2023 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION CHINA

accellera SYSTEMS INITIATIVE

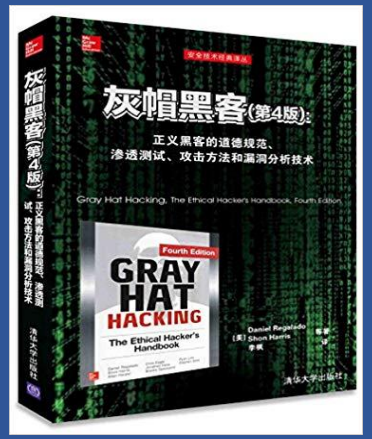
DESIGN AND VERIFICATION™ CONFERENCE AND EXHIBITION

Shanghai | September 20, 2023

Beyond UVM

Feng Li (李枫)
hkli2012@126.com
Sep 20, 2023

Who Am I



An indie developer from China：

- The main translator of the book «Gray Hat Hacking The Ethical Hacker's Handbook, Fourth Edition» (ISBN：9787302428671) & «Linux Hardening in Hostile Networks, First Edition» (ISBN: 9787115544384)
- Pure software development for ~15 years (~11 years on Mobile dev)
- Actively participating Open Source Communities:
  https://github.com/XianBeiTuoBaFeng2015/MySlides
- Recently, focus on infrastructure of Cloud/Edge Computing, AI, IoT, Programming Languages & Runtimes, Network, Virtualization, RISC-V, EDA, 5G/6G…

Agenda

I. Background
- Technology Stack
- Testbeds

II. Enhancing Cocotb
- Comparison
- Potential Enhancements

III. Project CocotbII
- Cocotb with emerging Python-based DSLs
- Cocotb with Dlang
- Cocotb with AI
- Chisel Verification with Cocotb?

IV. Wrap-up

I. Background

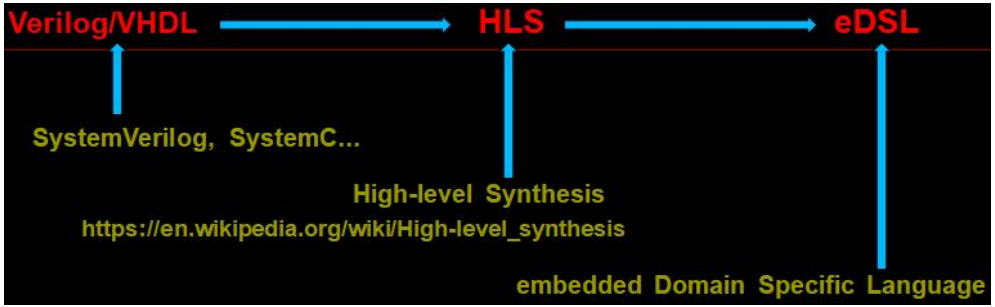
1) Technology Stack

1.1 FOSS EDA

- https://en.wikipedia.org/wiki/Comparison_of_EDA_software
- https://semiwiki.com/wikis/industry-wikis/eda-open-source-tools-wiki/
- https://fossi-foundation.org/
- https://ieeexplore.ieee.org/document/9398963
- https://ieeexplore.ieee.org/document/9398960
- https://ieeexplore.ieee.org/document/9336682
- https://ieeexplore.ieee.org/document/9105619
- …

I. Background

1.2 Evolution of HDLs

- https://en.wikipedia.org/wiki/Hardware_description_language
- https://hdl.github.io/awesome/items/
- 

Typical eDSLs

-

Host Language	Typical eDSLs
Haskell	Bluespec, Clash...
Scala	Chisel, SpinalHDL...
Python	Amaranth/FHDL, PyGears...

- Finally convert to Verilog/VHDL (https://en.wikipedia.org/wiki/Source-to-source_compiler)

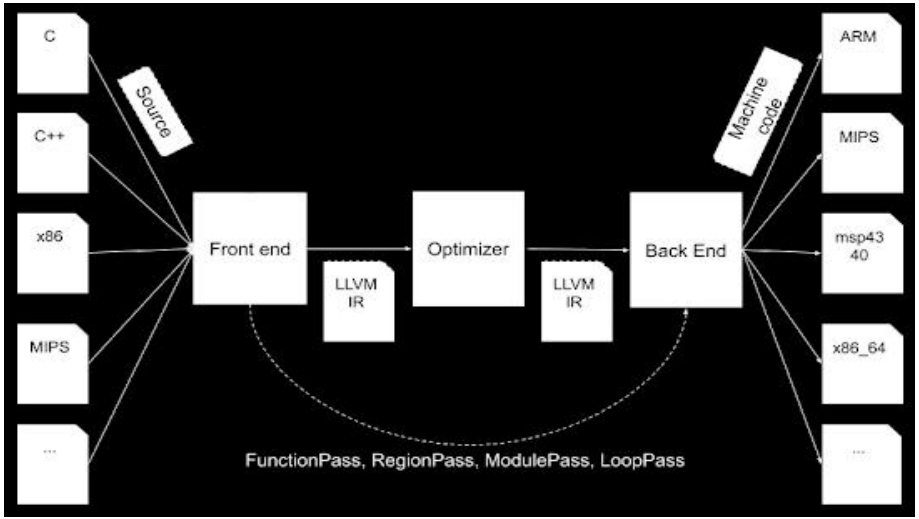
I. Background

1.3 LLVM

- https://en.wikipedia.org/wiki/LLVM

LLVM is a set of compiler and toolchain technologies,[5] which can be used to develop a front end for any programming language and a back end for any instruction set architecture. LLVM is designed around a language-independent intermediate representation (IR) that serves as a portable, high-level assembly language that can be optimized with a variety of transformations over multiple passes.[6]

- https://llvm.org
-



Source: http://blog.k3170makan.com/2020/04/learning-llvm-i-introdution-to-llvm.html

I. Background

1.3.1 MLIR

- https://mlir.llvm.org/

Multi-Level Intermediate Representation

The MLIR project is a novel approach to building reusable and extensible compiler infrastructure. MLIR aims to address software fragmentation, improve compilation for heterogeneous hardware, significantly reduce the cost of building domain specific compilers, and aid in connecting existing compilers together.

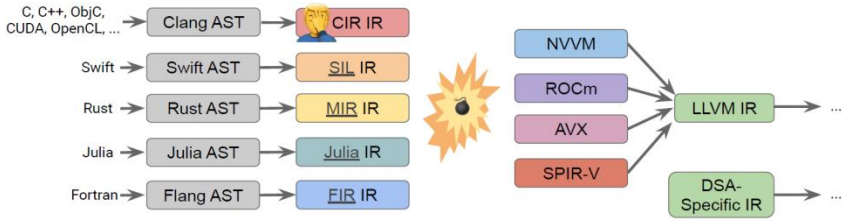
- Motivation

MLIR is intended to be a hybrid IR which can support multiple different requirements in a unified infrastructure. For example, this includes:

- The ability to represent dataflow graphs (such as in TensorFlow), including dynamic shapes, the user-extensible op ecosystem, TensorFlow variables, etc.
- Optimizations and transformations typically done on such graphs (e.g. in Grappler).
- Ability to host high-performance-computing-style loop optimizations across kernels (fusion, loop interchange, tiling, etc.), and to transform memory layouts of data.
- Code generation “lowering” transformations such as DMA insertion, explicit cache management, memory tiling, and vectorization for 1D and 2D register architectures.
- Ability to represent target-specific operations, e.g. accelerator-specific high-level operations.
- Quantization and other graph transformations done on a Deep-Learning graph.
- Polyhedral primitives.
- Hardware Synthesis Tools / HLS.

I. Background

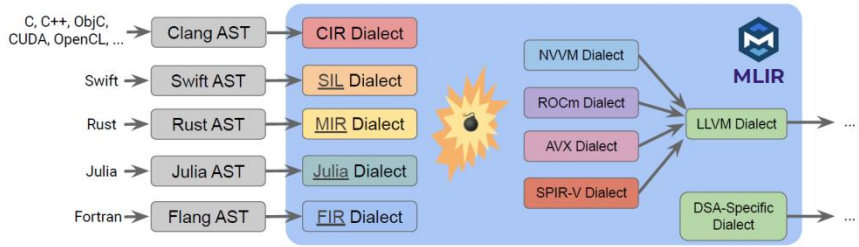
From LLVM to MLIR



- Different back-ends demand customized IR for optimization
- DSAs even cannot use LLVM for generating back-end codes and demand their own IR for code generation

Severe Fragmentation: IRs have different implementations and "frameworks"

MLIR: "Meta IR" and Compiler Infrastructure



MLIR is a **"Meta IR"** and **compiler infrastructure** for:

- Design and implement **dialect**
- Optimization and transform inside of a **dialect**
- Conversion between different **dialects**
- Code generation of **dialect**

Source: https://hanchenye.com/assets/pdf/Gatech_CIRCT_Slides_Hanchen.pdf

I. Background

1.3.2 CIRCT

- https://circt.llvm.org/

  Circuit IR Compilers and Tools

- **Motivation**

The EDA industry has well-known and widely used proprietary and open source tools. However, these tools are inconsistent, have usability concerns, and were not designed together into a common platform. Furthermore these tools are generally built with Verilog (also VHDL) as the IRs that they interchange. Verilog has well known design issues, and limitations, e.g. suffering from poor location tracking support.

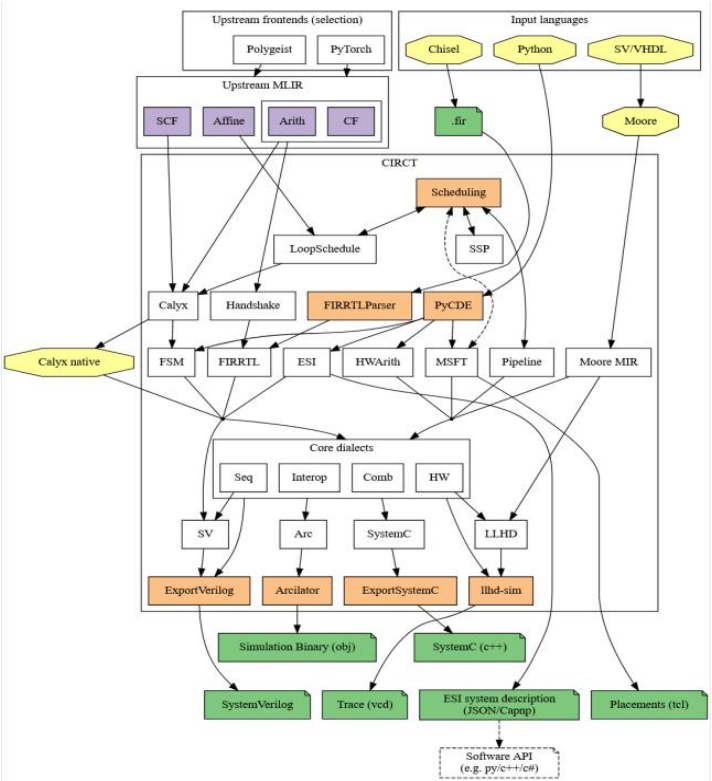
The CIRCT project is an (experimental!) effort looking to apply MLIR and the LLVM development methodology to the domain of hardware design tools. Many of us dream of having reusable infrastructure that is modular, uses library-based design techniques, is more consistent, and builds on the best practices in compiler infrastructure and compiler design techniques.

By working together, we hope that we can build a new center of gravity to draw contributions from the small (but enthusiastic!) community of people who work on open hardware tooling. In turn we hope this will propel open tools forward, enables new higher-level abstractions for hardware design, and perhaps some pieces may even be adopted by proprietary tools in time.

