DEBUG THIS!

CLASS-BASED TESTBENCH DEBUGGING WITH VISUALIZER

RICH EDELMAN, VERIFICATION TECHNOLOGIST, MENTOR GRAPHICS

V E R I F I C A T I O N

W H I T E P A P E R

w w w . m e n t o r . c o m

INTRODUCTION

Heard in the hall… "New School Debugger! Wow! I can't wait. But I'm skeptical. What makes it new? And does it even work? No one likes to debug a testbench. But it would be nice to have something to make life easier for testbench debug. Does it work in post-simulation mode? OK. I'll listen."

The testbench isn't the product. The testbench is not going to make any money and the testbench isn't what the boss is yelling about as tape out approaches. He wants the RTL bugs gone and a functionally correct RTL.

Guess what's a good way to find bugs and assure functional correctness? Have a good testbench.

Testbenches are different than RTL. They have behavioral Verilog code. They have files and data structures. They have threads. They have objects. Modern testbenches are likely class-based testbenches, using SystemVerilog, the UVM and object-oriented programming concepts.

Debugging these modern class-based testbenches has been painful for at least two reasons. First, they are designed and built using object-oriented coding styles, macros, dynamic transactions, phasing and other constructs completely foreign to RTL verification experts. Second, the tools for class-based debug have lagged the simulators ability to simulate them.

CHANGE IS HERE

Class based debug is different. It is not the same as debugging RTL. You don't have to be an object-oriented programmer in order to debug a class based testbench. It's just a bunch of objects – some statically created, some dynamic – which interact with the DUT. Class based debug doesn't have to be hard.

POST-SIMULATION DEBUG

Many RTL debug sessions are post-simulation. Simulation is run, and a PLI/VPI application records signal value changes into a database. In order to do debug the debug application loads the database and debug happens. Class based debug can operate in the same way, but the reliance on PLI or VPI can be problematic. The LRM defines VPI routines for class based access, but many if not all simulator vendors have not fully implemented those "class based VPI" routines.

Moving forward, post-simulation debug for class based testbench debug may become more simulator specific – at least until the vendors provide a complete VPI implementation. "Simulator independent" post-simulation debug may be a thing of the past – or at least have limited functionality.

That doesn't mean class-based testbench debug won't work. It will work – as you'll see below. It does, however, mean that really good class-based testbench debug will come from the vendor that does the best job integrating class based debug into traditional post-simulation RTL flows.

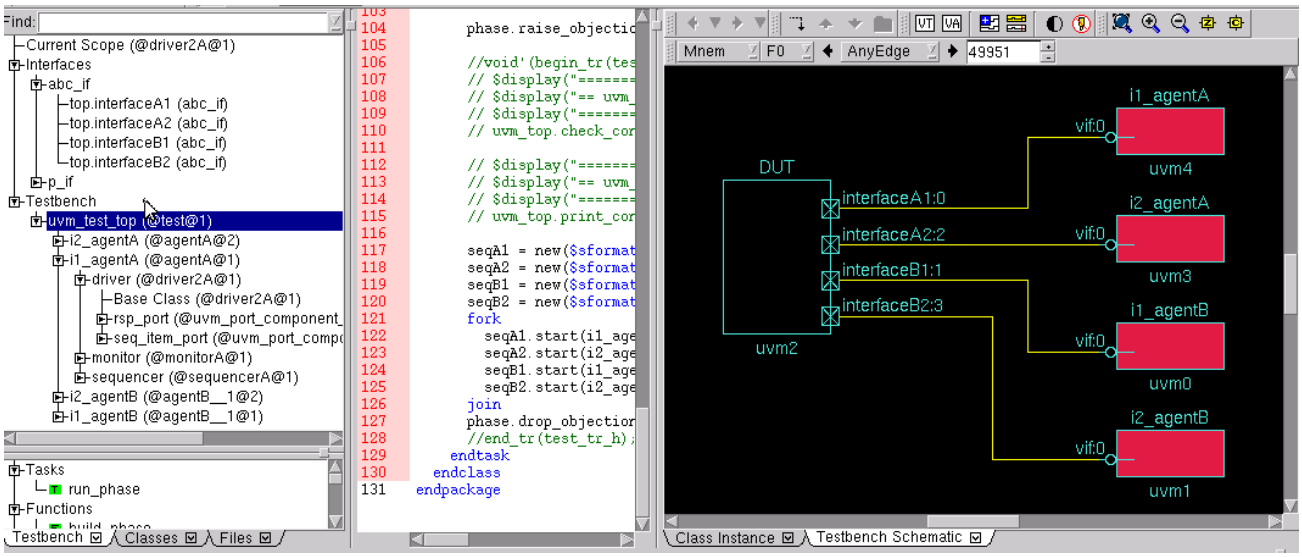
DEBUG THIS!

The rest of this article will discuss various class based debug techniques that you might find interesting. The example design is a small RTL design with a trivial bus interface and a small UVM testbench. We've applied these same techniques to customer designs that are large RTL with many standard bus interfaces, and a complex UVM testbench.

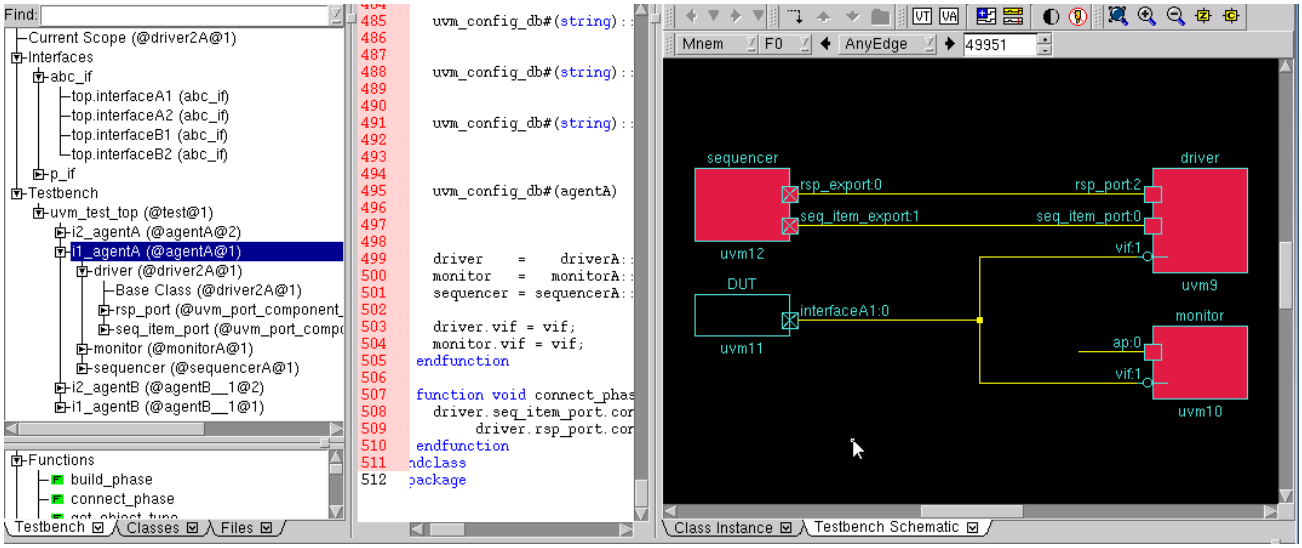
Your best use of this article would be to bring up your design, and try out the techniques outlined.

