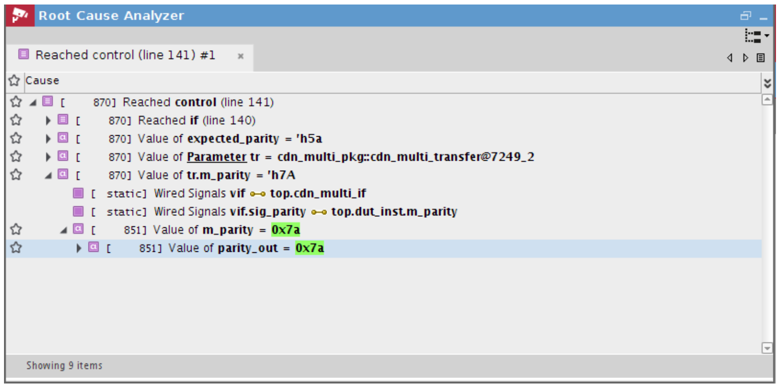
```
[      870] Reached control (line 141)
[      870] Reached if (line 140)  Goto Control
[      870] Value of expected_parity = 'h5a  Goto Assignment
[      870] Value of Parameter tr = cdn_multi_pkg::cdn_multi_transfer@7249_2  Goto Assignment
[      870] Value of tr.m_parity = 'h7A  Goto Assignment
OTHER LEADS (1)
[      870] Executed control (line 141)  Goto Control
```

Name	Type	Value
(no filter)	(no filter)	(no filter)
uvm_test_top	top.parity_test	top.parity_test@3026_3
this	cdn_multi_pkg::cdn_...	cdn_multi_pkg::cdn_multi_slave_monitor@327(...
tr	cdn_multi_pkg::cdn_...	cdn_multi_pkg::cdn_multi_transfer@7249_2
expected_parity	bit[7:0]	0x5a
cdn_cfg	cdn_regs_pkg::cdn_r...	worklib.cdn_regs_pkg::cdn_regs_cfg@3155_6
tr.m_parity	bit[7:0]	0x7a

Showing 6 items

Figure 6: Indago Debug Analyzer App provides the user with causal relationships to allow the user to quickly get to the root cause of any issue within a few clicks.

Using traditional debug tools, it is quite common for engineers to become "lost" in the tens of investigative decisions they have made while debugging. As shown in Figure 7, the Root Cause Analyzer window of the Indago Debug Analyzer App saves your decision tree, so you can quickly walk backward or forward through the entire debug path, or launch a new investigation from any node in the debug path.



```
[      870] Reached control (line 141)
  [      870] Reached if (line 140)
  [      870] Value of expected_parity = 'h5a
  [      870] Value of Parameter tr = cdn_multi_pkg::cdn_multi_transfer@7249_2
  [      870] Value of tr.m_parity = 'h7A
    [ static] Wired Signals vif ⊶ top.cdn_multi_if
    [ static] Wired Signals vif.sig_parity ⊶ top.dut_inst.m_parity
    [      851] Value of m_parity = 0x7a
      [      851] Value of parity_out = 0x7a
```

Figure 7: The Root Cause Analyzer window of the Indago Debug Analyzer App saves the debug path information, allowing users to review their debug steps and launch new investigations from any decision tree node.

