Dynamic Scheduling in Verilator

CHIPS Deep Dive Cafe Talks, 2021-06-15
Karol Gugala, kgugala@antmicro.com
Towards UVM support in open source tooling
ANTMICRO

- Founded 2009
- Turning ideas into software-driven products
- Industrial IoT and embedded systems: AI/ML in defense/security, mining, agriculture, autonomous vehicles, robotics, aerospace, industrial automation
- We use, develop, advocate open source
- Member of Linux Foundation, Zephyr Project, CHIPS Alliance, OpenPOWER Foundation, Strategic Founding member of RISC-V International
- Introducing new design methodologies and workflows based on open source
WHAT WE DO

HARDWARE
Proof of Concepts (PoC), demonstrators, prototyping, open source platforms

SOFTWARE & AI
OS porting, building BSPs, build systems, device management, edge & cloud AI

FPGA & ASIC
Custom IP blocks, SiP development, soft SoCs, heterogeneous processing systems

TOOLS
Tools, new software and hardware development and testing methodologies
WHAT WE DO

See our technology showcase on antmicro.com
ANTMICRO OPEN SOURCE PORTAL

Check out our Open Source portal!

https://opensource.antmicro.com/
WHY DO WE NEED UVM / SYSTEMVERILOG SUPPORT IN OPEN SOURCE TOOLS?
INNOVATE BY BRIDGING EXISTING METHODOLOGIES

- Sustainable results can be achieved by not excluding the existing pool of designs and designers
- Need to make it easy to combine and remix methodologies that have been impossible to use together
- There are new open source tools and methodologies that can radically improve the ASIC design space

Big progress has been made to enable that with important milestones already achieved - the latest being dynamic scheduling support in Verilator - but a lot more work is needed!
WHAT MAKES UVM/ SYSTEMVERILOG SUPPORT IN OPEN TOOLS IMPORTANT?

- Necessary to combine commercial ecosystem with open source tools and methodologies
- Chip-making companies can benefit from open source while keeping their existing UVM codebase
- Proprietary licensing of existing tools makes it hard to build scalable, reproducible CIs
- Number of open source cores and lots of pre-existing IP implemented in SystemVerilog, e.g.
  - SweRV
  - Ibex
WHAT IS MISSING?
SV-TESTS

- We created a test suite to determine the SystemVerilog support level in various open source tools
- Aims to pinpoint all the supported and missing SystemVerilog features in various tools
- Generates report from last passing master build at symbiflow.github.io/sv-tests
- Introduces three types of tests:
  - Testing individual features as per the SystemVerilog standard
  - Existing third party test suites
  - Selected open source IP cores, such as SweRV, Ibex and others
<table>
<thead>
<tr>
<th>Test</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibex RISC-V core</td>
<td>1/1</td>
</tr>
<tr>
<td>RSD RISC-V core</td>
<td>1/1</td>
</tr>
<tr>
<td>Various sanity checks</td>
<td>1/1</td>
</tr>
<tr>
<td>SCR1 RISC-V core</td>
<td>1/1</td>
</tr>
<tr>
<td>Swerv RISC-V core</td>
<td>1/1</td>
</tr>
<tr>
<td>UVM tests using assertions</td>
<td>1/1</td>
</tr>
<tr>
<td>Particular UVM classes</td>
<td>1/1</td>
</tr>
<tr>
<td>UVM-random</td>
<td>1/1</td>
</tr>
</tbody>
</table>

**Notes:**
- **ibex**: Full ibex core test
- **RSD**: RISC-V system tests
- **SCR1**: System C tests
- **Swerv**: System Verilog tests
- **UVM**: System UVM tests
- **UVM-random**: System UVM-random tests

**Build Information**

```
build:/lowrisc/ibex/ibex_simple_system_0/src/lowrisc_list_common_0.1/tools/verilator/common.wt
1 // Copyright lowRISC contributors.
2 // Licensed under the Apache License, Version 2.0, see LICENSE for details.
3 // SPDX-License-Identifier: Apache-2.0
4 //
5 // common waiver rules for verilator
6 //
7 // Do not warn about unconnected pins in module instantiations, e.g.
```
EVENT DRIVEN SIMULATIONS

- One of the biggest missing pieces in open source open source UVM simulations is fast, time aware and event driver simulator
- Verilator is the fastest simulator available on the market
  - The missing pieces was time awareness and events support
- To address this we decided to reimplement the way Verilator handles simulation scheduling
STRATIFIED SCHEDULER

- SystemVerilog simulation time is divided into time slots which are further divided into regions.
- Different types of statements fall into different regions:
  - Normal assignments are in Active,
  - Non-blocking assignments are in NBA,
  - Concurrent assertions are in Observed, etc.
- These regions are not simple divisions of time; the simulation can go back to an earlier region within a time slot, e.g. if a non-blocking assignment triggers an active block.
- Some events can get delayed to a following time slot.
DYNAMIC SCHEDULING

• So far, Verilator has partially implemented the stratified scheduler, but only by statically ordering the generated code
• That is not sufficient in cases where we cannot predict (during compilation) what events get scheduled or delayed and when it happens (such as delays or forks in a deep call stack, or in a virtual function)
• UVM requires a more dynamic approach that allows scheduling events at runtime
OUR APPROACH

- We completely changed the way Verilator schedules events
- Now each process is run in a separate thread
- The processes communicate with each other using standard synchronization primitives such as mutexes and condition variables
- The scheduling of their execution is left to the OS scheduler
PROCESSES AND THREADS