EXPLORING YOUR UVM-BASED TESTBENCH

How about using UVM Schematics? Everybody loves schematics as they make connectivity easy to see. In this testbench the test and environment are simple. There is a DUT with four interfaces. Each interface is connected to an agent. Select the instance in the UVM component hierarchy and choose "View Schematic."



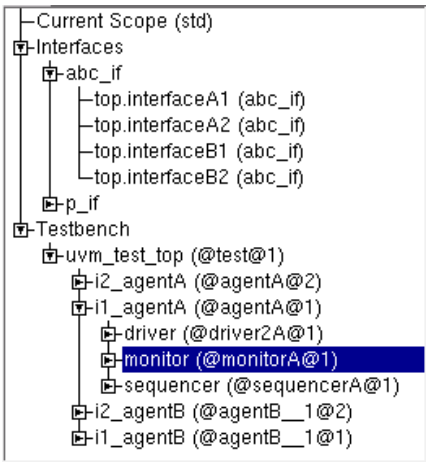
If we select the i1_agentA object, you can see that each agent has the usual structure – a sequencer, a driver and a monitor. 'AgentA' is drawn below.



Schematics are useful, but can get very busy, very quickly. A fabric with 20 or 30 interfaces and large blocks connected by AXI will become a tangled mess. Schematics are a tool we'll use in testbench debug, but we're going to do our debug here using 'objects'. We're going to explore the objects in the testbench – both the UVM Components– the drivers, monitors and agents; and the UVM Objects – the sequences and transactions (sequence items).

UVM COMPONENT HIERARCHY

The first way you can explore your UVM based testbench is by traversing the UVM component hierarchy. You can expand the 'uvm_test_top' and see the children, 'i2_agentA,' 'i1_agentA,' 'i2_agentB' and 'i1_agentB.' If you further expand 'i1_agentA,' then you see the children 'driver,' 'monitor' and 'sequencer.'

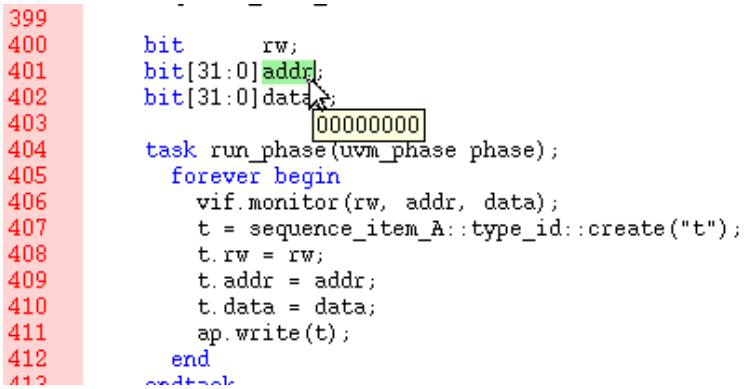


Select the 'monitor' object. That monitor object has a name – a UVM name like 'uvm_test_top.i1_agentA.monitor,' but also a simulator specific name – a shortcut for the actual physical address. You can see the UVM name by traversing the hierarchy shown – 'uvm_test_top', then 'i1_agentA,' then 'monitor.' This is the same that you would get by calling 'obj.get_full_name().' The shortcut name is '@monitorA@1.' This shortcut name can be parsed – it means that this object is the first object constructed of class type 'monitorA.'

Once the monitor is selected in the component hierarchy window, the source code for 'monitorA' is shown in the source code window.

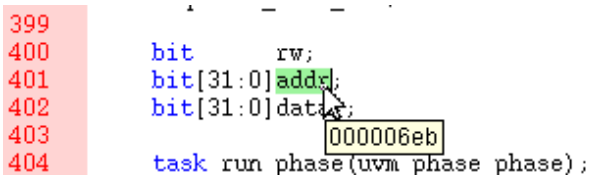
```
383   class monitorA extends uvm_monitor;
384     `uvm_component_utils(monitorA)
385
386     virtual abc_if vif;
387
388     uvm_analysis_port #(sequence_item_A) ap;
389
390     function new(string name = "monitorA", uvm_component parent = null);
391       super.new(name, parent);
392     endfunction
393
394     function void build_phase(uvm_phase phase);
395       ap = new("ap", this);
396     endfunction
397
398     sequence_item_A t;
399
400     bit      rw;
401     bit[31:0]addr;
402     bit[31:0]data;;
403
404     task run_phase(uvm_phase phase);
405       forever begin
406         vif.monitor(rw, addr, data);
407         t = sequence_item_A::type_id::create("t");
408         t.rw = rw;
409         t.addr = addr;
410         t.data = data;
411         ap.write(t);
412       end
413     endtask
414   endclass
```

So far not too astounding, nor interesting. The highlighted RED numbers mean that this source code is from the currently selected instance – it is the current context. If we hover over the 'addr' field then we see the value of 'addr' from the current time.



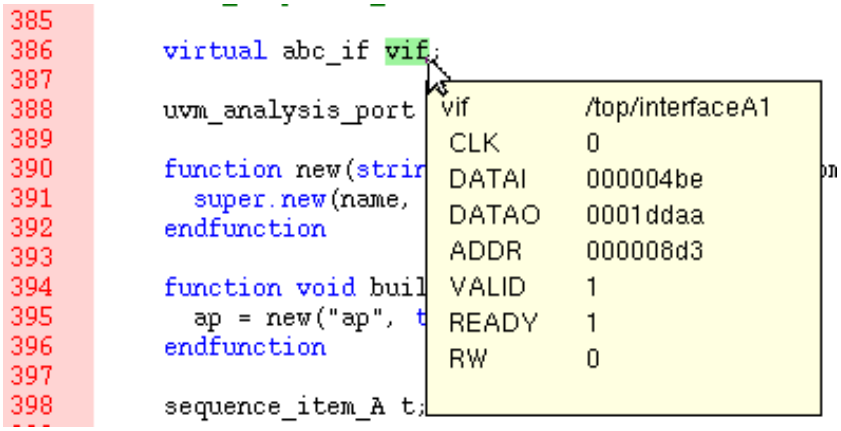
Hover over a value? Still not astounding. But wait. Remember this is post-simulation, and we're looking at the value of a ***member variable inside an instance of an object***. Cool.