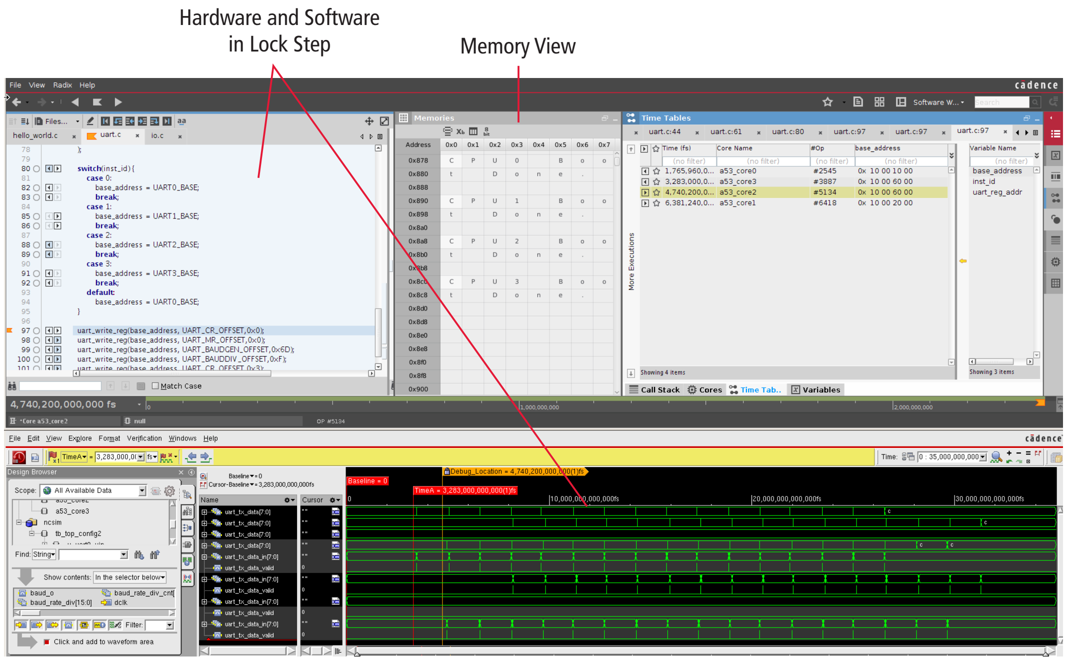
While debugging, if you find a value of interest, you can highlight it within the GUI. This highlight is applied to every location where this value appears, including in the waveform window. This helps cut debug time when switching between various contexts and comparing lengthy values (such as a 64-bit hex value). You can also query any variable to quickly understand when and where its value was updated or query for when specific source lines were executed throughout the recorded window. So, now, if we apply the patented RCA technology and the Big Data concept to the original problem as described in Figure 3, we may find out that the root cause was probably hidden in the OOP testbench code where, using traditional debug methods, we had to deal with very limited debug capabilities. Using the Indago Debug Analyzer App, the engineer would be able to run just one verification iteration and, using the analysis tools provided in the App, easily identify the root cause of the problem.

Embedded Software/Hardware Integration Debug

Embedded software delivers functionality on most of today's SoCs and is an important tool for software-driven verification. This growing software dependence creates demand for an efficient way to develop and debug these different types of software as well as any hardware bugs identified by the software. The Indago Embedded Software Debug App provides a software-centric environment to shorten the time to find the root cause of both in software and hardware bugs.

Since embedded software is developed for a range of needs, from software-driven verification to production operating systems and applications, embedded software debug must be available across a variety of platforms. It is important to consider both the performance and the model accuracy for both the development and application of the embedded software. For example, when developing software for hardware verification, it can be valuable to develop the software on a fast platform to improve developer productivity. However, because the software is testing the hardware, it will ultimately need to be run on a fully accurate platform. The Indago Embedded Software Debug App meets these requirements, providing a consistent debug environment across platforms and processor model abstractions.

In addition to the performance and accuracy tradeoff, another key characteristic is hardware visibility and hardware/software synchronization. Typically, the level of functionality and visibility into software is good, but when doing detailed debug of hardware and software interaction, the critical factors are the visibility of the hardware activities and how the hardware activity is synchronized with the software. As shown in Figure 8, the Indago Embedded Software Debug App uses post-process analysis of hardware and software trace information to provide the most accurate combined views of the hardware and software. The App leverages the core Indago technology to manage and visualize the "Big Data" of a complete multicore/multicluster software trace per core to arrive at the root cause more quickly. Having the complete context of software execution makes it easier to locate the origin of a problem.



```
);

switch(inst_id) {
   case 0:
      base_address = UART0_BASE;
      break;
   case 1:
      base_address = UART1_BASE;
      break;
   case 2:
      base_address = UART2_BASE;
      break;
   case 3:
      base_address = UART3_BASE;
      break;
   default:
      base_address = UART0_BASE;
}

uart_write_reg(base_address, UART_CR_OFFSET,0x0);
uart_write_reg(base_address, UART_MR_OFFSET,0x0);
uart_write_reg(base_address, UART_BAUDGEN_OFFSET,0x6D);
uart_write_reg(base_address, UART_BAUDDIV_OFFSET,0xF);
```

Time (fs)	Core Name	#Op	base_address
1,765,960,0...	a53_core0	#2545	0x 10 00 10 00
3,283,000,0...	a53_core3	#3887	0x 10 00 60 00
4,740,200,0...	a53_core2	#5134	0x 10 00 60 00
6,381,240,0...	a53_core1	#6418	0x 10 00 20 00

