February 25-28, 2013
DoubleTree, San Jose

accellera SYSTEMS INITIATIVE

Extendable Messaging Techniques for Debugging and Analyzing UVM Testbench Structure and Transaction Flow

by

Jun Zhao, Bindesh Patel, Rex Chen

Research & Development

Synopsys Inc.

SYNOPSYS®
Predictable Success

Overview:

This presentation contains

- Introduction
- Current UVM Debug Capabilities and Limitations
- Adding New Tracing Messages into the UVM Class Library
  - Tracing Component Creation and Port Connection
  - Tracing Transaction Flow at the Port Level
- Saving UVM Message Data into a Database
- Post-processing UVM Message Data and Enhanced Visualizations
- Conclusion

Introduction

- Universal Verification Methodology (UVM)
  - A standard verification methodology
    - Reuse and interoperability
  - A SystemVerilog class library
    - Testbench template
- How to debug UVM based testbenches?
  - No VPI standard for dynamic data dumping
  - VPI also is too low-level, can incur large overhead
- Transaction level debug v.s. Code level debug
  - No standard on transaction dumping
  - Acquire transaction level debug data from UVM

Introduction (1)

- Ideal scenario
  - Attach a debug system to UVM library as an extension
  - Dynamically access and process the internal data

- The fact
  - UVM does not allow reusable extensions
  - Virtual function-based callbacks does not help
    - Only the extended class can make use of these callbacks
    - Implementations of the callbacks can be only developed inside of a specific testbench
    - Cannot be reused in other testbenches
  - System messages become very important
    - Tracing messages for phase/objection, config/resource_db, etc
    - But still not covering all the key UVM functions

Overview:

This presentation contains

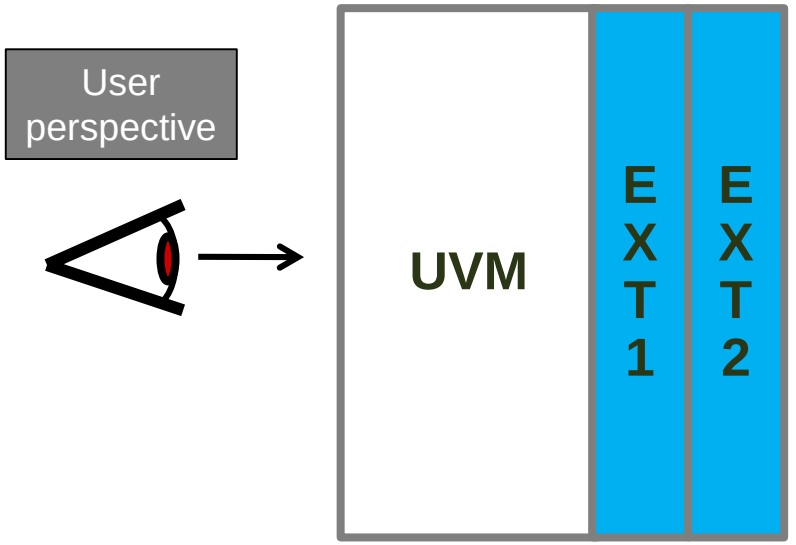
- Introduction
- **Current UVM Debug Capabilities and Limitations**
- Adding New Tracing Messages into the UVM Class Library
  - Tracing Component Creation and Port Connection
  - Tracing Transaction Flow at the Port Level
- Saving UVM Message Data into a Database
- Post-processing UVM Message Data and Enhanced Visualizations
- Conclusion

UVM Extensibility Issue

- How to add vendor-specific or tool-specific extensions?
  - For example, people may want to record the important details of UVM execution to help for post-simulation debugging
    - For end users, they have to add codes in their testbenches, but the codes may be generic to all testbenches
    - For tool vendor or design companies, they have to modify the UVM library, but to think every vendor/company have their own modified UVM libraries
  - There are features in UVM that can help but are limited
    - Transaction recorder
    - Report Catcher
    - etc

Improve UVM Extensibility

- Enable the UVM library to be extendible and the extension should be:
  - From external
  - Intact to UVM library
  - Transparent to end user
  - Stackable
- The extension can collect, process, or even modify the dynamic data in UVM during execution
- Analogy
  - PLI in Verilog