- https://circt.llvm.org/docs/Charter/

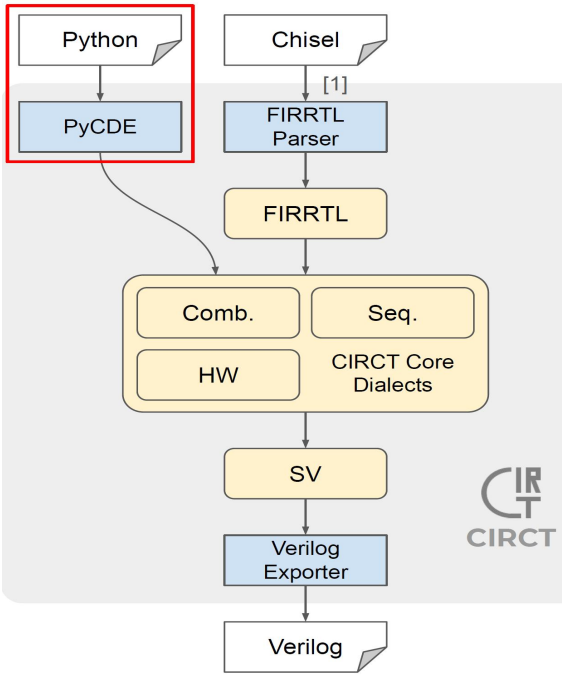
I. Background

Dialects and how they interact & PyCDE



Source: https://circt.llvm.org/includes/img/dialects.svg

https://circt.llvm.org/docs/PyCDE/



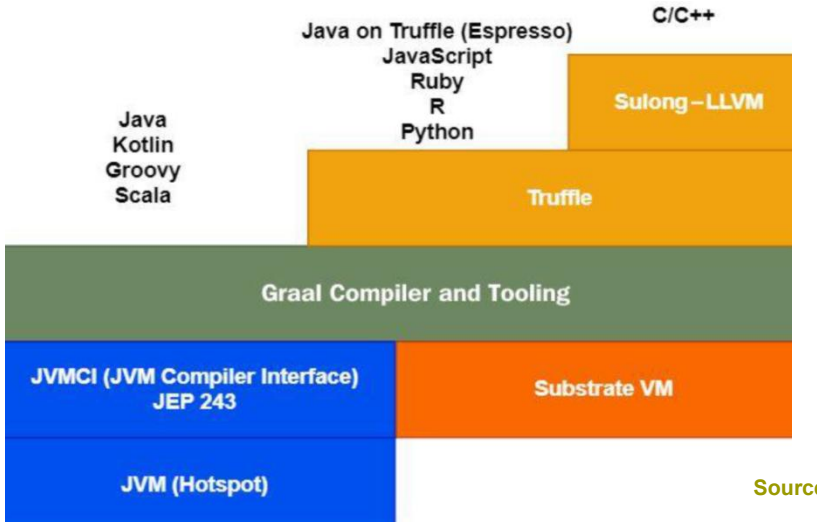
Source: "Compilers for Domain-Specific Accelerators", Hanchen Ye, Gatech ECE6100/CS6290 guest lecture.

I. Background

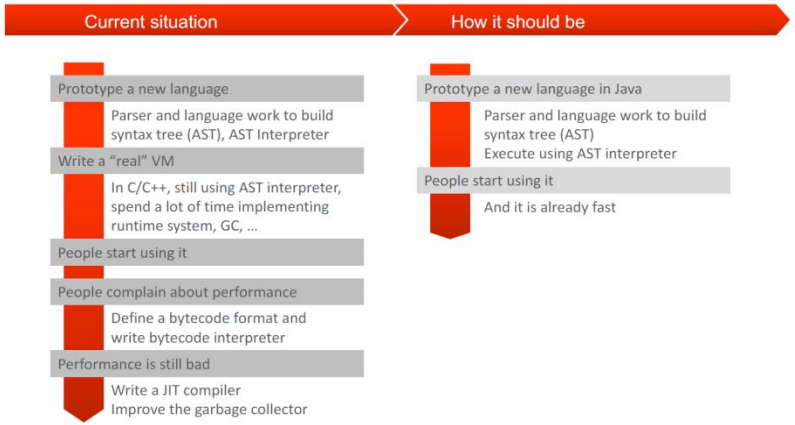
1.4 GraalVM

- https://en.wikipedia.org/wiki/GraalVM
- https://www.graalvm.org/

A high-performance JDK distribution designed to accelerate the execution of applications written in Java and other JVM languages like JavaScript, Python, and a number of other popular languages.



Source: https://static.packt-cdn.com/downloads/9781800564909_ColorImages.pdf

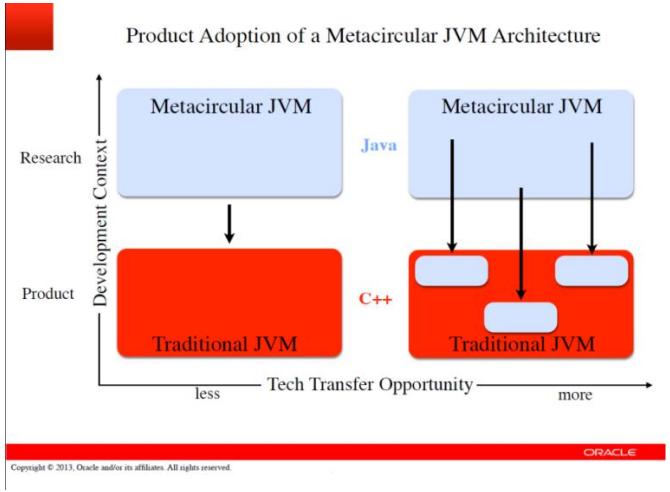


Source: "Turning the JVM into a Polyglot VM with Graal", Chris Seaton, Oracle Labs.

- https://github.com/oracle/graal/blob/master/truffle/docs/Languages.md

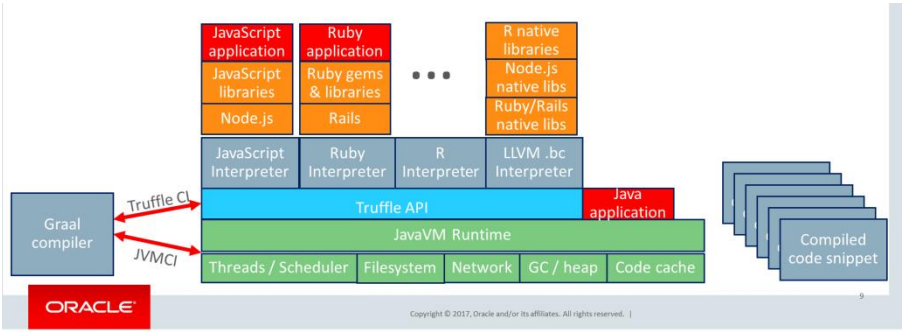
I. Background

Meta-Circular

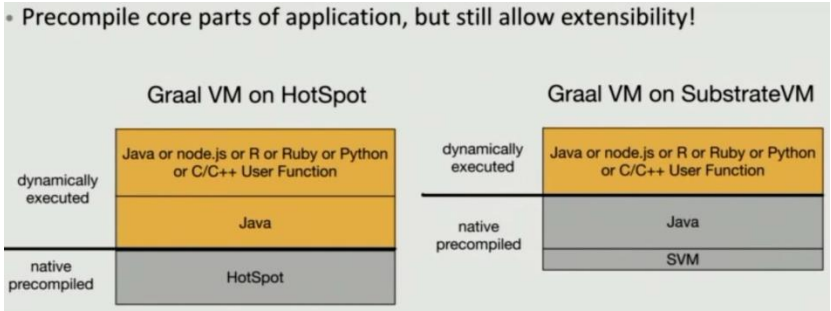


Source: https://chrisseaton.com/truffleruby/jokerconf17/

A hybrid of static & dynamic runtimes



Source: https://ics.psu.edu/wp-content/uploads/2017/02/GraalVM-PSU.pptx



Source: "Adopting Java for the Serverless world", Vadym Kazulkin, JUG London 2020.

- https://github.com/oracle/graal/tree/master/sulong
- A Universal High-Performance Polyglot VM.

I. Background

Current release and roadmap

- https://www.graalvm.org/release-calendar/

JDK 21

- https://openjdk.org/projects/jdk/21/

430: String Templates (Preview)
431: Sequenced Collections
439: Generational ZGC
440: Record Patterns
441: Pattern Matching for switch
442: Foreign Function & Memory API (Third Preview)
443: Unnamed Patterns and Variables (Preview)
444: Virtual Threads
445: Unnamed Classes and Instance Main Methods (Preview)
446: Scoped Values (Preview)
448: Vector API (Sixth Incubator)
449: Deprecate the Windows 32-bit x86 Port for Removal
451: Prepare to Disallow the Dynamic Loading of Agents
452: Key Encapsulation Mechanism API
453: Structured Concurrency (Preview)

JDK 19

405: Record Patterns (Preview)
422: Linux/RISC-V Port
424: Foreign Function & Memory API (Preview)
425: Virtual Threads (Preview)
426: Vector API (Fourth Incubator)
427: Pattern Matching for switch (Third Preview)
428: Structured Concurrency (Incubator)

I. Background

1.5 Python

- https://en.wikipedia.org/wiki/Python_(programming_language)

Ranking

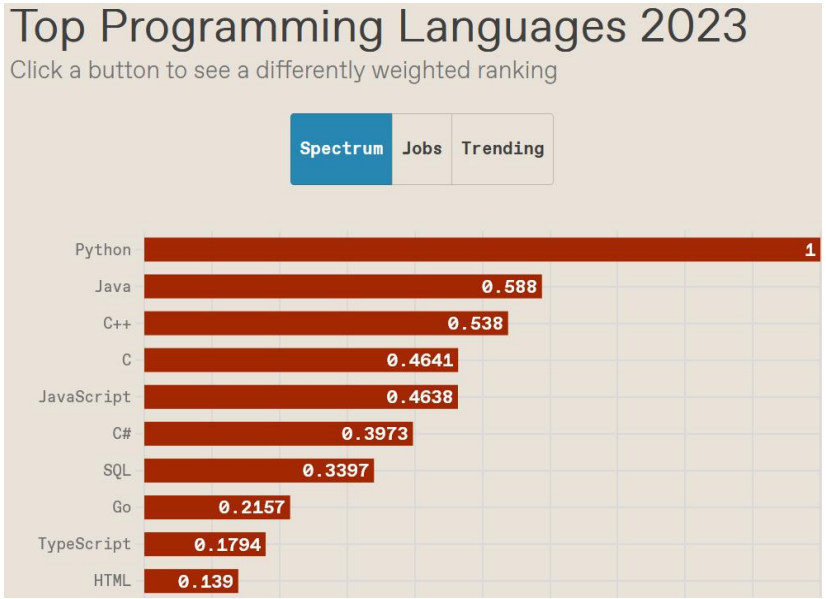
- PYPL

Worldwide, Sept 2023 :

Rank	Change	Language	Share	1-year trend
1		Python	27.99 %	+0.1 %
2		Java	15.9 %	-1.1 %
3		JavaScript	9.36 %	-0.1 %
4		C#	6.67 %	-0.4 %
5		C/C++	6.54 %	+0.3 %
6		PHP	4.91 %	-0.4 %
7		R	4.4 %	+0.2 %
8		TypeScript	3.04 %	+0.2 %
9	↑↑	Swift	2.64 %	+0.6 %
10		Objective-C	2.15 %	+0.1 %