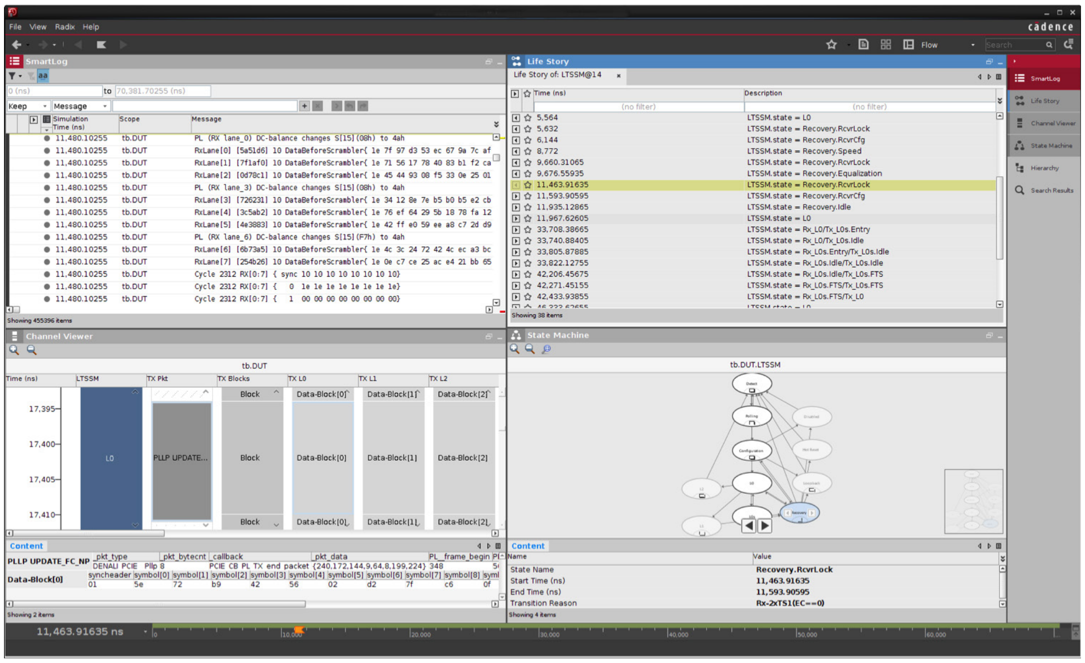
Figure 8: Synchronized hardware/software view in Indago Embedded Software Debug App

Visualization of Protocol Traffic and Events

Verifying the standard interfaces in IP blocks and SoCs can be one of the most resource-intensive tasks for verification teams. Popular interface specification families such as ARM AMBA, DDR, MIPI, and others have undergone major expansions in recent years. Verification engineers seldom have more than a couple weeks to ramp up on a new protocol before they have to start verifying designs incorporating it. Commercial VIP helps offload part of the verification effort, but engineers still need a window into the protocol-specific interactions between the design, the VIP, and the testbench to find the root causes of bugs.

Such a window is provided by the Indago Protocol Debug App, which presents a holistic picture of the verification environment in four views (see Figure 9):

- Channel Viewer
- State Machine Viewer
- Life Story
- SmartLog



```
Simulation Time (ns)  Scope   Message
11,480.10255  tb.DUT  PL (RX lane_0) DC-balance changes S[15](08h) to 4ah
11,480.10255  tb.DUT  RxLane[0] [5a51d6] 10 DataBeforeScrambler{ 1e 7f 97 d3 53 ec 67 9a 7c af
11,480.10255  tb.DUT  RxLane[1] [7f1af0] 10 DataBeforeScrambler{ 1e 71 56 17 78 40 83 b1 f2 ca
11,480.10255  tb.DUT  RxLane[2] [0d78c1] 10 DataBeforeScrambler{ 1e 45 44 93 08 f5 33 0e 25 01
11,480.10255  tb.DUT  PL (RX lane_3) DC-balance changes S[15](08h) to 4ah
11,480.10255  tb.DUT  RxLane[3] [726231] 10 DataBeforeScrambler{ 1e 34 12 8e 7e b5 b0 b5 e2 cb
11,480.10255  tb.DUT  RxLane[4] [3c5ab2] 10 DataBeforeScrambler{ 1e 76 ef 64 29 5b 18 78 fa 12
11,480.10255  tb.DUT  RxLane[5] [4e3883] 10 DataBeforeScrambler{ 1e 42 ff e0 59 ee a8 c7 2d d9
11,480.10255  tb.DUT  PL (RX lane_6) DC-balance changes S[15](F7h) to 4ah
11,480.10255  tb.DUT  RxLane[6] [6b73a5] 10 DataBeforeScrambler{ 1e 4c 3c 24 72 42 4c ec a3 bc
11,480.10255  tb.DUT  RxLane[7] [254b26] 10 DataBeforeScrambler{ 1e 0e c7 ce 25 ac e4 21 bb 65
11,480.10255  tb.DUT  Cycle 2312 RX[0:7] { sync 10 10 10 10 10 10 10 10}
11,480.10255  tb.DUT  Cycle 2312 RX[0:7] {   0  1e 1e 1e 1e 1e 1e 1e 1e}
11,480.10255  tb.DUT  Cycle 2312 RX[0:7] {   1  00 00 00 00 00 00 00 00}
Showing 455396 items
```