UVM Transaction Recording

- Record UVM sequence behavior and contents into preferred database via uvm_recorder
- The following *hook* functions are provided to be implemented by user or vendor:
  - uvm_create_fiber
  - uvm_set_index_attribute_by_name
  - uvm_set_attribute_by_name
  - uvm_check_handle_kind
  - uvm_begin_transaction
  - uvm_end_transaction
  - uvm_link_transaction
  - uvm_free_transaction_handle
- These *hook* functions are automatically called at the key stages of sequence generation
- Limitation: Only covers sequence part at very high level

UVM Tracing Messages

- Embedded inside of the UVM library at the major points of the execution
- Expose important runtime data into log for debug or post process
- Can be activated from command line options:

+UVM_PHASE_TRACE	turns on tracing of phase executions
+UVM_OBJECTION_TRACE	turns on tracing of objection activities
+UVM_RESOURCE_DB_TRACE	turns on tracing of resource DB access (read & write)
+UVM_CONFIG_DB_TRACE	turns on tracing of configuration DB access

- Limitations
  - Not cover all the functionalities of the UVM
  - Only output to a text format log file, difficult for post processing

Overview:

This presentation contains

- Introduction
- Current UVM Debug Capabilities and Limitations
- **Adding New Tracing Messages into the UVM Class Library**
  - Tracing Component Creation and Port Connection
  - Tracing Transaction Flow at the Port Level
- Saving UVM Message Data into a Database
- Post-processing UVM Message Data and Enhanced Visualizations
- Conclusion

Adding New Trace Messages

- Trace how the component hierarchy is built and how the ports/sockets are connected;
- Trace the UVM factory registration and override configuration;
- Trace the traffic at the TLM1 port interface and capture the pass-through transactions, requests and responses, etc.;
- Trace the TLM2 socket interface and capture the pass-through transaction (the generic payload), sync, phase, and basic protocol, etc.
- Trace the register access (read and write, mirror, etc) and how the register hierarchy has been built.

Tracing Component Creation and Port Connection

- Add tracing points where components/port are created and report/record the following information:
  - The parent full name
  - The component/port name
  - The full type name, e.g. “ubus_pkg::class ubus_master_driver”, of the component/port
  - Other component information (e.g. is port, export, or imp)

- Add report/recording points where ports are connected, and record the following information:
  - The caller port full name and port type, etc
  - The provider port full name and port type, etc

Example Code

```
function uvm_component::new (string name, uvm_component parent);
…
// Add a message whenever a new component has been created. The port component will be
// reported when creating the uvm_port_base, so won't be reported here.
    begin
        uvm_port_component_base port_component;
        if (!$cast(port_component,this))
            `uvm_info ("COMP_TRACE",{"Creating  component ",(parent==top?"":
                 {parent.get_full_name(),"."}),name,"(type=",get_type_name(),")"},UVM_LOW)
    end
endfunction

function uvm_port_base::new (string name, …);
…
// Add a message whenever a new base port component has been created.
    `uvm_info ("PORT_TRACE", {"Creating port ", m_comp.get_full_name(),
              " (type=",get_type_name(),")"}, UVM_LOW)
endfunction

function void uvm_port_base::connect (this_type provider);
…
// Add a message whenever two ports are connected.
    `uvm_info ("PORT_CONN_TRACE", {"Connecting ports ",this.get_full_name(),
              " with ",provider.get_full_name()}, UVM_LOW)
endfunction
```

Tracing Transaction Flow at the Port Level

- Add report/recording points at each port/export/imp methods like put(), get(), etc.