What happens if we change from time 0 to some other time? For example, if the current time is 49993, then hovering over addr looks like:



Now that's interesting. In post-simulation mode, we have access to all the class member variables for our monitor. Questa simulation recorded the testbench information so that you can have access to the objects that existed during simulation. You can have post-simulation debug and see your dynamic objects too. Have your cake and eat it too.

What about the virtual interface that the monitor is connected to? Hovering over the variable 'vif' gives us:

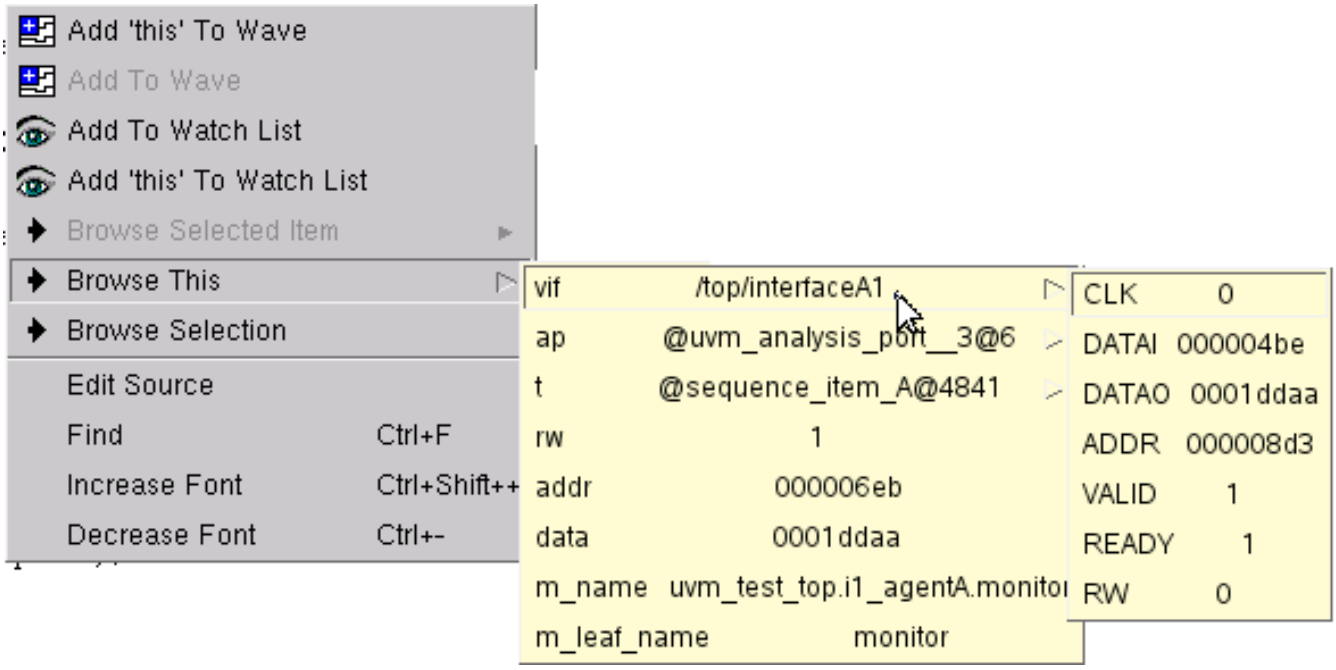


Neat. The popup menu is displaying the current values of the signals in the virtual interface.

BROWSE THIS!

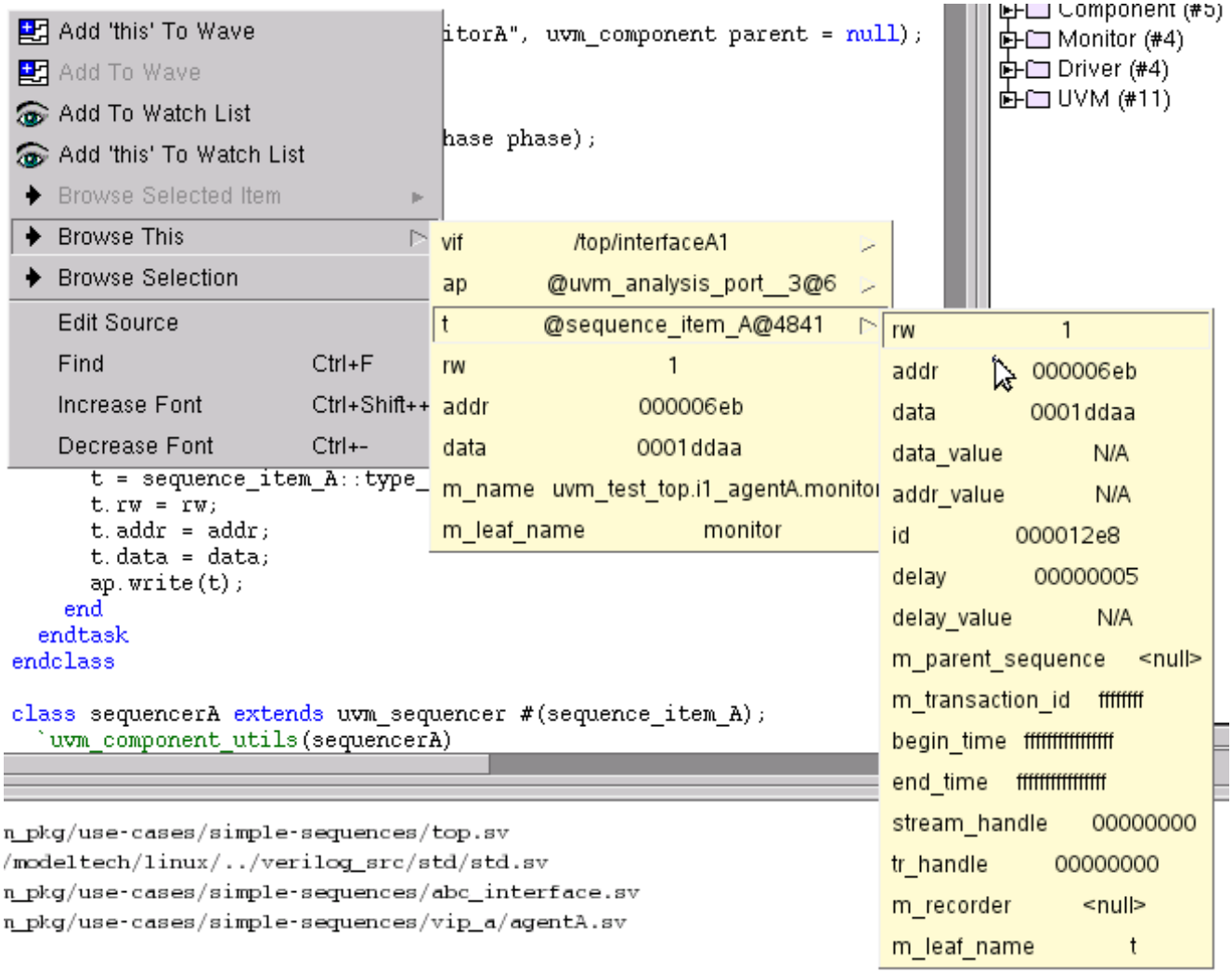
Is there an easier way to see ALL of the member variables at once for this object? How about fast exploration of class objects in post-simulation? That's what "Browse This" gives you.

