2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Paving a Path to Hardware-Based Acceleration in a Single UVM Environment

(Because there can be only one UVM environment!)

Axel Scherer, Cadence Design Systems, Chelmsford, MA, USA (axels@cadence.com)

Mark Azadpour, Western Digial, Longmont, CO, USA (Mark.Azadpour@wdc.com)

***Abstract*—2015 marks the anniversary of Moore's law and it is still going strong. Hence we continue to enjoy the ability of ever growing integration and chip complexity. This complexity drives simulation runtime. An increasing number of users face simulation runtimes that last multi-hours or multi-days. This makes verification difficult, increasing the need for hardware-based acceleration. The problem is that verification environments are targeted for simulation, rather than for acceleration, preventing reuse. In this paper we show how to model UVM environments in a way that allows for high-speed execution for acceleration *and* simulation in a single environment.**

Keywords—UVM, UVC, Verification Environment, Simulation, Acceleration, IEEE 1800, SystemVerilog

I. INTRODUCTION

In-circuit emulation has been in use for many years, on specialized hardware systems such as emulators. These hardware systems have also been capable of accelerating a design under test (DUT) for quite some time. Typically, one would move a DUT into an accelerator when the simulation runtime becomes unbearable for verification. In the past, this only occurred for very large systems or subsystems and therefore was relatively uncommon.

In the last few years however, the increased integration and transistor budget, in combination with new complexities in the mobile, cloud and automotive space, has made simulation runtimes grow significantly. Although many aspects can be verified with short simulations, a typical regression suite will contain several *long pole* tests that could run for several hours, or even several days. These runtimes, and the compound effect of iterations, has made this a significant problem for verification closure and bug fixing, because it affects the ability to verify projects on time. Subsequently, the need and application space for hardware-based acceleration has risen significantly. Now, even large IP blocks and many sub-systems are targets for acceleration.

However, it is not trivial to adopt acceleration in a typical simulation-based environment. This is because the constraints for simulation are much looser than for acceleration. As a consequence, one needs to invest significant engineering cycles, in particular by modifying the verification environment (VE), to gain the speed advantages of acceleration. In addition, the modifications often lead to parallel development streams that often get abandoned after the project has been completed.

This paper shows methods on how to ease the transition to acceleration and to build an environment that is both optimized for simulation speed, while enabling an acceleration flow.

II. FUDAMENTALS

For a conventional Universal Verification Methodology (UVM) based environment, you have to consider that the accelerator will only be accelerating the DUT, and not the class-based verification environment. This is a key consideration to contend with.

As UVM environments by themselves can be very compute intensive, it might not make sense to target every simulation-based test for acceleration. In an ideal target the DUT and not the verification environment consumes

the bulk of the simulation runtime. In addition, runtimes should be considered in terms of hours in order to make the move to acceleration economically viable.

As a starting point, long running, multi-hour or multi-day, tests will be the targets for acceleration. As a second step, one has to determine what portion of the runtime is consumed by the DUT. This is typically done with simulation performance profiling.

A. Heavy Verification Environment – 90% VE Runtime

Let's assume you have gathered the performance profile of your longest running tests, and you see that *test1*, runs for 30 hours. In the profile, you see that 90% of the runtime is actually consumed by the VE. Your initial thought might be: *This test is not suitable for acceleration*, because if you squeeze the DUT runtime to close to 0%, you will end up with a maximum acceleration potential of about 1.11. Subsequently, you might conclude that a 27-hour runtime versus a 30-hour runtime is not a sufficient enough improvement.

However, since the overall runtime was this long, you can look at this from a different perspective. What if you could optimize your VE run in less than 30 minutes and at the same time accelerate the DUT runtime by 100x? That would be 30 minutes for the verification environment and 1.8 minutes for the DUT, for a total of 31.8 minutes, or a total speedup of over 56 times—this becomes a totally different scenario!

Now, for example, we can run this test many times a day enabling high-frequency iterations for debug purposes and coverage closure. We can even think of modifications to *test1* that would allow us to get the DUT into states that would otherwise have been unrealistic to attain. For example, a theoretical *test1'* that would consume 300 hours of simulation runtime with a 56x speedup could now become feasible, as the expected runtime would be about 5.5 hours!

B. Extremely Heavy Verification Environment – 99% VE Runtime

In the second case, we call *test2*, we also assume a total runtime of 30 hours with 99% consumption by the VE. In this case, the initial seed up potential is only 1.01, or basically negligible noise.