Source: http://pypl.github.io/PYPL.html

IEEE Spectrum



Source: https://spectrum.ieee.org/top-programming-languages/

I. Background

- **GitHut**

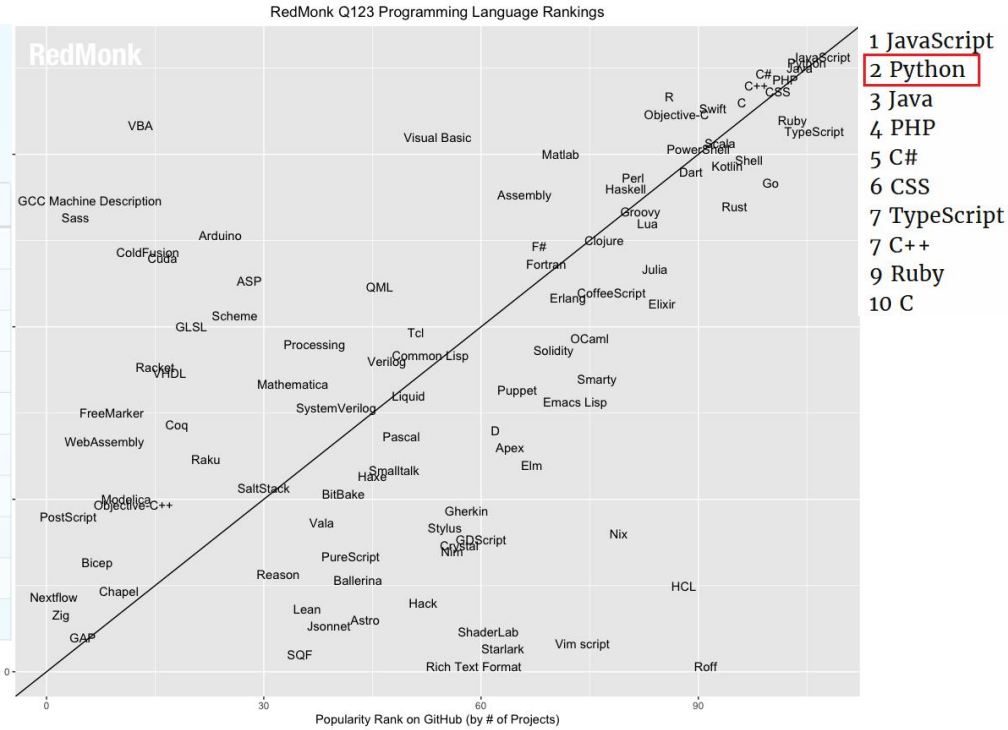
Year: 2023 | Quarter: 2

Number of languages: 50

# Ranking	Programming Language	Percentage (YoY Change)	YoY Trend
1	Python	17.355% (+0.673%)	
2	Java	11.387% (-0.212%)	
3	Go	10.877% (+1.574%)	^
4	C++	9.994% (+0.301%)	
5	JavaScript	9.542% (-0.869%)	v
6	TypeScript	7.673% (-0.988%)	
7	PHP	5.086% (-0.178%)	
8	C	4.615% (+0.514%)	^
9	Ruby	4.494% (-0.386%)	v
10	C#	3.270% (-0.111%)	

Source: http://https://madnight.github.io/githut/#/pull_requests/2023/2

- **RedMonk**



Source: https://redmonk.com/sogrady/2023/05/16/language-rankings-1-23/

- **Is Python the Top1 programming language in AI?**

I. Background

1.5.1 Implementations

- https://wiki.python.org/moin/PythonImplementations
- You may refer to our previous talk "A survey of current Python implementations" at PyCon China 2021 and the upcoming follow-ups.
- …

I. Background

1.5.1.1 CPython (Official)

- https://www.python.org/

***Bottlenecks** of current CPython implementation*

- No JIT (Just in time) compiler
- GIL (https://en.wikipedia.org/wiki/Global_interpreter_lock)
- …

I. Background

Make CPython Faster

- Plan from the Father of Python
  https://thenewstack.io/guido-van-rossums-ambitious-plans-for-improving-python-performance

- https://www.zdnet.com/article/python-programming-we-want-to-make-the-language-twice-as-fast-says-its-creator/

Microsoft currently has five core developers who contribute to the development of CPython, including Brett Cannon, Steve Dower, Guido van Rossum, Eric Snow, and Barry Warsaw — all veterans in the Python core developer community.

I. Background

- **The “Shannon Plan”**

- Posted to GitHub and python-dev last October
  - github.com/markshannon/faster-cpython
- Based on experience with “HotPy”, “HoyPy 2”
- Promises 5x in 4 years (1.5x per year)
- Looking for funding

Source: https://github.com/markshannon/faster-cpython/blob/master/plan.md

The stages to high performance

Stage 1 -- Python 3.10

The key improvement for 3.10 will be an adaptive, specializing interpreter. The interpreter will adapt to types and values during execution, exploiting type stability in the program, without needing runtime code generation.

Stage 2 -- Python 3.11

This stage will make many improvements to the runtime and key objects. Stage two will be characterized by lots of "tweaks", rather than any "headline" improvement. The planned improvements include:

- Improved performance for integers of less than one machine word.
- Improved peformance for binary operators.
- Faster calls and returns, through better handling of frames.
- Better object memory layout and reduced memory management overhead.
- Zero overhead exception handling.
- Further enhancements to the interpreter
- Other small enhancements.

Stage 3 -- Python 3.12 (requires runtime code generation)

Simple "JIT" compiler for small regions. Compile small regions of specialized code, using a relatively simple, fast compiler.

Stage 4 -- Python 3.13 (requires runtime code generation)

Extend regions for compilation. Enhance compiler to generate superior machine code.

Source: https://github.com/markshannon/faster-cpython/blob/master/plan.md

https://github.com/markshannon/
https://github.com/faster-cpython/
https://www.python.org/dev/peps/pep-0659/

…

I. Background

1.5.1.2 GraalPy

- https://www.graalvm.org/python/
- https://github.com/oracle/graalpython

A Python 3 implementation built on GraalVM.

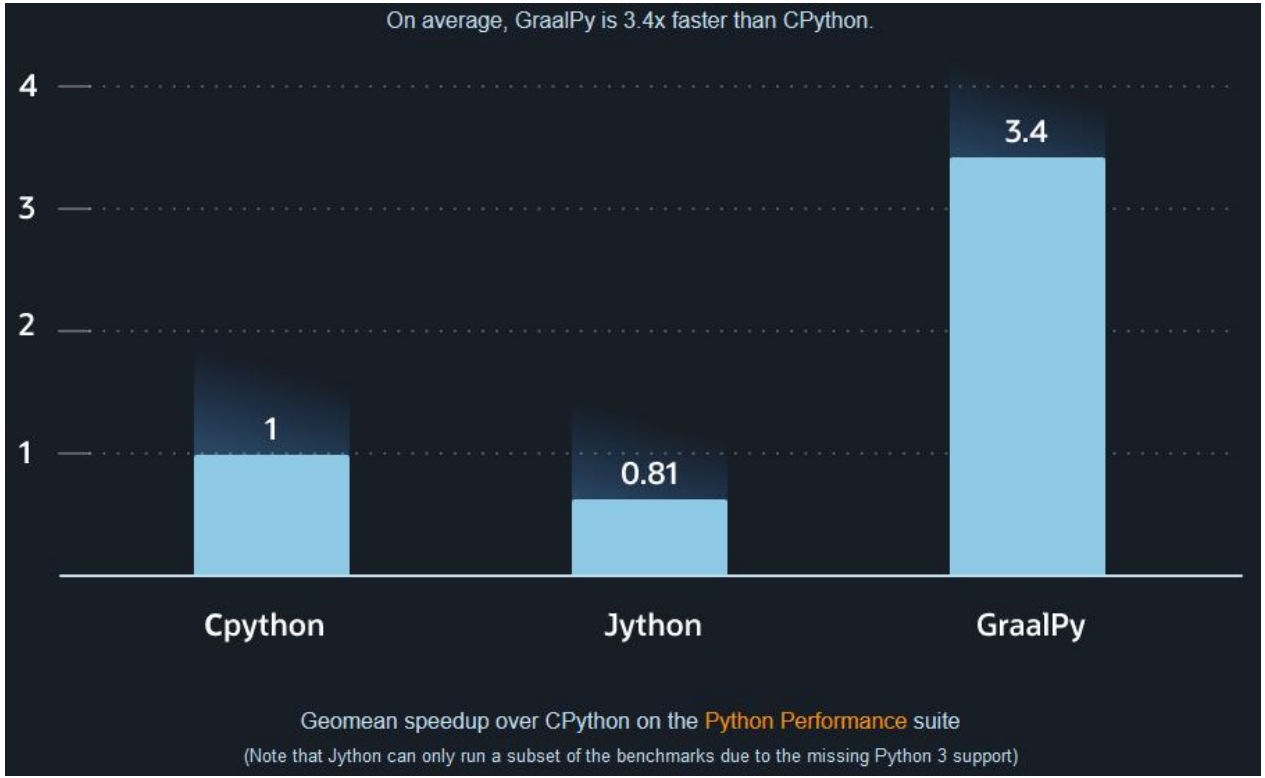
GraalPy, the GraalVM Implementation of Python

GraalPy is an implementation of the Python language on top of GraalVM. A primary goal is to support PyTorch, SciPy, and their constituent libraries, as well as to work with other data science and machine learning libraries from the rich Python ecosystem. GraalPy can usually execute pure Python code faster than CPython, and nearly match CPython performance when C extensions are involved. GraalPy currently aims to be compatible with Python 3.10. While many workloads run fine, any Python program that uses external packages could hit something unsupported. At this point, the Python implementation is made available for experimentation and curious end-users. We welcome issue reports of all kinds and are working hard to close our compatibility gaps.

- **Benefits**

III. Project Cocotbll

- **Performance**

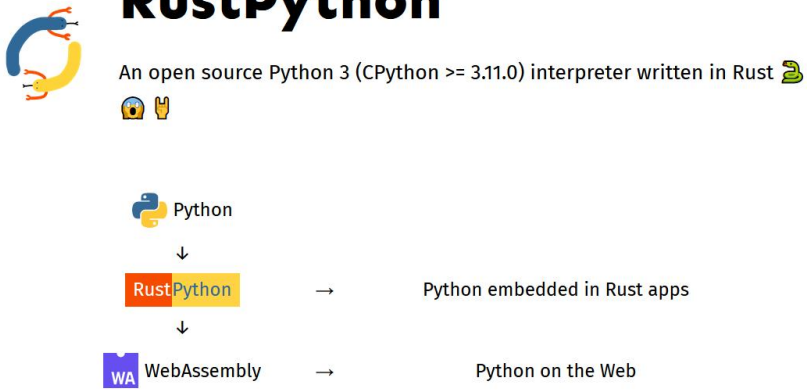


Source: https://www.graalvm.org/python

I. Background

1.5.1.3 RustPython

- https://rustpython.github.io/

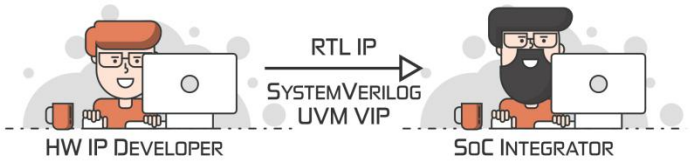


- https://github.com/RustPython/RustPython
- Support WASI!
- Built-in support for any platform that support by Rust toolchain.
- RustPython has a very experimental JIT compiler that compile Python functions into native code.

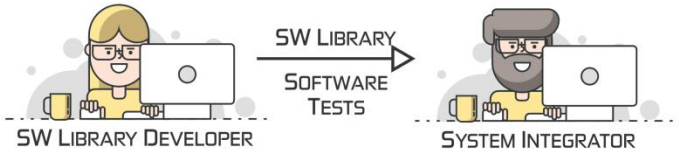
I. Background

1.6 Hardware Verification

- How does it impact the Hardware Verification domain? Traditionally, hardware and software are designed and developed in isolation. In an ASIC hardware design scenario, an IP team codes the RTL, and verifies it using SystemVerilog powered UVM testbenches before handing off the IP to the SoC integration team. An essential part of the IP -> SoC handoff is the UVM test suit that is required to be run at the SoC or at a subsystem level to make sure that the IP integration is seamless.



A similar work-flow is undertaken by software teams. A software IP (or library) developer passes on a set of tests to the application development team to make sure that the SW IP works without glitches in the application/system development environment.



FPGA based Hardware Accelerator technology requires a complete re-look at how verification is done today, and how it needs to evolve. Since a hardware accelerator integrates tightly with the processor, a lot more hardware-software coverification is obviously required.

Source: http://uvm.io/blog/2019/04/accelerated-uvm

Hardware Verification Language (HVL)

- https://en.wikipedia.org/wiki/Hardware_verification_language

I. Background

1.7 UVM

- **https://en.wikipedia.org/wiki/Universal_Verification_Methodology**

The **Universal Verification Methodology** (UVM) is a standardized methodology for verifying integrated circuit designs. UVM is derived mainly from the OVM (Open Verification Methodology) which was, to a large part, based on the eRM (e Reuse Methodology) for the e Verification Language developed by Verisity Design in 2001. The UVM class library brings much automation to the SystemVerilog language such as sequences and data automation features (packing, copy, compare) etc., and unlike the previous methodologies developed independently by the simulator vendors, is an Accellera standard with support from multiple vendors: Aldec, Cadence, Mentor Graphics, Synopsys, Xilinx Simulator(XSIM).

