



OpenHW Group

Proven Processor IP

Writing Tests for CORE-V-VERIF

Mike Thompson

mike@openhwgroup.org



OPENHW GROUP™
— PROVEN PROCESSOR IP —

© OpenHW Group

April, 2022

Objectives

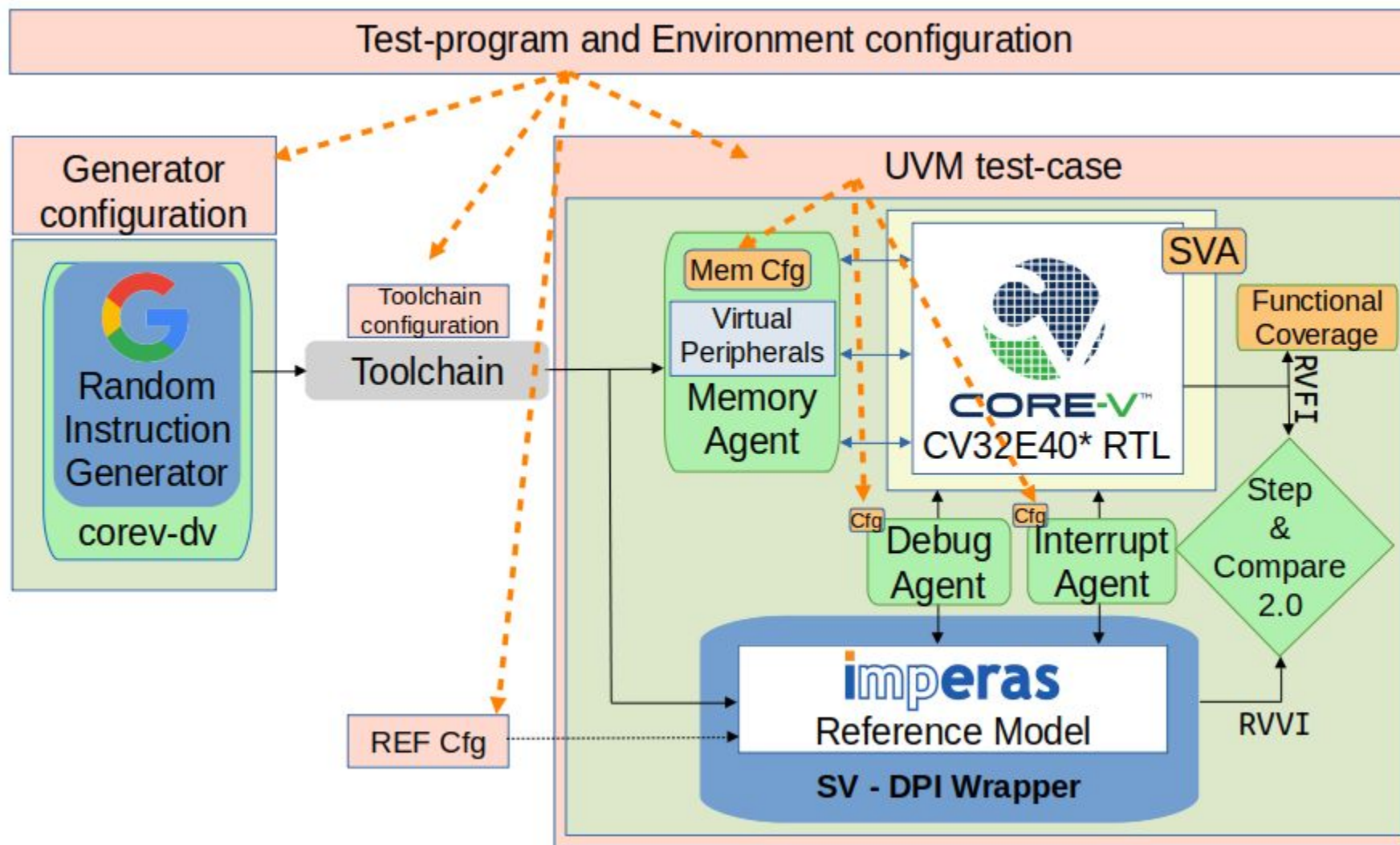
- These slides are one of a set of presentations designed to provided an overview of the CORE-V-VERIF SV/UVM environment:
 - Understanding the verification environment architecture.
 - Verification Planning for TRL-5 projects.
 - UVM Agents in CORE-V-VERIF.
 - Writing and Running Test-cases ← This presentation
- Goals of these presentations:
 - Introduction for new Contributors.
 - Communicate the current state of CORE-V-VERIF.
 - Create consensus around standardized specifications, architecture and implementation of UVM Environments used in CORE-V-VERIF.

