ETH zürich

Institut für Integrierte Systeme
Integrated Systems Laboratory

Department of Information Technology and Electrical Engineering

EFCL Winter School 2026

Sample Solution Exercise 02

Simulation with Verilator

F. K. Gürkaynak
Prof. L. Benini

Last Changed: 2026.02.09

1 Overview

In this exercise, we will explore the practical application of *Verilator 5*, the leading open-source tool for hardware simulation. Verilator converts Verilog code into an executable binary that can then be run to simulate the design. Our focus will be on simulating a digital design and testbench written in *SystemVerilog*, first with a small example design and then, to get an understanding of the Croc SoC, we move on to a more complete testbench with Croc.

We will learn:

- How to translate a SystemVerilog design into an executable binary using Verilator.
- How to specify signals to track in the wave and how to run a simulation.
- How to debug the simulation results using waveforms.

1.1 About the Style

We will use a number of different styles to identify different types of actions as shown below:

Student Task: Parts of the text that have a gray background, like the current paragraph, indicate steps required to complete the Exercise.

Actions that require you to select a specific menu will be shown as follows:
`menu→ sub-menu→ sub-sub-menu`

Whenever there is an option or a tab that can be found in the current view/menu we will use a BUTTON to indicate such an option.

Throughout the exercise, you will be asked to enter certain commands using the command-line[1].

```
sh>  command to be entered on the Linux command line
```

1.2 Getting Started

Student Task 1:

- Start by navigating to your home and then the exercise directory

```
sh>  cd ~/ex02
```

- Enter the `oseda` container's shell

```
sh>  oseda bash
```

Note 1: If you ever open a new terminal window or tab, don't forget to re-enter the oseda containers shell, otherwise you won't be able to start the open-source tools.

[1] There are many reasons for using a command-line, some functionality can not not be accessed through GUI commands, and in some cases, using the command-line will be much faster. Most importantly, things you enter on the command line can be converted into a script and executed repeatedly.

2 File-based Testbenches

We begin with a simple 8-bit adder to gain a base understanding of simulation and testbenches. The adder we have here first adds two 8-bit numbers to generate a sum and a carry, then saves them in flip-flops and in the next cycle the result becomes visible at the outputs.

2.1 File-Based Testbenches

Once a circuit is described in SystemVerilog, it must be functionally verified against the specification using an HDL simulator such as Verilator. This requires a verification environment, known as a testbench, which performs the following tasks:

- Generate and apply test stimuli to the Design Under Test (DUT), here the 8-bit adder.
- Capture and record the DUT's actual responses.
- Compare the DUT's output against expected responses.

A common approach to testing a design is a file-based testbench, as illustrated in Figure 2. The advantage of this method is that stimuli and expected responses can be modified without recompiling the testbench, making the verification process more flexible and efficient.

In this part of the exercise, we will use the following Python scripts:

- **`gen_stimuli.py`** randomly generates input stimuli for the testbench. (the two terms going into the adder)
- **`gen_exp_response.py`** serves as the golden model, generating expected responses (the result of the 8-bit addition).

The file-based testbench written in SystemVerilog typically consists of three main processes.
Clock and Reset Generation

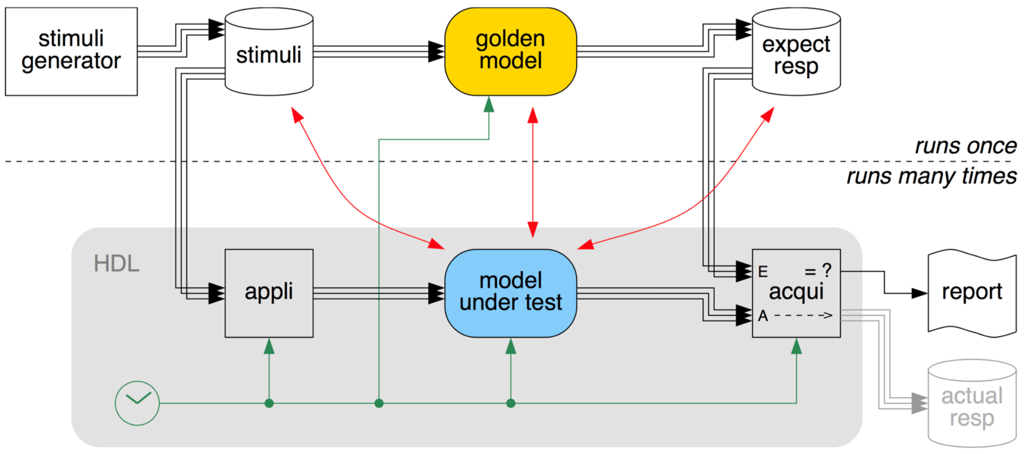
- Generates a periodic clock until the end-of-computation (`eoc`) signal is asserted.
- Generates a low-active reset (`reset_n`) signal at the beginning of the test to initialize the DUT.
- The clock period and duty cycle are usually configurable through parameters at the start of the testbench.