History [edit]

In December 2009, a technical subcommittee of Accellera — a standards organization in the electronic design automation (EDA) industry — voted to establish the UVM and decided to base this new standard on the Open Verification Methodology (OVM-2.1.1),[1] a verification methodology developed jointly in 2007 by Cadence Design Systems and Mentor Graphics.

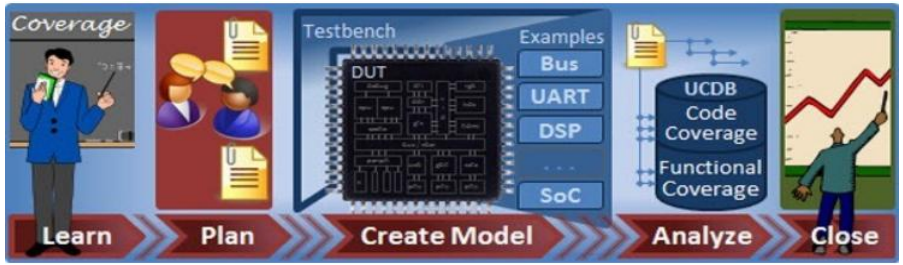
On February 21, 2011, Accellera approved the 1.0 version of UVM.[2] UVM 1.0 includes a Reference Guide, a Reference Implementation in the form of a SystemVerilog base class library, and a User Guide.[2]

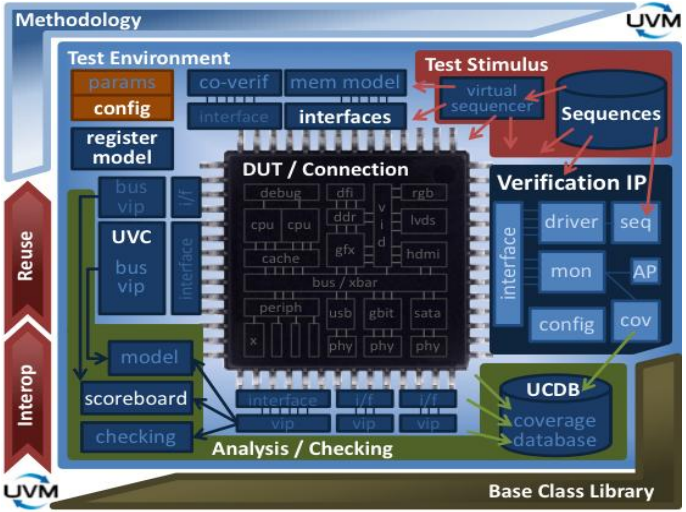
Definitions [edit]

- Agent - A container that emulates and verifies DUT devices
- Blocking - An interface that blocks tasks from other interfaces until it completes
- DUT - Device under test, what you are actually testing
- DUV - Device Under Verification
- Component - A portion of verification intellectual property that has interfaces and functions.
- Transactor - see component
- Verification Environment Configuration - those settings in the DUT and environment that are alterable while the simulation is running
- VIP - verification intellectual property

...

I. Background





Source: https://verificationacademy.com/cookbook

- https://ieeexplore.ieee.org/document/9195920
- https://www.accellera.org/community/uvm/
- The latest version of UVM is 1.2, and 2.0 is on the way.
  https://www.accellera.org/downloads/standards/uvm
- https://www.accellera.org/downloads/standards/uvm
- https://www.accellera.org/downloads/drafts-review
- https://en.wikipedia.org/wiki/SystemVerilog

I. Background

1.8 Vlang

- https://github.com/coverify-org/vlang-docs

Next Generation Verification Language that base on 

Vlang Features at a Glance

Multicore Vlang enables concurrent programming. End user can fine-tune the number of concurrently running threads at module level. Vlang also enables concurrency at a higher abstraction by allowing multiple simulators running in parallel.

Constrained Randomization Full blown and efficient. Concurrency enabled.

UVM Compliance Word-to-word translation of SystemVerilog UVM. More efficient and user-friendly due to generic programming.

Object Oriented Programming Support for function/operator overloading.

Safety and Productivity Automatic Garbage Collection. Exception Handling. Unittests.

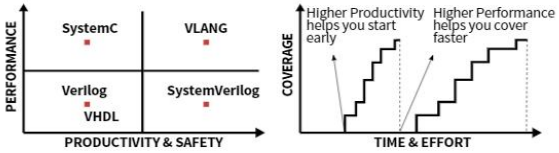
Systems Programming Allows low level access to hardware resources. Allows embedded assembly language.

Interface with other Languages Full blown C++ interface. VHPI/VPI bindings with VHDL and SystemVerilog.

Licensing Provided free under open source boost license. Vlang UVM library is available under Apache2 license.

Verification with Vlang

Even as the chip complexity keeps increasing, we continue to rely on same old RTL methodology to design our chips. As a result the abstraction gap between the design and the specification is increasing exponentially.
Vlang attempts to bridge this gap by providing you a high productivity and high performance verification environment.



Higher Productivity means that you take less time in building your verification infrastructure and higher performance means that your regression runs much faster.
If you have ESL as part of your SoC development flow, there are additional reasons that you should use Vlang to verify your ESL models. Vlang supports much better integration with C++ compared to SystemVerilog. Vlang is ABI compatible with C/C++. Vlang also allows you to call any method (including virtual methods) on C++ objects right from Vlang without any boilerplate code. In comparison SystemVerilog DPI interface is limited to C language. As a result any interface between SystemC and SystemVerilog tends to be highly inefficient.
Yet another advantage is that Vlang is free and open source just like your SystemC simulator.

Source: https://fdocuments.in/document/vlang-flyer.html?page=2 (Date Post 28-Dec-2015)

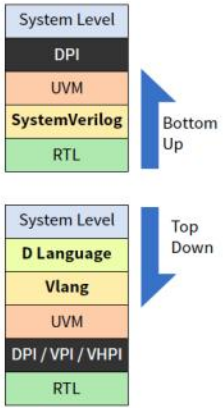
I. Background

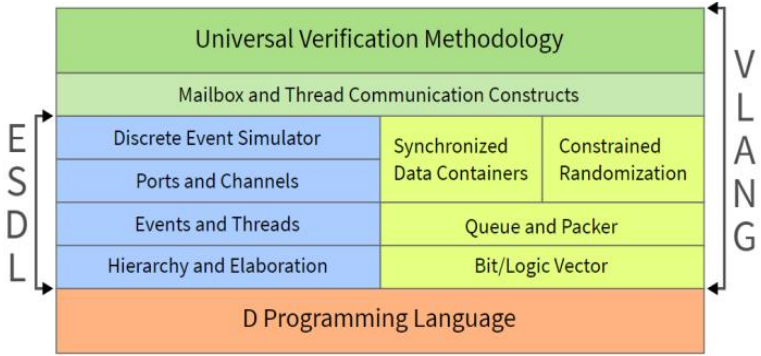
Top Down Verification Stack:

System Verilog integrates tightly with RTL;
Vlang with System Level

- Vlang is built on top of D Programming Language which provides ABI compatibility with C/C++
- Vlang interfaces with RTL using DPI
  - DPI overhead is compensated by parallel execution of testbench
- Vlang offers zero communication overhead when integrating with Emulation Platforms and with Virtual Platforms

SystemVerilog Simulation





Source: https://dvcon-proceedings.org/wp-content/uploads/introduction-to-next-generation-verification-language-vlang-presentation.pdf (DVCon Europe 2014)

Vlang vs SystemVerilog vs SystemC:

https://fdocuments.in/document/vlang-flyer.html?page=2 (Date Post 28-Dec-2015)

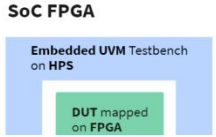
Just as a reference since a little out of date…

For more details, you may refer to our report "D-based Next Generation Verification Language" and its update.

I. Background

Embedded UVM:

http://uvm.io/

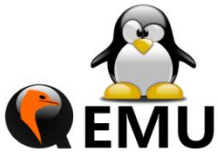


Spawn your own Emulation Platform for $100

Create your own SoCFPGA based Emulator for $100 and upto 100X speedup, with an **Embedded UVM** testbench running on HPS and DUT mapped on FPGA.

Run UVM Tests with Vivado and with GHDL

Opensource and Free **IEEE UVM 1.0** port complete with Constrained Reandomization, released under Apache2/Boost license.

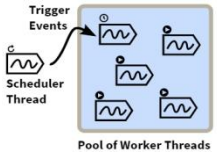


Co-Simulate your DUT with Device Drivers

LLVM powered native compilation on ARM and other embedded processors, with runtime Footprint small enough to run UVM on Raspberry PI and Beaglebone.

Deploy UVM Testbenches on Software Stack

LLVM powered native compilation on ARM and other embedded processors, with runtime Footprint small enough to run UVM on Raspberry PI and Beaglebone.



Scale your Testbench to Multicore Servers

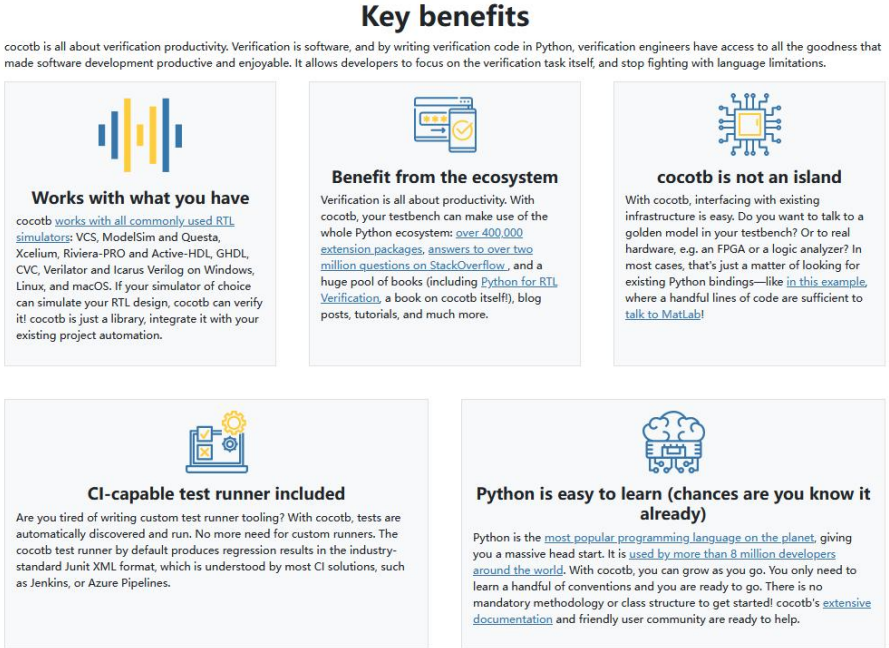
The first and yet the only UVM implementation that lets your testbench run on multiple cores. Lets your testbench scale on Multicore server machine.

I. Background

1.9 Cocotb

- https://www.cocotb.org/

A Coroutine based Cosimulation TestBench environment for verifying VHDL and Verilog RTL using Python.



Key benefits

cocotb is all about verification productivity. Verification is software, and by writing verification code in Python, verification engineers have access to all the goodness that made software development productive and enjoyable. It allows developers to focus on the verification task itself, and stop fighting with language limitations.

Works with what you have

cocotb works with all commonly used RTL simulators: VCS, ModelSim and Questa, Xcelium, Riviera-PRO and Active-HDL, GHDL, CVC, Verilator and Icarus Verilog on Windows, Linux, and macOS. If your simulator of choice can simulate your RTL design, cocotb can verify it! cocotb is just a library, integrate it with your existing project automation.

Benefit from the ecosystem

Verification is all about productivity. With cocotb, your testbench can make use of the whole Python ecosystem: over 400,000 extension packages, answers to over two million questions on StackOverflow, and a huge pool of books (including Python for RTL Verification, a book on cocotb itself!), blog posts, tutorials, and much more.

cocotb is not an island

With cocotb, interfacing with existing infrastructure is easy. Do you want to talk to a golden model in your testbench? Or to real hardware, e.g. an FPGA or a logic analyzer? In most cases, that's just a matter of looking for existing Python bindings—like in this example, where a handful lines of code are sufficient to talk to MatLab!

CI-capable test runner included

Are you tired of writing custom test runner tooling? With cocotb, tests are automatically discovered and run. No more need for custom runners. The cocotb test runner by default produces regression results in the industry-standard Junit XML format, which is understood by most CI solutions, such as Jenkins, or Azure Pipelines.

