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

Mahesha Shankarathota (maheshas@synopsys.com)

Vybhava S (vybhava@synopsys.com)

Indrajit Dutta (indrajit@synopsys.com)

Synopsys India Pvt Ltd. Bangalore-India

***Abstract*— This paper discusses a methodology in which the compile time of a complex verification environment is brought down from 12-20 minutes to 2 minutes achieving 6X reduction in the compile time. By adopting partition compile methodology and good coding practices, an overall gain of 6X is achieved in the compilation process, thus increasing the productivity of the whole team. This paper describes the basic requirements for the partition compile methodology, motivation, our solution, and challenges faced while upgrading a complex verification environment. The good coding practices in SystemVerilog necessary to meet the partition compile requirements are presented with examples in this paper. The above flow is tested on DesignWare IP core verification at Synopsys including the DesignWare USB3.0 component. The simulator used is VCS® simulator version vcsmx-2011.12.**

***Keywords*—** Partition, Compilation, System Verilog, USB, Intellectual Property, Verification, DesignWare

## INTRODUCTION

The conventional verification techniques of SOCs and IP blocks are becoming extremely challenging due to the increasing complexities of the designs. When the time-to-market pressure of these SOCs and IP blocks are high, it becomes critical to have efficient and faster verification techniques to meet this demand. In standard verification methodologies, utmost care should be taken to achieve the shortest compile and simulation time to shorten the overall verification cycle and increase productivity.

The DesignWare USB3.0 IP verification environment consists of 5 major components namely AHB, AXI VIP (Verification IP), USB3.0 VIP, USB Physical Layer IP, and the USB3.0 Core. The USB3.0 Core is a complex IP supporting multiple modes (like Device, Host, Hub, OTG, and DRD), multiple interfaces, and all USB speeds. The legacy RAL based VMM constrained random verification environment does not have incremental compile support. This results in compiling all the components every time, taking about 12-20 minutes of compile time. A complex environment of about 2500 files for parsing and widely used by over 60 engineers causes a significant decrease in productivity. This paper presents our effort to find an efficient solution to this problem, provides implementation details and highlights the results achieved.

## I. LEGACY USB3.0 TEST ENVIRONMENT ARCHITECTURE

The USB3.0 verification environment is shown in Figure 1.The legacy compile flow uses the VCS® single compile flow where all files of the verification environment are input to the simulator. This environment consists of two top-level entities.

- A Verilog Subsystem instantiates the USB3.0 DUT and the USB3.0 Physical layer (PHY) IP.
- A program block which is the Test Environment (TE) top and instantiates all the TE components.

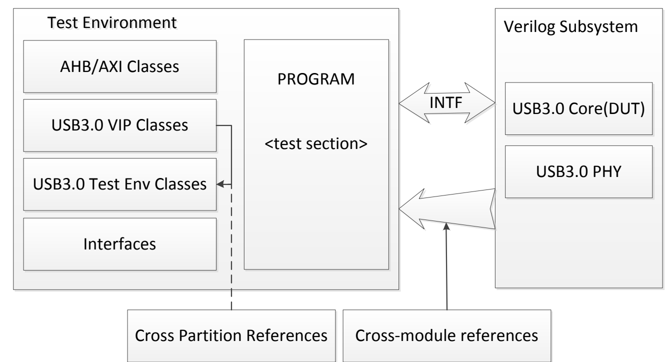


**Figure 1**: USB3.0 Verification Environment

The TE components AHB, AXI VIP, USB3.0 VIP, and numerous TE common classes reside in the compilation-unit scope in the single compile flow. In this flow, all the modules, classes, and program blocks are in a single compilation-unit. The interaction between the Verilog subsystem and the TE happens through multiple virtual interfaces. The functional verification of the design requires multiple Cross Module Reference(s) (XMR) to get access to Verilog subsystem signals in the TE domain. The flow supports the use of XMRs and forward reference of classes. For example, a reference from the USB3 VIP class to the test environment class (in Figure 1 above) is depicted as a cross partition reference.