Topics

- Definitions.
- Test Programs.
- Test Cases.
- Writing your own Test Programs.
- Running and Managing Tests.

Definitions

CORE-V-VERIF Implementation as of 2022-03



Test-Cases, Test-Programs and Tests



- UVM makes a distinction between a **Test-Case** and a **Test**.
- CORE-V-VERIF makes a distinction between a **Test-Case** and a **Test-Program**.
- A Test-Case is a SystemVerilog class that is an extension of uvm_test:
 - In the UVM, the test-case is the top-level object in the hierarchy.
 - The primary role of a test-case is to constrain the configuration objects of the environment and its components (Agents, Scoreboards, Sequences, etc.)
 - In a typical UVM environment, there are only a few unique test-cases.
 - To date, in core-v-verif there is only one (!) test-case per core.
- A Test-Program is machine code that runs on the simulation model of the core.
 - Test-programs execute within the core-v-verif programming environment.
 - Test-programs can be manually or machine generated.
 - In core-v-verif there are many test-programs.
- A Test is a single simulation of a test-case:
 - Each simulation of a constrained-random test-case is a unique test.
 - Each simulation of a directed test-case is the same as all others.



CORE-V-VERIF Programming Environment



- Test-Programs run on the RTL model of the core-under-verification, which is instantiated in a UVM environment.
- Test-Programs are translated from source (C or assembly) to machine code by a cross-compiler referred to as the “Toolchain”.
- The Toolchain is controlled by a set of files collectively called the “**Board Support Package**” (bsp).
- The core-v-verif BSP has limited support for compile-time configuration.
- Together these are collectively referred to as the “CORE-V-VERIF Programming Environment”:
 - It is a very, very bare-metal environment.
- The CORE-V-VERIF Programming Environment affects all Test-Programs:
 - The Test-Program has access to limited resources - do not call malloc()! 😊
 - Maintaining consistency between the Test-Program, BSP and the UVM environment is a challenge.



Verification Environment Consistency



- Maintaining consistency of all components of a verification environment is a difficult problem.
- The UVM architecture was deliberately designed to address this problem:
 - All components must have a configuration object.
 - All components controlled with an explicit run-flow.
 - Shared information maintained in a single, unified configuration database.
 - Top-down control of test-case configuration.
- Unfortunately, major components of core-v-verif are not UVM components:
 - The **riscv-dv** generator is written in SystemVerilog as a set of *uvm_component* extensions.
 - No concept of the UVM run-flow.
 - No configuration objects.
 - Does not make use of the configuration database.
 - Controlled via SystemVerilog “plusargs” instead of constraints on random class members.
 - The RISC-V cross-compiler (**toolchain**) has no concept of UVM.
 - Neither the Spike or Imperas **Reference Models** have any concept of UVM.
- To (partially) address these issues, core-v-verif adds a configuration layer based on YAML test descriptions.



Test-Cases

The UVM test-case in CORE-V-VERIF



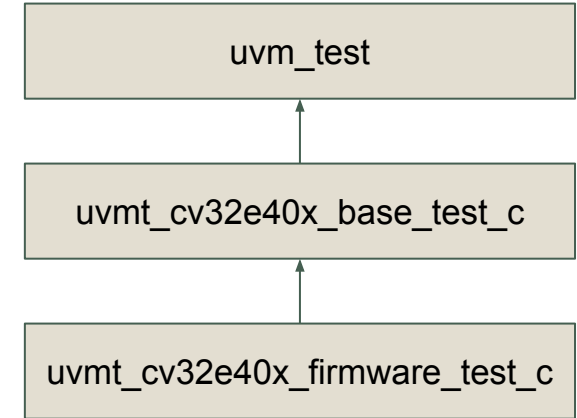
- The preliminary design of core-v-verif identified three types of UVM test-cases:
 - See the “[Test-Programs in the CORE-V-VERIF UVM Environment](#)” chapter of the Verification Strategy (on ReadTheDocs) for more information.
- What actually happened is that a single UVM test-case supported all of our needs: [core-v-verif/<core>/tests/uvmt/compliance-tests/uvmt_<core>_firmware_test.sv](#)
 - This may not always be true in the future.
- Multiple reasons for this:
 - Over-use of “plusargs” to control randomization (a bad practise).
 - Much of the “stimulus” required to verify a core are the instructions.
 - Other types of stimulus are (mostly) independent of the instructions:
 - Cycle-time behavior of external interface (e.g. request-to-grant delay).
 - External, asynchronous events (e.g. interrupts).
 - Memory subsystem behavior (e.g. page faults).