In the source code for this object press the right-mouse-button. The menu appears. Select 'Browse This' and then select a variable you want to see; like 'vif':

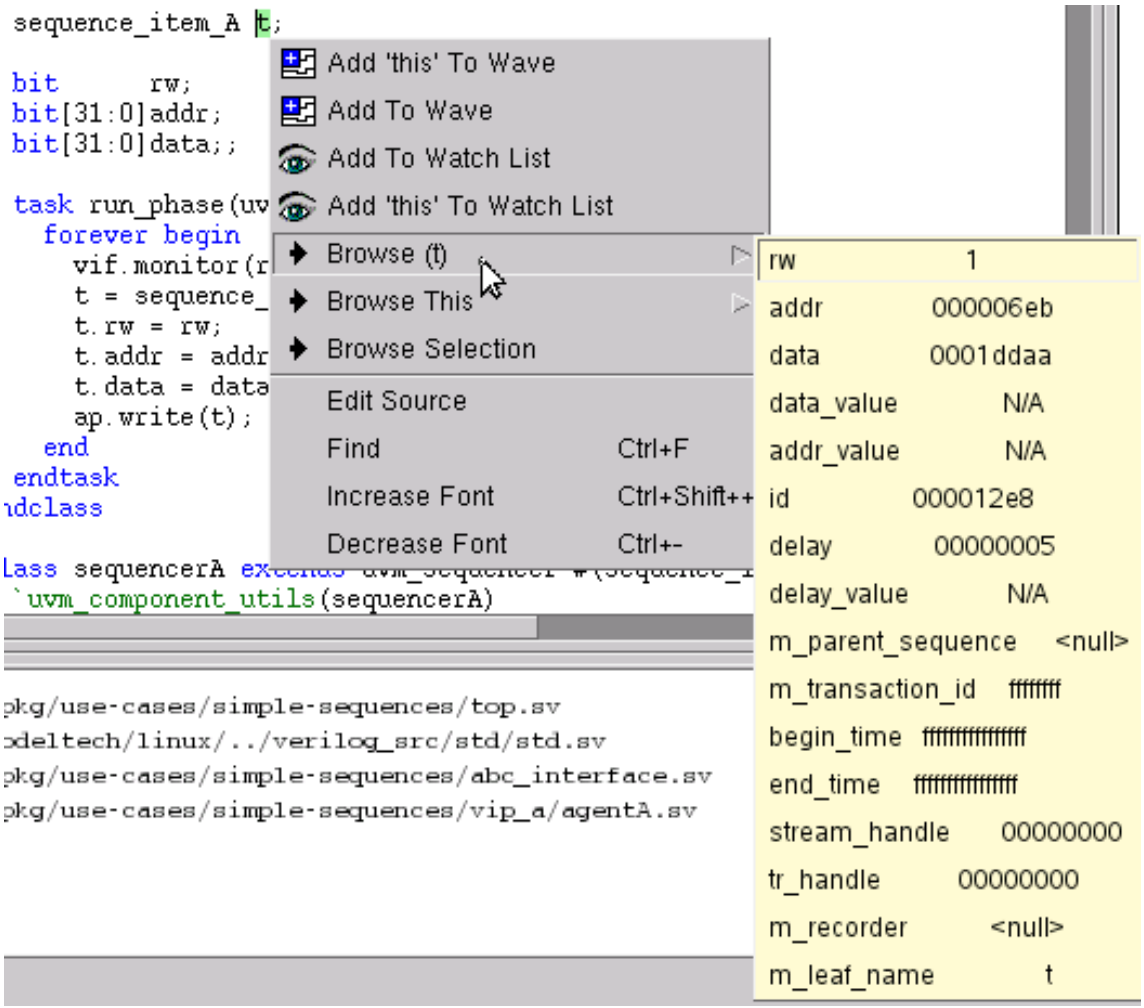


The vif for this monitor is displayed right there with a right-mouse-button and a select. You can see the instance name of the virtual interface – "/top/interfaceA1," and you can see the current values of the signals on the interface.

Using the right-mouse-button in the source code window is a very powerful way to instantly browse a class object, and follow any class handles. For example, right-mouse-button, select 't' and see the object which is pointed to by 't' at the current time ('t' points at the object named "@sequence_item_A@4841").



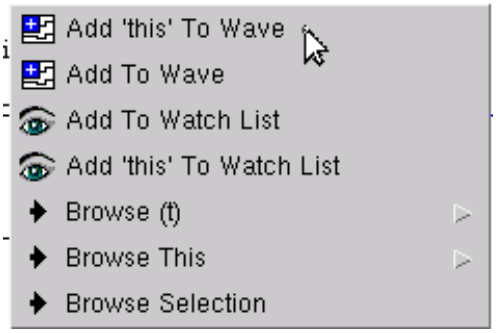
Of course, we could have gotten the value of 't' more easily in this case by selecting 't', then right-mouse-button then "Browse (t)."



The popup menu is displaying the values of the member variables in the object pointed to by 't' at the current time.

CLASS HANDLES IN THE WAVEFORM WINDOW

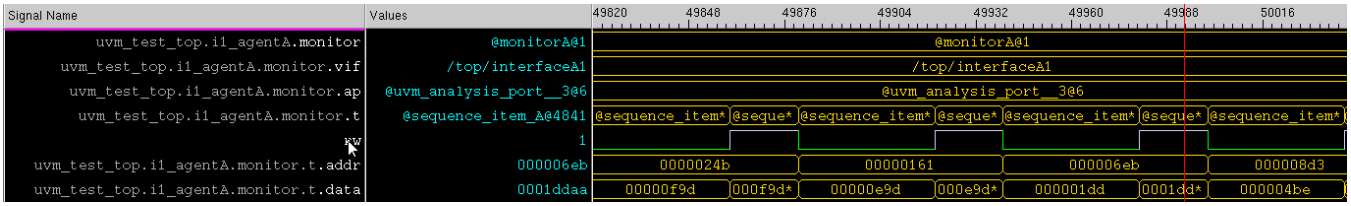
'Browse This' is very powerful, but sometimes there is nothing better than a waveform to view what is happening over time. In the monitor source code window, hit the right-mouse-button, and select "Add 'this' To Wave."



Now the 'this' pointer – the current monitor object – is added to the wave window. The object 'monitor' has been added to the wave window! Wow, a class handle in the wave window.

Expand the 'monitor' handle by clicking on the plus sign in the Signal Name column. Now we can see the class member variables in the wave window. We see 'vif,' 'ap' and the transaction 't.' The transaction pointer 't' is the most interesting thing about the monitor.

