

# Shakti C-Class Multicore Verification

Megna Premkumar (Guided by Prof. Kamakoti V, mentored by Lavanya J.)

## I. INTRODUCTION

The Shakti C-Class is a controller class processor featuring a 5-stage in-order design with support for a Memory Management Unit (MMU). This architecture is capable of running full-fledged operating systems such as Linux. The processor supports the standard RV64GCSUN ISA. When instantiated with four C-Class cores, a multicore configuration is achieved. This configuration utilizes the OpenSMART Network-on-Chip (NoC) with criticality as the underlying fabric. For cache coherency, the MSI protocol is employed alongside a customized bus protocol. Additionally, a 256KB Block RAM (BRAM) is used as a substitute for the main memory. Ensuring the correct functioning of a processor is one of the cornerstones of the modern microprocessor development. Functional verification is used for this purpose. In this paper, I present some tests for verification of Shakti C-Class Multicore.

## II. RISC-V-DV

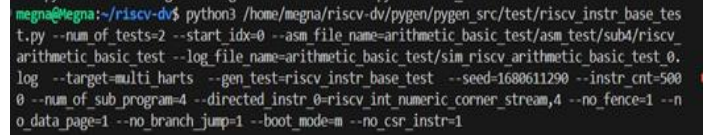
RISC-V-DV is an open-source, Python-based instruction generator designed for the verification of RISC-V processors. It supports multiple hardware threads and generates tests where each test produces a program with distinct assembly instructions for each hardware thread (hart ID). RISC-V-DV tests also offer co-simulation support with Spike, an Instruction Set Simulator (ISS). For verification, I conducted simulations using Spike, which emulates an ideal RISC-V core, and compared the results with simulations from the Shakti core. Spike simulation of Shakti C-Class with the generated assembly programs.

## III. PYGEN

RISC-V-DV is a purely Python based open-source instruction generator for RISC-V processor verification. It uses PyVSC as the main library for randomization and coverage collection. There are tests specific to the ISA, which can be modified using the *target* argument. The argument *simulator* is chosen as *Pyflow* which invokes the python generator. The script which is used to generate the tests is *run.py*. The test can be specified using the argument *test*. Finally, the command used to run a single test is:

```
python3 run.py --test=riscv_arithmetic_basic_test
--simulator=pyflow
```

*run.py* imports all of the source files from *pygen*. This script generates a command to compile *riscv\_base\_instr\_test.py* like this:



```
megna@megna:~/riscv-dv$ python3 /home/megna/riscv-dv/pygen/pygen_src/test/riscv_instr_base_test.py --num_of_tests=2 --start_idx=0 --asm_file_name=arithmetic_basic_test/asm_test/sub4/riscv_arithmetic_basic_test --log_file_name=arithmetic_basic_test/sim_riscv_arithmetic_basic_test_0.log --target=multi_harts --gen_test=riscv_instr_base_test --seed=1680611290 --instr_cnt=500 0 --num_of_sub_program=4 --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
```

Fig.1: Command used to run *riscv\_base\_instr\_test.py*

This command calls the functions inside the class *riscv\_asm\_program\_gen()* which is defined in the file *riscv\_asm\_program\_gen.py*. This function creates the main assembly program for all hart. Main program has separate assembly instructions for every hart ID. The threads have the same boot address and jump to the section according to the hart ID. The number of harts can be modified by setting the value of *NUM\_HARTS* as 4 in the file *riscv\_core\_setting.py*. It is also used to create sub programs for every hart based on the value given to the argument *num\_of\_sub\_programs* in the command. The argument *num\_of\_tests* is used to one or more test programs for a given riscv-dv test. In *riscv\_instr\_base\_test.py*, multiprocessing module in python is used to generate these tests parallelly.

## IV. NUM\_OF\_SUB\_PROGRAMS

Issue with the tests was when the *num\_of\_sub\_programs* was non zero. The test generation failed and assembly instructions were not generated.

The class *riscv\_asm\_program\_gen()* has the function *gen\_program* which generates the main program. It calls the function *gen\_sub\_program* which takes number of harts and number of sub programs as arguments. Then after the subprograms are generated, it calls another function *gen\_callstack* which takes the following arguments: *main\_program*, *sub\_program*, *sub\_program\_name*, and *num\_sub\_program*.