Stimuli Application

- Reads the pre-generated input stimuli from a file into an array.
- Applies each stimulus, one per clock cycle, until all stimuli have been processed.
- Ends the simulation when all stimuli have been applied.

Response Acquisition and Verification

- Reads the expected responses from a file.
- Compares the DUT's actual outputs against the expected responses.
- If a mismatch is detected, error information is displayed for debugging.



Student Task 2:

- Navigate to the `adder8` directory and open the testbench file `sourcecode/tb_adder8.sv` using your preferred text editor.
- How many cycles will the DUT need for resetting?

  Sample solution: The reset is low for one clock cycle (`T_CLK`).

Next, we will walk through the steps required to simulate the design in Verilator and debug the circuit using waveform analysis.

2.2 First Simulation with Verilator

Simulating a design with Verilator happens in three stages: the design is first translated to either SystemC or C++, it is then compiled to an executable binary on the host PC, and finally, the binary is run. This process enables cycle-accurate hardware execution on any host PC.

To compile the design into an executable binary, we need to provide some information to Verilator, e.g., the paths and names of the design files, and the name of the top-level design. For this simple design, we can specify all source files in a single command. However, for larger designs, we typically use a file list containing all required source files.

Student Task 3:

- Complete the following command by replacing the < > placeholders with the appropriate filenames and module names to compile the design into a binary:

```
sh>  verilator <source files' and testbench file> --top <top module
       name> --binary -Wno-fatal --trace --trace-structs
```

The `--trace --trace-structs` options enable waveform generation while preserving hierarchical structures, making signal visualization easier for debugging. We will discuss and use these trace files in a later section of the exercise.

After compilation, Verilator creates an `obj_dir` directory in the working directory. This directory contains the compiled binary along with other necessary files used by the tool. HINT: You can change the name of the output object directory with the `--Mdir <directory>` option.

Now, we can proceed with running the simulation by executing the compiled binary.

Student Task 4:

- Run the simulation by executing the following command:

```
sh >  ./obj_dir/Vtb_adder8
```

- Does the simulation complete successfully? Are there any reported mismatches?