Now we will try even more aggressive optimization of the VE to go down from over 29 hours to 30 minutes. In this case the DUT overhead is so small that going to acceleration does not make sense. 1% of 30 hours is just 18 minutes. And, if we accelerate these 18 minutes by 100x, we end up with less than a minute. However, 48 minutes spent in pure simulation compared to 30 minutes in acceleration, is not significantly different enough to justify the use of an accelerator.

C. 50:50 Verification Environment Versus DUT Runtime

In *test3*, we assume a total runtime of 16 hours, with 8 hours consumed by the VE, and 8 hours by the DUT. Assuming that the VE has been optimized already, we could achieve a maximum speed-up of about 2x. Whether this gain is sufficient depends on multiple factors. An 8-hour runtime might actually make a qualitative difference for the project. In the morning, one could view the results of the last run, make some modifications, and start another run. By the end of the workday, this run might be completed and we now double our chances to view results and make changes. In the past, this would have been possible only once a day, for example. In some cases, this might actually be enough of an improvement to justify going to acceleration.

If the VE could be optimized further, to 20 minutes for example, we have a much better case, as the total runtime could be approximately 20 minutes (VE) + about 5 minutes (DUT) = 25 minutes total, or a speedup of 38x allowing for many iterations during the day.

D. Heavy DUT – 90% VE Runtime

In this case, the overall runtime is already dominated by the DUT. Even without much VE optimization we could achieve a speedup of about 10x. With a test consuming 30 hours, this might already be sufficient justification to move to acceleration as it brings down the runtime by one order of magnitude.

Now if we assume a *test4*, with 30 hours runtime, and where the VE can be optimized further from 3 hours to 5 minutes, we are reaching the ideal situation. In this case we have a potential runtime of 5 minutes (VE) + 16 minutes (DUT at 100x acceleration) = 21 minutes or a 77x speedup.

Speedups of over 100x are possible and realistic. It all depends on the VE and DUT contribution to the runtime.

III. THE CHANNEL FACTOR

The various cases discussed in section II are oversimplified. The runtime is not just consumed by the VE and the DUT, but also by the channel, the interface between the software-based simulator and the hardware-based accelerator. In some cases this can be a significant factor, in particular if there is a lot of traffic occurring between the VE and the DUT.

However, the ideal test is DUT heavy and has minimized the traffic between simulator and accelerator.

This paper will not discuss the various forms of channel interfaces. Instead it will show one basic approach and a verification environment that does allow for various, more sophisticated channel options. It will also focus on the principals on how to build Universal Verification Components (UVCs) and VEs that can support simulation and acceleration in one environment.

IV. THE IDEAL TEST

The ideal test should be built to optimize the test intent, while achieving maximal acceleration performance. Such a test should have most of the action in the DUT, while the UVM environment is used to load memories and registers, to start basic operations and eventually determine the pass/fails status in a standardized UVM manner. Furthermore, the ideal test minimizes run-time checking both for protocols as well as for data via scoreboard and instead performs post-mortem checking so that interactions between the VE and DUT are minimized.

The ideal test executes as follows:

1. Load memories using *back-door access* (see section VIII).
2. Start the clock and deactivate the *reset* signal.
3. Load the register using a front-door bus interface.
4. Start the device operation.
5. Start threads to determine end of test
6. Check for *Pass/Fail* status.

If used properly, a test like this could be rerun in case of failure by enabling the scoreboard to find the cause at an earlier time in the test. These aspects will be discussed in section X.

V. CLEAN SIGNAL AND TRANSACTION SEPARATION

In classic UVM testbenches, the driver and the monitor, and in non-ideal situations even other components or objects, directly interact with signals of the DUT model. This means the testbench contains both transaction-based content and signal based content. This setup works, however it slows down hardware acceleration dramatically. Instead, we want to marry the conceptual benefits of keeping the testbench purely transaction-based, as well as keeping the resulting performance benefits.

One of the fundamental problems of acceleration is the requirement to minimize the communication overhead between a simulator and an accelerator. Every single interaction with a signal in the testbench requires synchronization. Hence, if we move from individual signal driving and monitoring to transaction based data

interaction, the associated overhead and amount of synchronization is reduced significantly. This will reduce the burden on both the VE and the channel leading to much faster acceleration potential. The bus functional model () associated with driving and monitoring will be in a top module associated with the DUT.

