Verilator 5.040 User Guide for UVM-Based Verification

Version 1.0

1. Introduction

Verilator is an **open-source** simulator and synthesizable SystemVerilog/Verilog compiler that converts RTL designs into optimized C++ or SystemC models. It has gained widespread adoption in both academic research and industrial verification projects due to its high-performance simulation capabilities, scalability for large designs, and integration with modern verification methodologies such as **UVM** (Universal Verification Methodology).

Unlike traditional event-driven simulators, Verilator employs a **cycle-based simulation model**, which allows for faster execution of complex designs with minimal overhead. This is particularly advantageous for regression-heavy test environments**,** where simulation speed and reproducibility are critical.

This guide provides a comprehensive workflow for engineers and verification professionals to simulate an **APB UART design** using a **UVM testbench**. It covers everything from installation, directory setup, compilation and simulation, waveform tracing, and scripting.

By following this guide, verification engineers will be able to leverage Verilator's strengths to:

- Achieve high-speed, cycle-accurate simulation of APB UART designs.
- Seamlessly integrate DUT behavior with UVM testbench components including sequences, drivers, monitors, and scoreboards.
- Implement reliable, repeatable regression tests with automated scripting and waveform generation for debugging and analysis.

This document serves both as a practical step-by-step reference and as a detailed technical guide for professional verification engineers who require a robust, open-source solution for high-performance, UVM-compliant simulation workflows.

2. Advantages of Verilator

Verilator is ideal for **large-scale**, **regression-heavy** verification, particularly where **licensing cost** is a concern.

Feature	Verilator	Commercial Tools
Cost	Free and open-source	Expensive Licensing
Simulation Speed	Cycle-based, optimized	Event-driven, slower for large designs
UVM Support	Supported via custom libraries	Native, full-featured
Scalability	Efficient for large designs	Resource-intensive for very large designs
Debugging	Console-based, VCD Waveforms	GUI-based waveform debugging

3. Installation of Verilator v5.040

This section provides detailed guidance on installing Verilator. For maximum flexibility and consistency across verification environments, it is recommended to install Verilator directly from the official Git repository. The following commands can be executed in a Linux terminal to install Verilator version 5.040, which is currently the most stable release for UVM-based simulations. (Note: The newer version 5.042 introduces compatibility issues with UVM libraries and is therefore not recommended for this flow.)

```
# Prerequisites:
sudo apt-get install git help2man perl python3 make autoconf g++ flex bison ccache
sudo apt-get install libgoogle-perftools-dev numactl perl-doc
sudo apt-get install libfl2                         # Ubuntu only (ignore if gives error)
sudo apt-get install libfl-dev                      # Ubuntu only (ignore if gives error)
sudo apt-get install zlibc zlib1g zlib1g-dev        # Ubuntu only (ignore if gives error)
sudo apt-get install libsystemc libsystemc-dev
sudo apt-get install z3                             # Optional solver
sudo apt-get install mold                           # If present at build, needed for run
sudo apt-get install lcov                           # to generate html coverage reports

git clone https://github.com/verilator/verilator    # Only first time

# Every time you need to build:
unset VERILATOR_ROOT                  # For bash
cd verilator
git pull                              # Make sure git repository is up-to-date
git tag                               # See what versions exist
git checkout v5.040                   # Switch to specified release version e.g. v5.040

autoconf                              # Create ./configure script
./configure                           # Configure and create Makefile
make                                  # Build Verilator itself
sudo make install

verilator --version                   # To Check Installed Verilator's Version
verilator --help                      # show the help
```

Verilator is now successfully installed. You may open a new terminal session and invoke the `verilator` command to begin using the tool.

4. Simulation Workflow

A Verilator-based simulation flow consists of translating the RTL and UVM testbench into a C++ executable, executing the compiled simulation binary, and optionally producing waveform traces for debugging and analysis.

4.1. Compilation Flow

To generate the C++ model and build the corresponding simulation executable in Verilator, execute the following command.

```
verilator --exe --cc --build --binary --Mdir <obj_dir_name> -o <exe_fname>
-I<dir> -sv <design.sv> <testbench.sv>
```

4.2. Simulation Flow

Use the following command to execute the Verilator-generated simulation.

```
./<obj_dir_name>/<exe_file_name>
```

5. Verilator-Compliant UVM Testbench Example

This chapter provides a detailed, end-to-end guide for compiling and simulating the **APB UART** RTL design using a **UVM-based testbench** under **Verilator 5.040**. It covers preparation steps, Verilator command usage, build processes, integration of UVM components, test execution, and debugging practices.

