An Approach for Faster Compilation of Complex Verification Environment: The USB3.0 Experience

By

Mahesha Shankarathota (maheshas@synopsys.com)
Vybhava S (vybhava@synopsys.com)
Indrajit Dutta (indrajit@synopsys.com)

Synopsys India Pvt Ltd. Bangalore-India
Overview

- 6X compilation time gain using VCS® (2011.12) Partition Compile Methodology
- Example of a complex USB3.0 Configurable VMM Verification Environment
- Motivation and Methodology
- Coding guidelines and good practices
- Summary of Results and Conclusions
program p;
virtual SBus VirDutBus = tb.DutBus;
import TransactorPkg::*;
SBusTransctor xactor = new(VirDutBus);

initial begin
fork
begin
xactor.request();
xactor.wait_for_bus();
end
begin
wait(tb.DUT.DutBus.req == 1); //XMR
//wait(VirDutBus.req == 1);
wait(tb.DUT.DutBus.grant == 1);
//wait(VirDutBus.grant == 1); // XMR replaced by Virtual Interface
end
join
$unit::cov1.sample();
end
endprogram

module dut(SBus DutBus);
initial begin
wait(DutBus.req == 1);
dutBus.grant = 1;
dutBus.addr = 0;
dutBus.data = 15;
end
endmodule

interface SBus;
logic req,grant;
logic [7:0] addr, data;
endinterface

module tb;
Sbus DutBus();
dut DUT(DutBus);
endmodule

package TransactorPkg;
class SBusTransctor;
virtual SBus bus;
function new( virtual SBus s);
    bus = s;
endfunction

    task request();
        bus.req <= 1;
    endtask

    task wait_for_bus();
        @(posedge bus.grant);
    endtask

endclass

endpackage

// Code in $unit scope
covergroup cov;
option.per_instance = 1;
cp1: coverpoint tb.DutBus.req ;
cp2: coverpoint tb.DutBus.grant ;
cp3: coverpoint tb.DutBus.addr;
cp4: coverpoint tb.DutBus.data;
endgroup

cov cov1 = new();
$unit is the name of the scope that encompasses a compilation unit. Compilation unit refers to module, interface, package, and program blocks.

**USB 3.0 verification environment**

This environment uses VCS single compile flow for simulation.

Typedef of class where definition of class is not defined in same scope

Direct accessing of DUT signals inside the program block

Verilog Subsystem

**Verilog Subsystem**

- USB3 DUT
- USB3 PHY

Cross Partition Reference

Direct accessing of DUT signals inside the program block
Adopting Partition compile Flow

- Partitions need to be made as packages/modules/programs.
- Avoid code in $unit (code-changes in $unit scope trigger recompilation of the entire design).
- Identify the Partitions - USB3 TE partitioned well into AHB/AXI, USB3 VIP, DUT+PHY, COM, TST partitions.
- SV code modifications to remove XMRs (Cross module references) and Cross Partition References.
- The biggest challenge for us was converting all the XMRs to interface signals.
- Update of makefiles and scripts.
SystemVerilog packages provide an additional mechanism for sharing parameters, data, type, task, function, sequence and property declarations. Packages can be imported or referenced in the SystemVerilog module, interface, and program blocks.

USB 3.0 verification environment

This environment uses VCS partition compile flow for simulation.

A virtual interface allows the same subprogram to operate on different portions of a design and to dynamically control the set of signals associated with the subprogram. Instead of referring to the actual set of signals directly, users are able to manipulate a set of virtual signals.
PARTITION COMPILe FLOW STEPS

vlogan AHB_AXI_VIP_PKG INTF

vlogan USB3_VIP_PKG

vlogan USB3_RTL

vlogan USB3_TE_COM_PKG

vlogan PROG_BLK

vlogan topcfg

vcs topcfg -partcomp

Analysis of different partitions

Analysis of the configuration file

elaboration which generates simv (executable to run the simulation)

Change in program block

Change in common file of program block

Change in DUT/RTL only

Change in USB3 VIP pkg

Change in AHB or AXI pkg

For compilation from scratch, run all steps

config topcfg;
design tb work_dir.prog;
partition package AHB_AXI_VIP_PKG;
partition package USB3_VIP_PKG;
partition package USB3_TE_COM_PKG;
partition cell USB3_RTL;
partition cell PROG_BLK;
default liblist DEFAULT work_dir;
endconfig
Partition Compile Coding Guidelines

- Avoid code in $unit scope – It triggers re-compilation
- No forward references to different compilation-unit scope
- No direct access of DUT signals in program scope
- All DUT signal accesses should be through virtual interface
- No force, release constructs of the signals
No direct access of signals from TE

- All signal accesses should be done through interface signals.