- Report/record the following information:
  - The request and/or response transactions
  - The return value if any
  - The function name, e.g. “put”, “get”
  - The time entering and leaving the methods
  - The port info (full name, type, recording_details, and other config data, etc)

TLM1 and TLM2 Interface

- TLM Ports

```
task put (TYPE arg);
task get (output TYPE arg);
task peek (output TYPE arg);
```

- Sequence Iteem Pull Ports

```
task get_next_item(output REQ req_arg);
function void item_done(input RSP rsp_arg = null);
function void put_response(input RSP rsp_arg);
```

- Analysis Ports

```
function void write (input T t);
```

- TLM2 Sockets

```
function uvm_tlm_sync_e nb_transport_fw (T t, ref P p, input uvm_tlm_time
delay);
function uvm_tlm_sync_e nb_transport_bw(T t, ref P p, input uvm_tlm_time delay);
function task b_transport (T t, uvm_tlm_time delay);
```

Example Code


```
// A container class that wraps the
// data to be recorded.

class uvm_port_recording_object extend
uvm_object;
    // The base port component handle
    uvm_port_component_base port_comp;

    // The port interface method name
    string func_name;

    // The transaction payload
    uvm_object req;

    // The begin time of the method call
    time begin_time;

    // The end time of the method call
    time end_time;
endclass
```

```
// The macro to be added at beginning of each port interface method.
// It initiates the container object and records the beginning time.
`define UVM_IF_METHOD_BEGIN \
   uvm_port_recording_object port_value = new; \
   port_value.begin_time = $time;

// The macro to be added at the ending of each port interface method.
// It records the method name, the base port component,the ending time,
// and the transaction payload. The uvm_report_record() method will
// call the UVM transaction recording hook functions and record the
// data into database.
`define UVM_IF_METHOD_END(req_arg,method_name) \
   port_value.func_name = method_name; \
   port_value.port_comp = m_comp; \
   port_value.end_time = $time; \
   if ($cast(port_value.req,req_arg)) \
    uvm_report_record ("PortIF", "Port level recording …", port_value);

// Add the macros to each TLM or sequence port method
`define UVM_SEQ_ITEM_PULL_IMP(imp, REQ, RSP, req_arg, rsp_arg) \
 task get_next_item(output REQ req_arg); \
   `UVM_IF_METHOD_BEGIN \
   imp.get_next_item(req_arg); \
   `UVM_IF_METHOD_END(req_arg,"get_next_item") \
 endtask \
 function void item_done(input RSP rsp_arg = null); \
   `UVM_IF_METHOD_BEGIN \
   imp.item_done(rsp_arg); \
   `UVM_IF_METHOD_END(rsp_arg,"item_done") \
 endfunction \
```

Overview:

This presentation contains

- Introduction
- Current UVM Debug Capabilities and Limitations
- Adding New Tracing Messages into the UVM Class Library
  - Tracing Component Creation and Port Connection
  - Tracing Transaction Flow at the Port Level
- **Saving UVM Message Data into a Database**
- Post-processing UVM Message Data and Enhanced Visualizations
- Conclusion

Log File v.s. Database

- Text format log file
  - Huge number of messages
  - Extraordinary large file
  - Difficult to organize and process the data
  - Hard to locate useful data

- Well-organized database with good user interface
  - Data organization
    - Predefined properties (e.g. verbosity, severity, etc.)
    - User-defined properties with values in different data types
    - Transactions and their payloads
  - User interface – a set of PLI tasks
    - Direct PLI task instrumentation
    - Use UMV report catcher to capture the messages and hook PLI tasks
    - Take advantage of UVM recorder

