2015 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Accelerating RTL Simulation Techniques

Lior Grinzaig, Verification Engineer

Adva Optical Networks, Tel Aviv, Israel

liorgrinzaig@gmail.com

Abstract— Long simulation run times are a bottleneck in the verification process. This article presents a variety of methods for combating this performance issue: utilizing different tools, such as SystemVerilog properties; understanding of the design as a system, making changes in different levels of implementation such as line/block/macro level. This generally speeds up the full regression, or, in some situations, at least the debug run. The article also presents several tips on how to do analysis for performance issues.

Keywords— performance; efficiency; simulation; simulator; acceleraion; time; runtime; turn around; system; analysis; test; verification; coding style

I. INTRODUCTION

Long simulation run times are a bottleneck in the verification process. A lengthy delay between the start of a simulation run and the availability of simulation results has several implications:

- Long turn-around times cause the code development (design and verification) and the debug process to be slow and clumsy. Due to the long time, some scenarios become not feasible to verify on a simulator and must be verified on faster platforms — such as an FPGA or emulator, which have their own weaknesses.
- Engineers must make frequent context-switches, which can reduce efficiency and lead to mistakes.

Coding style has a significant effect on simulation run times. Therefore it is imperative that the code writer examine their code, not only by asking the question “does the code produce the desired output?” but also “is the code efficient, and if not, what can be done to improve it?”

Previous papers have examined the effect on simulator performance when implementing the exact same functionality while using different coding styles[1], or emphasize the different coding style approach that is needed when the traditional verification environments are migrating from Verilog to SystemVerilog and are implementing the widely spread UVM methodology[2].

While this paper gives some optimizations methods based on the abilities of SystemVerilog, it is also adding another layer of performance optimization that comes from the understanding of the design as a system, and optimizations that are made specifically for debug runs purposes.

This article also presents several tips on how to analyze for performance issues.

TWO TYPES OF CODE MODIFICATIONS

There are two types of code modifications that can accelerate simulations: changes in line/block level (which we will call micro level) and changes in the module or component level (which we will refer to as macro level).

II. MICRO CODE MODIFICATIONS

A. *Sensitivity Lists/Triggering Events*

The key thing to remember about a sensitivity list at an *always block* or a trigger event at a *forever block* is that when the trigger occurs, the simulator starts to execute some code. This is trivial and fine in the functional sense, but when considering code efficiency, it is desirable to determine when a signal can be exempt from the sensitivity list or which event should be chosen for triggering.

1) Synchronous example:

Consider the following example (counting the number of transactions over a synchronous bus):

```
always @(posedge clk)
begin
  if ((VALID == 1) && (READY ==
1))
  begin
    count++;
  end
end
```

The above code is the most intuitive way to implement the counting code. It is intuitive since this is how it would have been written in the design. Notice, however, that during the time the clock is toggling and no transactions are present on the bus, the *if* condition is unnecessarily checked, over and over again.

Now consider the following adjusted code:

```
initial begin
  forever
  begin
    wait ((VALID == 1) && (READY == 1))
    count++;
    @(posedge clk);
  end
end
```

You can see that this code functionally counts the same thing, but much more efficiently, with respect to the number of calculations needed. This implementation realizes that in a system, a bus is in idle state a significant percent of the time, allowing us to achieve performance optimization.

The exact speedup of this change cannot be calculated using a simple formula since it depends on several factors, such as the effort of a particular simulator on a given machine to execute the count++ relative to the other commands, and the ratio between the idle cycles and the cycles with actual transaction. Nevertheless, in order to get an idea on the speedup potential, when using a ratio of 1:3 transaction to idle, the result was that the "each cycle" code took 47% more time to be executed than the alternative code[1].

2) Asynchronous example:

The following is taken from actual code found inside an actual IP. It is a BFM code of an internal PHY. For this example, the code has been edited to use only eight phases; the original code included 128 phases.

[1] All of the measurements in this article have been taken from code running on Questasim 10.4

```
wire          IN0 = IN;
wire #(25) IN1 = IN0;
wire #(25) IN2 = IN1;
wire #(25) IN3 = IN2;
wire #(25) IN4 = IN3;
wire #(25) IN5 = IN4;
wire #(25) IN6 = IN5;
wire #(25) IN7 = IN6;

always @(*)
begin
  case (DELAY_SEL)
    4'd0 : OUT = IN0 ;
    4'd1 : OUT = IN1 ;
    4'd2 : OUT = IN2 ;
    4'd3 : OUT = IN3 ;
    4'd4 : OUT = IN4 ;
    4'd5 : OUT = IN5 ;
    4'd6 : OUT = IN6 ;
    4'd7 : OUT = IN7 ;
  endcase
end
```

Examining this code carefully shows that for each change of IN, the *always block* is invoked eight times. This is due to the cascading changes of the INx signals: IN0 changes at "t" invoke the *always block* that initially processes the case logic; then IN1 changes at "t+25" and invokes the *always block* again, and so on, until IN7 invokes it at "t+175". Remember that the code originally supported 128 phases, so for each change there where 128 invocations. The case itself was composed of 128 options, ***and*** this module was implemented on every bit of the PHY's 128-bit bus!

