Project Report

FPGA Based C-Class Verification

Omkar Bhilare
omkarbhilare45@gmail.com
Mentor: Lavanya J
lavanya.jagan@gmail.com
Shakti Project, IIT Madras

1 Abstract:

C-Class is a member of the SHAKTI family of processors. It is an extremely configurable and commercial-grade 5-stage in-order core supporting the standard RV64GCSUN ISA extensions. Verification is a very important process in chip design. Usually, the software RTL simulation tests on host computers take a very large amount of time. This paper explores the possibility of FPGA-based verification of SHAKTI processors to reduce verification time. AAPG is a tool that is intended to generate random RISC-V programs to test RISC-V cores. In this paper, we ran AAPG tests on FPGA and compared the spike's golden signature dump with the FPGA signature dump. This paper also explores the self checking tests generated by AAPG. Self-checking tests have the advantage of running on FPGA or silicon without much intervention from the host, thereby accelerating the speed verification significantly.

Key Words: C-Class, FPGA, RTL, AAPG, RV64GCSUN, ISA, Verification

2 Introduction:

C-Class processors are configurable and commercial-grade 5-stage in-order core supporting the standard RV64GCSUN ISA extension. Nowadays processor generation is becoming faster and faster. To verify these processors the verification tests runtime must be faster but at the same time verifying directed and random test stimuli.
The verification and debugging fall into following categories: [1]

- **Software RTL Simulation:**
  In software RTL simulation features like assertion and waveforms helps in verification and debugging. COroutine based COsimulation TestBench(CoCoTb) environment can also be used to verify and debug the RTL code in python. The software RTL simulation comes with the advantage of High controllability and visibility. But the Major disadvantage of Software RTL simulation is the slow speed which is low throughput.
- **Hardware Emulation Engines:**
  It takes billions of verification cycles to boot an operating system and executing software applications. The high throughput required for nowadays verification process can only be achieved using specialized hardware engines or FPGA prototyping. The Hardware emulation engines like Cadence Palladium Z1 can be used for faster hardware and software integration. The major disadvantage of Hardware Emulation Engines is the higher cost of actual devices.
- **FPGA based Prototyping:**
  FPGAs are the cheap alternatives to Hardware Emulation Engines. In the last decade or so there are lots of improvements happened in the FPGA Software Toolchains which allow to verify or debug the HDL code on the FPGA itself. The major advantage of FPGA prototyping is the fast speed, low cost. The only issue with it is the less or limited visibility

Comparision of RTL Verification Approach			
RTL Verification Approach	**Speed**	**Controllability**	**Visiblity**
Software RTL Simulation	Very Slow	High	Full
Hardware Emulation Engines	Very Fast	High	Full
FPGA based Prototyping	Fast	Low	Limited

Table 1: Comparision of RTL Verification Approach

3 SHAKTI FPGA Framework:

The SHAKTI FPGA Framework for verification consist of following tools:

1. **RISCV GDB:**
   RISCV GDB is the GNU debugger for the RISC-V platforms. RISC-V GDB is a debugger which let's you control the flow of assembly code which is running on the FPGA.
   In this framework we have used the following GDB features: [2]

   (a) *file Code.riscv*: Selects the binary of code which needs to loaded into FPGA.

   (b) *load*: Loads the all sections from binary to the FPGA.

   (c) *break function-name*: Used for creating breakpoints in the ASM code.

   (d) *dump binary memory mem.bin start end*: Used for Dumping the FPGA signature to the binary file stored in host computer.