<table>
<thead>
<tr>
<th>Single Compile</th>
<th>Partition Compile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tp1926.sv</strong></td>
<td><strong>DWC_usb3_if.sv</strong></td>
</tr>
<tr>
<td>begin</td>
<td>interface DWC_usb3_tb_signals_if ();</td>
</tr>
<tr>
<td>wait (</td>
<td>logic [7:0] ulpi_tx_data;</td>
</tr>
<tr>
<td>tb.U_DWC_usb3_subsys.ulpi_tx_data[7:0] == 8'h00);</td>
<td>endinterface //DWC_usb3_tb_signals_if</td>
</tr>
<tr>
<td></td>
<td><strong>ec_hst.sv</strong></td>
</tr>
<tr>
<td></td>
<td>gbl.tb_signals_if=</td>
</tr>
<tr>
<td></td>
<td>`TB_INST_NAME.tb_signals_if;</td>
</tr>
<tr>
<td></td>
<td>gbl.clk_rst_if = `TB_INST_NAME.clk_rst_if;</td>
</tr>
<tr>
<td></td>
<td>gbl.misc_if = `TB_INST_NAME.misc_if;</td>
</tr>
<tr>
<td><strong>tp1926.sv</strong></td>
<td><strong>begin</strong></td>
</tr>
<tr>
<td>begin</td>
<td>wait ( gbl.tb_signals_if.ulpi_tx_data[7:0] == 8'h00);</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>
typedef bit[3:0] TYPE;

package p1;
typedef bit[3:0] TYPE;
endpackage

import p1::*;
class update_vecs_class;
TYPE n1;
TYPE n2[4:0];
TYPE n3[3:0][3:0];
TYPE n4;
operator_class op;
task put(TYPE in[3:0]);
op = new;
n1 = in[0];
n2[3:0] = in;
n3 = '{4{in[3:0]}};
n4 = op.invert(n1);
endtask;
function TYPE get(int j);
return n2[j];
endfunction
delclass

class operator_class;
TYPE x1;
function TYPE invert(TYPE in);
x1 = ~in;
invert = x1;
endfunction
delclass

// Common Program in both flows
program p;
    update_vecs_class T1 = new;
    initial begin
        TYPE in[3:0],in0,in1, in2, in3;
in[0] = 10;
        T1.put(in);
in0 = T1.get(2);
        $display( "in data = %d get data = %d shift_data = %d\n", in[0], in0, T1.n4);
    end
endprogram
Removal of XMRs - Example

- The RAM “XX” initialization logic from the program scope is moved to Verilog Subsystem

<table>
<thead>
<tr>
<th>ad_xactor.sv</th>
<th>DWC_usb3_subsys.sv</th>
</tr>
</thead>
<tbody>
<tr>
<td>`ifdef DWC_USB3_DPRAM_PORT_EN</td>
<td>`ifdef DWC_USB3_DPRAM_PORT_EN</td>
</tr>
<tr>
<td>`ifdef DWC_USB3_RAM0_PORT1_EN</td>
<td>`ifdef DWC_USB3_RAM0_PORT1_EN</td>
</tr>
<tr>
<td>`ifdef DWC_USB3_RAM0_PORT1_EN</td>
<td>always</td>
</tr>
<tr>
<td>gbl.misc_if.dpram_initx[0] = 1'b1;</td>
<td>begin</td>
</tr>
<tr>
<td>wait(gbl.misc_if.dpram_initx[0] == 1'b1);</td>
<td>#1;</td>
</tr>
<tr>
<td>gbl.misc_if.dpram_initx[0] = 1'b0;</td>
<td>@(posedge misc_if.dpram_initx[0]);</td>
</tr>
<tr>
<td>`PRINT_NORMAL($psprintf(&quot;%0s: Re-init of DPRAM0 with X &quot;,prefix));</td>
<td>for (int i = 0; i &lt; U_RAM0_dpram.DEPTH; i = i+1)</td>
</tr>
<tr>
<td>`endif</td>
<td>begin</td>
</tr>
<tr>
<td>`endif</td>
<td>$display(&quot;DWC_usb3_xmrs:Done Re-init of DPRAM0 i=%d&quot;,i);</td>
</tr>
<tr>
<td>`endif</td>
<td>U_RAM0_dpram.mem_array[i] =</td>
</tr>
<tr>
<td>`ifdef DWC_USB3_RAM0_PORT1_EN</td>
<td>{U_RAM0_dpram.DATA_WIDTH{1'bx}};</td>
</tr>
<tr>
<td>gbl.misc_if.dpram_initx[0] = 1'b1;</td>
<td>end</td>
</tr>
<tr>
<td>wait(gbl.misc_if.dpram_initx[0] == 1'b1);</td>
<td>misc_if.dpram_initx_done[0] = 1'b1;</td>
</tr>
<tr>
<td>gbl.misc_if.dpram_initx[0] = 1'b0;</td>
<td>end</td>
</tr>
<tr>
<td>`PRINT_NORMAL($psprintf(&quot;%0s: Done Re-init of DPRAM0 with X &quot;,prefix));</td>
<td>`endif</td>
</tr>
<tr>
<td>`endif</td>
<td>`endif</td>
</tr>
</tbody>
</table>
Summary of Results

- Legacy Compile
  - single compile
  - scratch compile
  - AHB_AXI_VIP_PKG
  - USB3_VIP_PKG
  - USB3_TE_COM_PKG
  - USB3_RTL
  - PROG_BLK
  - RSIM

Compile time in minutes

- Options:
  - * Time taken for each partitions to be recompiled

12 of 13
Conclusions

- The legacy single compile flow takes about 12 minutes of compile time each time. The typical usage is change in the test (PROG_BLK), which takes about 2 minutes to be compiled. This provides a 6X gain compilation time.

- When the simulation run-time options change (RSIM), there is no compile to be done and we see maximum improvement.

- Adhering to good coding practices makes migration easier.

- Migration has overhead of maintaining both the flows in sync until partition compile flow is stable.