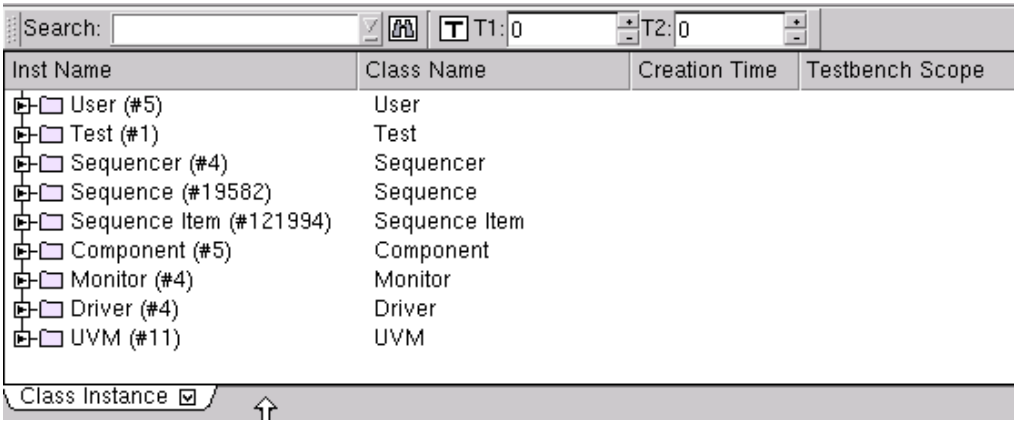
This one handle, when displayed in the waveform window, displays all the transactions that the monitor has created; that's **all the transactions that the monitor has created and sent out to any UVM subscriber**. Now that's powerful.



MORE EXPLORING YOUR UVM TESTBENCH

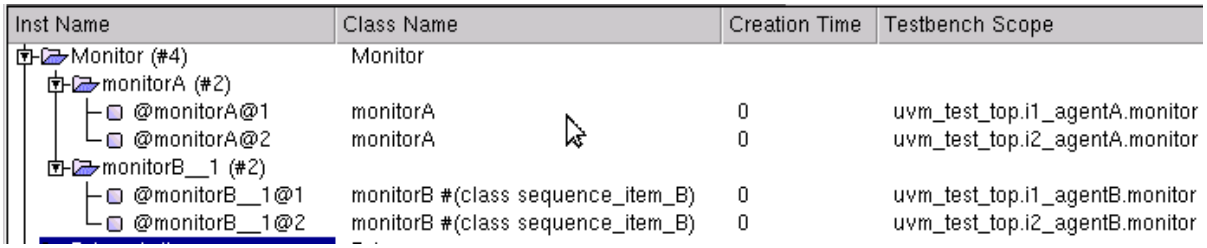
What if you know there is a class you want to debug, but you don't know where it is in the UVM hierarchy? Or the class is not a uvm_component? Maybe you want to debug a sequence? Or even debug a monitor or driver? Try the Class Instance window.

CLASS INSTANCE WINDOW



The Class Instance window is organized by base class. In this case there are 4 driver instances, 4 monitor instances, and 121,994 sequence items along with many other instances.

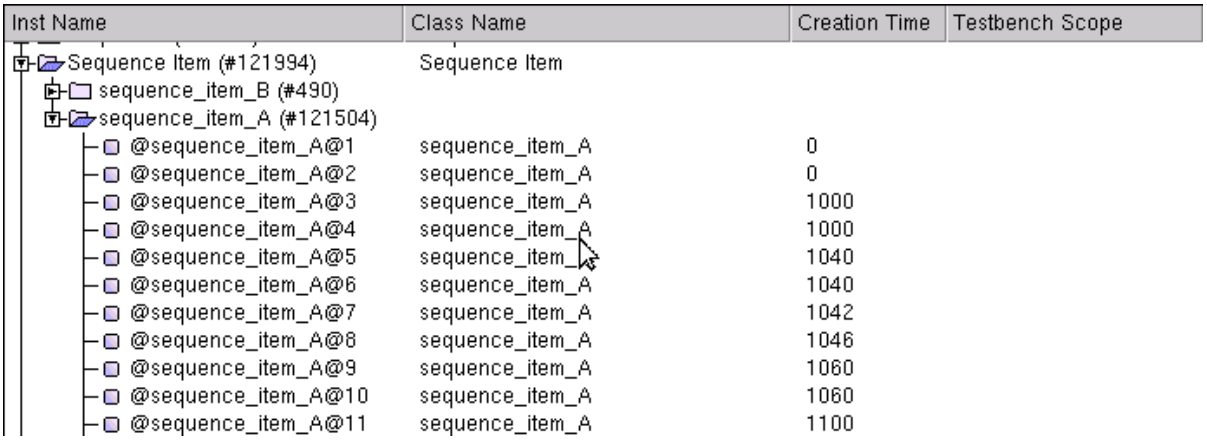
If we expand the Monitor, we can see more details about the four monitors in our testbench.



Inst Name	Class Name	Creation Time	Testbench Scope
Monitor (#4)	Monitor		
monitorA (#2)			
@monitorA@1	monitorA	0	uvm_test_top.i1_agentA.monitor
@monitorA@2	monitorA	0	uvm_test_top.i2_agentA.monitor
monitorB__1 (#2)			
@monitorB__1@1	monitorB #(class sequence_item_B)	0	uvm_test_top.i1_agentB.monitor
@monitorB__1@2	monitorB #(class sequence_item_B)	0	uvm_test_top.i2_agentB.monitor

We can see the @monitorA@1 instance name, the Class name, either parameterized or not, the time this object was created, and the UVM testbench name (if it has one). If we select one of these objects – for example "@ monitorA@1," the current context changes to that object, and we can "Browse This," just like we did above coming from the UVM component hierarchy tree.