This resulted in a complexity magnitude of ~$M*N^2$ (where M is the number of bits in the bus, and N is the number of phases).

Now, consider this adjusted code:

```
wire          IN0 = IN  &&  (DELAY_SEL=='d0);
wire #(25)   IN1 = IN  &&  (DELAY_SEL=='d1);
wire #(50)   IN2 = IN  &&  (DELAY_SEL=='d2);
wire #(75)   IN3 = IN  &&  (DELAY_SEL=='d3);
wire #(100) IN4 = IN  &&  (DELAY_SEL=='d4);
wire #(125) IN5 = IN  &&  (DELAY_SEL=='d5);
wire #(150) IN6 = IN  &&  (DELAY_SEL=='d6);
wire #(175) IN7 = IN  &&  (DELAY_SEL=='d7);

always @(*)
begin
  case (DELAY_SEL)
    4'd0 : OUT = IN0 ;
    4'd1 : OUT = IN1 ;
    4'd2 : OUT = IN2 ;
    4'd3 : OUT = IN3 ;
    4'd4 : OUT = IN4 ;
    4'd5 : OUT = IN5 ;
    4'd6 : OUT = IN6 ;
    4'd7 : OUT = IN7 ;
  endcase
end
```

Based on the system assumption that the delay configuration is not configured simultaneously with the modules' functional data flow, we have reduced the code complexity to M*N. Actually, if we do not care in our simulation about the "analog delay" on the bus, we can simply write OUT=IN and reduce the complexity to M only.

To emphasize the importance of being efficiency aware, this simple code change alone, having reduced the calculation complexity to M*N, accelerated some full-chip tests (SoC of ~40M gates) by a factor of two!

B. Wrong or inefficient modeling:

The following code is a small part of a memory model.

```
logic [31:0] mem [(2<<20)-1:0];

always @( negedge rst_n)
begin
  if (!(rst_n))
    for ( i=0 ; i < (2<<20) ;  i++ )
      mem[i] = 0;
end
```

This example code seems fine, but it can actually be optimized as well.

This is a large array and looping over one million entries will take a long time. Fortunately, this time can be saved during the initial reset of the chip (before the memory is filled) by masking the first reset *negedge* — as the array is already filled with zeros. Beyond that, however, a different approach can be applied. Using an associative array instead of a fixed array enables the array to be nullified with one command, instead of by using a loop:

```
logic [31:0] mem[*];

always @( negedge rst_n)
begin
  if (!(rst_n))
    mem= {};
end
```

Even with a relatively small memory with 256 entries, the efficiency of the implementation with an associative array code is 10 times better.

C. Time tracking:

Many times we want to execute code after some time has passed from a previous event. There are two options to keep track on the time that have passed. The first option is by using explicit delays by the '#' operator. This approach has one major drawback – what if the clock's frequency is not known in advance, or can be different in a future project? Therefore, instead, the clock cycles counter is usually used:

```
int cycleDelay=100;
event startDelay; //triggered by some logic

initial
begin
  @( startDelay);
  repeat (cycleDelay) @(posedge clk);
  $display("%t %0d count", $realtime, cycleDelay); //just some action
end
```

However, this is very wasteful manner to track time, since a code is being executed each clock cycle. Instead, consider the following code using a more sophisticated way of using the '#' operator:

```
int cycleDelay=100;
realtime samplePosedge1, samplePosedge2, period;
event startDelay; //triggered by some logic

initial
begin
  @(posedge clk);
  samplePosedge1 = $realtime;
  @(posedge clk);
  samplePosedge2 = $realtime;
  period = samplePosedge2 - samplePosedge1;
  -> startDelay;
end

initial
begin
  @(startDelay);
  #( period * ( cycleDelay – 2 ) ); //since 2 clocks were "wasted" on the sampling
  $display("%t %0d * delay", $realtime, cycleDelay); //just some action
end
```

Using this delay method consumes only a tiny fraction of simulator resources compared to the cycle count method.

III. MACRO CODE MODIFICATIONS

A. *Using Different Code for Development and Debug*

In a design or a verification environment that is composed of many different components, where not all of them must be active at all times, it may be beneficial to eliminate parts of the code that are not essential for the majority of the tests.

If the code is a module in the design itself, a "stub" or a simple BFM can be created to replace that module, and then a *generate if* block with an *else* option is added. Depending on the global parameter added during simulation time, the simulator will generate the real code or the simplified code. Engineers can decide when to use which option. If they want a specific test to always use the simplified code, set the parameter at the regular *run* command. Alternatively, if they want to support a simplified model during "debug mode" only, the parameter is set only when debugging or developing, but not when running the full regression.