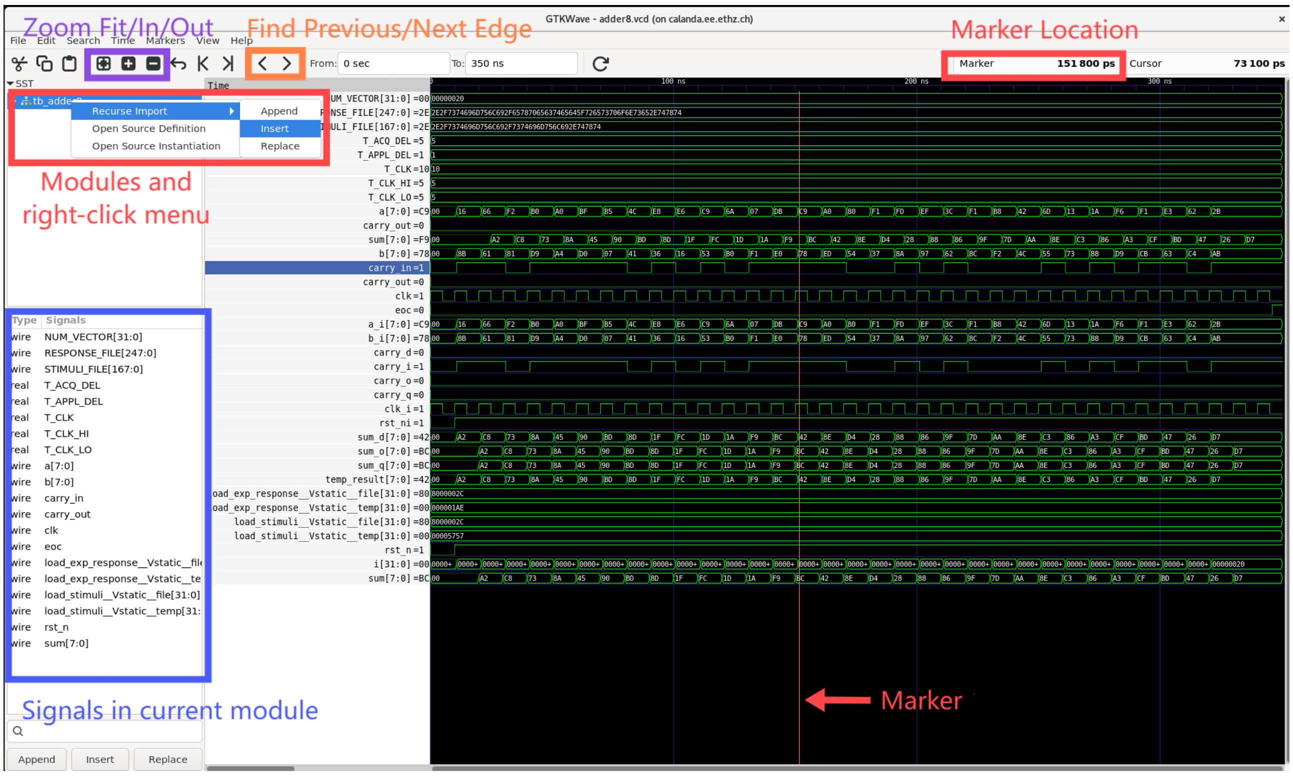
Sample solution: It does not, there are multiple incorrect results.

The messages reported in the terminal only indicate whether the design passed or failed by comparing the DUT's outputs against the golden model. This provides a quick verification check but does not help identify the root cause of failures.

To debug issues in our `Adder8` design, we need to observe the circuit's behavior during simulation.

2.3 GTKWave

Since Verilator does not have a built-in waveform viewer, we must export the waveform data to a Value Change Dump (VCD) file during simulation and open it using an external tool. In this exercise, we will use GTKWave to visualize the signal traces.



GTKWave has many different ways to interact with the guy but for our purposes using the controls from the toolbar as well as clicking in the waves to set a marker and drawing an area to zoom to by holding is sufficient.

As mentioned earlier, the `--trace --trace-structs` option tells the tool to record the wave files. Besides, we need to tell the tool the time and signal names we want to record, as well as the location where we want to store the file. These settings are configured within the testbench. We have already included the necessary commands in the Stimuli Application section of the testbench, so you won't need to modify them.

Student Task 5:

- Before opening the waveform, take a moment to re-examine the printed messages from the previous task. Identifying patterns in the failed input vectors can help narrow down the debugging scope. Can you find any common characteristics among the failed cases?
- The VCD file **`adder8.vcd`** was generated in the previous step. We can now open this file with GTKWave to visualize how the design is working during the simulation.

```
sh >  gtkwave adder8.vcd
```

- After running the command, the GUI should open. On the left column, you will see the design hierarchy with `tb_adder8`. To import all signals from the testbench and DUT, you can either right-click on the `tb_adder8` and select `Recurse Import→ Insert`, or click on `tb_adder8`, then select the signals you want to see and click 'Insert' in the bottom left. These steps will load all available signals, allowing you to analyze how the design behaves during simulation.
- Now, we can click the `Zoom Fit` in the top toolbar to observe all the transactions.
- Try to fix the bug. When does it occur, do you recognize a pattern)? Is there any signal that is suspicious looking? (eg too much or too little activity) You may work together with others and ask the assistants if you are stuck.