} These are controlled by the Test-Case, not the Test-Program.



The view from the top: *uvm_test*

- All test-cases extend from *uvm_test*:
 - In the CV32E40X, almost all testing is handled by one UVM test: *uvmt_cv32e40x_firmware_test_c*, which extends from *uvmt_cv32e40x_base_test_c*.
 - The base test has a “vsequencer” member of type *uvme_cv32e40x_vsqr_c*.
 - Like most component objects in core-v-verif, the vsequencer has a configuration (cfg) and context (cntxt) members.
 - The most interesting members of the vsequencer are the handles to all the Agent sequencers.
- The test-case’s sequencers are the key to the UVM environment’s behaviour.



```
class uvme_cv32e40x_vsqr_c extends uvm_sequencer#(  
    .REQ(uvm_sequence_item),  
    .RSP(uvm_sequence_item)  
);  
  
// Objects  
uvme_cv32e40x_cfg_c    cfg;  
uvme_cv32e40x_cntxt_c cntxt;  
  
// Sequencer handles  
uvma_clknrst_sqr_c    clknrst_sequencer;  
uvma_interrupt_sqr_c  interrupt_sequencer;  
uvma_debug_sqr_c      debug_sequencer;  
uvma_obi_memory_sqr_c obi_memory_instr_sequencer;  
uvma_obi_memory_sqr_c obi_memory_data_sequencer;
```

uvmt_<core>_firmware_test



- All tests in CV32E40P, E40X and E40S use this UVM test-case.
- All environment randomization is controlled from here, following the standard UVM randomization control hierarchy.
- [uvmt_cv32e40*_firmware_test](#) extends from [uvmt_cv32e40*_base_test](#) which implements most of the env and component constraints.
- In core-v-verif, randomization is enabled or disabled using SystemVerilog “plusargs”:
 - [+gen_random_debug](#) will enable generation of random debug requests.
 - [+gen_irq_noise](#) will enable generation of random interrupts.
 - For cores using OBI v1.2, random memory bus errors can also be enabled.
 - Note: randomization of OBI cycle timing is always enabled.
- In core-v-verif, control of the plusargs is handled by the Test-Program Configuration.
 - More on this later...



Test-Programs

Test-Programs



- Two types:
 - **Manually Generated:** e.g. hello-world, misalign, etc.

```
$ make test TEST=hello-world
```
 - **Machine Generated:** e.g. rand_jump_stress_test

```
$ make corev-dv // only needed once
```

```
$ make gen_corev-dv test TEST=corev_rand_jump_stress_test
```
- Manually generated tests may be, but do not need to be, self-checking.
 - Many manually generated test are at least partially self-checking.
 - CORE-V-VERIF environment will check the core's execution against a reference model by default, so in theory the self-checking aspects of the test-program are not required.
- Machine generated tests are rarely self-checking:
 - Require a check against a reference model to determine pass/fail.

Test-program and Environment Configuration



- Each Test-program has its own directory:
 - e.g. [core-v-verif/cv32e40x/tests/programs/custom/hello-world/](#)
- Manually written test-programs will have:
 - A [test.yaml](#) is the test-program definition for the toolchain and SystemVerilog simulator.
 - One or more C and/or RISC-V assembly source files for the test-program itself (e.g. [hello-world.c](#)).
- Corev-dv generated test-programs will have:
 - A [test.yaml](#) configuration file. Serves the same function as for manually written test-programs.
 - A [corev-dv.yaml](#) is the test-program definition for the corev-dv instruction generator.
- These YAMLS are parsed by python scripts to produce:
 - Shell environment variables used by the Makefiles.
 - Command-line arguments for corev-dv, toolchain and SystemVerilog simulator.
- Detailed documentation is available in the [Test-Program Definitions](#) section of [core-v-verif/mk/TOOLCHAIN.md](#).