An object named *callstack\_gen* is used, to create a random or structured call flow involving the main program and sub-programs. This object points to the class *riscv\_callstack\_gen()*. There are two specific functions defined inside the class *riscv\_callstack\_gen* which caused the issue:

- ❑ *problem\_definition()* – This function is used to create a range of sub program IDs, which

is then randomised by the function `randomize()`.

- ❑ `randomize()` – This function is used to randomize the unique sub program IDs.

The function `gen_callstack.randomize()` is failing because `problem_definition` isn't called before using `randomize()`, and therefore range of subprogram IDs likely remains empty. Thus, it creates error trying to randomize an empty list.

After calling the function `problem_definition()` before `randomize()`, the error got fixed and the test generated sub programs for every hart.

## V. TEST GENERATION

Simulator used to run instruction generator is `pyflow`. This command is used to run the simulator to generate assembly instructions.

```
python3 run.py --test=riscv_arithmetic_basic_test --simulator=pyflow --steps gen
```

After running this command, it generates another command to run the file `riscv_instr_base_test.py`.

```
magna@magna:~/riscv-dv$ python3 run.py --test=riscv_arithmetic_basic_test --simulator=pyflow --steps gen
Tue, 21 May 2024 16:36:11 INFO      Creating output directory: out_2024-05-21
Tue, 21 May 2024 16:36:11 INFO      Processing regression test list : /home/magna/riscv-dv/target/multi_harts/testlist.yaml, test: riscv_arithmetic_basic_test
Tue, 21 May 2024 16:36:11 INFO      Processing regression test list : /home/magna/riscv-dv/yaml/base_testlist.yaml, test: riscv_arithmetic_basic_test
Tue, 21 May 2024 16:36:11 INFO      Found matched tests: riscv_arithmetic_basic_test, iterations:2
Tue, 21 May 2024 16:36:11 INFO      Processing simulator setup file : /home/magna/riscv-dv/yaml/simulator.yaml
Tue, 21 May 2024 16:36:11 INFO      Found matching simulator: pyflow
Tue, 21 May 2024 16:36:11 INFO      Building RISC-V instruction generator
Tue, 21 May 2024 16:36:11 INFO      Running RISC-V instruction generator
Tue, 21 May 2024 16:36:11 INFO      Generating 2 riscv_arithmetic_basic_test
Tue, 21 May 2024 16:36:11 INFO      Running riscv_arithmetic_basic_test with 1 batches
Tue, 21 May 2024 16:36:11 INFO      Running riscv_arithmetic_basic_test, batch 1/1, test_cnt:2
```

Fig. 2: Running the `pyflow` simulator

```
magna@magna:~/riscv-dv$ python3 /home/magna/riscv-dv/pygen/pygen_src/test/riscv_instr_base_test.py --num_of_tests=2 --start_idx=0 --asm_file_name=arithmetic_basic_test/asm_test/sub4/riscv_arithmetic_basic_test --log_file_name=arithmetic_basic_test/sim_riscv_arithmetic_basic_test_0.log --target-multi_harts --gen_test=riscv_instr_base_test --seed=1680611290 --instr_cnt=500 0 --num_of_sub_program=4 --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
```

Fig. 3: Command generated after compiling `run.py`

The command in Fig.3 is generated and it can be run independently after modifying the `no_of_tests`, `num_of_sub_programs`, `instr_cnt`, etc. After the assembly test programs are generated, we can use this command to get a binary file –

```
riscv64-unknown-elf-gcc -nostartfiles -T link.ld
riscv_arithmetic_basic_test_0.S
```

## VI. SPIKE SIMULATION

Spike, the RISC-V ISA Simulator, implements a functional model of one or more RISC-V harts. Spike simulation results are used to compare with Shakti's results for verification.