  Sample solution: The carry signal never toggles and the failures only occur for numbers with a sum larger than 255, meaning there is a problem with the carry generation.

- Once you are done, close GTKWave and return to the **`ex02`** directory

```
sh >  cd ~/ex02
```

2.4 Croc SoC

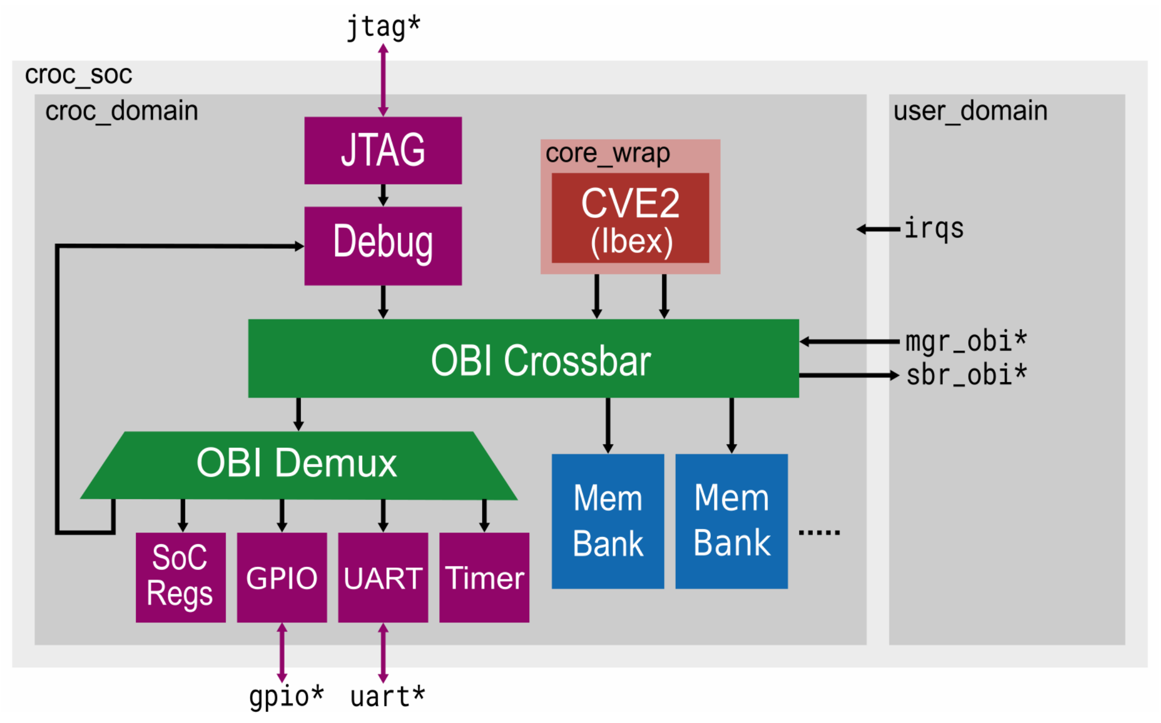
With the knowledge how to build and run simulations using Verilator and then visualize and debug problems using GTKWave, we will now shift to a more complex design. We again use the Croc SoC you have explored in the previous exercise, we will continue to use this design through the remaining exercises.

Croc SoC is designed as a streamlined, efficient microcontroller system based on the RISC-V architecture, utilizing a lightly parameterizable SystemVerilog codebase. Croc focuses on simplicity while still providing a solid platform for students and others to build on for their own projects. One important design choice its its simple CVE2 (Ibex) core. It is a two-stage RV32I core with some more extensions available by changing a few parameter (eg the multiplication extension). The core is maintained and verified by the OpenHWGroup and includes their X-interface, making it easy to extend the core with custom or standardized instructions. The second important choice is the OBI (Open Bus Interface) protocol for the on-chip interconnect. OBI is an easy to understand and fairly minimal protocol with a lot of optional features defined in the spec to cover more demanding tasks. It represents a good middle-ground between the more common APB and AXI, making it ideal for educational purposes.

The system further includes two banks of SRAM, allow for simultaneous access to instruction and data, a RISC-V debug module with a standard JTAG interface as well as some basic peripherals to get started.

The SoC level of Croc only consists of five simple RTL files.