Example Test-Program Configurations

- Each test directory must have a `test.yaml`.

```
name: hello-world
uvm_test: uvmt_$(CV_CORE_LC)_firmware_test_c
description: >
  Simple hello-world sanity test
```

`cv32e40x/tests/programs/custom/hello-world/test.yaml`

- By default, the `cfg/default.yaml` is also used.

```
name: default
description: Default configuration for CV32E40X simulations
compile_flags:
  +define+ZBA_ZBB_ZBC_ZBS
ovpsim: >
  --showoverrides
  # --trace --tracechange --traceshowicount --monitornets
cflags: >
plusargs: >
  +enable_zba_extension=1
  +enable_zbb_extension=1
  +enable_zbc_extension=1
  +enable_zbs_extension=1
cv_sw_march: rv32imc_zbalp00_zbb1p00_zbc1p00_zbs1p00
```

`cv32e40x/tests/cfg/default.yaml`

- You can select a different cfg
YAML via the CFG variable:
`$ make test CFG=my.yaml TEST=my_test`

CORE-V-VERIF Programming Environment



- All test-programs must be able to run in the CORE-V-VERIF programming environment.
 - It is a very “bare metal” environment.
 - Each programming environment is core-specific.
 - Largely defined in the “Board Support Package” in [core-v-verif/<core>/bsp](#).
 - The BSP documentation is in the README.
- Key features of the BSP that the test-program writer (human or machine) must be aware of:
 - **link.ld:**
 - Memory map, start address, stack address and debug address.
 - **syscalls.c:**
 - Implementation of **_write()** to support printf() in test-programs.
 - Catcher for all other unimplemented system calls.
 - **corev_uvmt.h:**
 - Macros to access virtual peripherals in the SystemVerilog testbench.

Warning: few of the OpenHW Contributors are expert in the definition and implementation of the programming environment.

CORE-V-VERIF Virtual Peripherals



- The CV32E40P, X and S verification environments support a set of “Virtual Peripherals” that are resources available to the programmer.
 - These are documented in the Verification Strategy ([link](#)).
 - Implemented as a set of UVM virtual sequences that run on the Memory Interface Agent.
- Address map for Virtual Peripherals must match the BSP (corev_uvmt.h):
 - The address map can be overridden via the test-program definition (test.yaml).
- The most popular Virtual Peripherals are:
 - **virtual printer:** write to stdout.
 - **status flags:** signal end-of-test to simulation.

Virtual Peripheral	VP Address (data_addr_i)	Action on Write
Address Range Check	$\geq 2^{20}$	Terminate simulation (unless address is one of the virtual peripherals, below).
Virtual Printer	32'h1000_0000	\$write("%c", wdata[7:0]);
Interrupt Timer Control	32'h1500_0000	timer_irg_mask <= wdata;
	32'h1500_0004	timer_count <= wdata; This starts a timer that counts down each clk cycle. When timer hits 0, an interrupt (irq_o) is asserted.
Debug Control	32'h1500_0008	Asserts the debug_req signal to the core. debug_req can be a pulse or a level change, with a programmable start delay and pulse duration as determined by the wdata fields:
		wdata[31] = debug_req signal value
		wdata[30] = debug request mode: 0= level, 1= pulse
		wdata[29] = debug pulse duration is random
		wdata[28:16] = debug pulse duration or pulse random max range
		wdata[15] = start delay is random
		wdata[14:0] = start delay or start random max range



Developing Your Own Tests

Manually Written Test-Programs

- Create a new directory under [core-v-verif/<core>/tests/programs/custom](#)
 - The directory name is the test-program name (e.g. `make test TEST=test_program_name`)
 - Add all .c and .S source files for the test-program to that directory.
(If the test-program comprises multiple source files, the Makefiles will recognize this).
 - Add test.yaml.
 - Most custom programs use the `cfg/default.yaml`.
- Manually written test-programs must work within the CORE-V-VERIF programming environment:
 - Please read [core-v-verif/<core>/bsp/README.md](#).
 - Easy to write if you avoid exceptions, traps, interrupts and debug-requests.
 - See [tests/program/custom/debug_test](#) and [tests/program/custom/interrupt_test](#) for good examples of manually written test-programs that exercise Interrupt and Debug features of the core.
- If you enable exceptions, traps, interrupts or debug-requests, you may need to write your own handler:
 - The BSP will simply return to the main execution flow (which may be sufficient for your needs).

