UVM Transaction Debugging
Agenda

- TLM Introduction
- Transaction visualization in Riviera-PRO
- Transaction recording in UVM
- Graphical Debugging for TLM and UVM in Riviera-PRO
- UVM-TLM simulation example demo
Introduction

- As size of typical digital design grows, you have to raise **abstraction level** while creating it.
- Higher abstraction levels can be achieved in different areas:
  - When handling individual bits is no longer feasible, you can use arrays, records/structures or even associative arrays.
  - When simple functions/tasks/procedures are not enough to manage your code, you can switch to **Object Oriented Programming** (OOP).
  - When data transferred in your design gets too diverse and too complicated, you should consider raising data transfer abstraction to **Transaction Level**...
Transactions Overview

- **Transaction** is an abstraction of information transfer.
- If you have a problem with the term *transaction*, try to replace it with *message*.
- In languages supporting OOP, a transaction is typically executed by calling a method of some design object. In other languages, it can be a procedure/function call.
- In UVM a transaction is a class object (*uvm_transaction*), that includes whatever information is needed to model the communication between two components.
- The amount and detail of the information encapsulated in a transaction is an indication of the abstraction level of the model.
Basic TLM Communication

- The most basic transaction-level operation allows one component to put a transaction to another.

- The producer generates transactions and sends them out through its **port** (green square).

- The actual implementation of the transaction is supplied by the consumer.

- The transaction implementation (in consumer) connects with requester via **export** (red circle).
Put vs. Get

- If directions of data and control flow agree, **producer puts transaction into consumer:**

- If directions of data and control flow disagree, **consumer gets transaction from the producer:**

- No matter if it is **put** or **get** situation, the process with **export** (the executor) is responsible for implementation of the transaction; requester is using its **port** to call services of the executor.
FIFOs

- Simple transaction models (direct consumer to producer connection) work OK only when data traffic is slow.

- It may be necessary for components to operate independently, where the producer is creating transactions in one process while the consumer needs to operate on those transactions in another.

- TLM FIFO is used to synchronize data flow between producer and consumer.

- So, the producer puts the transaction into the TLM FIFO fifo, while the consumer independently gets the transaction from the FIFO.
Analysis Ports

- TLM provides additional, specialized transaction port: *analysis port*. (Look for blue diamond in the diagram.)

- `tlm_analysis_port` has just one interface method `write` (void function) and can be connected to analysis exports of multiple data-collecting components (scoreboards, coverage collectors, etc.)
Transaction Recording in Riviera-PRO

- The transaction defined in the source code can be recorded in the simulation database in Riviera-PRO.
- The transaction happens on a transaction stream.
- Transactions have both their beginning and end times and can overlap one another.
Transaction Attributes

- The transaction attribute is a user-defined property assigned to a transaction.
- An attribute has a name and a value.
- Attributes can be assigned any arbitrary meaning.
Linking Signals to a Transaction

- Signals could be linked to a transaction stream.
- Easy association between the transactions and the signals.
- Linked signals are automatically traced in asdb.
Linking Transactions

- Two transactions can be related as source transaction and target transaction.
- Helps with better understanding of the data flow in the design.
- The relation interpretation is abstract and up to the user.
Using Transaction Recording

- Simple case using Riviera-PRO’s transaction recording functions.

```plaintext
begin
  task doXaction (abus ibus);
    $beginTransaction ( my_stream, $time);
    trans1 = $beginTransaction ( my_stream, $time);
    tmp = ibus.addr;
    $addAttribute ( trans1, "addr", tmp );
    tmp = ibus.data;
    $addAttribute ( trans1, "data", tmp );
    tmp = ibus.write;
    $addAttribute ( trans1, "write", tmp );
    #100
    $endTransaction ( trans1, $time );
  endtask

  initial begin
    abus t;
    my_stream = $createStream ("ABUS");
    t = new;
    t.randomize();
    doXaction(t);
  end
end
```

![Transaction Recording Example](image-url)
Transaction Recording in UVM

- Let’s take a look at UVM’s sample design: ‘Hello World’.