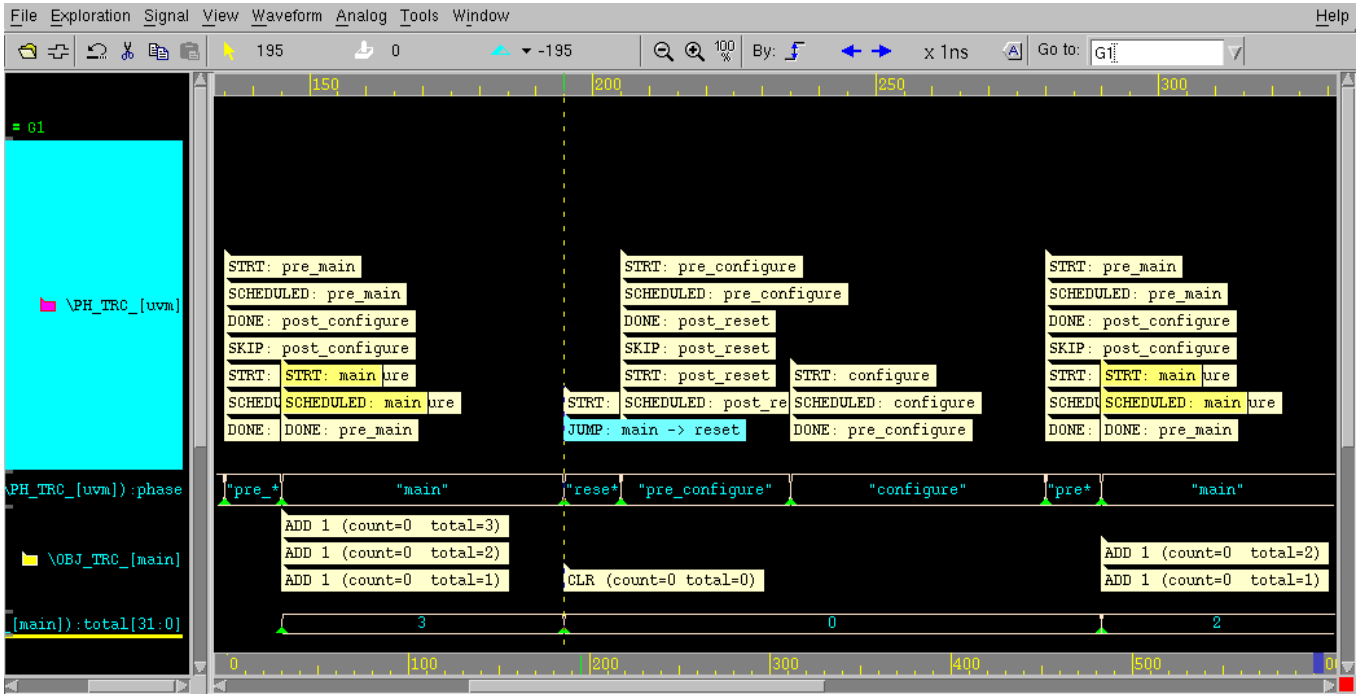
Log File Example


```
…
UVM_INFO ../../../../src/base/uvm_phase.svh(1410) @ 0: reporter [PH/TRC/SCHEDULED]
Phase 'uvm.uvm_sched.pre_main' (id=378) Scheduled from phase
uvm.uvm_sched.post_configure
UVM_INFO ../../../../src/base/uvm_phase.svh(1158) @ 0: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched.pre_main' (id=378) Starting phase
UVM_INFO ../../../../src/base/uvm_phase.svh(1235) @ 0: reporter [PH/TRC/SKIP] Phase
'uvm.uvm_sched.pre_main' (id=378) No objections raised, skipping phase
UVM_INFO ../../../../src/base/uvm_phase.svh(1387) @ 0: reporter [PH/TRC/DONE] Phase
'uvm.uvm_sched.pre_main' (id=378) Completed phase
UVM_INFO ../../../../src/base/uvm_phase.svh(1410) @ 0: reporter [PH/TRC/SCHEDULED]
Phase 'uvm.uvm_sched.main' (id=390) Scheduled from phase uvm.uvm_sched.pre_main
UVM_INFO ../../../../src/base/uvm_phase.svh(1158) @ 0: reporter [PH/TRC/STRT] Phase
'uvm.uvm_sched.main' (id=390) Starting phase
UVM_INFO @ 0: main [OBJTN_TRC] Object
uvm_test_top.ubus_example_tb0.ubus0.masters[0].sequencer.loop_read_modify_write_seq
raised 1 objection(s): count=1 total=1
UVM_INFO @ 0: main [OBJTN_TRC] Object
uvm_test_top.ubus_example_tb0.ubus0.masters[0].sequencer added 1 objection(s) to its
total (raised from source object ): count=0  total=1
UVM_INFO @ 0: main [OBJTN_TRC] Object uvm_test_top.ubus_example_tb0.ubus0.masters[0]
added 1 objection(s) to its total (raised from source object ): count=0  total=1
UVM_INFO @ 0: main [OBJTN_TRC] Object uvm_test_top.ubus_example_tb0.ubus0 added 1
objection(s) to its total (raised from source object ): count=0  total=1
…
```