COREV-DV Test-Programs

- COREV-DV is a set of class extensions of specific classes in Google's riscv-dv.
- To develop a corev-dv test-program:
 - Create a directory: `core-v-verif/<core>/tests/programs/corev-dv/my_corev_test_program`
 - Add test.yaml and corev-dv.yaml
- Test.yaml performs the same function as manually written test-programs.
- Corev-dv.yaml is almost identical to the test-program configuration used by riscv-dv.

```
# Test definition YAML for corev-dv test generator

name: corev_rand_arithmetic_base_test
uvm_test: $(CV_CORE_LC)_instr_base_test
description: >
    RISC-V-DV generated arithmetic test
plusargs: >
    +instr_cnt=10000
    +num_of_sub_program=0
    +directed_instr_0=riscv_int_numeric_corner_stream,4
    +no_fence=1
    +no_data_page=1
    +no_branch_jump=1
    +boot_mode=m
    +no_csr_instr=1
```

COREV-DV extensions of RISCV-DV

- Corev-dv extends (does not modify) riscv-dv.
 - Implemented in [core-v-verif/lib/corev-dv](#)
- Makefiles will clone riscv-dv from GitHub and compile corev-dv as needed:
 - `$ make corev-dv`
- Two types of corev-dv extensions:
 - Compatibility with CORE-V-VERIF Programming Environment.
 - Instruction stream generation extensions, implemented in [core-v-verif/lib/corev-dv/instr_lib](#)



COREV-DV Compatibility with CORE-V-VERIF



- The Google riscv-dv project assumes a specific programming environment that core-v-verif does not support.
- An extension of riscv_asm_program_gen resolves this issue by overriding the following methods:
 - **gen_program_header()**: use symbols as required by BSP.
 - **gen_ecall_handler()**: riscv-dv assumes ecall signals the end of simulation, which is not how it is done in core-v-verif.
 - **gen_test_done()**: use virtual-peripheral status flags to signal end of sim.



COREV-DV Instruction Stream Extensions

- These range from trivial to complex.
- All of them are extensions of a riscv-dv instruction stream class:

```
class corev_interrupt_csr_instr_stream
extends riscv_load_store_rand_instr_stream;
```
- Implemented to either create a new instruction stream, or alter an existing stream to:
 - Improve function coverage.
 - Integrate with the core-v-verif programming environment.

```
class corev_nop_instr extends riscv_directed_instr_stream;

  `uvm_object_utils(corev_nop_instr)

  function new(string name = "");
    super.new(name);
  endfunction : new

  function void post_randomize();
    riscv_instr_name_t allowed_nop[$];
    riscv_instr      nop;

    allowed_nop.push_back(NOP);
    if (!cfg.disable_compressed_instr)
      allowed_nop.push_back(C_NOP);

    nop = riscv_instr::get_rand_instr(.include_instr(allowed_nop));
    nop.m_cfg = cfg;
    randomize_gpr(nop);
    insert_instr(nop);
  endfunction : post_randomize
endclass : corev_nop_instr
```


Updating/Extending COREV-DV

- Corev-dv is a SystemVerilog Package.
- To create a non-core-specific extension to corev-dv, add the class to [core-v-verif/lib/corev-dv/corev_instr_test_pkg.sv](#)
 - Should have a **corev_** prefix in both the filename and class name.
- Core-specific extensions should be placed in [core-v-verif/<core>/env/corev-dv](#)



Running and Managing Tests

(specific to the CV32E4 cores)

Running Tests

- Documentation for setting up your shell environment and running tests with the CV32E4 cores is in [core-v-verif/mk/README.md](https://github.com/core-v-verif/mk/README.md).
 - If you have a question or spot a problem with the documentation, please open an Issue.
- Tests are either “directed tests”, that produce the same stimulus each time they are run, or “constrained random tests” that produce different stimulus each time they are run.
- All files generated when a test is run are placed in a unique, per-test directory.

Directed vs. Constrained-Random Tests



- **Directed Tests** produce the same stimulus each time they run.
- **Constrained-Random Tests** produce unique stimulus each time, controlled by a seed passed as a run-time argument.
- In core-v-verif there are two testing dimensions:
 - **Test-programs** can be manually-written or machine (corev-dv) generated.
 - The **UVM environment** is “random by design”.
- This means that tests using manually-written test-programs are not necessarily directed tests:
 - The test-program is static, but the behavior of the environment is not.
 - Cycle-timing of memory buses, interrupts, debug requests and memory errors are all subject to constraints.
- Corev-dv generated test-programs are all (slightly) different.
- Seed management:
 - For manually-generated tests, you need the seed for the UVM environment simulation.
 - For corev-dv generated tests you need both the seed for the corev-dv run that produced the test-program and the seed for the UVM environment simulation.