The previous command will create a binary file `a.out`. This file can be used to run on spike using the

following command. This will create the log file named as `spike.log`. `-p4` is a flag to simulate 4 cores.

```
spike -l --log=spike.log -p4 a.out
```

After running on spike, we can generate a dump file which can be used for comparison with the shakti's dump file. The command use is –

```
spike -l --log=spike.dump -p4 a.out
```

## VI. EXECUTION ON SHAKTI MULTICORE

The assembly tests cannot be directly run on shakti cpu. The test has to be converted into hex files using the `elf2hex` command. Now, in order for the shakti's out executable to work the following files are requires in addition to the `boot.LSB`, `boot.MSB`, and `boot.mem` files.

```
elf2hex 8 33554432 a.out 2147483648 > code.mem
```

Then the following command will generate dump of shakti's execution –

```
./out +rtldump
```

The multiharts are simulated on shakti multicore setup and `rtl.0 dump`, `rtl.1 dump`, `rtl.2 dump`, `rtl.3.dump` files are generated for the respective cores

## VII. SPIKE DUMP VS SHAKTI DUMP

Spike is a simulator which simulates an ideal RISC-V core. After the tests are run on spike, log and memory dump files are obtained. The log files are used to find the coverage of the tests. Coverage refers to the extent to which your test addresses different functionalities of the core being tested. The memory dump is used to verify the Shakti multicore setup. The tests are run on shakti multicore and it's dump files are compared against the spike's dump files.

## VII. TESTS

There are several tests in the `baselist.yaml`. Each test case employed for verification is likely to stimulate distinct functionalities within the assembler. The generated assembly code will tend to vary based on the constructs incorporated into the test. Here are some tests which I used for verifying.

### A) RISC\_V\_ARITHMETIC\_BASIC\_TEST:

Is specifically designed to generate assembly instructions for basic arithmetic operations on an RISC-V processor. It's configurable for different RISC-V instruction set variants (e.g., RV32I, RV64I, multiharts). The test generates assembly instructions for fundamental arithmetic operations like addition, subtraction, multiplication and division.

However, it has support only for RV32I/64I Base Integer, RV32M/64M Multiply and RV32C/64C Compressed extensions. As it doesn't support RV32F/64F Floating – Point extension, it may not cover more advanced arithmetic features like overflow handling or floating-point operations.

The following command is used to generate arithmetic basic tests using *pyflow*.

```
python3 run.py --test=riscv_arithmetic_basic_test --simulator=pyflow --steps gen
```

Based on the *num\_of\_tests*, it will generate that many number of assembly tests with given instruction count named as *riscv\_arithmetic\_basic\_test\_0.S* and *riscv\_arithmetic\_basic\_test\_1.S* and so on

Benefits of *riscv\_arithmetic\_basic\_test*:

- ❑ Verifies the correct functionality of the RISC-V processor's arithmetic unit by providing basic coverage for arithmetic operations.
- ❑ Building block for more complex verification tests involving arithmetic.

#### B ) RISC\_V\_JUMP\_STRESS\_TEST:

Is specifically designed to generate assembly instructions that stress-test the jump functionality of an RISC-V processor. It is configurable for RISC-V Instruction Set variants like RV32/64. The test generates a large number (stressful amount) of assembly instructions for various jump types supported by the target like unconditional jumps (e.g., JAL, JALR), conditional jumps (e.g., BEQ, BNE, BLT, BGE), etc.

The following command is used to generate jump stress tests using *pyflow*.

```
python3 run.py --test=riscv_jump_stress_test --simulator=pyflow --steps gen
```

Based on the *num\_of\_tests*, it will generate that many number of assembly tests with given instruction count named as *riscv\_jump\_stress\_test\_0.S* and *riscv\_jump\_stress\_test\_1.S* and so on.

The spike and Shakti dump files are generated using the commands mentioned before.

Benefits of *riscv\_jump\_stress\_test*:

- ❑ Helps identify issues in the processor's jump logic, especially under stressful conditions with many jumps or complex control flow and handling branch instructions.