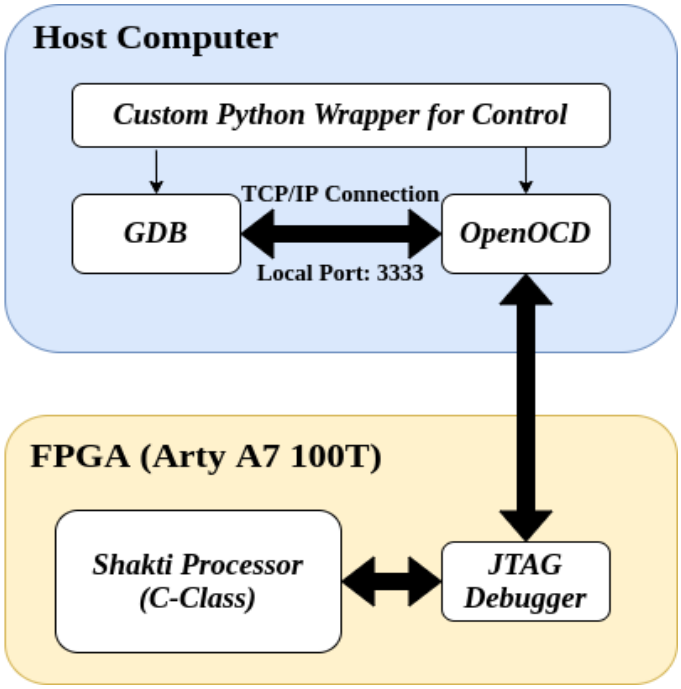


Figure 1: Shakti FPGA framework

2. **OpenOCD:**
   Open On-Chip Debugger (OpenOCD) provides an interface for the RISCV-GDB to connect to the target device (Arty A7-100T). It allows the RISC-V GDB (gdbserver) to connect to the target processor which is running on the FPGA.

3. **JTAG Debugger:**
   An additional hardware device is required to allow the Host Computer to communicate and introspect the target hardware. These devices are often referred to as Debugger devices or Debug adapters. The compiled program is transferred to the target hardware using the Debugger device.

4. **Python Wrapper:**
   AAPG On FPGA (AOF) is a python wrapper which is used on HOST Computer to generate tests and compare the golden and FPGA dump.
   The features of AOF as follows:

   (a) *Extract the FPGA Dump and compare to the Golden (spike) signature*:
   The Python Wrapper automatically runs AAPG Tests on the FPGA and afterwards extract the FPGA Signature into host computer with the help of GDB. It also compares the FPGA and golden signature Dump.

(b) *Multiple Tests Support*:
The Python wrapper can also automatically run N number of tests one after another on the FPGA.
The Steps followed by wrapper for multiple tests are as follows:

i. *python signature.py -tests=N* : Following command runs N number of tests on the FPGA one after other.
ii. First wrapper set up the test number 0, runs the AAPG test on FPGA and extracts the Dump from FPGA. Afterwards compare the FPGA and golden Signature dump.
iii. If both dumps are same, then wrapper deletes the last test folder and setup the next test.
iv. This is done until N-1 Tests. If any tests are fail then they will be automatically stored in folder named *fail tests-number*

(c) *Increases Visibility of the FPGA*:
The main drawback of FPGA based prototyping for debugging and verification of processors is the less visibility and control in the FPGA flow. AAPG has a feature called Self-Checking Tests. In these tests at regular intervals the checksums of CPU Status is stored in memory of CPU at particular location. This set of checksums is nothing but signature. The golden signature (set of checksums) generated by spike will be always true. The python wrapper compares the golden signature generated by the spike to the signature of the FPGA.
The python wrapper can detect exactly where is the mismatch in the checksum and accodingly find the group of instruction causing issue resulting more visibility and control in the verification flow.

5. **Arbitrary Assembly Program Generator (AAPG):**
Arbitrary Assembly Program Generator (AAPG), is a code level verification tool built in-house as part of the Shakti Program. Self Checking tests are a feature in AAPG that setup an assembly test with checksums placed at regular intervals throughout the test, and the values of these checksums provided in a designated reference data section. The concept of Self Checking tests arose for the purpose of verification at the Field Programmable Gate Array (FPGA) level. Currently, one can create and use a self checking test, as an single entity/program to be run in the environment needing verification and identify if the test is passing or failing.[3]
AAPG is a python based package that can be installed directly from the PyPI repository with the command:

```
pip install aapg
```

One can also access the AAPG code from its gitlab repository and install it locally onto their machines from there using the commands:

```
git clone https://gitlab.com/shaktiproject/tools/aapg.git
python setup.py bdist_wheel
pip install dist/*
```

This method of installation helps developers make changes locally to AAPG, test and use them in their machines.