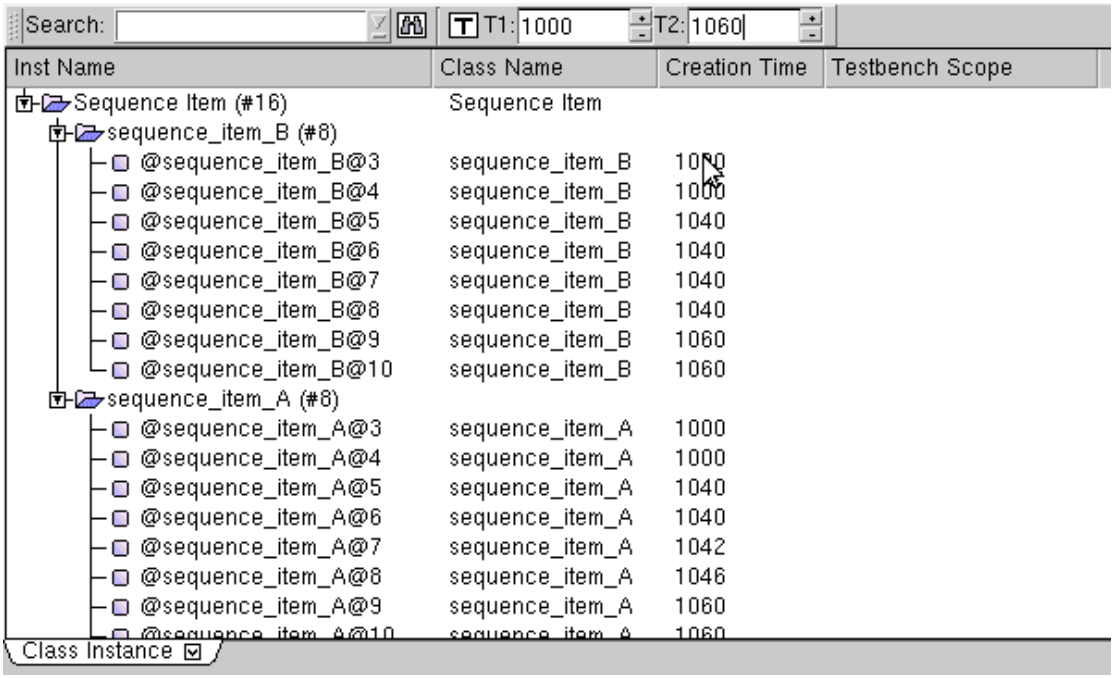
If we expand the Sequence Items, we can see some interesting creation times:



Inst Name	Class Name	Creation Time	Testbench Scope
Sequence Item (#121994)	Sequence Item		
sequence_item_B (#490)			
sequence_item_A (#121504)			
@sequence_item_A@1	sequence_item_A	0	
@sequence_item_A@2	sequence_item_A	0	
@sequence_item_A@3	sequence_item_A	1000	
@sequence_item_A@4	sequence_item_A	1000	
@sequence_item_A@5	sequence_item_A	1040	
@sequence_item_A@6	sequence_item_A	1040	
@sequence_item_A@7	sequence_item_A	1042	
@sequence_item_A@8	sequence_item_A	1046	
@sequence_item_A@9	sequence_item_A	1060	
@sequence_item_A@10	sequence_item_A	1060	
@sequence_item_A@11	sequence_item_A	1100	

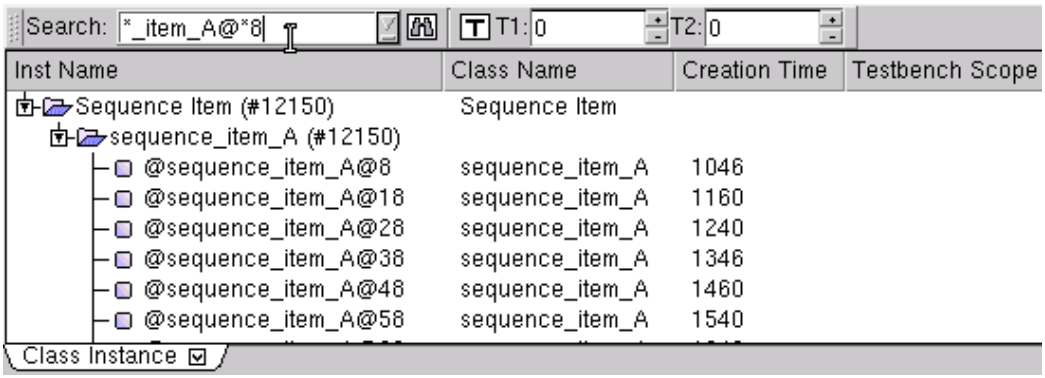
SEARCHING FOR INSTANCES BY TIME

We can create a search – to find the objects created between two times – 1000 and 1060.