These assembly files are converted into a binary file to simulate on spike using –

```
riscv64-unknown-elf-gcc -nostartfiles -T link.ld  
riscv_arithmetic_basic_test_0.S
```

```
spike -l --log=spike.log -p4 a.out
```

```
spike -l --log=spike.dump -p4 a.out
```

Now, the binary file *a.out* has to be converted into hex files to be executed on Shakti multicore setup using the following command -

```
elf2hex 8 33554432 a.out 2147483648 > code.mem  
/out +rtldump
```

This execution will create dump files like *rtl.0 dump*, *rtl.1 dump*, *rtl.2 dump*, *rtl.3 dump* separately for each hart.

## VIII. COMPARISION OF DUMP FILES

The spike.log created will create one single file with memory traces of all 4 cores. A python script *gendump.py* is used to separately obtain memory dump for each core. Now the memory dump file for each core is compared using a python script *diff.py* (it uses the *diff* command) to compare and store the difference in another file.

```
core 0: 3 0x0000000080000054 (0x301d9073) c769_misa 0x8000000000001105  
core 0: 3 0x0000000080000054 (0x301d9073) c769_misa 0x800000000000141105  
core 0: 3 0x0000000080000088 (0x304d9073) c772_mie 0x0000000000001000  
core 0: 3 0x000000008000008c (0x30200073) c768_mstatus 0x000000a000001880  
core 0: 3 0x0000000080000088 (0x304d9073) c772_mie 0x0000000000000000  
core 0: 3 0x000000008000008c (0x30200073) c768_mstatus 0x000000a000000080
```

Fig. 4: The machine instructions which are different

In Fig. 4, the CSR registers seem to have mismatches as the machine instructions are dependent on the CPU architecture. It will be conveyed to the design team for debugging to confirm if it is a RTL bug or not. All other instructions tested seem to be functioning as expected, as there are no reported mismatches.

## IX. TEST COVERAGE

Coverage is crucial in verification because it provides a metric for how thoroughly you've tested a design, particularly in the context of hardware verification especially like using RISC-V DV tests. By analysing coverage metrics, you can create tests specifically targeting those functionalities.

Coverage for *riscv\_arithmetic\_basic\_test*:

The spike.log file is used to create the coverage report using the following command –

```
python3 cov.py --dir arithmetic_basic_test/asm_test/ --
simulator pyflow --enable_visualization
```

In Fig. 5, the number of groups in the table indicate the number of different instructions used in the tests. For example, in the *riscv\_arithmetic\_basic\_test* all basic arithmetic instructions are used except for floating point instructions and jump instructions. As you can see in Fig. 6, the coverage scores for floating and jump instructions are 0.

Groups Coverage Summary		
Total groups in report: 147		
SCORE	WEIGHT	NAME
18.75	1	opcode_cg
19.7917	1	csrrw_cg
60	1	rv32i_misc_cg
50	1	mepc_alignment_cg
35.1562	1	beq_cg
0	1	jal_cg
26.0417	1	lui_cg
47.7679	1	addi_cg

Fig. 5: Coverage report for *riscv\_arithmetic\_basic\_test*

0	1	flw_cg
0	1	fld_cg
0	1	fsw_cg
0	1	fadd_s_cg
0	1	fadd_d_cg
0	1	fsub_s_cg
0	1	fsub_d_cg
0	1	fmul_s_cg
0	1	fmul_d_cg

Fig. 6: The floating-point instructions having score 0

Coverage for *riscv\_jump\_stress\_test* :

The *spike.log* file is used to create the coverage report using the following command –

```
python3 cov.py --dir jump_test/asm_test/ --simulator
pyflow --enable_visualization
```

In Fig. 7, in *riscv\_jump\_stress\_test* all jump instructions are used. The jump instruction scores are non-zero.

Groups Coverage Summary		
Total groups in report: 147		
SCORE	WEIGHT	NAME
21.875	1	opcode_cg
19.7917	1	csrrw_cg
80	1	rv32i_misc_cg
50	1	mepc_alignment_cg
48.0469	1	beq_cg
50.7812	1	jal_cg
26.0417	1	lui_cg

Fig. 7: Coverage report for *riscv\_jump\_stress\_test*

## X. REFERENCES

- [1] <https://github.com/chipsalliance/riscv-dv>
- [2] <https://gitlab.com/shaktiproject/saferv/-/tree/master/multi-core>
- [3] <https://github.com/riscv-software-src/riscv-isa-sim>