VCS® parses all the files of the environment, compiles, and simulates. In the legacy verification flow we parse around 2500 files for a single simulation and it takes about 12-20 minutes for the compilation (analysis + elaboration). For a

change in any of the test-bench files, we need a fresh simulation to verify the change, and hence the engineer needs to wait for the same amount of compile time for each simulation. And, because this legacy setup is widely used by many engineers, it has a significant negative impact on productivity. This is a typical example of huge compile time dependency in the design verification cycle for any large SystemVerilog based verification setup. Huge compile time can shift the project delivery schedule and increase the overall product development time and finally can contribute significantly to the "time-to-market" of the product. This is the precise reason we started to look for better compilation flows to reduce the compile time.

After looking at VCS® separate and partition compile flows, we selected VCS® partition compile flow. The verification environment with well-defined partitions like AHB, AXI VIP (Verification IP), USB3.0 VIP, USB3.0 Physical Layer IP, USB3.0 DUT, and the SystemVerilog test-bench fitted perfectly into the flow.

## II. SYSTEMVERILOG CONCEPTS USED IN PARTITION COMPILE

System Verilog is rich in constructs and operators. Knowledge of SV packages, SV virtual interfaces, cross module references (XMRs), and $unit scope are necessary to adopt partition compile methodology. This section describes these concepts with examples.

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. Types, task, functions, sequences, and properties may be declared within a package. These declarations may be referenced within the module, interfaces, programs and other packages by either import or fully resolved name. In the Figure 2, package "TransactorPkg" is declared and is imported in the program "p" in Figure 3.

*package TransactorPkg;*
*import TransactorPkg::*;*

Cross-module reference (XMR) means accessing signals of other modules or accessing signals of other partitions with respect to partition compile. These XMRs are extensively used to check the status or value of the DUT or test-bench signals. One way of avoiding cross module references is by using virtual interfaces. In Figure 3, program "p" shows DutBus.req and DutBus.grant signals being accessed through XMRs.

*wait(tb.DUT.DutBus.req == 1);*

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


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();

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

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

**Figure 2**: $unit Scope, Package, and Interface Declaration

```
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);     // XMR replaced by Virtual Interface
       wait(tb.DUT.DutBus.grant == 1);
       //wait(VirDutBus.grant == 1);
     end
   join
   $unit::cov1.sample();
  end
endprogram
```

**Figure 3**: Program Block, Virtual Interface, and XMR

Virtual interfaces provide a mechanism for separating abstract models and test programs from the actual signals that make up

the design. 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. Changes to the underlying design do not require the code using interface to be rewritten. By abstracting the connectivity and functionality of a set of blocks, virtual interface promote code-reuse.
Physical interfaces are not allowed in object oriented programming, as a physical interface is allocated at compilation time itself. Virtual interfaces which are set at run time allow object oriented programming with signals rather than with only variables. Virtual interface object can be passed as arguments to task, function, class, and program. A virtual interface must be initialized before it can be used. By default it points to NULL. In Figure 3 virtual interface of type "SBus" is declared and connected to physical interface in program block "p".

*virtual SBus VirDutBus = tb.DutBus*

$unit is the name of the scope that encompasses a compilation unit. Compilation unit refers to module, interface, package, and program blocks. Its purpose is to allow the unambiguous reference to declarations at the outermost level of a compilation unit (that is, those in the compilation-unit scope). This is done through the same scope resolution operator used to access package items. The compilation-unit scope allows users to easily share declarations (for example, types) across the unit of compilation, but without having to declare a package from which the declarations are subsequently imported. Thus, the compilation-unit scope is similar to an implicitly defined anonymous package. Because it has no name, the compilation-unit scope cannot be used with an import statement, and the identifiers declared within the scope are not accessible through hierarchical references. Within a particular compilation unit, however, the special name $unit can be used to explicitly access the declarations of its compilation-unit scope. In Figure 2, the covergroup cov is declared and the object is created in $unit scope. The same object is called in program block "p" in Figure 3 by referring to $unit scope.

Definition in $unit scope:

*covergroup cov;*
*cov cov1 = new();*

Reference in program block:

*$unit::cov1.sample();*

## III. ADOPTING PARTITION COMPILE

### A. Requirements

Partition Compile follows the use model of "Unified Use Model" (UUM) of VCS® compilation. The technology is more robust in quality and performance. Partition Compile can be used across a wider variety of design flows. Following are the requirements of Partition Compile flow to achieve best turnaround time.