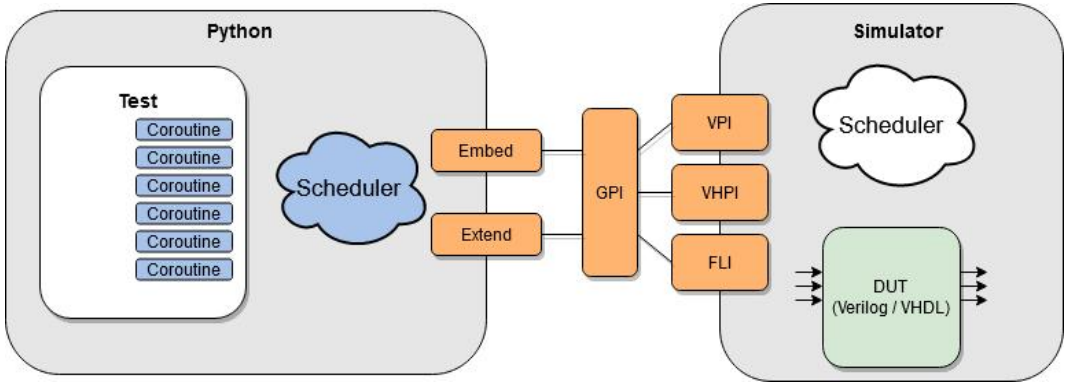
Python is easy to learn (chances are you know it already)

Python is the most popular programming language on the planet, giving you a massive head start. It is used by more than 8 million developers around the world. With cocotb, you can grow as you go. You only need to learn a handful of conventions and you are ready to go. There is no mandatory methodology or class structure to get started! cocotb's extensive documentation and friendly user community are ready to help.

For more details, you may refer to our previous talk **"Cocotb: a Swiss Army Knife for hardware verification"** at PyCon China 2022 and the upcoming follow-ups.

I. Background

Workflow:




```python
# test_my_design.py (simple)

import cocotb
from cocotb.triggers import Timer


@cocotb.test()
async def my_first_test(dut):
    """Try accessing the design."""

    for cycle in range(10):
        dut.clk.value = 0
        await Timer(1, units="ns")
        dut.clk.value = 1
        await Timer(1, units="ns")

    dut._log.info("my_signal_1 is %s", dut.my_signal_1.value)
    assert dut.my_signal_2.value[0] == 0, "my_signal_2[0] is not 0!"
```

...

Cocotb contains a library called `GPI` (in directory `cocotb/share/lib/gpi/`) written in C++ that is an abstraction layer for the VPI, VHPI, and FLI simulator interfaces.

...

The interaction between Python and GPI is via a Python extension module called `simulator` (in directory `cocotb/share/lib/simulator/`) which provides routines for traversing the hierarchy, getting/setting an object's value, registering callbacks etc.

...

Source: https://docs.cocotb.org/en/stable/

I. Background

Extension:

https://docs.cocotb.org/en/stable/extensions.html

Cocotb gives its users a framework to build Python testbenches for hardware designs. But sometimes the functionality provided by cocotb is too low-level. One common example are bus drivers and monitors: instead of creating a bus adapter from scratch for each new project, wouldn't it be nice to share this component, and build on top it? In the verification world, such extensions are often called "verification IP" (VIP).

In cocotb, such functionality can be packaged and distributed as extensions. Technically, cocotb extensions are normal Python packages, and all standard Python packaging and distribution techniques can be used. Additionally, the cocotb community has agreed on a set of conventions to make extensions easier to use and to discover.

https://github.com/cocotb/cocotb/wiki/Further-Resources

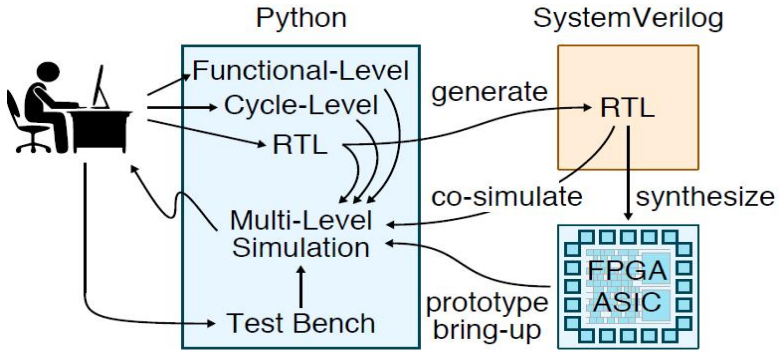
- cocotbext-eth: Ethernet (GMII, RGMII, XGMII, PTP clock)
- cocotbext-pcie: PCI Express (PCIe), and hard IP core models for UltraScale and UltraScale+
- cocotbext-axi: AXI, AXI lite, and AXI stream
- cocotbext-i2c: I2C interface modules
- cocotbext-uart: UART interface modules
- cocotbext-wishbone: Drive and monitor Wishbone bus
- cocotbext-uart: UART testing
- cocotbext-spi: Drive SPI bus
- cocomod-fifointerface: FIFO testing
- cocotbext-interfaces: "generalization of digital interfaces and their associated behavioral models"; Avalon ST
- cocotbext-ral: A port of the uvm-python RAL to use BusDrivers
- cocotbext-apb: AMPBA APB (Transaction, Master, Slave, Monitor)
- cocotb-ahb AHB bus functional model
- cocotb-tilelink TileLink UL bus functional model

I. Background

1.10 PyH2

- https://github.com/pymtl/pymtl3/

PyH2 creatively Adopts PBT(Property-based Testing) for software to test hardware. It combines PyMTL (a unified Python framework for open-source hardware modeling, generation, simulation and verification) with Hypothesis(a PBT framework for Python software and creates a property-based testing framework for hardware, https://github.com/HypothesisWorks/hypothesis).



	Complete Random Testing	Iterative Deepened Testing	PyH2
Small number of test cases to find bug	✓	X	✓
Small number transactions in bug trace	X	✓	✓
Simple transactions in bug trace	X	✓	✓
Simple design instance for bug trace	X	✓	✓

Source: "A New Era of Open-Source Hardware", Christopher Batten, Cornell University.

I. Background

1.11 Chisel

- https://www.chisel-lang.org/
- https://github.com/chipsalliance/chisel3

FIRRTL

- https://github.com/chipsalliance/firrtl-spec

  Flexible Internal Representation for RTL.
- https://github.com/chipsalliance/firrtl

  This project is in maintenance mode

  Pull Requests should only be made for bug fixes against versions 1.6 and below (Chisel 3.6 and below).

  Please see CIRCT for the next generation FIRRTL compiler. Also see Chisel.

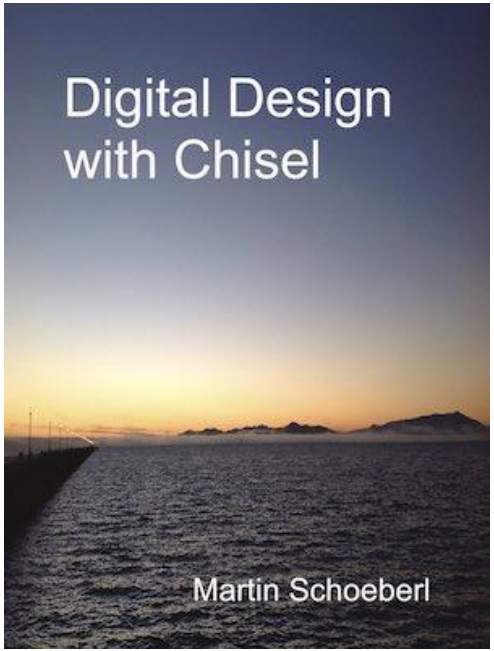
Treadle

- https://github.com/chipsalliance/treadle

  Chisel/Firrtl Execution Engine.

Rocket Chip Generator

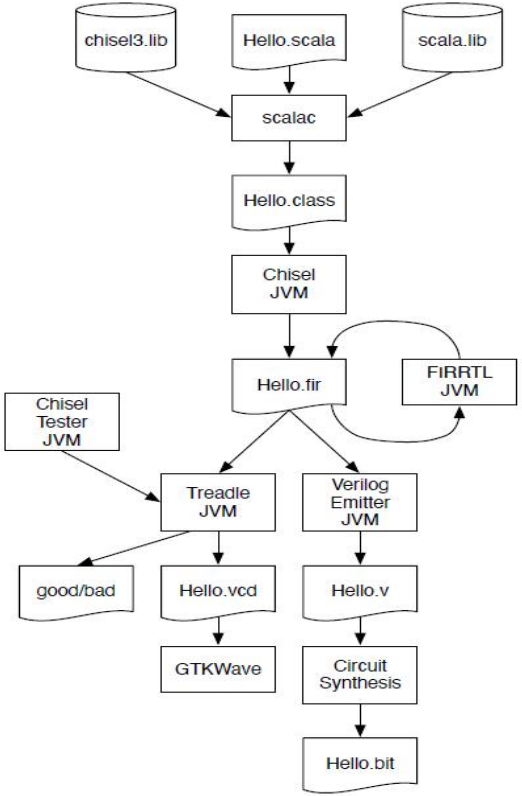
- https://github.com/chipsalliance/rocket-chip



Source: http://www.imm.dtu.dk/~masca/chisel-book.html

I. Background

Tool flow of the Chisel ecosystem:



Source: http://www.imm.dtu.dk/~masca/chisel-book.pdf

I. Background

1.12 SpinalHDL

- **https://github.com/SpinalHDL/SpinalHDL**
  - A language to describe digital hardware
  - Compatible with EDA tools, as it generates VHDL/Verilog files
  - Much more powerful than VHDL, Verilog, and SystemVerilog in its syntax and features
  - Much less verbose than VHDL, Verilog, and SystemVerilog
  - Not an HLS, nor based on the event-driven paradigm
  - Only generates what you asked it in a one-to-one way (no black-magic, no black box)
  - Not introducing area/performance overheads in your design (versus a hand-written VHDL/Verilog design)
  - Based on the RTL description paradigm, but can go much further
  - Allowing you to use Object-Oriented Programming and Functional Programming to elaborate your hardware and verify it
  - Free and can be used in the industry without any license
- **https://spinalhdl.github.io/SpinalDoc-RTD/**
- **https://github.com/SpinalHDL/VexRiscv**

...