Code example for using a *generate if* block:

```
generate if ( ! CLKDIV_DIGRF_SIMPLIFIED_MODEL )
begin :package_model //simulation will take longer time
  clkdiv_digrf   u_pll_clkdiv
  (
    .CKOUT_624M  (CKOUT_624M),
    .CKOUT_499M  (CKOUT_499M),
    .CKOUT_416M  (CKOUT_416M),
    .CKOUT_312M  (CKOUT_312M),
    .CLKIN       (CLKIN)
  );
end
else
begin : simplified_model
  clkdiv_digrf_simplified   u_pll_clkdiv
  (
    .CKOUT_624M  (CKOUT_624M),
    .CKOUT_499M  (CKOUT_499M),
    .CKOUT_416M  (CKOUT_416M),
    .CKOUT_312M  (CKOUT_312M),
    .CLKIN       (CLKIN)
  );
end
endgenerate
```

Please note that each begin-end pair is named differently, so the module path will be different depending on the parameter value. This is relevant when probing into that module, so the probing path should be also depended on the parameter value.

If the code is not in the design, engineers can use the *generate if* method as well, or simply add a parameter that interacts with the testbench component directly to disable the component. Alternatively, if the code is of a *class* type, the parameter may be used to prevent the component creation.

For example, in the universal verification methodology (UVM), the configuration object of an agent should hold variables indicating whether to create subscribers for the monitor, and even indicating whether to run or disable the monitoring of the monitor itself.

By using these types of methods we have managed, with minimal effort, to speed up a SoC (10M gates) environment by a factor of 10.

B. System Modes

In some cases, leaving some modules in a reset state, or with no clock, is a valid design mode. Even when it is not a valid system mode, if the test(s) are not affected by it, *force*s can be used to override the normal behavior. Again, this can be controlled by a parameter.

In a design with a complex clock scheme, engineers may try to find the best clock ratios that are relevant for that type of test. If the test depends on cores, it may help to increase the core clock frequency. If the test depends on DMA activity, the core frequency can be reduced when the core is idle. It is good practice to choose ratios that are used by default for most of the tests and make adjustments to only specific ones.

IV. PERFORMANCE ANALYZING TOOLS

A problem with trying to optimize the performance of a simulator is that it is often impossible to know exactly where the bottlenecks are and what is slowing it down. As shown in the code examples above, sometimes even small – almost negligible – code changes can have a large effect on the entire simulation. How can one find this code among millions of code lines and inside of IPs, especially if others wrote them?

Performance analyzing tools, provided by simulator vendors, are used to identify the parts of the code that consume the most cycles of the simulator. As a side note, since there is a correlation between the usage of the

simulator calculation resources and the power consumption of the chip, it is sometimes possible to find design bugs in the early stages of the project.

A. *Simulation Phases*

When using analysis tools, it is best to perform different analysis for the different stages of the simulation (i.e., different time-frames). These stages include the out-of-reset phase, the configuration phase, and the run phase (which can be further sub-divided). Using small time frames produces more accurate analysis per simulation stage, since different parts of the design are active at these different stages; thereby consuming different simulator resources. Conversely, examining the simulation run globally makes it harder to analyze the design for anomalies.

B. *Acceleration Measurements*

After finding the critical components in the code that affect simulation time and finding the right solution for them, it is recommended to measure the benefit from those optimizations. Here are some tips regarding those measurements:

- When measuring, know exactly what is being measured. For example, when measuring simulation run time, do not include the time required to load the simulator software and do not include the time taken to load the design code into the simulator. These times may be important, and may be optimized as well, but they are irrelevant for this type of calculation.
- Make real comparisons: compare A to A, not A to B.
    - *Random elements* Usually, engineers disable random elements by using the same random seed to simulate the same scenario. However, there are cases where the change itself is the cause of a different random generation result. For such cases, using the same seed is not recommended; instead, continue using random seeds along with a statistical analysis method as described below.
    - *Statistical analysis* The run-time can be affected by external things, beyond the content of the code, such as, server usage by other processes or data in the server's cache. For a good comparison, run the compared test several times (20–30), with and without the change, and compare the average times. Also check that the standard deviation is reasonable (around 20% of the average). If there are abnormal measurements, restart the process or throw away specific "off the chart" measurements that may be the result of some specific server problem.

V. CONCLUSION

Slow simulations are not necessarily decreed by fate. Engineers as well as managers should pay attention to the importance of coding efficiently for simulation as well as the different ways to analyze simulations and tackle simulation bottlenecks.

REFERENCES

[1] Clifford E. Cummings "Verilog coding styles for improved simulation efficiency", ICU 1997

[2] Frank Kampf, Justin Sprague and Adam Sherer "Yikes! why is my SystemVerilog testbench so slooooow?", DVCon 2012