![Diagram of UVM design]

- The design contains two producers, consumer and FIFO:
  - 1\textsuperscript{st} producer talks to consumer directly.
  - 2\textsuperscript{nd} producer talks to consumer via tlm_fifo.
  - Producers generate randomized packets and sends them via ports.
  - Consumer receives packets and generates transactions.
class top extends uvm_component;

    producer #(packet) p1;
    producer #(packet) p2;
    uvm_tlm_fifo #(packet) f;
    consumer #(packet) c;

    `uvm_component_utils(top)

    function new (string name, uvm_component parent=null);
        super.new(name,parent);

            p1 = new("producer1",this);
            p2 = new("producer2",this);
            f = new("fifo",this);
            c = new("consumer",this);

            p1.out.connect( c.in );
            p2.out.connect( f.blocking_put_export );
            c.out.connect( f.get_export );
    endfunction

endclass
Direct Port/Export Connection

- Producer 1 connects to Consumer directly: port to export.
- Producer 1 makes call to `put()` function.
- Consumer provides implementation for `put()` function.

```
uvm_blocking_put_port #(T) out;
...
task run();
    T p;
    for (count =0; count < num_packets; count++)
    begin
        ...
        void'(p.randomize());
        out.put(p);
        #10;
    end
endtask
```

```
uvm_blocking_put_imp #(T,consumer #(T)) in;
...
task put (T p);
    lock.get();
    count++;
    accept_tr(p);
    #10;
    void'(begin_tr(p));
    #30;
    end_tr(p);
endtask
```
Connection via TLM-FIFO

- Producer 2 connects to Consumer via TLM FIFO.
- Producer and consumer operate independently.

```verilog
uvm_blocking_put_port #(T) out;
...

task run();
    T p;
    for (count =0; count < num_packets; count++)
        ...
        void'(p.randomize());
    out.put(p);
    #10;
end
endtask
```

```verilog
uvm_get_port #(T) out;
...

task run ();;
    T p;
    while(out.size()) begin
        out.get(p);
        put(p);
    end
endtask
```
‘Hello World’ - Consumer

• Let’s have a closer look at the transaction implementation in UVM starting with the Consumer.

```verilog
class consumer #(type T=packet) extends uvm_component;
...
  task put (T p);
  lock.get();
  count++;
  accept_tr(p);
  #10;
  void'(begin_tr(p));
  #30;
  end_tr(p);
...  
endtask
endclass
```

**Call stack**

- `consumer :: begin_tr(p)`
- `uvm_component :: begin_tr (uvm_transaction tr)`
- `uvm_transaction :: begin_tr`
- `uvm_recorder_aldec :: begin_tr` (overrides default uvm_recorder)
**uvm_recorder_aldec Class**

- Default **uvm_recorder** class provides methods (functions) with basic recording functionality for the transactions
  - simple output to a text file.

- **uvm_recorder_aldec** extends **uvm_recorder** with
  - Transaction recording to Aldec’s simulation database.
  - This enables visualization of transactions on the waveform.

- Using **uvm_recorder_aldec**
  - If no recorder object is instantiated in Consumer **uvm_default_recorder** will be used
  - To override instantiate **global_recorder_aldec** in Consumer:
    ```
    task run_phase(uvm_phase phase);
    . . .
    this.recorder = global_recorder_aldec;
    . . .
    ```
Attributes Recording

- A user does not have to worry about specifying each property of the transaction object he/she wants to be recorded.
- The following UVM techniques take care of it:
  1. Register object properties with `uvm_field_int` macro.

```verbatim
class packet extends uvm_transaction;
rand int addr;
endclass

class consumer #(type T=packet) extends uvm_component;
...
`uvm_component_utils_begin(consumer #(T))
  `uvm_field_int(count,UVM_ALL_ON + UVM_READONLY + UVM_DEC)
  `uvm_component_utils_end
...
```
Attributes Recording – cont.