I. Background

Cocotb in SpinalHDL

- ```
  [submodule "tester/src/test/python/cocotblib"]
          path = tester/src/test/python/cocotblib
          url = https://github.com/SpinalHDL/CocotbLib.git
  ```

```
[mydev@fedora SpinalHDL-dev]$ tree tester/src/test/python/cocotblib
tester/src/test/python/cocotblib
├── AhbLite3.py
├── Apb3.py
├── Axi4.py
├── ClockDomain.py
├── Flow.py
├── __init__.py
├── LICENSE
├── misc.py
├── Phase.py
├── Scorboard.py
├── Spi.py
├── Stream.py
└── TriState.py
```
```

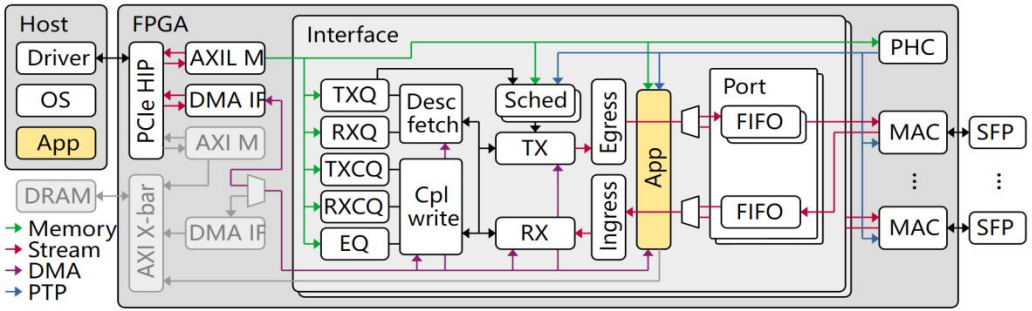
I. Background

1.13 Corundum

- https://github.com/corundum/corundum/

  Open source FPGA-based NIC and platform for in-network compute.

- **Block Diagram**



Block diagram of the Corundum NIC. PCIe HIP: PCIe hard IP core; AXIL M: AXI lite master; DMA IF: DMA interface; AXI M: AXI master; PHC: PTP hardware clock; TXQ: transmit queue manager; TXCQ: transmit completion queue manager; RXQ: receive queue manager; RXCQ: receive completion queue manager; EQ: event queue manager; MAC + PHY: Ethernet media access controller (MAC) and physical interface layer (PHY).

Source: https://github.com/corundum/corundum/

Testing

- Running the included testbenches requires cocotb, cocotbext-axi, cocotbext-eth, cocotbext-pcie, scapy, and Icarus Verilog. The testbenches can be run with pytest directly (requires cocotb-test), pytest via tox, or via cocotb makefiles.

...

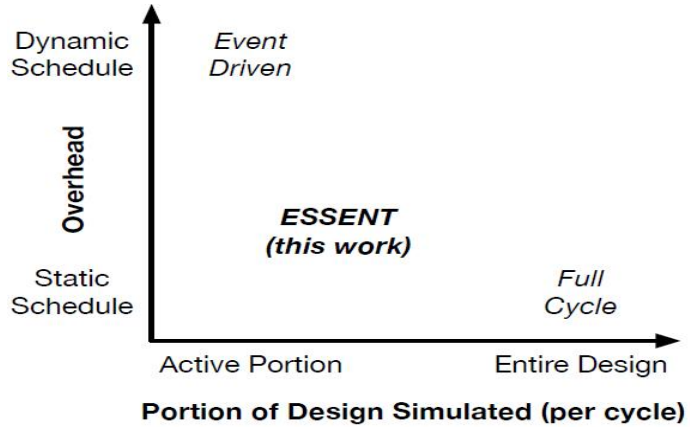
I. Background

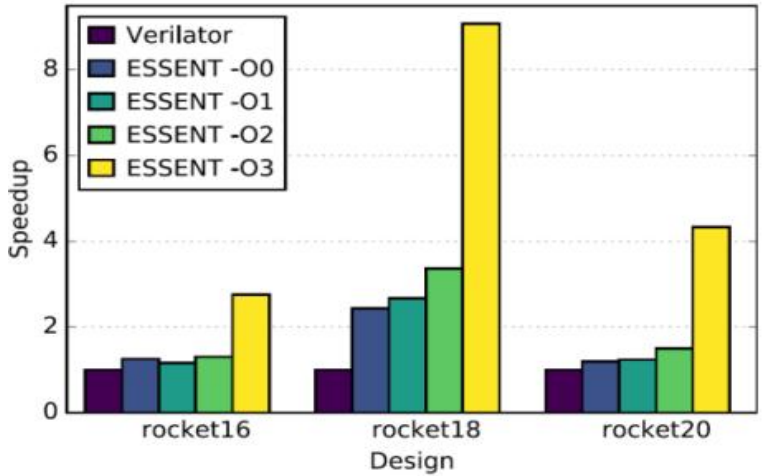
1.14 ESSENT/RepCut

- https://github.com/ucsc-vama/essent/

  Essential **S**ignal **S**imulation **E**nabled by **N**etlist **T**ransformations

  A high-performance RTL simulator generator which operates on hardware designs in the form of **FIRRTL**. The secret of ESSENT is to represent hardware designs as directed graphs so that the classic simulation approaches could be simplified. Even compared with **Verilator**, ESSENT can still have performance advantages at various optimization levels.





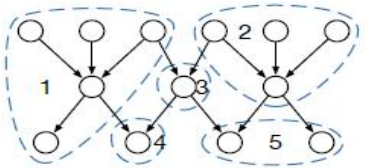
Source: "ESSENT: A High-Performance RTL Simulator", Scott Beamer et al, WOSET 2021.

I. Background

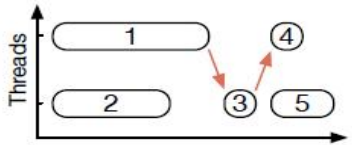
Repcut:

https://github.com/ucsc-vama/essent/tree/repcut

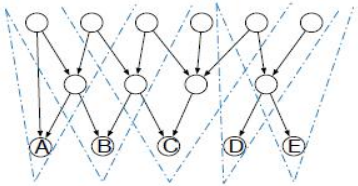
A parallel version of ESSENT that enabled by replication-aided partitioning approach (cuts the circuit into balanced partitions with small overlaps).



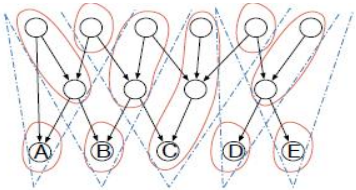
(a) Verilator Partitioning



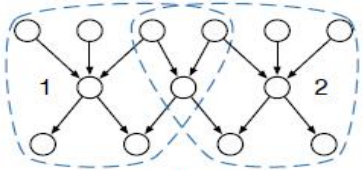
(b) Verilator Schedule



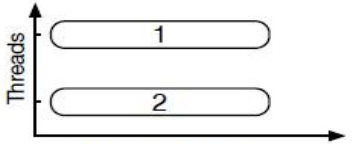
(a) Traverse Cones



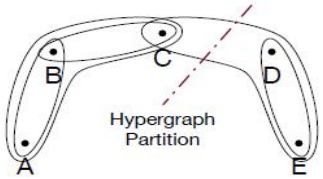
(b) Divide Clusters



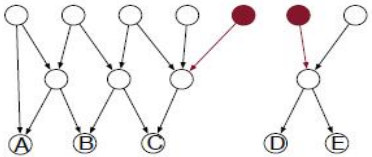
(c) RepCut Partitioning



(d) RepCut Schedule



(c) Build & Partition Hypergraph



(d) Acquire Partitions

Source: "RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning", Haoyuan Wang and Scott Beamer, ASPLOS 2023.

I. Background

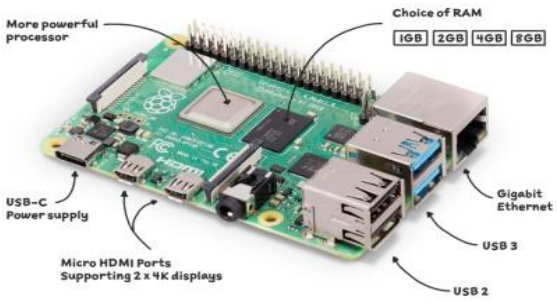
2) Testbeds

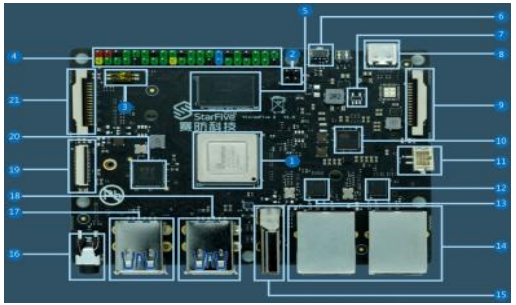
- **HW/SW**

  Testbed1: Intel NUC X15 LAPAC71H(32GB DDR5) with Fedora 38(Linux Kernel 6.3.11/6.4.15);

  Testbed2: Raspberry Pi 4 (8GB LPDDR4) with Fedora 37(Linux Kernel 6.3.8/6.4.12);

  Testbed3: VisionFive 2(8GB LPDDR4) with Debian 12(Linux Kernel 5.15).





II. Enhancing Cocotb

1) Comparison

From official guide:

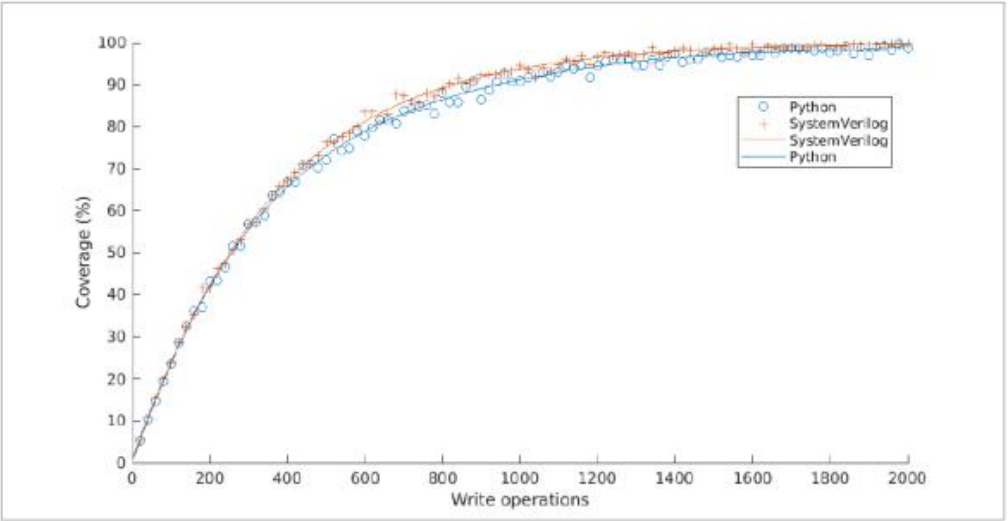
All verification is done using Python which has various advantages over using SystemVerilog or VHDL for verification:

- Writing Python is **fast** - it's a very productive language.
- It's **easy** to interface to other languages from Python.
- Python has a huge library of existing code to **re-use**.
- Python is **interpreted** - tests can be edited and re-run without having to recompile the design.
- Python is **popular** - far more engineers know Python than SystemVerilog or VHDL.

II. Enhancing Cocotb

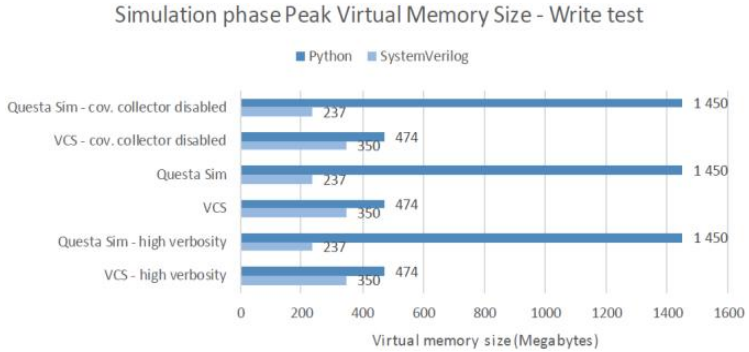
From the theses and papers:

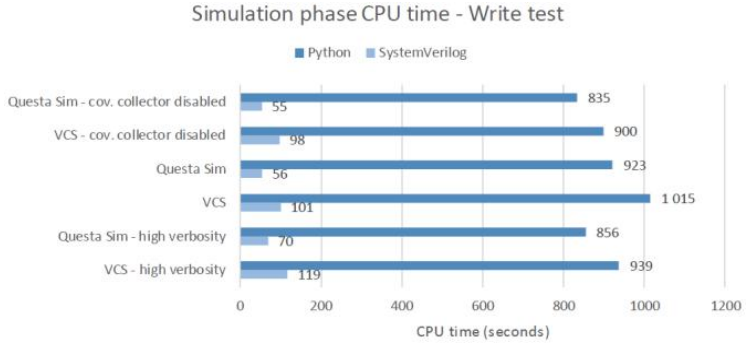
Testbench	Lines of code	Comment lines	Blank lines	Total file size (kB)
Python	1634	237	223	82
SystemVerilog	2325	134	428	103

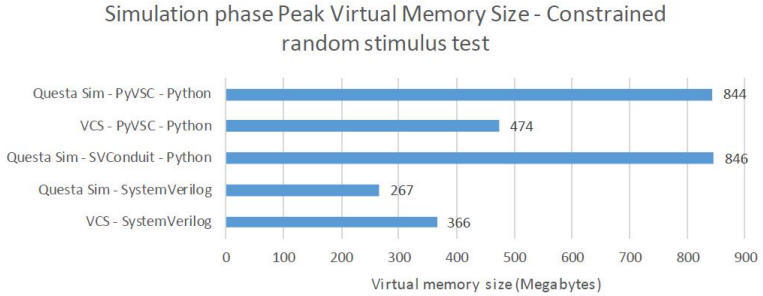


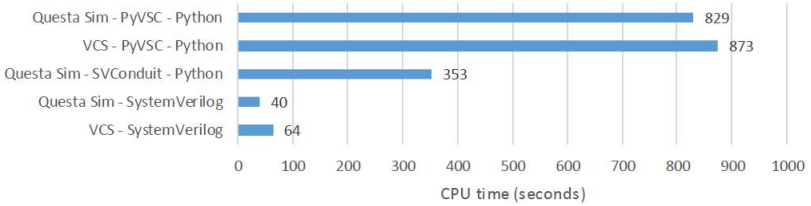
Source: "UVM TESTBENCH IN PYTHON: FEATURE AND PERFORMANCE COMPARISON WITH SYSTEMVERILOG IMPLEMENTATION", Miikka Sinervä, master's thesis 2023.

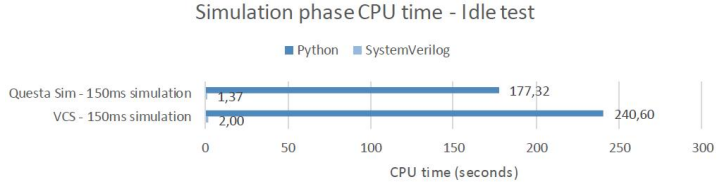
II. Enhancing Cocotb

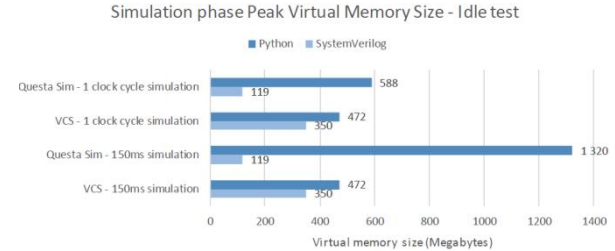












Source: "UVM TESTBENCH IN PYTHON: FEATURE AND PERFORMANCE COMPARISON WITH SYSTEMVERILOG IMPLEMENTATION", Miikka Sinervä, master's thesis 2023.

II. Enhancing Cocotb

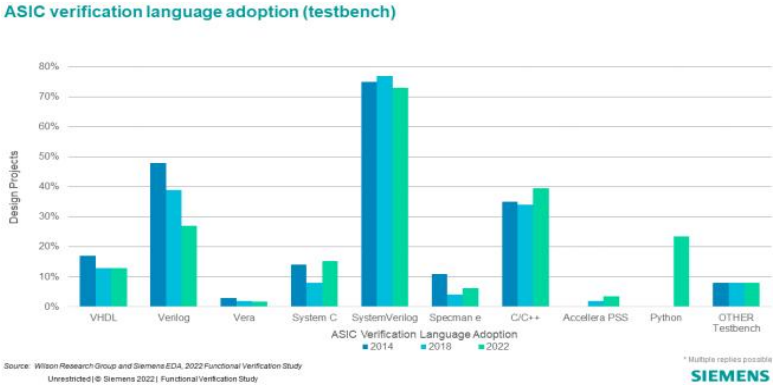
From AI's perspective:

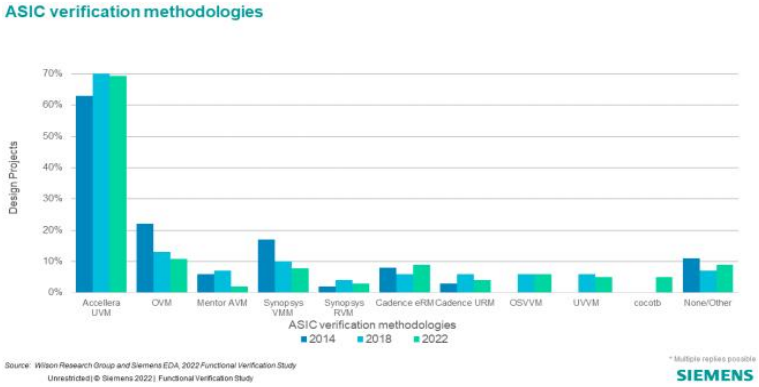
Paper	Aims	Main Advantages	Drawbacks	Algorithm Used
[12]	Functional coverage fulfillment; near-miss tracking; discovering errors by focusing to reach certain functional states of the DUT	Focusing on several aspects of FV	Few details on RL-based algorithms' configuration	Q-learning
[17]	State transition coverage fulfillment	Use of ML algorithms to increase the performance of an RL approach	Only a type of coverage in focus (state transition coverage) and only one type of device in focus (memories)	K-NN, Bayesian optimization
[18]	Functional coverage	The library created allows the use of many RL algorithms without the need to know many of their implementation details	Need to use cocotb testbenches instead of widely used ones (e.g., RTL-based testbenches)	Soft actor critic
[21]	Reaching a target state of a DUT	The degree of automation is increased by analyzing the results provided by RL algorithms using deep neural networks	The proposed approach requires many computational resources	Q-learning
[22]	Functional coverage fulfillment	Using recurrent neural networks for learning DUT behavior	The algorithms used are very complex, and the time required for verification engineers to be comfortable with them could be significantly large	Rainbow agent, based on Q-learning
Current work	Reaching a target state of a DUT	Detailed presentation of configuration possibilities for RL-based systems; development of efficient inference mechanisms	Not using ML algorithms; not working with Q-learning, which is the widely used option due to its proven performance	Semigradient temporal-difference (SGTD)

Source: "Reinforcement Learning Made Affordable for Hardware Verification Engineers", Alexandru Dinu and Petre Lucian Ogrutan, Micromachines 2012.

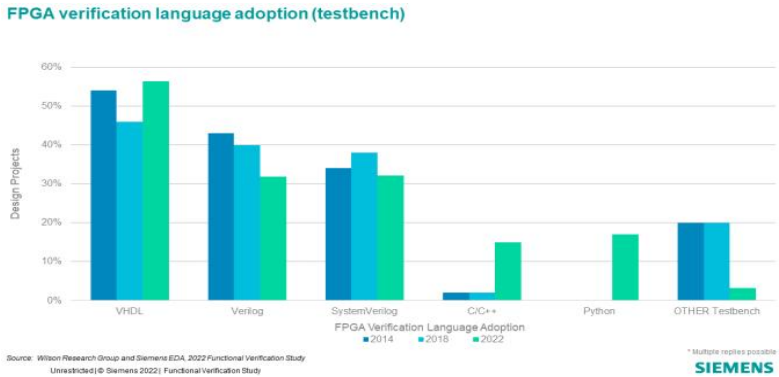
II. Enhancing Cocotb

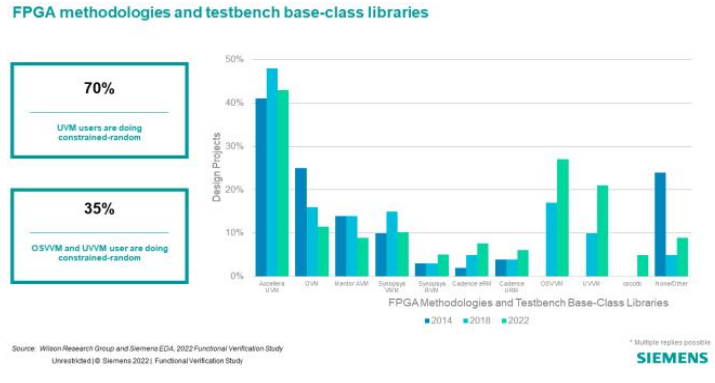
From the trend:





Source: https://blogs.sw.siemens.com/verificationhorizons/2022/12/26/strongpart-10-the-2022-wilson-research-group-functional-verification-study-strong/





Source: https://blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/

II. Enhancing Cocotb

2) Potential Enhancements

2.1 New Python Runtimes

2.1.1 SpinalHDL

Testbed1

- **Comparison**

Time-consuming	CPython 3.11.4 + OpenJDK 17.0.7 (s)	GraalPy 3.10.8 + GraalVM CE(v23.1.0 dev release 20230609 for Java 20) (s)
Compile	45	49
Test	325	407

With Linux Kernel 6.3.11: time-consuming for Compile and Test of SpinalHDL (on dev branch with last commit c5553fcd7cdf05edd6d35fb2424115c3654528b7) with CPython+OpenJDK and GraalPy+GraalVM.

II. Enhancing Cocotb

Time-consuming	CPython 3.11.5 + OpenJDK 17.0.8 (s)	GraalPy 3.10.8 + GraalVM CE(v23.1.0 dev release 20230817 for Java 21) (s)
Compile	2 (?)	48
Test	227	383

With Linux Kernel 6.4.15: time-consuming for Compile and Test of SpinalHDL (on dev branch with last commit 2fdc3aadfc859bbfcf0fb559658606f3dbe4f705) with Cpython+OpenJDK and GraalPy+GraalVM.

