Reusing UVM Test Benches in a Cycle Simulator

Kristina Hager, IBM corp.

Carter Alvord, Andrew Lynch, Umer Yousafzai, Cadence Design Inc.
SOC designs require integration of diversely sourced design and test bench components.
Our challenge is to integrate off-the-shelf Verilog IP and TB with cycle simulation.

How can you use an event driven TB in a cycle based methodology without rewriting code?
Let’s explore techniques we used to create our “Hybrid Test Bench” co-simulation solution
The SystemVerilog space is wide open, so we needed to pick a TB methodology to scope down
We exploited the scope and features of the UVM methodology to enable our co-sim solution
We also took advantage of UVM’s feature set, such as phasing, to build our solution.
We extended `uvm_phase` and used objections to keep the simulators in sync.

Extensible approach to create additional sync-points, e.g., `Post_configure`, `post_main`, etc.

`Post_run` required for VIPs that don’t use run-time phases.
We extended the UVM report_server to pass messages from ncsim to HTB.
We needed an easy way to connect a test bench to the DUT and sync the simulators
Simply connect the test bench to the DUT by replacing the logic with a system task

demodule dut{
    input wire req_master_0,
    output reg gnt_master_0,
    input wire req_master_1,
    output reg gnt_master_1,
    input wire clock,
    input wire reset);

    bit[2:0] st;

    always @(posedge clock or posedge reset) begin
        if(reset) begin
            start <= 1'b0;
            st<=3'h0;
        end
        else
            case(st)
                0: begin //Begin out of Reset
                    start <= 1'b1;
                    st<=3'h3;
                end
                3: begin //Start s


```verilog
module dut(
    input wire req_master_0,
    output reg gnt_master_0,
    input wire req_master_1,
    output reg gnt_master_1,
    input wire clock,
    input wire reset);

    initial $htb_register_portlist();

    endmodule // dummy
```
However, hierarchical references into the DUT from the test bench require modification

```verilog
module foo;
    reg [31:0] a1, a2;

    MesaDut.a1 = a1;
    a2 = MesaDut.a2;
endmodule // foo

module foo;
    reg [31:0] a1, a2;

    initial
        begin
            $htb_register_write("MesaDut.a1",a1);
            $htb_register_read("MesaDut.a2",a2);
        end
endmodule // foo
```
Also, VPI cannot drive wires.
In this case the test bench must be modified

```plaintext
module dut(inout wire [31:0] cout,
           input wire en);

// won't work!
initial
  $htb_register_portlist();
endmodule // dut

module dut(inout wire [31:0] cout,
           input wire en);

reg [31:0] cout_data;
assign cout = cout_data;

initial
  $htb_register_portlist();
endmodule // dut
```
We use a VPI trick to simplify initialization

tfData.type = vpiSysTask;
tfData.sysfunctype = vpiSysTask;
tfData.tfname = (PLI_BYTE8 *) 
"#htb_register_portlist";
tfData.calltf = htb_setup_calltf;
tfData.compiletf = htb_setup_compiletf;
tfData.sizetf = 0;
tfData.user_data = 0;
 vpi_register_systf(&tfData);

The compileTF routines collect connection data.

We use an end of compile callback to process the data collected by
the compilef routines and register with HTB.

The callTF routine does nothing.

```c
int htb_setup_calltf(char * p)
{
    vpi_printf((PLI_BYTE8 *) "In htb_setup_calltf\n");
    return 0;
}
```
Custom VPI code synchronizes IUS with HTB during each simulation interval.

IUS Initialization

Compiletf routines execute
End of compilation routine execute
After Delay Callback registered

IUS Simulates

After Delay Callback executes
vpi_get_value

IUS Exports Signal Data

Mesa Imports Signal Data

Mesa Simulates

Mesa Exports Signal Data

vpi_put_value
After Delay Callback registered
We used the following VPI routines to enable the function outlined on previous slides:

- `vpi_register_systf`
- `vpi_register_cb`
  - `cbAfterDelay`
- `vpi_iterate/vpi_scan`
  - `vpiPort`
  - `vpiArgument`
- `vpi_put_value`
- `vpi_get_value`
- `vpi_get_vlog_info`
- `vpi_handle`
- `vpiSysTfCall`
- `vpi_control`
- `vpiFinish`
- `vpi_get`
  - `vpiTimePrecision`
  - `vpiType`
  - `vpiDirection`
- `vpi_get_str`
  - `vpiName`
We needed the software solution to support all simulator features, be modular, and be fast.
Using processes preserved simulator features and improved debuggability as a bonus.
We created distinct software components to achieve modularity
We relied on the posix shared memory libraries to coordinate between processes.

<table>
<thead>
<tr>
<th>Pipes</th>
<th>Producer and consumer type (FIFO) data sharing between two processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets</td>
<td>Similar to pipe, data is transferred using I/O operations between processes on local or separate machines</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Information is shared between processes on a single machine by R/W operations from a common segment of memory.</td>
</tr>
</tbody>
</table>

Fast! Versatile!
Shared memory is an extremely fast mechanism to share data between processes.

Handy reference: The Linux Programming Interface
HTBlib hides the complexity of shared memory and semaphores

Graphics obtained from:
Handy reference: The Linux Programming Interface
Many thanks to:

HTB “Beta Testers” in IBM’s System & Technology group:
  Peng Fei Gou, Yu Xuan Zhang

Additional Collaborators at Cadence Design Systems:
  Carter Alvord, Amit Kohli

Additional Collaborators in IBM’s System & Technology group:
  Ron Cash, Walt Kowalski, Wolfgang Roesner
Thanks! Any questions?