- Wrap test-bench code in SystemVerilog packages.
- Partitions need to be made as packages/modules/programs.
- Avoid code in $unit (code-changes in $unit scope trigger recompilation of the entire design).
- Where possible, avoid XMRs in test-bench code. XMRs prevent putting code in packages (LRM restriction). Use SystemVerilog virtual interfaces instead of XMRs.
- Do not combine source code from multiple partitions in the same file.

To adopt Partition Compile flow, the first step is to identify the partitions in the entire test environment. During development and debug of test-bench and DUT, a separate partition should be created for each module that is being changed. This way, other parts of the design which do not change often will not be recompiled. Hence external IP/VIP modules can be made separate partitions as these don't need to be recompiled for change in other components in the test-bench.
As recompile time of large partitions is higher than that of smaller partitions, the partitions created need to be balanced in terms of compile time. The recommendation is to create reasonable number of partitions such that the recompile time is within acceptable limits. Also it can be noted that if there are multiple parallel tops in the hierarchy, separate partitions can be created for each such top. A separate partition for test (program) can be created so that only the test partition gets recompiled when there is change in the test and in no other test-bench components.
Taking all these into consideration, we have come up with the following partitions for our USB3.0 test-bench:

- RTL and PHY (USB3_RTL)
- AHB/AXI VIP (AHB_AXI_VIP_PKG)
- USB3.0 VIP (USB3_VIP_PKG)
- TE common class partition (USB3_TE_COM_PKG)
- Test partition (PROG_BLK)

Partition compile flow requires a top-level configuration file. This file describes the intended partition structure. The top-level configuration file for the USB3.0 environment is shown in Figure 4. Here, "tb" is the top module name of the Verilog Subsystem and "prog" is the program block name which is the top of the entire TE. The "work_dir" is the logical name of the compiled lib directory. Module based partitioning is used for USB3_RTL and PROG_BLK and no packages are created for these modules. The configuration name "topcfg" is passed to

VCS® command line options while compiling each of the partitions.

```
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
```

**Figure 4**: Top-Level Configuration File

*B. SystemVerilog code modification*

The existing USB3.0 verification environment is modified to fit into the Partition Compile flow. The main modification is to create the partitions needed for the flow. Following changes are done to create the partitions.

- AHB, AXI VIP (AHB_AXI_VIP_PKG) – This legacy VIP is available as SV Package.
- Existing USB3.0 VIP (vmm_subenv) is moved to a package (USB3_VIP_PKG) for making this a partition.
- All TE common classes are moved to a package (USB3_TE_COM_PKG) for making this a partition.
- RTL and PHY (USB3_RTL) – this code resides within module and hence qualifies for being a partition.
- Test partition (PROG_BLK) – this code resides within program block and qualifies for being a partition.

The legacy test-bench architecture of having parallel tops is continued for Partition Compile by having the Verilog Subsystem and TE program block as the two parallel top entities. The program block now imports other TE components within this block -- AHB/AXI VIP, USB3.0 VIP and TE common class packages. Refer to Figure 6 for USB3.0 verification environment for the partition compile flow.

The next challenge we faced were the large number of Cross-module references and cross partition references used in the environment. Partition compile supports XMRs but there will be degradation in simulation time. To avoid this degradation we decided to remove the XMRs.

The following procedure was done to remove the XMRs:

- New virtual interfaces with the required signals were defined.
- Object of the virtual interface is created and connected to physical interface in program block. Object can be created anywhere in the test-bench but connection should be done in the beginning of the program block.

The term cross partition reference used in this paper refers to a forward reference created by using typedef of the class where the class is defined in different compilation-unit scope.