- `croc_pkg.sv`: contains the parameters and definitions of the SoC
- `croc_chip.sv`: defines the chip-level design including the ASIC pads
- `croc_soc.sv`: instantiates different 'domains' and connects them together, namely the croc domain and the user domain
- `croc_domain.sv`: instantiates and connects all components (core, debug module, peripherals) using the OBI interconnect
- `user_domain.sv`: empty per default, provides a very simple way for students to add their own creations



2.5 Croc Testbench

Although the testbench for the Croc SoC follows the same fundamental principles as the `Adder8` design, it is more complex to support all the different interfaces and features. Additionally, instead of having the simple 'data goes in, data comes out' flow mirrored in the file-based testbench. Most of the interesting things happen internally when the core executes a program. So for the Croc testbench, you can imagine the software we load as the stimuli file and the things we check for (end state of an SRAM, outputs of the SoC eg via UART) as the vectors we compare against our expectations.

The complete simulation flow for the Croc SoC is illustrated below, where the process is divided into the following key steps:

Software Compilation

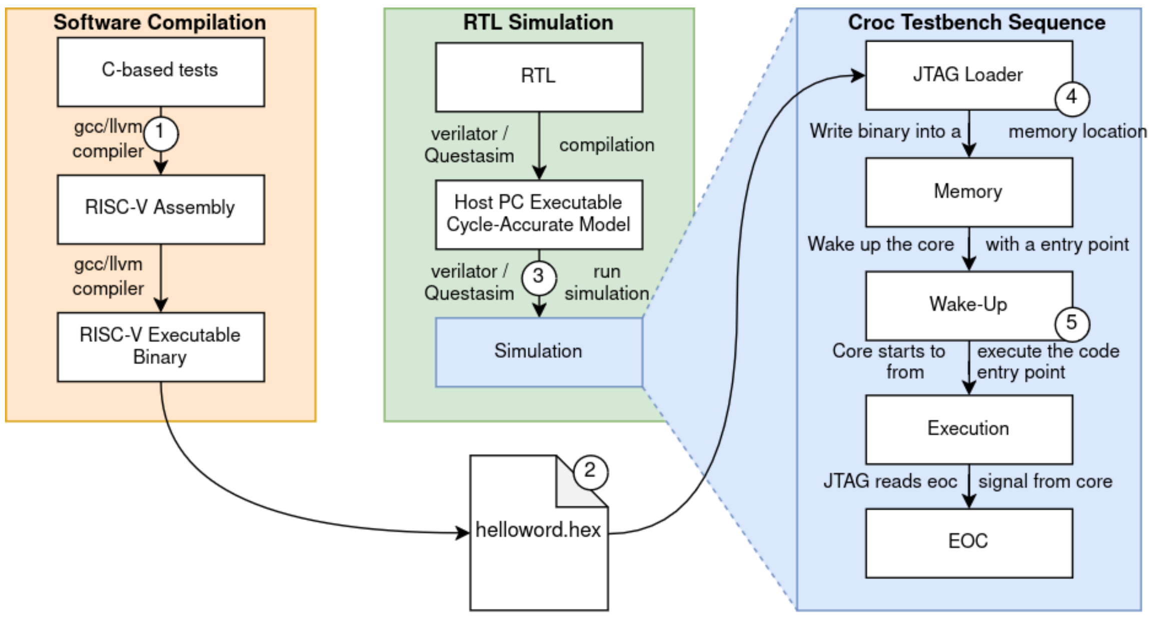
- The simulation starts with C-based tests, which are compiled into RISC-V assembly code and RISC-V executable binaries.
- The input stimuli and expected responses are typically embedded within these C-based tests.

RTL Simulation

- Similar to the `Adder8` example, the RTL source code is compiled into an executable binary.
- The simulation is then launched, executing the compiled binary.

Croc Testbench Sequence

- The JTAG loader in the testbench reads the RISC-V executable binary and loads it into the SRAM banks.
- Once loaded, the core is woken up and begins executing instructions from the entry point.
- The core writes to a memory-mapped register, asserting the eoc (end-of-computation) signal upon completion.
- The JTAG module continuously polls this register until the eoc signal is detected, at which point the simulation ends.