Post-Process and Visualizaton

- Post-process: visualization, filtering, searching, ordering, highlighting, reorganization

- Waveform illustration for phasing execution and objection activities:



Overview:

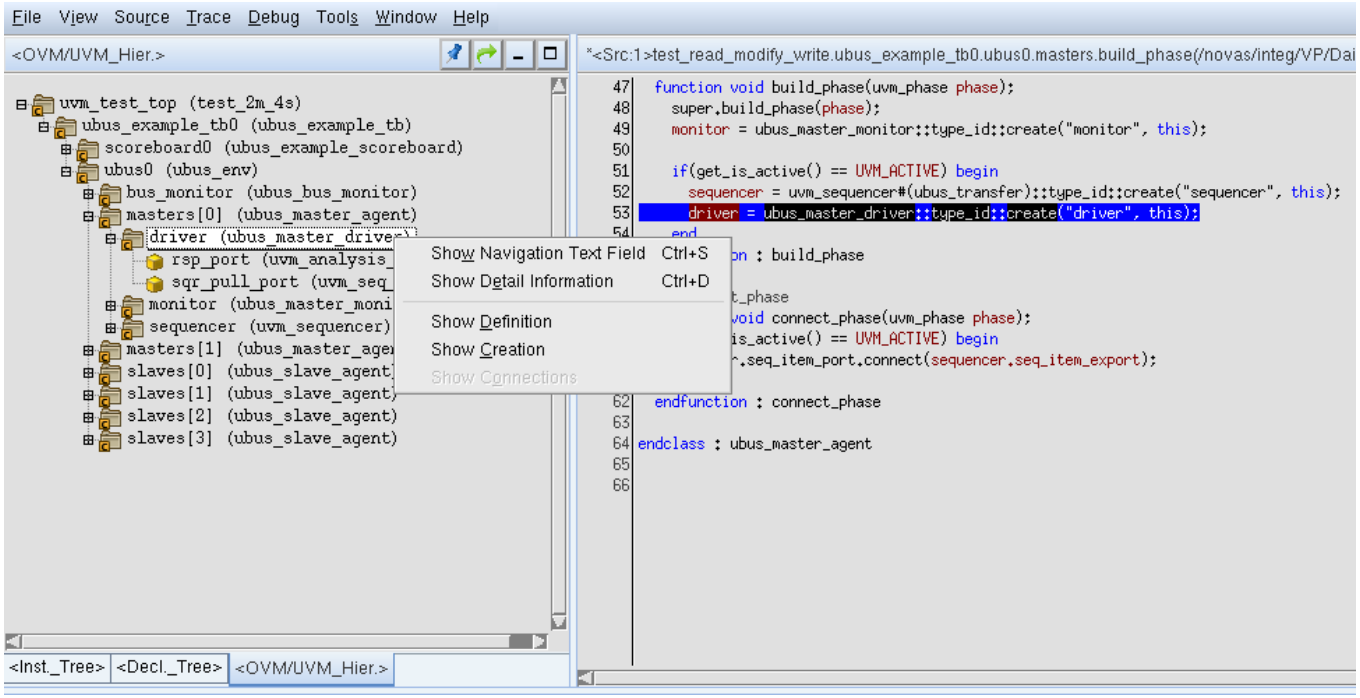
This presentation contains

- Introduction
- Current UVM Debug Capabilities and Limitations
- Adding New Tracing Messages into the UVM Class Library
  - Tracing Component Creation and Port Connection
  - Tracing Transaction Flow at the Port Level
- Saving UVM Message Data into a Database
- **Post-processing UVM Message Data and Enhanced Visualizations**
- Conclusion

Testbench Component Hiearchy

- Collect the testbench hierarchy and component parent-child relationship data from the added tracing messages where components/ports are created