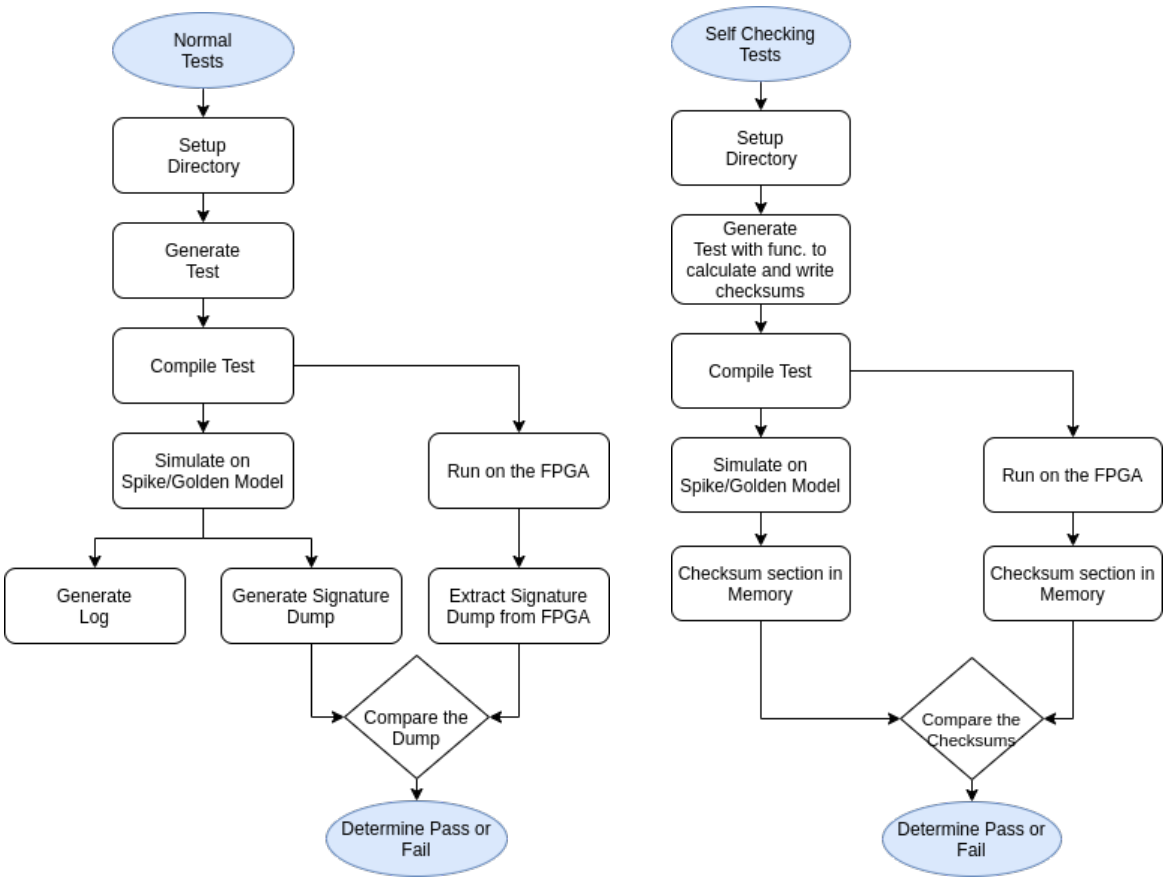
4 AAPG Tests on FPGA:

1. **Normal AAPG Tests on the FPGAs:**
A user who wants to verify their riscv design, will generate a random program using AAPG[3]. Then compile it on the golden model spike. Then using the custom wrapper run it on the FPGA. The wrapper runs the same test on the soft SOC running on the FPGA. Then extract the memory dump from FPGA and compare with the memory dump of the spike. Depending on the comparison test result is decided.

2. **Self-Checking AAPG Tests on the FPGAs:**
The config file of AAPG has a section labelled self-checking, which contains:

(a) **Rate**: This takes an integer in the range (1,infinity) as input. It controls the number of instructions between two check sums. If rate is 10, then a checksum will be added every 10 instructions.[3]

In Self-Checking AAPG tests, these sets of checksums are embedded in particular part of memory. First the test program is run on the golden spike model from which we get golden signature dump. Then python wrapper runs these Self-Checking AAPG tests on the FPGA and extracts the set of checksums from the memory region and compares with the reference golden dump. Depending on the comparison test result is decided.



(a) Normal AAPG Tests on the FPGA

(b) Self-Checking AAPG Tests on the FPGA

Figure 2: FPGA Test Flow

5 Installation:

In this paper we have ran AAPG tests on the **SHAKTI's Vajra SOC** which was running on **Arty A7-100t FPGA**. So to program the FPGA with soft SOC we need following tools installed on the system:[4]

1. Bluespec Compiler
2. Device Tree Compiler
3. Vivado 2017 or above
4. Miniterm
5. OpenOCD

The detailed steps can be found in *section 3.3.2* of shakti user manual.