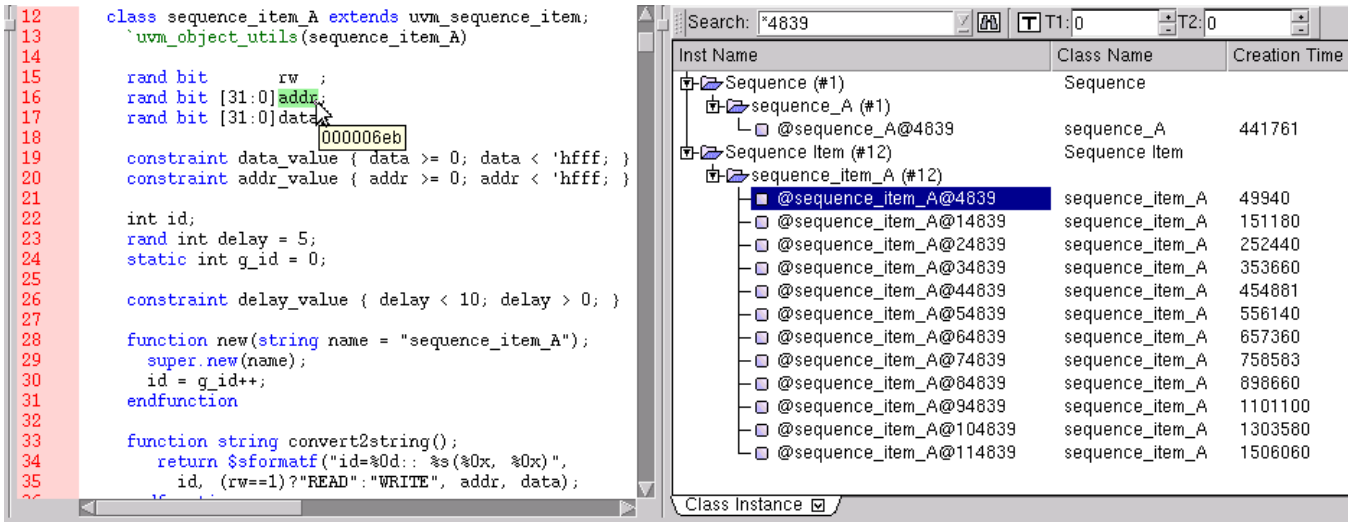


SEARCHING FOR INSTANCES BY REGULAR EXPRESSION

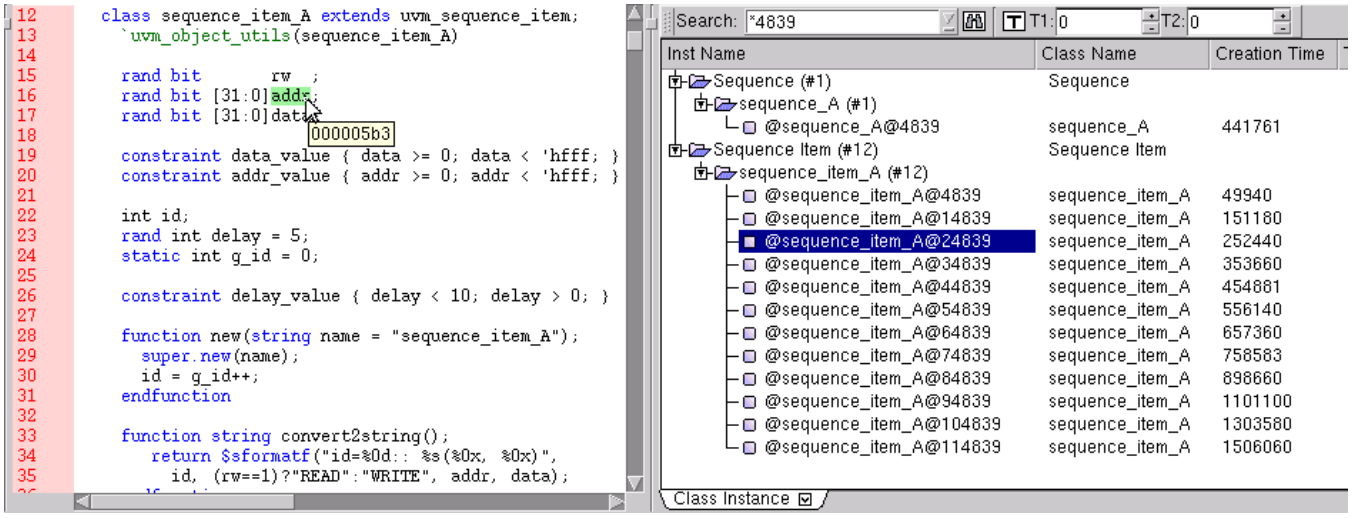
Or we can search using a regular expression. Perhaps we are interested in any sequence item type 'A,' which ends in the digit 8. Create a regular expression in the Search box – "*_item_A@*8."



Or we could search for any object ending in 4839. Change the regular expression to "*4839." The listing shows all the objects with 4839 as a suffix. Select the first sequence_item_A, and see the source code for THAT object. Hover over the 'addr' field and see the address for that transaction (that sequence item).

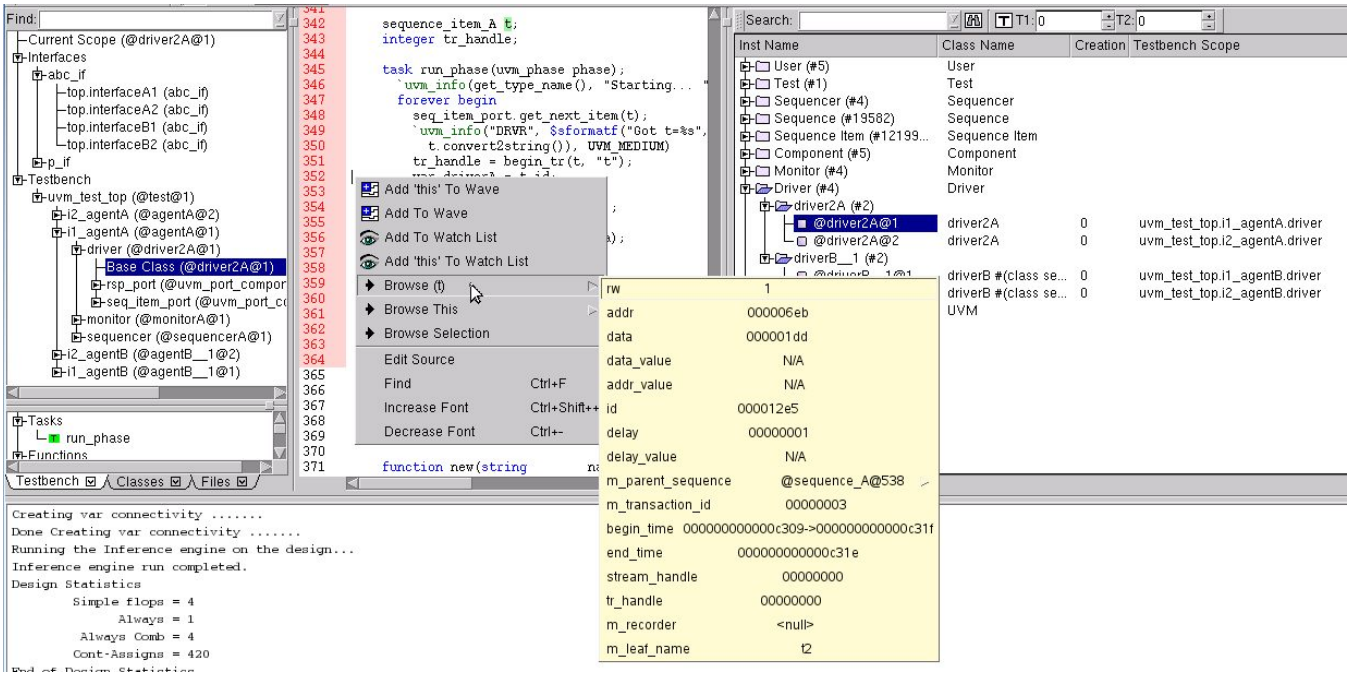


Switch to the object @sequence_item_A@24839, by selecting it. Now hover over the 'addr' in the source code window.

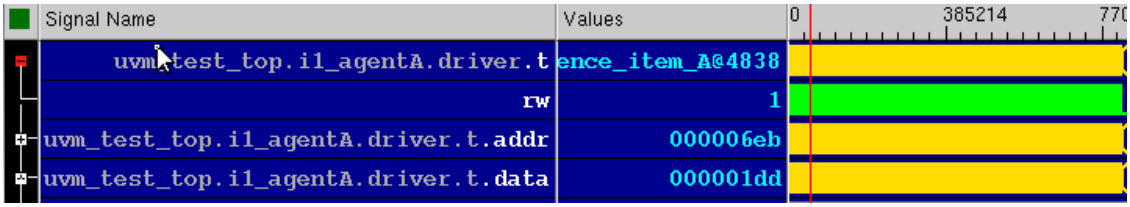


HANDLES IN OBJECTS – DRIVERS TOO

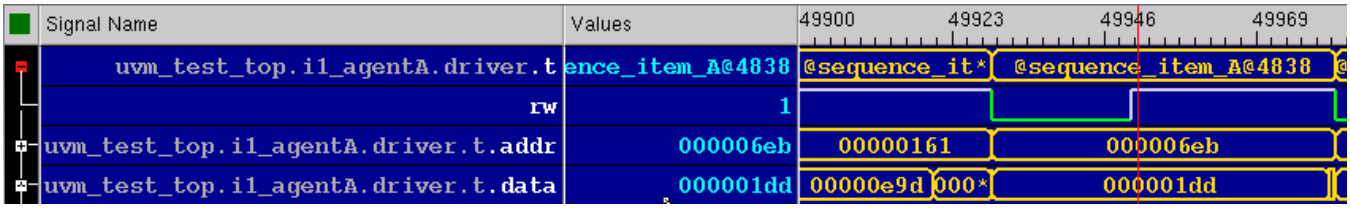
Just as with the monitor, we can select a driver from the Class Instance window (@driver2A@1). Then we go into the Base Class, and select the handle 't.' Right-mouse-button and Browse 't' shows the current value for the handle 't.'