The example in Figure 5 shows how a cross partition reference gets created and provides one of the methods to remove this cross partition reference. In single compile flow, the declaration "*typedef class operator_class*" is needed since update*_vecs_class* uses the object of *operator_class*.If the operator_class is moved inside a package, the above typedef becomes a cross partition reference, since the class definition is present in different compilation-unit scope. For partition compile flow, to remove this cross partition reference, we have to update the code as shown on the right hand side in Figure 5. The "*typedef class operator_class*" is removed and the package p1 is imported before the *update_vecs_class*.

```
//-------Single compile flow

typedef class operator_class;
typedef bit[3:0] TYPE;

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
endclass


class operator_class;

 TYPE x1;

 function TYPE invert(TYPE in);
   x1 = ~in;
   invert = x1;
 endfunction
endclass
```

```
--------Partition Compile flow

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

  class operator_class;

     TYPE x1;
  function TYPE invert(TYPE in);
       x1 = ~in;
       invert = x1;
   endfunction
  endclass
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
endclass
```

```
// Common Program in both modes
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
```

**Figure 5**: Cross Partition Reference Example

Both the below methods were used to remove the cross partition references.

- Imported the relevant packages in the current package/partition.
- Moved the definition of the class and the usage to the relevant package.

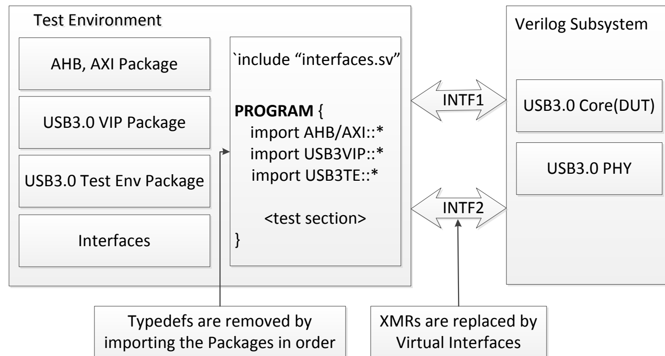


**Figure 6**: USB3.0 Verification Environment for Partition Compile

### *C. Updates in scripts*

The legacy verification environment's single compile flow is a two-step flow where all the files are compiled and then simulated. We adopted the UUM three-step partition compile flow which consists of analyze, elaborate and simulate steps. The compilation consists of analyze and elaborate steps.

The in-house Makefile infrastructure was updated to use UUM flow to incorporate incremental compilation for the separate partitions identified. We have created separate targets to help compilation of each partition.

```
---------Partition compile flow steps----------
1. vlogan  AHB_AXI_VIP_PKG  INTF
2. vlogan  USB3_VIP_PKG
3. vlogan  USB3_RTL
4. vlogan  USB3_TE_COM_PKG
5. vlogan  PROG_BLK
6. vlogan  topcfg
7. vcs  topcfg -partcomp

1 to 6 is analysis and 7 is elaboration.  Analysis and elaboration together is compilation.
1, 2, 3, 4 and 5 are analysis of different partitions.
6 is analysis of the configuration file.
7 is elaboration which generates simv (executable to run the simulation).

 For compilation from scratch, run 1 to 7.
 Change in AHB or AXI pkg  - run 1 and 7.
 Change in USB3 VIP pkg – run 2 and 7.
 Change in DUT – run 3 and 7.
 Change in common file of program block  - run 4, 5 and 7.
 Change in program block – run 5 and 7.
```

**Figure 7**: Steps Involved in the Partition Compile Flow

For compilation of different partitions, a "VCS_PC = <option>" gmake command-line option is provided. The available options are listed below:

- ALL => Compile all partitions and simulate
- VIP => Compile USB3_VIP_PKG and simulate
- COM => Compile USB3_TE_COM_PKG partition and simulate
- TST => Compile USB3 PROG_BLK and simulate
- RTL => Compile Verilog Subsystem and simulate
- RSIM => Run simulation only with different run time options

In RSIM case, none of the partitions are analyzed or compiled; only simulation is intended to be run with different run time options. So ideally this option takes 0-1 minute compile time.

## IV. CODING GUIDELINES PROPOSAL

During the process of migration to partition compile, we encountered several challenges that were a result of coding style. Based on that experience we recommend the following coding practices as a must to adopt the partition compile flow. We also believe that these are best practices for any SystemVerilog verification environment.