Simulation Outputs

- Running a simulation can create a lot of files:
 - Compilation and simulation logs
 - Transaction and Trace logs
 - Toolchain outputs
 - Waveforms
- The CV32E4 Makefiles place all of these in a common directory structure: `<simulator>_results/<cfg>/<test-program>/<run>/`
 - By default these are located in `core-v-verif/<core>/sim/uvmt`.

Reproducing Results of Constrained-Random Simulations



- The philosophy of core-v-verif is that all test-cases should be capable of generating unique stimulus each time they run:
 - The difference between tests is determined by the initial seed provided to the simulator at run-time.
 - The seed is written to the logfile and can be passed to the simulator via a Makefile variable on the command-line.
- Each simulator uses a distinct command-line argument to set the seed.
 - Typically, but not always, the simulator or it's core-v-verif Makefiles will set the seed to '1' if it is not specified on the command line.



Example 1: Manually Generated Test-Programs

- In this simple scenario, an invocation of the 'hello-world' test produces an interesting result:
`$ make test TEST=hello-world`
- First, open the run-time log file to determine the seed used:
 - Each simulator uses a unique argument for this (groan).
- Next, run the test again with RNDSEED set on the command-line:

```
$ make test TEST=hello-world RNDSEED=<seed>
```

```
xrun
-R
-xmlibdirname ../../xcelium.d
-l xrun-hello-world.log
-covoverwrite
-64bit
-licqueue
+UVM_VERBOSITY=UVM_MEDIUM
+USE_TSS
-svseed 1234567890
-sv_lib /home/v51366/mydata/GitHubRepos/openhwgroup/core-v-verif
-sv_lib /home/v51366/mydata/GitHubRepos/openhwgroup/core-v-verif
-sv_lib /home/v51366/mydata/GitHubRepos/openhwgroup/core-v-verif
-nowarn XCLGNOPTM,RNDNOXCEL,PRLDYN,COVNBT
-covtest hello-world
+UVM_TESTNAME=uvmt_cv32e40x_firmware_test_c \
+enable zba_extension=1 +enable zbb_extension=1 +enable zbc_exte
+elf_file=/home/v51366/mydata/GitHubRepos/openhwgroup/core-v-ver
+firmware=/home/v51366/mydata/GitHubRepos/openhwgroup/core-v-ver
+itb_file=/home/v51366/mydata/GitHubRepos/openhwgroup/core-v-ver
+nm_file=/home/v51366/mydata/GitHubRepos/openhwgroup/core-v-veri
```

Example 2: corev-dv Generated Test-Programs

- In this simple scenario, an invocation of the ‘random jump stress’ test produces an interesting result:

```
$ make gen_corev-dv test TEST=corev_rand_jump_stress_test
```

- Open the run-time log file to determine the seed used:
 - Each simulator uses a unique argument for this.

- Run the test again with RNDSEED set on the command-line:

```
$ make gen_corev-dv test TEST=hello-world RNDSEED=<seed>
```


Thank You

Additional Info



Statistics from CV32E40P Regressions

- The “cv32e40p_full_covg_no_pulp” regression defines:
 - 44 test-programs.
 - 30 are manually written.
 - 14 are generated by corev-dv.
- Manually written tests are typically only run once.
- Corev-dv generated tests are run many times (up to 200 times each).
- Total number of tests in “cv32e40p_full_covg_no_pulp” regression is ~1800.
 - This regression will achieve ~99% coverage.
 - Achieving 100% coverage requires merging coverage results from 3 to 4 regressions.

[core-v-verif/cv32e40p/regress/cv32e40p_full_covg_no_pulp.yaml](#)

```
# List of tests
tests:
  hello-world:
    build: uvmt_cv32e40p
    description: uvm_hello_world_test
    dir: cv32e40p/sim/uvmt
    cmd: make test COREV=YES TEST=hello-world

  corev_rand_arithmetic_base_test:
    build: uvmt_cv32e40p
    description: Generated corev-dv arithmetic test
    dir: cv32e40p/sim/uvmt
    cmd: make gen_corev-dv test COREV=YES TEST=corev_rand_arithmetic_base_test
    num: 200
```

The Goal: a universal UVM environment for CORE-V cores



UVM Test Configurations replace all YAML files.