```
copying runtime jar...
[info] welcome to sbt 1.6.0 (GraalVM Community Java 21)
error:
  bad constant pool index: 0 at pos: 48445
     while compiling: <no file>
        during phase: globalPhase=<no phase>, enteringPhase=<some phase>
     library version: version 2.12.15
    compiler version: version 2.12.15
  reconstructed args: -classpath /home/mydev11/.sbt/boot/scala-2.12.15/lib/scala-library.jar -Yrangepos

  last tree to typer: EmptyTree
      tree position: <unknown>
           tree tpe: <notype>
              symbol: null
          call site: <none> in <none>

== Source file context for tree position ==
```

Failed to run due to an issue of SBT, a workaround as below:

```
[mydev11@koonuc15x-1 SpinalHDL-dev]$ git diff
diff --git a/project/build.properties b/project/build.properties
index 1e70b0c1c..304098715 100644
--- a/project/build.properties
+++ b/project/build.properties
@@ -1 +1 @@
-sbt.version=1.6.0
+sbt.version=1.9.4
```

Gradually move to **Mill** or **Bloop**.

II. Enhancing Cocotb

Testbed2

- **Comparison**

Time-consuming	CPython 3.11.5 + OpenJDK 17.0.8 (s)	GraalPy 3.10.8 + GraalVM CE(v23.1.0 dev release 20230817 for Java 21) (s)
Compile	649	Similar issues as Testbed1 and CPython+OpenJDK
Test	14517	Similar issues as Testbed1 and CPython+OpenJDK

With Linux Kernel 6.4.12: time-consuming for Compile and Test of SpinalHDL (on dev branch with last commit 2fdc3aadfc859bbfcf0fb559658606f3dbe4f705) with CPython+OpenJDK and GraalPy+GraalVM.

Coredump

A workaround:
disable parallel testing

```
[mydev@fedora SpinalHDL]$ cd SpinalHDL-dev/
[mydev@fedora SpinalHDL-dev]$ du -sh
2.6G    .
[mydev@fedora SpinalHDL-dev]$ git diff
diff --git a/build.sbt b/build.sbt
index c56d64953..06aaadb64 100644
--- a/build.sbt
+++ b/build.sbt
@@ -16,13 +16,13 @@ val defaultSettings = Defaults.coreDefaultSettings ++ xerial.sbt.Sonatype.sonaty
   scalafmtPrintDiff := true,

   //Enable parallel tests
-  Test / testForkedParallel := true,
-  Test / testGrouping := (Test / testGrouping).value.flatMap { group =>
+  Test / testForkedParallel := false,
+//  Test / testGrouping := (Test / testGrouping).value.flatMap { group =>
 //    for(i <- 0 until 4) yield {
 //      Group("g" + i,  group.tests.zipWithIndex.filter(_._2 % 4 == i).map(_._1), SubProcess(ForkOptions()))
 //    }
-    Seq(Group("g",  group.tests, SubProcess(ForkOptions())))
-  },
+//    Seq(Group("g",  group.tests, SubProcess(ForkOptions())))
+//  },
 //  concurrentRestrictions := Seq(Tags.limit(Tags.ForkedTestGroup, 4)),

   libraryDependencies += "org.scala-lang" % "scala-library" % scalaVersion.value,
[mydev@fedora SpinalHDL-dev]$
```

II. Enhancing Cocotb

2.1.2 Corundum

- **Comparison**

Time-consuming	CPython 3.11.4 + OpenJDK 17.0.7 (s)	GraalPy 3.10.8 + GraalVM CE(v23.1.0 dev release 20230609 for Java 20) (s)
Test (run with tox, venv)	4012.37 (setup[33.03]+cmd[3979.34] seconds)	3916.22 (setup[15.08]+cmd[3901.11] seconds)

On Testbed1 with Linux Kernel 6.3.11: time-consuming of testing **Corundum** (on master branch with last commit 56c89640e0deb8083a4b899b2e7c344c52f89053) with CPython+OpenJDK and GraalPy+GraalVM.

Time-consuming	CPython 3.11.5 + OpenJDK 17.0.8 (s)	GraalPy 3.10.8 + GraalVM CE(v23.1.0 dev release 20230817 for Java 21) (s)
Test (run with tox, venv)	3914.69 (setup[15.03]+cmd[3899.66] seconds)	Infeasible
Test (run without tox)	4125.86 4170.97(enable verbose)	N/A

On Testbed1 with Linux Kernel 6.4.15: time-consuming of testing **Corundum** (on master branch with last commit ed4a26e2cbc0a429c45d5cd5ddf1177f86838914) with CPython+OpenJDK and GraalPy+GraalVM.

II. Enhancing Cocotb

Issues

- run without tox

  pytest -n auto [--verbose]

but