2. Consumer calls end_tr()

```
    uvm_transaction::end_tr()
    uvm_object ::record -
      __m_uvm_field_automation () – automatically extracts all the fields
    uvm_recorder_aldec :: record_field
    uvm_recorder_aldec :: set_attribute
      $addAttribute(...); - Aldec’s PLI function to record the
    transaction attribute in the simulation database (asdb)
```
'Hello World' Example – output

{INFO}/producer.sv(46) @ 0 ns: top.producer2 [producer] Starting.
{INFO}/producer.sv(62) @ 0 ns: top.producer2 [producer] Sending producer2-0
{INFO}/producer.sv(46) @ 0 ns: top.producer1 [producer] Starting.
{INFO}/producer.sv(62) @ 0 ns: top.producer1 [producer] Sending producer1-0
{INFO}/producer.sv(62) @ 10 ns: top.producer2 [producer] Sending producer2-1
{INFO}/producer.sv(62) @ 20 ns: top.producer2 [producer] Sending producer2-2
{INFO}/consumer.sv(57) @ 40 ns: top.consumer [consumer] Received producer1-0 local_count=1
{INFO}/producer.sv(62) @ 50 ns: top.producer1 [producer] Sending producer1-1
{INFO}/consumer.sv(57) @ 80 ns: top.consumer [consumer] Received producer2-0 local_count=2
{INFO}/producer.sv(62) @ 90 ns: top.producer2 [producer] Sending producer2-3
...

www.aldec.com
Graphical Debugging in Riviera-PRO

1. Call Stack window

2. Watch window
3. Transaction Data Viewer

```
<table>
<thead>
<tr>
<th>Begin Time</th>
<th>End Time</th>
<th>addr</th>
<th>accept_time</th>
<th>initiator</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10ns</td>
<td>40ns</td>
<td>120</td>
<td>0</td>
<td>468</td>
<td>1</td>
</tr>
<tr>
<td>50ns</td>
<td>80ns</td>
<td>34</td>
<td>40</td>
<td>488</td>
<td>1</td>
</tr>
<tr>
<td>90ns</td>
<td>120ns</td>
<td>238</td>
<td>80</td>
<td>468</td>
<td>1</td>
</tr>
<tr>
<td>130ns</td>
<td>160ns</td>
<td>211</td>
<td>120</td>
<td>488</td>
<td>1</td>
</tr>
<tr>
<td>170ns</td>
<td>200ns</td>
<td>188</td>
<td>160</td>
<td>488</td>
<td>1</td>
</tr>
<tr>
<td>210ns</td>
<td>240ns</td>
<td>72</td>
<td>200</td>
<td>488</td>
<td>1</td>
</tr>
</tbody>
</table>
```

4. Waveform
Graphical Debugging – cont.

5. Breakpoints and single stepping

6. Class Viewer is coming soon...
Summary

- Start using UVM in your Testbench
- UVM provides mechanism for doing many things automatically, recording transactions is one of them
- Aldec enhances UVM recording function with graphical visualization of transactions
Aldec

**Founded**
1984 (Privately Held)

**Corporate Headquarters**
2260 Corporate Circle
Henderson, NV USA

**Office Locations**
Katowice, Poland
Kharkov, Ukraine
Paris, France
Tokyo, Japan
Shanghai, China
Taipei, Taiwan
Bangalore, India
Raanana, Israel

**Worldwide Distribution**

**Website:** [http://www.aldec.com](http://www.aldec.com)
**Tel. USA:** +1-702-990-4400
**E-mail:** sales@aldec.com

**Tel. Europe:** +33-6-80-32-60-56
**E-mail Europe:** sales-eu@aldec.com

**Webinars:** [http://www.aldec.com/events](http://www.aldec.com/events)

**Technology Patents**

- **US Patent#5,479,355:** System and method for a closed loop operation of schematic designs with electrical hardware
- **US Patent#5,051,938:** Simulation of selected logic circuit designs
- **US Patent#4,827,427:** Instantaneous incremental compiler for producing logic circuit designs
- **US Patent#4,791,357:** Electronic circuit board testing system and method
- **US Patent#7,003,746:** Method and apparatus for accelerating the verification of application specific integrated circuit designs
- **US Patent#6,915,410:** Compiler synchronized multi-processor programmable logic device with direct transfer of computation results among processors

**Products**

- **Active-HDL™**
  FPGA Design and Simulation Made Easy
- **Riviera-PRO™**
  Fast RTL Simulation Engine
- **ALINT™**
  Design Rule Checking
- **HES™**
  Emulation, Acceleration and Prototyping System

**Military and Aerospace Solutions**

- **DO-254/CTS™**
  DO-254 Functional Verification Solution
- **Actel™ Prototyping Adaptors**
  RTAX-S/SL and RTSX-SU Rad-Tolerant, Space-Flight System Designs