Dynamic Scheduling in Verilator

- Process A
- Process B
- Process C

Thread pool

- Thread 1
- Thread 2

- Event triggered
- Sensitivity list triggered
- Waiting on event
- Delay

Antmicro
Dynamic Scheduling in Verilator

SIMULATION STEP

previous time slot

Run all processes

all processes are idle

Perform non-blocking assignments

need to re-run some processes (e.g. a sensitivity list was triggered)

Execute Postponed region (e.g. $strobe)

next time slot
WHAT IS ALREADY POSSIBLE?
DELAYS

- Delay controls allow you to postpone the execution of a given statement by the specified amount of time.
- This simple example of generating a clock wasn’t possible in Verilator before (clocks would be generated in the driving C++ code created by the user).

```verilog
module t;
    logic clk;
    initial forever begin
        clk = 1'b0;
        #1;
        clk = 1'b1;
        #1;
    end
endmodule
```
EVENT VARIABLES

- Event controls and event triggers allow processes to communicate with each other.
- Here we have an example of two processes triggering each other in each time slot.
- The `event.triggered` method is available as well. It returns 1 if the event has been triggered in a given time slot, 0 otherwise.
- You can also use events in block sensitivity lists and trigger them from within a block.

```verilog
module t;
  event ping;
  event pong;

  initial forever begin
    @ping;
    #1;
    ->pong;
  end

  initial forever begin
    #1;
    ->ping;
    @pong;
  end
endmodule
```
module t;

initial begin
  fork
    begin
      $display("forked process 1");
      end
  begin
      $display("forked process 2");
      end
  begin
    $display("forked process 3");
    end
  join
  $display("join in main process");
$finish;
endmodule

• Forks are used to spawn multiple concurrent processes
• A possible output for this example:
  forked process 2
  forked process 1
  forked process 3
  join in main process

  (depends on the order of execution of the forked processes)
OTHER TYPES OF JOINS

- All types of joins are supported:
  - `join` (wait for all children to finish)
  - `join_any` (wait for any one child)
  - `join_none` (do not wait, continue the main process immediately)

- Output for this example:
  - `join_none in main process`
  - `forked process`

(main process continued execution without waiting)

```verilog
dynamic scheduling in verilator

module t;
  initial begin
    fork
      begin
        #1 $display("forked process");
        $finish;
      end
    join_none
      $display("join_none in main process");
    endmodule
```
WAIT STATEMENT

- Wait statements block the execution of a process until a specific condition is met
- This condition can be any SystemVerilog expression
- In this example, one process waits for the other to assign the values it expects to some variables

```verilog
module t;
    int a = 0;
    int b = 0;
    int c = 0;

    initial begin
        #1 b = 1; // First wait
        #1 a = 2;
        #1 c = 4; // Second wait
        #1 b = 5; // Third wait
    end

    initial begin
        wait(a > b);
        wait(a + b < c);
        wait(a < b && b > c);
        $finish;
    end
endmodule
```
CONstrained RANDOMIZATION

- Not related to scheduling, but needed for UVM
- Allows you to generate random values constrained by certain conditions
- Extremely useful for testing: you can easily generate a huge number of test inputs that still make sense due to your constraints
- Only basic constraints are supported so far; the next step is to add support for a real solver

```verilog
typedef enum bit[7:0] { ONE, TWO, THREE, FOUR } Enum;

class Cls;
  constraint A { a inside {ONE, THREE}; }
  constraint B { b > 8; b < 42; }
  rand Enum a;
  rand logic[7:0] b;
  rand int c;
endclass

module t;
  Cls obj;
  initial begin
    int rand_result; obj = new;
    rand_result = obj.randomize() with { c < 64; };
    assert(obj.a inside {ONE, THREE});
    assert(obj.b > 8 && obj.b < 42);
    assert(obj.c < 64);
    $finish;
  end
endmodule
```
DYNAMIC SCHEDULER EXAMPLES

- We’ve created a public repository showing example usage of all the new features we added to Verilator
- The repository is available on GitHub [github.com/antmicro/verilator-dynamic-scheduler-examples](github.com/antmicro/verilator-dynamic-scheduler-examples)
- The CI there runs simulations of all the examples in the repository
WHAT ELSE NEEDS TO BE DONE?
REAL-WORLD OPEN SOURCE UVM

- Ongoing work on extending Verilator with SystemVerilog features required by UVM:
  - Stratified scheduler
  - Randomize methods
  - Full Class support
- Big milestone achieved recently introducing stratified scheduling into Verilator
  - This still requires some work - especially in runtime optimization
- By introducing the concept of delayed execution to Verilator we have enabled event driven simulations
MERGING TO MAINLINE

- The dynamic scheduling work needs to be merged to Verilator mainline
  - The first step is to introduce a mechanism for choosing the scheduler
- Necessary for commercial entities to start adopting open source tooling in their verification suites
- This step is anticipated by Verilator maintainers, who helped achieve this goal and the project community
NEXT STEPS

• Extending Verilator with support for more SystemVerilog features used in UVM such as:
  ▫ Clocking blocks
  ▫ Assertions
  ▫ Class parameters
  ▫ Cyclic randomization
  ▫ Unpacked structs
• Runtime optimization
• Integration with UHDM work done in parallel
• Identifying issues preventing integration with verification test suites used by different projects like e.g. riscv-dv and addressing them
FUTURE GOALS

- Long-term goal is to enable full UVM support in Verilator
  - The plan is to choose a real life design and implement all the functionality required to run its validation
  - The preference is to focus on an open source SoC design like SweRV or OpenTitan
- Open source design verification will lead to modular, collaboration-driven chip design workflows
THANK YOU FOR YOUR ATTENTION!