Life Story of: LTSSM@14

Time (ns)	Description
5,564	LTSSM.state = L0
5,632	LTSSM.state = Recovery.RcvrLock
6,144	LTSSM.state = Recovery.RcvrCfg
8,772	LTSSM.state = Recovery.Speed
9,660.31065	LTSSM.state = Recovery.RcvrLock
9,676.55935	LTSSM.state = Recovery.Equalization
11,463.91635	LTSSM.state = Recovery.RcvrLock
11,593.90595	LTSSM.state = Recovery.RcvrCfg
11,935.12865	LTSSM.state = Recovery.Idle
11,967.62605	LTSSM.state = L0
33,708.38665	LTSSM.state = Rx_L0/Tx_L0s.Entry
33,740.88405	LTSSM.state = Rx_L0/Tx_L0s.Idle
33,805.87885	LTSSM.state = Rx_L0s.Entry/Tx_L0s.Idle
33,822.12755	LTSSM.state = Rx_L0s.Idle/Tx_L0s.Idle
42,206.45675	LTSSM.state = Rx_L0s.Idle/Tx_L0s.FTS
42,271.45155	LTSSM.state = Rx_L0s.FTS/Tx_L0s.FTS
42,433.93855	LTSSM.state = Rx_L0s.FTS/Tx_L0

Showing 38 items

Name	Value
State Name	Recovery.RcvrLock
Start Time (ns)	11,463.91635
End Time (ns)	11,593.90595
Transition Reason	Rx-2xTS1(EC==0)

Showing 4 items

```
PLLP UPDATE_FC_NP  _pkt_type    _pkt_bytecnt  _callback                  _pkt_data                             PL_frame_begin
                   DENALI PCIE  Pllp 8        PCIE CB PL TX end packet   {240,172,144,9,64,8,199,224}          348
Data-Block[0]      syncheader symbol[0] symbol[1] symbol[2] symbol[3] symbol[4] symbol[5] symbol[6] symbol[7] symbol[8]
                   01         5e        72        b9        42        56        02        d2        7f        c6        0f
Showing 2 items
```

11,463.91635 ns

Figure 9: Indago Protocol Debug App showing (counter-clockwise from bottom-left) Channel Viewer, State Machine Viewer, Life Story, and SmartLog.

The Channel Viewer portrays the communication between the design and VIP with a lab equipment-style view showing the sequence of transactions with protocol-specific labeling. The State Machine Viewer shows a graphical view of the state machines within the VIP components along with the causes of state transitions. The Life History and SmartLog provide detailed views of all events pertaining to simulation objects and smart filtering of messages. Together, these views present an integrated picture of the design and verification environment to simplify the debugging process.

Summary

Debug has long been a verification bottleneck, commonly taking half of the verification project time. As SoCs have grown in complexity, debug is now more challenging than ever. There are now many more components and layers to debug, and traditional debug methodologies are falling short. The Indago Debug Platform offers new patented, automated RCA technology, a Big Data approach, and a unified debug environment for RTL/testbench, hardware/software, and VIP protocol debug that has demonstrated the ability to reduce the time to find bugs by 50%. This technology is essentially taking the guesswork and excessive iterations out of the debug equation and improving the overall debug productivity.

For Further Information

Learn more about the Cadence Indago Debug Platform at: www.cadence.com/products/fv/indago_debug_platform/pages/default.aspx.

Footnotes

[1] As found through Cadence customer interviews.

cadence®

Cadence Design Systems enables global electronic design innovation and plays an essential role in the creation of today's electronics. Customers use Cadence software, hardware, IP, and expertise to design and verify today's mobile, cloud and connectivity applications. **www.cadence.com**

© 2015 Cadence Design Systems, Inc. All rights reserved. Cadence, the Cadence logo, Incisive, and Palladium are registered trademarks and Indago and vManager are trademarks of Cadence Design Systems, Inc. ARM and AMBA are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All others are properties of their respective holders. 3703 04/15 CY/DM/PDF