VI. Reset and Clock Control

As stated in the previous section, the testbench must not reference any signals. As the clocks are the highest frequency signals in the environment, it is particularly important not to have any clock references in the testbench. This means that clocks must not be in an interface UVC or, in associated sequence items, sequence clocks must also never be referenced.

We will need to be able to control signals associated with a particular interface as well as signals controlling the clocks and resets in the device.

To do so, we introduce a UVC specifically targeting the control of these signals. The actual signal driving will happen in a module instance inside the top module associated with the DUT. The UVC is controlling the generation instead by setting value in an associated SystemVerilog interface. In other words, our VE will not have direct access to clock events and therefore can run in parallel to the DUT without the need to constantly sync up. The virtual sequence of the UVM VE starts off of the clock in the VE and then the clock activity runs purely in the hardware domain.

```
clock_and_reset_seq = clock_and_reset_sequence::type_id::create("clock_and_reset_seq");
clock_and_reset_seq.clock_period =  10;
clock_and_reset_seq.reset_delay  =  30;
clock_and_reset_seq.run_clock    =   1;

clock_and_reset_seq.start(p_sequencer.clock_and_reset_sqr);
```

In the example above, we show that the factory creates a sequence, three values are being set, and then the sequence is started. This clock and reset initiation could be put into a macro to look like this:

```
`clk_rst_start(10, 30)
```

VII. Basic Channel

In a single UVM VE that supports acceleration and simulation, we need to minimize other traffic between the VE and the DUT besides clock and reset. As discussed, ideally we would only send transactions or data, instead of accessing signals between the two domains, and we would minimize the number of these transactions.

In a classic UVM environment the driver and monitor have very frequent interactions with DUT signals. In fact, the driver contains the BFM that sets values on various signals, and the monitor reads collections of signals to extract from the signal-level to the transaction-level. Every such interaction is costly.

As a consequence, we are now making the UVM VE completely signal free by having it reside exclusively in the transaction-level domain. The signal interaction shifts from the driver and monitor to the SystemVerilog interface. These interfaces then run in the accelerator. The interaction between simulator and accelerator will become truly be transaction-based, reducing the events between the two by one or two orders of magnitude for standard industry protocols.

In the example below we have moved the driving BFM from the UVM driver to an interface task call.

Once the UVM driver receives the transaction it will no longer drive signals, instead it will request that the interface, by way of the virtual interface handle, drives the DUT.

```
// Inside SystemVerilog Interface
task drive_transaction(
    instruction_type_e instruction_type,
```

```
    int input_addr,
    int input_data
  );
    case (instruction_type)
      WRITE: $display(" driving a WRITE:  address 'h%h, data 'h%h!!",
                      input_addr, input_data);
      READ:  $display(" driving a READ:   address 'h%h, data 'h%h!!",
                      input_addr, input_data);
     endcase
     if (instruction_type == WRITE) begin
      @(posedge clk);
      @(posedge clk);
      addr    <= input_addr;
      data_i  <= input_data;
      @(posedge clk);
      req     <= 1'b1;
      @(posedge clk);
      @(posedge clk);
      while (ack == 0) begin
         @(posedge clk);
      end
       @(posedge clk);
      req     <= 1'b0;
      addr    <=   '0;
      data_i  <=   '0;
    end
     // SNAP
  endtask
```

Similar to the UVM driver, the monitor will no longer operate on signals. It will also request that the interface starts to collect transactions and then it will send the collected information back to the monitor.

```
    task collect_transaction(
        output instruction_type_e instruction_type,
        output int  output_addr,
        output int  output_data
      );
      @(posedge clk);
     // SNAP
     endtask
```

On the UVM side, these tasks will be called on their virtual interfaces and exchange data.

VIII. Memory Interactions

One of the biggest contributors to test traffic is interaction with memory. In modern devices, the memory space is large and the amount of data required to start an operation, or to analyze the result of a test, can be very large. If you use the bus interface to load memory before traffic starts, or to read the result into memory at the end of a test, the overhead can dominate the overall runtime. Hence, one should access the memory in the hardware-based accelerator directly, and load and read it using a so-called *front-door operation*. For this purpose, accelerators have special APIs to access memory efficiently.

In a write or read operation one associates the DUT hierarchy to the memory element directly with a UVC that transfers the data.