1. Now that we have all the softwares, we need to flash the FPGA with Shakti Vajra SOC:[5]

```
git clone https://gitlab.com/shaktiproject/gc2020.git
cd gc2020/c64-a100/
pip3 install -r requirements.txt
python3 -m configure.main
make -j<jobs> generate_verilog
make generate_boot_files ip_build arty_build generate_mcs program_mcs JOBS=<jobs>
```

2. Connecting to the Target to check whether Vajra SOC is running or not:

   (a) In a New Terminal window

```
openocd -f shakti-arty.cfg
```

(b) In yet Another Terminal window

```
riscv64-unknown-elf-gdb -x gdb.script
```

(c) Launch UART Console

```
sudo miniterm /dev/ttyUSB1 19200
```

(d) On pressing the reset-button on the board, UART console should display the following: *S*hakti-Boot-Logo

Now that we have FPGA ready with Vajra SOC running, we can now run AAPG tests on the FPGA.

1. Single AAPG Normal/Self-Checking tests on the FPGA:

   (a) Setup AAPG Test and clone the Python wrapper for running tests on FPGA:

   ```
   git clone https://gitlab.com/verif-group/verif-interns/aapg_fpga_test
   cd aapg_fpga_test
   aapg setup
   cd work
   # Changes in the config.yaml according to the need
   cd ..
   aapg gen    #For normal single tests
   aapg gen --self_checking #For normal self checking tests
   cd work
   make
   ```

   (b) Run the Single AAPG Normal/Self-Checking Tests on the FPGA:

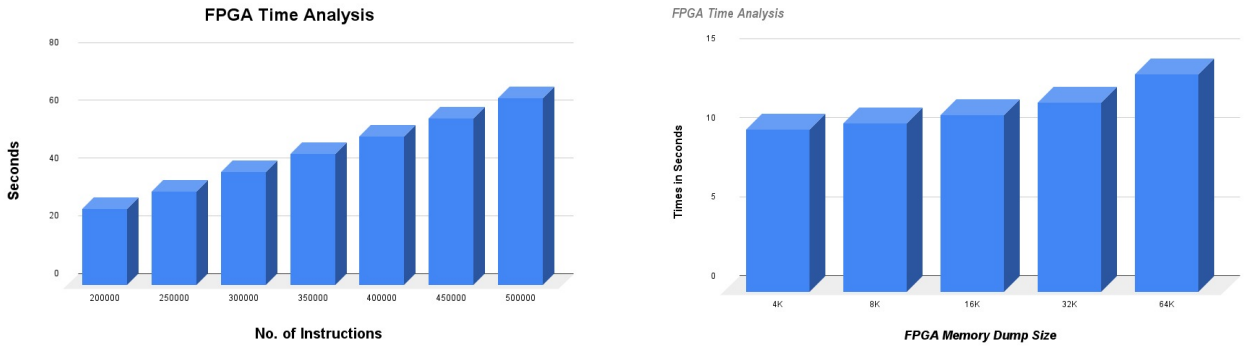
   ```
   cd aapg_fpga_test
   python signature.py -run
   ```

2. N number of multiple AAPG Self-Checking tests on the FPGA:

   (a) The script automatically creats and make the tests, the user only needs to do changes in the config_defined file. Passed tests folder will be deleted by the script automatically, If any tests fails, it will be stored in test_number folder. Result Can be found in *s*tatus.txt

   (b)
   ```
   cd aapg_fpga_tests
   # Do changes in config_defined for tests
   python signature -tests=N #N continue tests will be performed on FPGA
   # for example: python3 signature.py -tests=5  #For 5 Regression Tests, 5 can be any number
   ```

6 Results:



(a) Instruction Test

(b) Memory Dump test

Figure 3: FPGA Time Analysis

These tests were ran on the Shakti's VAJRA SOC. The Major Advantage of FPGA in verification flow is the Speed. As shown in the Figure 3, 5,00,000 RISC-V Instruction took only 64 Seconds to run on the FPGA. The

visibility and control is required in any Verification Framework. Traditional FPGA flow has usually less visibility and control but with the help of checksums in AAPG and python wrapper we can see where is the mismatch. In future we need to run this framework on the Shakti's I-class.

References

[1] Esperanto Technologies, University of California, Berkeley, "DESSERT: Debugging RTL Effectively with State Snapshotting for Error Replays across Trillions of cycles"

[2] "RISC-V GDB TUTORIAL", Developed by: SHAKTI Development Team @ iitm '20

[3] "SELF CHECKING TESTS FROM AAPG", Developed by: SHAKTI Team @ iitm, ABISHEK TAIKAD SHYAMSUNDER.

[4] "SHAKTI DEVELOPMENT BOARD USER MANUAL", DEVELOPED BY: SHAKTI DEVELOPMENT TEAM @ IITM '19

[5] "Gitlab Repository of C64-A100 (Vajra)", gitlab.com/shaktiproject/sp2020/c64-a100