5.1. Overview

The APB UART verification environment comprises the following RTL and UVM-based components:

- **APB UART RTL:** The APB-compliant UART module under verification.
- **UVM Testbench Top:** A UVM-based top-level testbench module that instantiates the DUT, interfaces, and the overall testbench infrastructure.
- **UVM Environment:** Standard UVM components, including the APB and UART agents, configuration objects, and the scoreboard for result verification.
- **Tests and Sequences:** Transaction-level stimulus designed to exercise UART register accesses and validate functional behavior.

For further details on the VIP, please refer to the **`APB_UART_VIP_Doc.pdf`** located in the **`docs/`** directory of the **`verilator-uvm-apbuart`** repository.

5.2. Obtaining the APB UART Example

The complete APB UART RTL and UVM-based verification environment can be obtained from the Git repository. Users can clone the repository using the following commands:

```
git clone git@github.com:10x-Engineers/verilator-uvm-apbuart.git
cd verilator-uvm-apbuart
git submodule update --init
```

After cloning, the repository will contain all necessary directories and files. Users should ensure they have the required Verilator version and dependencies installed before building and running the simulation.

5.3. Directory Structure

The repository follows a modular directory organization for easy navigation and integration.

```
verilator-uvm-apbuart
    |
    ↳ design/          # APB UART RTL files
    |
    ↳ uvm_lib/         # Contains custom UVM library forked from ANTMICRO
    |
    ↳ uvm_tb/          # UVM testbench files
    |
    ↳ sim/             # Contains simulation scripts/database
    |
    ↳ docs/            # Documentation
```

5.4. Custom UVM Library

Verilator does not include a native UVM library, as full UVM support has not been officially provided. Consequently, to compile a UVM-based testbench, users must specify the path to a Verilator-compliant custom UVM library. For this example, we utilize a custom UVM library developed by **ANTMICRO**, which incorporates specific workarounds to address Verilator's limited UVM support. This custom library is forked from ANTMICRO's uvm-verilator repository and included as a submodule in the **`uvm_lib/`** directory of this example project.

5.5. Verilator UVM Support Limitations

Verilator does not provide full support for UVM, as several UVM constructs are either partially implemented or unsupported. Consequently, any UVM-based testbench must be Verilator-compliant to compile and simulate successfully. Certain workarounds are required to prevent or resolve warnings and errors that arise due to Verilator's partial UVM support. In this testbench, these workarounds have been implemented using conditional compilation directives such as **`` `ifndef VERILATOR.``**

The following section highlights some of the issues encountered while running this example. Users may encounter additional warnings or errors when using UVM constructs that are not yet fully supported and will need to apply similar workarounds as necessary.

5.5.1. Unsupported Modport Clocking

In Verilator version 5.040, modport clocking is not supported, which results in compilation errors such as:

```
%Error-UNSUPPORTED: */apbinterface.sv:40:22: Unsupported: Modport clocking
40 |     modport DRIVER  (clocking driver_cb , input PCLK,PRESETn);
```

```
.. For error description see https://verilator.org/warn/UNSUPPORTED?v=5.040
```

To work around this limitation, users must remove modports from the interface and rely solely on **clocking blocks**. This issue has been addressed and resolved in Verilator version 5.042.

5.5.2. `uvm_do_with Limitation

In Verilator, using more than four instances of `uvm_do_with()` within the `body()` task of a sequence can result in a **`null pointer dereference`** error during simulation, for example:

```
%Error: /uvm_tb/apb_sequence.sv:75: Null pointer dereferenced
Aborting…
```

To avoid this issue, it is recommended to replace `uvm_do_with()` with a combination of `randomize()` and `uvm_send()` methods. The following conditional compilation demonstrates the approach:

```
`ifndef VERILATOR
        `uvm_do_with(apbuart_sq,{apbuart_sq.PWRITE == 1'b1;
                        apbuart_sq.PADDR  == cfg.baud_config_addr;})
`else
        assert(apbuart_sq.randomize());
        apbuart_sq.PWRITE = 1'b1;
        apbuart_sq.PADDR      = cfg.baud_config_addr;
        `uvm_send(apbuart_sq)
`endif
```

This method ensures compatibility with Verilator while maintaining functional equivalence of the sequence operations.