IX. Signal Level Protocol Compliance

Signal level protocol compliance checking should not occur in the UVM monitor. Instead, the checks should be implemented as SystemVerilog Assertions (SVA) inside the SystemVerilog interface. SVA provides full syntax to implement protocol compliance. Additionally, these checks can run both in acceleration as well as in formal analysis. The example below shows a check for enable that cannot occur without select.

```
master_no_enable_without_select: assert property (@(posedge pclk)
   disable iff (!presetn)
      !(!psel && penable));
```

X. ENHANCED CONFIGURABILITY

As stated initially in this paper, VEs are typically built for simulation in mind, operating under a loose set of constraints. Further on, VEs often have limited configurability. To optimize speed in general, and for acceleration in particular, it is important to have a sophisticated and flexible configuration mechanism.

A test should be able to turn almost any aspect of the VE on or off. In particular, a test should be able to configure any type of checking, such as dynamic data checking in a scoreboard, protocol and data checking, post mortem checking, and so on. It also needs to control any form of coverage. Finally, a test should be able to choose between different ways of randomization in order to enable high-speed constraint solving.

```
master_no_enable_without_select: assert property (@(posedge pclk)
  disable iff (!presetn || !checks_enable)
     (!psel && penable));
```

For example, a VE should not only push the protocol compliance checking from the VE to the SystemVerilog interface to avoid frequent signal interaction. It should also control whether those checks should run or not. Hence, the disable clause is not only sensitive to typical signals, such as reset, but also has an enabling bit called `checks_enable`. The UVC that drives and monitors this interface will also control the `checks_enable` bit according to the needs of the test. Similarly, it may control a `coverage_enable` bit if signal level coverage is collected.

In typical UVC architecture a primary configuration field called `is_active` is built into the agent. This field only controls whether or not an agent is driving. We want to expand configurability to incorporate control of monitoring separately. This way, you can generate stimulus without having to monitor it for high-speed simulation or an acceleration run. In the example below, we added a field `monitor_active` to conditionally build the monitors. The same condition will be used when connecting the analysis ports of the monitors.

```
// in build_phase
if(agent_cfg.monitor_active == UVM_ACTIVE) begin
  monitor = apb_monitor::type_id::create("monitor",this);
  monitor.agent_cfg = agent_cfg;
end else
      `uvm_info(get_type_name(), "Monitor disabled via config", UVM_LOW)
```

A VE should provide all the hooks to allow you to write a test that can trade off visibility and performance. This is not limited to just checking, coverage, or constraint levels, but includes the VE scope of granularity as well. A VE should allow any use case: High speed, full debug, partial debug, and any others. This also means that the VE needs to set certain configuration bits in the interface. Specifically, the UVM agent needs to pass not only relevant enabling configuration information down via its virtual interface such as `checks_enable` `coverage_enable` but also other values that are important for the operation of driving or monitoring. In simple cases, just a few fields might need to be set. In more complex cases, a struct might be used to pass along a complete configuration set.

```
vif.set_config(cfg_s);
```

Subsequently, the UVCs and top-level environments need to contain the right fields in their configuration objects and act accordingly. For checking and coverage, this might be straightforward. For constraint modeling, this might be more alien to your everyday simulation engineer. However, there are classes of tests where constraint solving should be limited. Where either the transaction should be pre-calculated for speed, or where only a subset of constraints should be active. For example, it might not make sense to randomize data values, in particular for large and complex transactions and protocols.

XI. PEFORMANCE IMPLICATIONS

The modifications to the UVM VE, and its building blocks manifested in the UVC, shown in this paper do not negatively impact simulation performance. In fact, we have seen that it actually improves performance while at the same time it opens up significant performance gains when moving the DUT into the accelerator.

XII. SUMMARY

As soon as long pole simulations reach a few hours they can become bottlenecks for verification closure and debugging, as they affect iteration time and therefore productivity significantly. In the majority of such cases it is possible to speed up run time significantly using verification, or by up to two orders of magnitude in realistic scenarios. However, this requires some work if you start with a conventional simulation focus. This paper showed an approach that used a few adjustments to get you to acceleration much more efficiently in a single UVM verification environment, without sacrificing anything in the pure simulation space. You will neither loose speed, convenience nor flexibility. Instead, you are gaining the option to improve productivity.

REFERENCES

[1] D. Cohen, P. Edstrom, A. Scherer, “Developing a single environment for software simulation and hardware acceleration” Cadence Design Systems, April 2015.

[2] A. Scherer, “Will my UVM simulation accelerate?” Cadence Design Systems, March 2015

This template has been prepared and adapted for use in DVCon Europe 2015.