We could instead select 't' from the driver, and "Add to Wave." Zoom fit will show us ALL the transactions that the driver received from the sequencer. That's a lot of transaction handles – fast.

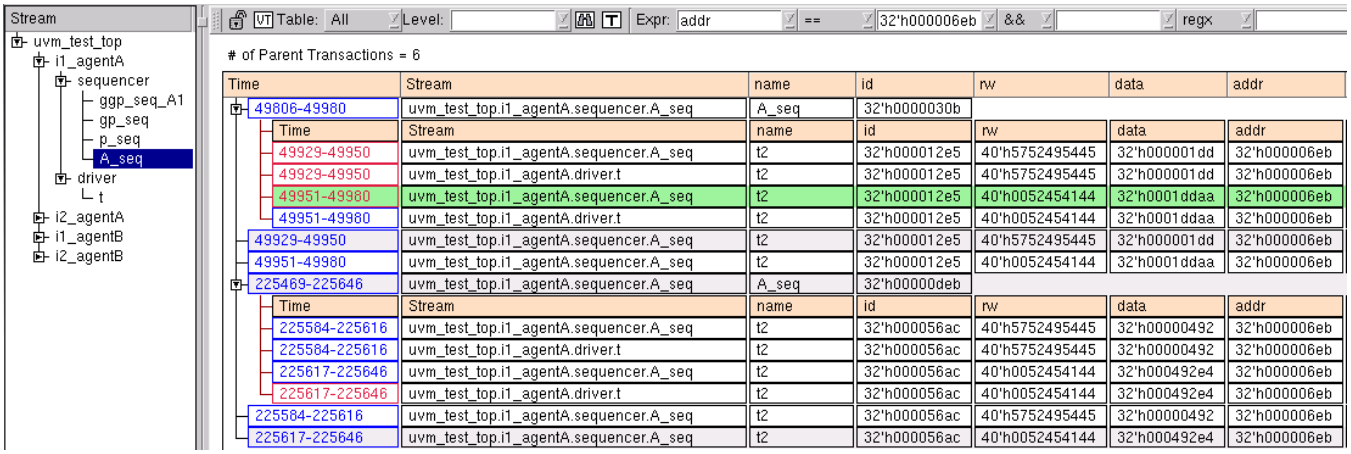


Zooming in to our handle – at time 49993.

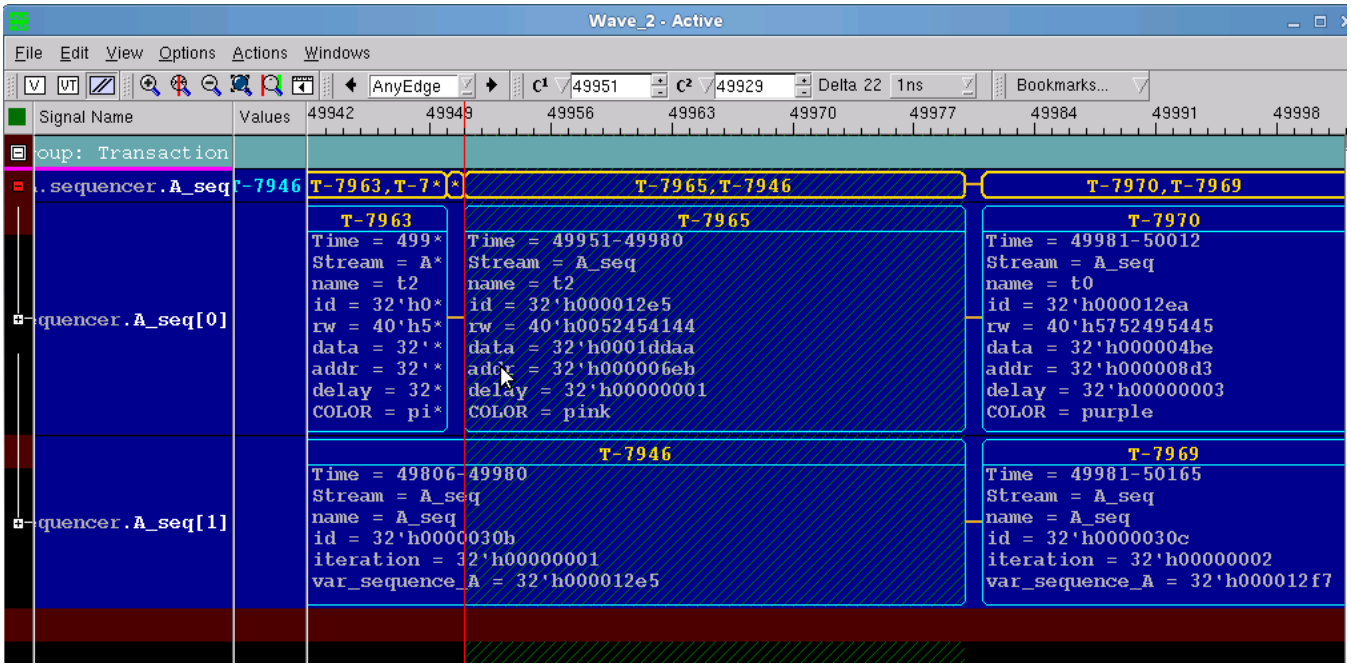


TRANSACTIONS

There are many classes and objects. If we're only interested in the transactions which have the address value 32'h000006eb, then we could open a StripeViewer, and create an expression – addr == 32'h000006eb. Then press Search.



With a quick click of "Add to Wave" the transaction stream is in the wave window. Click on a transaction (Selected green above), and see the cursor move to the transaction.



But now we're getting ahead of ourselves with Transaction debug. More on that later.

SUMMARY

Wow! These techniques are different. You get full visibility with little effort. You get windows synchronized by context – you get objects viewable over time.

Remember EVERYTHING you saw here is post-simulation. There is no live simulator connection. And I forgot to say – it's all leg smacking fast. Goodbye long coffee break while your window repaints or you re-read your OOP handbook. You can do class-based testbench debug as fast as you can think.

Just a switch to vopt and a switch to vsim; No change to your flow; Simulation speed faster than doing it the old way; Database sizes smaller; Post-simulation debug ready for you to try on your design.

I hope these UVM class-based debug techniques were of use to you. We've applied them to real customer designs and found real bugs. See http://www.mentor.com/products/fv/visualizer-debug for more information.

Happy debugging!

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics
Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
11F, No. 120, Section 2,
Gongdao 5th Road
HsinChu City 300,
Taiwan, ROC
Phone: 886.3.513.1000
Fax: 886.3.573.4734

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Garden
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

MGC 2-15 TECH12710-w