5.5.3. Sequencer CLASSREFDTYPE Issue

Verilator does not always generate correct type names for certain parameterized classes. In particular, the `uvm_sequencer` class, which is parameterized with `req` and `resp` of `seq_item_type`, may not correctly propagate a single argument to both parameters. This can result in errors such as:

```
%Error:apb_uart_verif/uvm_lib/uvm-1.2-current-patches/src/seq/uvm_sequencer
.svh: 155:21: Function Argument expects a CLASSREFDTYPE
'uvm_sequencer__Tz59_TBz59', got CLASSREFDTYPE 'uvm_sequencer__Tz59'
     : ... note: In instance 'testbench_top.assertions'
  155 |   seq_item_export = new ("seq_item_export", this);
```

To resolve this issue, explicitly pass **two arguments** of the `seq_item_type` when declaring the `uvm_sequencer`, as demonstrated below:

```
`ifndef VERILATOR
       class apb_sequencer extends uvm_sequencer#(apb_transaction);
`else
       class apb_sequencer extends uvm_sequencer#(apb_transaction,
apb_transaction);
`endif
```

This approach ensures correct type resolution and compatibility with Verilator while maintaining standard UVM behavior.

5.5.4. Limited Assertion Constructs Support

Verilator provides support for **only a subset of SystemVerilog assertions**. Certain assertion constructs, such as sequences, throughout, cycle delay ranges (##()), and logical operators within sequences, are not currently supported. Attempting to use these unsupported constructs results in compilation errors, for example:

```
%Error-UNSUPPORTED: /uvm_tb/apbuart_property.sv:15:4: Unsupported: sequence
   15 |    sequence APB_WRITE_CYCLE;

%Error-UNSUPPORTED: /uvm_tb/apbuart_property.sv:16:12: Unsupported:
throughout (in sequence expression)
   16 |    (PWRITE throughout (PSELx && PENABLE)) ;

%Error-UNSUPPORTED: /uvm_tb/apbuart_property.sv:48:24: Unsupported: ## ()
cycle delay range expression
   48 |      $rose(PSELx) |->  ##1($rose(PENABLE));

%Error-UNSUPPORTED: /uvm_tb/apbuart_property.sv:127:13: Unsupported: and
(in sequence expression)
  127 |     (PREADY and APB_READ_CYCLE) |-> !($isunknown(PRDATA));
```

Users can write only assertions using constructs that are fully implemented and supported in Verilator.

Users should restrict assertions to constructs that are fully supported by Verilator to ensure successful compilation and simulation.

5.6. Compiling the APB UART RTL in Verilator

The first key step in Verilator-based simulation is translating the RTL and UVM testbench into a cycle-accurate `C++` model. This is accomplished using the `--cc` flow. During this step, Verilator generates makefiles, header files, and `C++` source files in the `obj_dir/` directory. These files collectively represent the compiled form of the APB UART RTL and TB.

Once the `C++` files are generated, the next step is to build the simulation executable using the Verilator-generated Makefile. Upon successful completion, an executable is created within `obj_dir/`, which is linked with the Verilator runtime library.

Both steps, `C++` model generation and executable build, can be combined into a single command by using the **`--build`** flag:

```
verilator $(ARG_VERILATOR) --top-module testbench_top --Mdir apb_uart_obj_dir
-o apb_uart_simv -j 1 $(DISABLED_WARNINGS) -I$(RTL) -I$(TB) -I$(UVM_LIB)
$(UVM_LIB)/uvm_pkg.sv $(UVM_LIB)/uvm.sv $(RTL)/apb_uart_top.sv $(TB)/testbench.sv
```

- **$(ARG_VERILATOR):** contains all necessary compilation flags.
- **$(DISABLED_WARNINGS):** disables warnings arising from partial UVM support in Verilator.

The table below provides a detailed description of all Verilator command-line options used to compile and build this example.