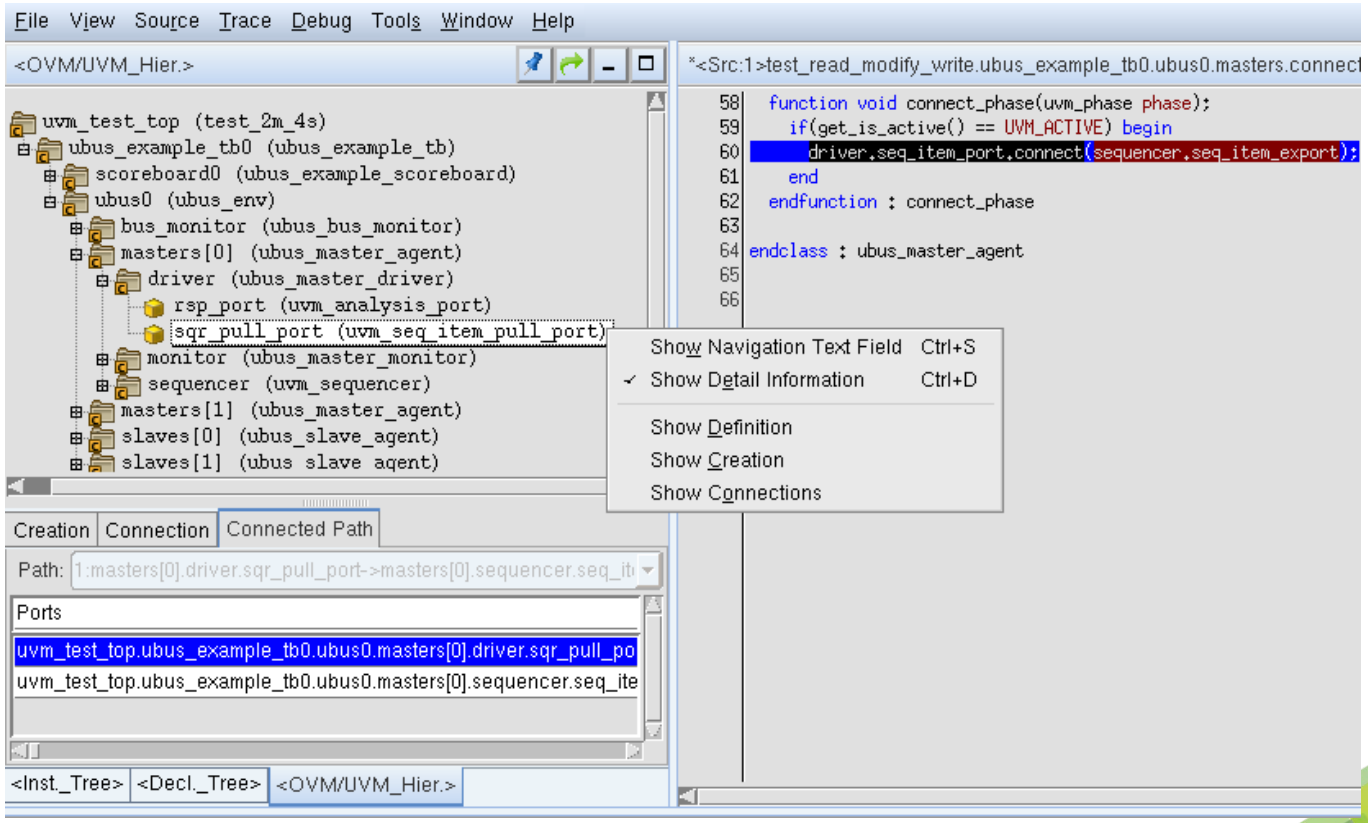
- Illustration of UVM component hierarchy tree and source code synchronization:



Port Connections

- Collect the port connection data and connection path from the added tracing messages where ports are connected