- Avoid code in $unit scope - code changes in $unit scope trigger recompilation of the entire design.
- Do not use forward references (cross partition references) by creating typedef of a class where the class is defined in other compilation-unit scope.

```
Single Compile:

votg_xactor.sv

typedef class otg_xfer_wts_c;
typedef class com_ec_utility_tasks_c;

class votg_xactor_c ....
endclass
```

```
Partition  Compile:

votg_xactor.sv
// The typedefs are removed and the required functionality
// moved to the otg_xfer_wts_c and com_ec_utility_tasks_c
class votg_xactor_c ....
endclass
```

**Figure 8**: Code Example for Avoiding Cross Partition Reference

In the legacy test-bench shown in Figure 8, we used typedef construct for otg_xfer_wts_c and com_ec_utility_tasks_c

classes, because compiler encounters these classes (used in votg_xactor_c class) even before they are declared. But since these classes reside in a different partition, to comply with partition compile flow, typedef declarations are now removed and the original functionality is moved to the partition that required the typedef construct.

- Direct access of DUT signals (XMRs) from the program block and test environment needs to be avoided.
  These types of accesses can be done by defining new virtual interfaces in the program domain and connecting the virtual interface to the physical interface during the initialization process in the program scope.

```
Single compile:

tp1926.sv
begin
wait ( tb.U_DWC_usb3_subsys.ulpi_tx_data[7:0] == 8'h00);
end
```

```
Partition  Compile:

DWC_usb3_if.sv
 interface DWC_usb3_tb_signals_if ();
  logic [7:0] ulpi_tx_data;
endinterface //tb_signals_if

ec_hst.sv
 gbl.tb_signals_if= `TB_INST_NAME.tb_signals_if;
 gbl.clk_rst_if = `TB_INST_NAME.clk_rst_if;
 gbl.misc_if = `TB_INST_NAME.misc_if;

tp1926.sv
 begin
wait ( gbl.tb_signals_if.ulpi_tx_data[7:0] == 8'h00);
end
```

**Figure 9**: Code Example-1 for XMR Replacement

The code example in Figure 9 shows how the XMR used in test tp1926.sv is removed in partition compile flow.  A new interface DWC_usb3_tb_signals_if with signal ulpi_tx_data is created and this signal is used in the test. The connection of virtual interface to the physical interface takes place in program block (ec_hst.sv file).

- XMRs which are not easy or possible to convert to virtual interface signals need to be handled differently. Here, the code segment may be moved to the partition where it is needed and virtual interface signals can be used to control the behavior of the logic.

```
Code in Program block:
`ifdef DWC_USB3_DPRAM_PORT_EN
 `PRINT_NORMAL($psprintf("%0s: Re-init DPRAM0 with X ",prefix));
 `ifdef DWC_USB3_RAM0_PORT1_EN
  gbl.misc_if.dpram_initx[0] = 1'b1;
  wait(gbl.misc_if.dpram_initx_done[0] == 1'b1);
  gbl.misc_if.dpram_initx[0] = 1'b0;
  `PRINT_NORMAL($psprintf("%0s: Done Re-init of DPRAM0 with X ",prefix));
`endif
```

```
Code in Verilog Subsystem:
`ifdef DWC_USB3_DPRAM_PORT_EN
 `ifdef DWC_USB3_RAM0_PORT1_EN
  always
  begin
   @(posedge misc_if.dpram_initx[0]);
   for (int i = 0; i < U_RAM0_dpram.DEPTH; i= i+1) begin
    $display("DWC_usb3_xmrs:Done Re-init of DPRAM0 i=%d",i);
    U_RAM0_dpram.mem_array[i] = {U_RAM0_dpram.DATA_WIDTH{1'bx}};
   end
   misc_if.dpram_initx_done[0] = 1'b1;
  end
`endif
```

**Figure 10**: Code Example -2 for XMR Removal

In our legacy verification setup, *dpram* residing in Verilog subsystem is being accessed and updated in TE domain using XMRs.  It is difficult to create virtual interface signals to access and update *dpram*. Hence in Partition Compile flow, the code which updates the *dpram* is moved to Verilog Subsystem and the control signals for this purpose are generated from TE domain using newly created virtual interface signals. In essence, the XMR issue is resolved while the functionality remains the same.