Option	Description
`--exe`	Link to create executable
`--cc`	Create C++ output
`--build`	Build model executable/library after Verilation
`--binary`	Build model binary
`--top-module <name>`	Name of top-level input module
`--Mdir <dir>`	Name of output object directory
`-o <file>`	Name of final executable
`-I<dir>`	Directory to search for includes
`-j <jobs>`	Parallelism for --build-jobs/--verilate-jobs
`-sv`	Enable SystemVerilog parsing
`--assert`	Enable all assertions
`--trace`	Enable VCD waveform creation
`--coverage`	Enable all coverage
`--timing`	Enable timing support

`--hierarchical`	Enable hierarchical Verilation
`--timescale` <timescale>	Sets default timescale
`-Wno-fatal`	Disable fatal exit on warnings
`-Wno-lint`	Disable all lint warnings
`-Wno-style`	Disable all style warnings

5.7. Running UVM Tests with Verilator

Once the DUT model is built and the UVM-testbench has been compiled and linked, the simulation is executed using:

```
./<apb_uart_obj_dir>/<apb_uart_simv> +UVM_TESTNAME=<test> +UVM_VERBOSITY=<level>
```

Key Runtime Controls:

- `+UVM_TESTNAME=<testname>` selects the UVM testname.
- `+UVM_VERBOSITY=<level>` adjusts verbosity levels.

5.8. Waveform Dumping and Debugging

Verilator doesn't support native GUI based waveform debugging. Users can use console-based UVM messaging for debugging and analysis. Users can adjust verbosity levels and observe transaction logs. Secondly users can enable VCD waveform dumps by adding **`--trace`** flag in compilation command and **`$dumpvars`** in the testbench top module.

```
initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
end
```

This **`dump.vcd`** file can be viewed in **GTKWave** or other viewers.

5.9. Automating the Build and Run Flow

A best practice for Verilator-based verification is to encapsulate RTL compilation, C++ model generation, build steps, and simulation execution within a script. In this example, the workflow is managed through a **Makefile**; however, users may alternatively use Bash, Python, or other scripting languages.

Automating the process in this way ensures consistent and reproducible results across multiple regression runs and simplifies error handling.

5.10. Summary

This chapter presented a complete workflow for compiling and simulating the APB UART RTL design using a Verilator-compliant UVM testbench. Key steps covered include:

- **C++ Model Generation:** Translating the RTL and testbench into a cycle-accurate C++ model using Verilator's `--cc` flow.
- **Executable Build:** Building the simulation executable from the generated C++ sources, either separately or combined using the `--build` option.
- **UVM Testbench Integration:** Instantiating the DUT, interfaces, agents, and sequences in a top-level UVM testbench for comprehensive verification.
- **Simulation Execution:** Running UVM test sequences with proper plusargs and optional waveform generation for debugging.
- **Automation:** Streamlining compilation, build, and simulation using scripts or Makefiles to enable reproducible regression testing.

By following this methodology, verification engineers can leverage Verilator's high-performance simulation capabilities while maintaining full compatibility with UVM-based verification environments, enabling efficient, scalable, and license-free verification of APB UART designs.

6. Verilator Limitations

While Verilator offers high-performance, open-source simulation, users should be aware of the following limitations:

- **No Native GUI Waveform Viewer:** Verilator does not include a built-in graphical waveform viewer; third-party tools such as GTKWave must be used for waveform visualization.
- **Limited UVM Support:** Certain UVM constructs are partially supported or unsupported, requiring custom workarounds for full testbench functionality.
- **Partial SystemVerilog Support:** Only synthesizable SystemVerilog constructs are fully supported; some advanced language features may not be available.
- **Two-state simulation only:** Verilator does not support 4-state logic (X/Z), which may limit certain verification scenarios.
- **Limited Assertions and Coverage Support:** Assertions are restricted to constructs that are compatible with Verilator's synthesizable simulation engine. Verilator provides limited functional coverage support.

7. Best Practices

To ensure a stable and efficient verification workflow with Verilator, consider the following best practices:

- **Pin Verilator Version:** Use version 5.040 to maintain stable UVM integration, avoiding known issues in later versions.
- **Modular Directory Structure:** Organize the project with separate directories for RTL, UVM testbench files, and generated simulation outputs to enhance maintainability and clarity.
- **Waveform Dumps:** Always generate .vcd waveform files to facilitate regression debugging and detailed analysis.
- **Automated Regression:** Use scripts or Makefiles to compile, build, and run multiple test scenarios consistently, ensuring reproducible and efficient regression testing.

8. Revision History

Version	Date	Author	Description
1.0	Dec 2025	Zeshan Ali	Initial Draft