Since the compilation process is beyond the scope of this exercise, we have prepared a Makefile and already compiled the provided software using the RISC-V GCC compiler. If you want to experiment and change the software, you will need to recompile it. Do so by navigating to the **`sw`** directory and then run `make`. The compiled binaries are located in **`sw/bin`** directory. During simulation, the JTAG module will automatically load these binaries into the SoC's SRAM.

We are now almost ready to compile and simulate the SoC. However, unlike the `Adder8` design, our SoC consists of many RTL files, making it impractical to manually specify each file in the compilation command.

To simplify this process, we use a file list containing all RTL source files required for the simulation and the necessary macro definitions used in those files.

Student Task 6:

- Navigate to the `verilator` directory.

```
sh>  cd verilator
```

- Open the file list **`croc.f`** in the **`verilator`** directory. How many files are being imported for the simulation? Can you find the files that implement a specific feature eg the core or UART peripheral?

Sample solution: You can run `grep "\.\./" croc.f | wc -l` to get the answer. (164)

- Enter the following command to compile the design. Note that we now use `trace-fst` and we give the files via the file list argument `-f`:

```
sh>  verilator --binary -j 0 -Wno-fatal --trace-fst --trace-structs
         --top tb_croc_soc -f croc.f
```

Note: the first-time compilation may take a few minutes. If no RTL changes are made, you do not need to recompile the design for subsequent runs.

- Once compiled, execute the binary to start the simulation:

```
sh>  ./obj_dir/Vtb_croc_soc
```

- Examine the test output and compare them against the flow in the testbench **`croc/rtl/test/tb_croc_soc.sv`** and the software you ran. Did the simulation produce the expected results?

Now that we have completed the SoC simulation, we can use the waveform traces to examine key events in the execution sequence, including program loading and the assertion of the end-of-computation signal.

As before, the simulation has recorded the waveforms to a a file, here a FST file which is like a compressed VCD.

Student Task 7:

- Open the waveform file using GTKWave. Since the SoC is complex, avoid importing all signals at once. Focus on the JTAG signals and memory interface, which are essential for the boot sequence. Add the five JTAG interface signals to the waveform view at the croc_soc level: `tb_croc_soc/i_croc_soc`, and the main memory interfaces at `tb_croc_soc/i_croc_soc/i_croc/gen_sram_bank[0]`.
- Before the JTAG module loads the executable binary, a simple memory write & read test is performed. Identify the time and data of memory write and read operations. Compare these with the test log output. Do the timestamps match? If not, why?

Sample solution: They are best identified by searching for two rather isolated peaks of the `bank_req` signal. The write occurs at $61\,650\,\mathrm{ns}$ and the read back occurs at $80\,150\,\mathrm{ns}$.

- After the simple memory test, JTAG writes the binary code to the starting address. What data is written to address `0x1000_0008`? Open the **`sw/bin/helloworld.dump`** file, which contains the assembly code for the test we execute. Identify the assembly instruction fetched at this moment.

Sample solution: A bit after the previous two requests there is a whole serious of requests. It starts at `0x1000_0000` (the start address of the SRAMs) and counts up by 4 (because we have 32-bit per write, which 4 byte). So going to the third peak will show the request for the given address. The value is `0x0000_1117`.

- The `eoc` signal comes from a memory-mapped register called `core_status` inside the `tb_croc_soc/i_croc_soc/i_croc/i_soc_ctrl` module. Can you determine the full memory address of this register from the waveforms?

Sample solution: The idea is to use the previous knowledge and add the OBI request coming into the module to the waveforms as well as the `core_status` register signals. We know that it has to change at the end of simulation so the testbench knows to terminate. Searching at the very end reveals a change in the registers value and the corresponding write goes to the address `0x0300_0008`.

- At this point feel free to explore the SoC further, change the software or modify the flow in the testbench. Adventurous attendees may also want to head over to the VLSI website[a] and have a look at the "RTL, understanding/extending Croc" exercise.

[a] https://vlsi.ethz.ch/wiki/VLSI_Exercises

$\mathcal{E}$ **You are done with Exercise 2. Discuss your results with an assistant and/or your peers.** $\mathcal{E}$