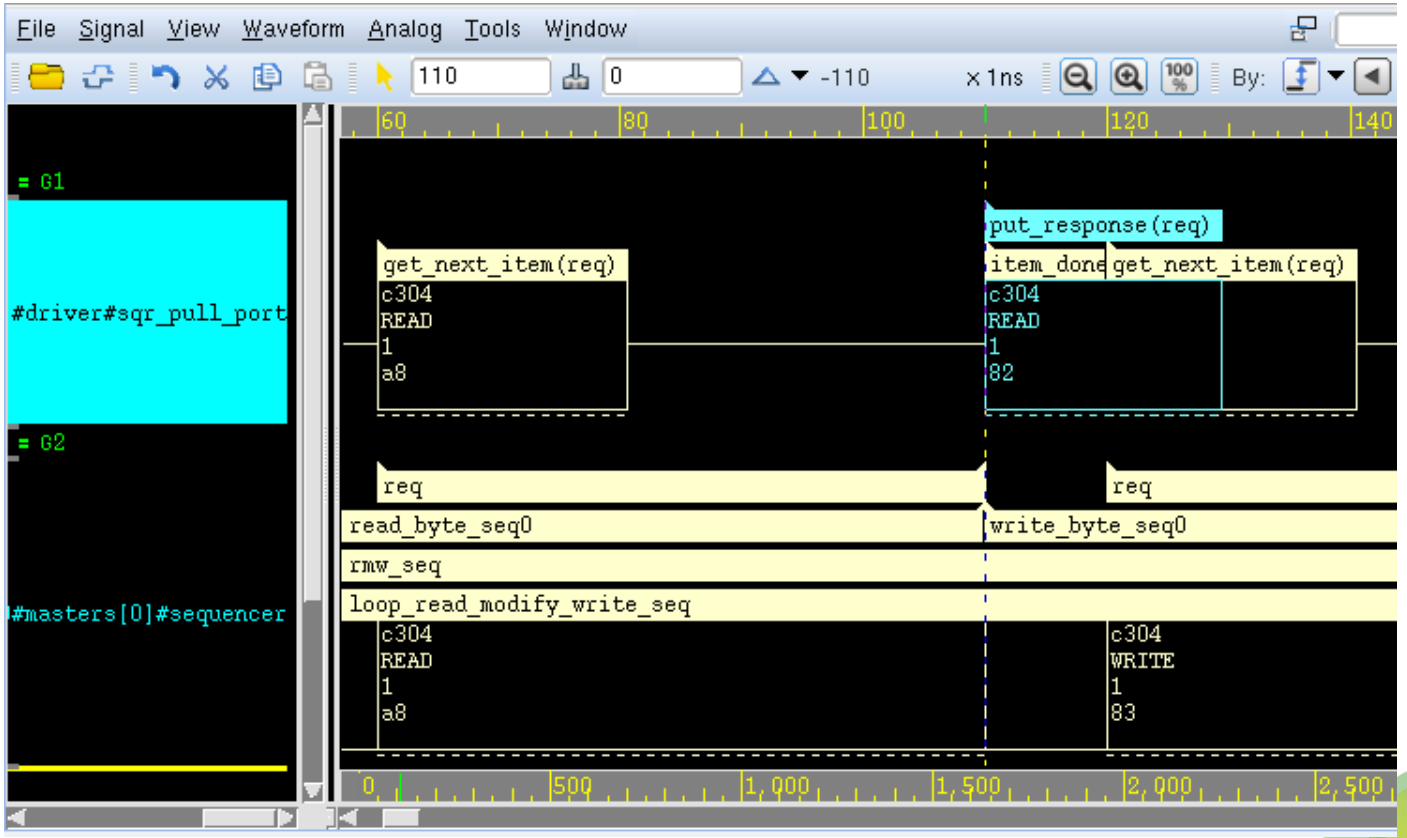
- Displaying ports and port connections in UVM hier tree:



Port-level Transaction Flow

- The data resulting from the tracing messages added for the transaction flow at the port level is captured and saved in the debug database

- Illustration of port-level transaction flow in contrast with transaction recording:



Conclusion

- Additional system trace messages, as described in this paper, should be instrumented in the UVM standard library
- UVM library should be enhanced such that the messages can be easily captured and diverted into a debug database
- Each message can be a recording point to collect internal runtime data
- Further processing of the database can enable more efficient post-simulation analysis and greater understanding of UVM testbenches