In the Figure 10 code example-2, the program block code sets interface signal gbl.misc_if.dpram_initx and the code in the Verilog Subsystem does the dpram initialization.

- It is not recommended to use force and release constructs in partition compile flow.

```
Single compile:

if (gbl.te_test_mode == `TE_SOC_LOOPBACK_TEST_MODE) begin
  force tb.U_DWC_usb3_subsys.U_DWC_usb3.pipe3_PhyStatus_async = 1'b0;
end
```

```
Partition Compile:

if (gbl.te_test_mode == `TE_SOC_LOOPBACK_TEST_MODE) begin
  gbl.rtl_internal_signals_if.rtl_pipe3_PhyStatus_async = 1'b0;
end
```

**Figure 11**: Code Example for Replacing Force and Release Constructs

The force constructs are replaced with simple assignments since the XMRs are now converted to virtual interface signals.

As far as possible, avoid dependencies on the parameters of other packages. This prevents the unnecessary compilation of this package when the parameters of the other packages change.

## V. SUMMARY OF RESULTS

The above flow was tested on DesignWare IP core verification in Synopsys including the DesignWare component USB3.0. In the beginning of the project, single compilation flow was adopted in the verification environment. By adopting partition compile flow to our verification environment we achieved significant improvement in compile time. Any change in the test now takes up-to 2 minutes to compile. Time taken to recompile any of the partitions is about five minutes. This achieved a gain of 6X in compile time and enhanced the productivity significantly. The total effort for the migration took about five man-months. The effort provided us with good coding practices which are useful for any verification project.
Figure 12 is a graphical representation of compilation times with different modes.

- The legacy single compile flow takes about 12 minutes of compile time each time.
- The first compile with partition compile takes about 12 minutes. We use –fastpartcomp switch to compile partitions in parallel, which takes advantage of multi-core machines. We achieved 12 minutes using 8-core machines.
- The typical usage is change in the test (PROG_BLK), which takes about 2 minutes to be compiled.
- When the simulation run-time options change (RSIM), there is no compile to be done and we see maximum improvement.
- Compilation of individual partitions takes approximately 4 to 6 minutes.

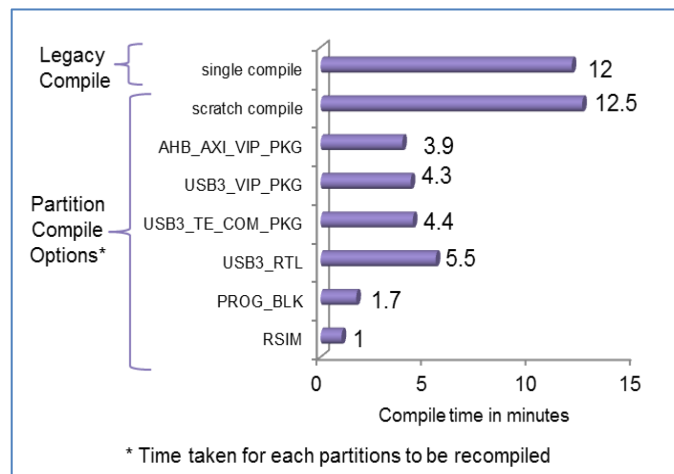


**Figure 12**: Summary of Results

## VI. CONCLUSIONS

This flow can be used as a reference flow for our future SystemVerilog verification projects.

- Adopting partition compile flow for a new project will be easy once the user adheres to the above guidelines.
- Migration of the existing complex environment becomes challenging if it is not well organized into partitions and is constantly getting updated with new changes.
- It is necessary to maintain the compatibility to the older flow in the process of migration.
- The coding guidelines are useful for any SystemVerilog verification project.

We would like to take up the following tasks in future based on the work already done for this project.

- Update the existing DesignWare USB 3.0 Core regression environment to use the partition compile flow.
- Use the partition compile flow for upcoming projects.

## REFERENCES


[I].SystemVerilog 3.1a Language Reference Manual from Accellera
[II].Partition Compile Guidelines and Recommendations – VCS® Partition compile Documents