```
[mydev11@koonuc15x-1 /]$ cat ~/.local/bin/pytest
#!/usr/bin/python3
# -*- coding: utf-8 -*-
import re
import sys
from pytest import console_main
if __name__ == '__main__':
    sys.argv[0] = re.sub(r'(-script\.pyw|\.exe)?$', '', sys.argv[0])
    sys.exit(console_main())
[mydev11@koonuc15x-1 /]$
```

hack the shebang of pytest script with that of GraalPy and RustPython:

Various issues will occur to block the test, especially for RustPython, we are still trying to resolve them…

II. Enhancing Cocotb

2.2 Alternatives to GHDL

2.2.1 Try to replace GHDL with NVC in the future

1)

```
[mydev11@koonuc15x-1 nvc-master]$ tree -L 2 -d lib
lib
├── ieee
├── ieee.08
├── ieee.19
├── nvc
├── nvc.08
├── nvc.19
├── std
├── std.08
├── std.19
├── synopsys
└── vital
```

```
[mydev11@koonuc15x-1 nvc-master]$ tree -L 3 -d src
src
├── jit
├── psl
├── rt
├── vhpi
└── vlog
```

2) NVC is still not as mature as GHDL…

3) Options for GHDL and NVC:

https://gritbub-ghdl.readthedocs.io/en/latest/using/InvokingGHDL.html#options

https://www.nickg.me.uk/nvc/manual.html

II. Enhancing Cocotb

2.2.2 Re-implement GHDL by SPARK

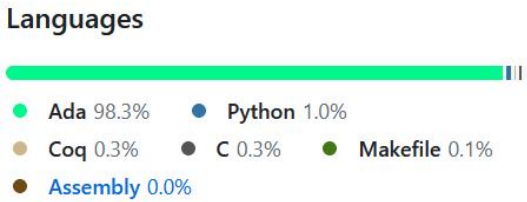
What is SPARK:

SPARK is a high-level computer programming language consisting of a well-defined subset of Ada. Like Ada before it, SPARK was designed for the development of high-integrity software used in systems where predictable and highly reliable operation is essential. SPARK uses a language feature known as "contracts" to specify components in a form that is suitable for static verification using formal methods.

"From the perspective of programming language capabilities, the paradigms are very similar to C and C++," said Dhawal Kumar, a principal software engineer at NVIDIA and one of their first SPARK users. "SPARK is an imperative programming language. You can write procedure-oriented or object-oriented code, and there are other facilities for programming in the large."

Source: https://www.adacore.com/uploads/techPapers/222559-adacore-nvidia-case-study-v5.pdf

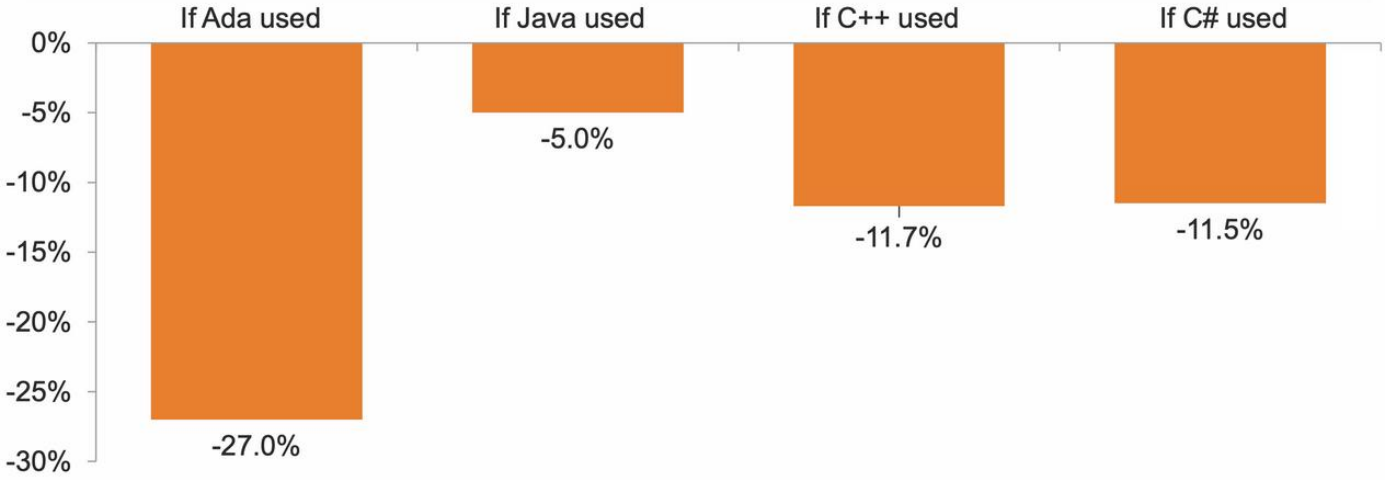
https://github.com/AdaCore/spark2014



II. Enhancing Cocotb

Reduces development costs:

The demand for cost-effective tools and methodologies has greatly increased in the automotive and industrial domains over the past few years. Using Ada and SPARK reduces the cost of developing safety and security-critical software by automating many verifications that would otherwise need to be done through manual code reviews or testing. It also allows the detection of potential issues early in the development process, reducing the amount of errors needing to be fixed in later stages. For industries that have strong safety, reliability, and security standards, like aerospace and automotive, these benefits can translate to nearly 40 percent cost and time savings from enhanced software verification, according to a study by VDC Research.



Source: https://www.adacore.com/nvidia

II. Enhancing Cocotb

2.3 Evaluate the support of RepCut

1) ESSENT/RepCut are written in Scala and using SBT for build. So it can also run on GraalVM like Chisel and SpinalHDL. A typical flow using the tool will: use ESSENT to generate head file from the Firrtl input, then write a C++ harness for the emitted code, compile everything, and finally run the simulation;

2) ESSENT/RepCut still need developer to manually write C++ harness to generate the simulator, so the developers need to know both Scala and C++. Shall we use Python-based DSL to re-implement it for better developer experience and more easily integrated with Cocotb? Let's further discuss them in another paper "Python-based emerging DSLs for FOSS EDA" of mine @ DVCon China 2023;

3) During practicing ESSENT/RepCut on Testbeds, we found that it lacks of detailed user guide and there are some mistakes in old tests that need to be updated.

II. Enhancing Cocotb

2.4 AI-assisted Cocotb

2.4.1 VeRLPy

- **https://github.com/aebeljs/VeRLPy**

VeRLPy is an open-source python library developed to improve the digital hardware verification process by using Reinforcement Learning (RL). It provides a generic Gym environment implementation for building cocotb-based testbenches for verifying any hardware design.

VeRLPy is currently dependent on OpenAI Gym, cocotb, cocotb-bus, and Stable Baselines3. These packages should get installed alongside VeRLPy when installing using `pip`. For running the verification, a simulator compatible with cocotb is additionally required. Please refer to the official cocotb documentation to set this up.

https://github.com/openai/gym → **https://github.com/Farama-Foundation/Gymnasium**

II. Enhancing Cocotb

Inheriting CocotbEnv

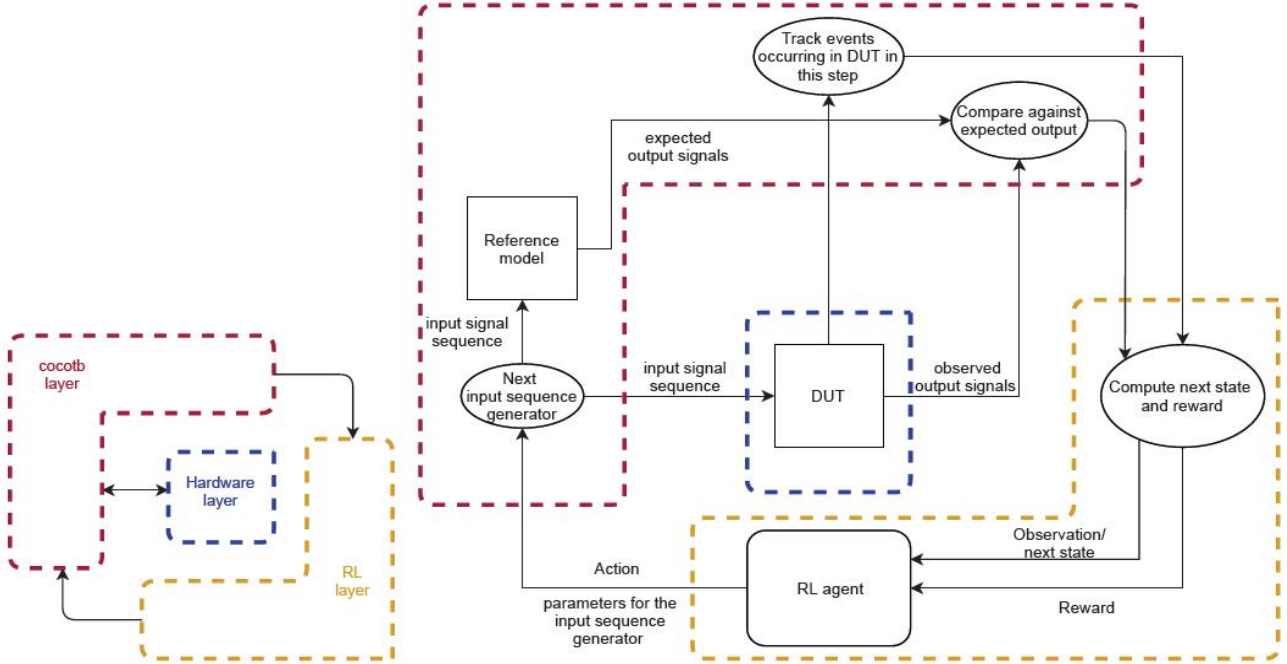
```python
# test_my_example_design.py

import cocotb
from verlpy import CocotbEnv

class MyExampleDesignCocotbEnv(CocotbEnv):
    def __init__(self, dut, observation_space):
        super().__init__()
        self.dut = dut # DUT object used for cocotb-based verification
        self.observation_space = observation_space # state space of the RL agent

        # add here any "self." variables that need to be accessed in
        # other functions below

    @cocotb.coroutine
    def setup_rl_episode(self):
        # add here the logic to be
        # executed on each call to reset() by the RL agent
```

```python
@cocotb.coroutine
def setup_rl_episode(self):
    # add here the logic to be
    # executed on each call to reset() by the RL agent

@cocotb.coroutine
def rl_step(self):
    # add here the verification logic to be
    # executed on each call to step() by the RL agent


@cocotb.coroutine
def terminate_rl_episode(self):
    # add here the logic to be executed at the end
    # of each RL episode when done == 1 for the Gym env


def finish_experiment(self):
    # add here the logic to be executed after all
    # the episodes are completed
```

Source: https://github.com/aebeljs/VeRLPy

II. Enhancing Cocotb

Architecture & Design



(a) A top level view of the framework (b) Block diagram of the complete framework with the component layers demarcated using magenta (cocotb layer), blue (Hardware layer) and orange (RL layer)

Source: "VeRLPy: Python Library for Verification of Digital Designs with Reinforcement Learning", Aebel Joe Shibu et al, 2021.

II. Enhancing Cocotb

Comparison of coverage and actions chosen during the 1K iterations

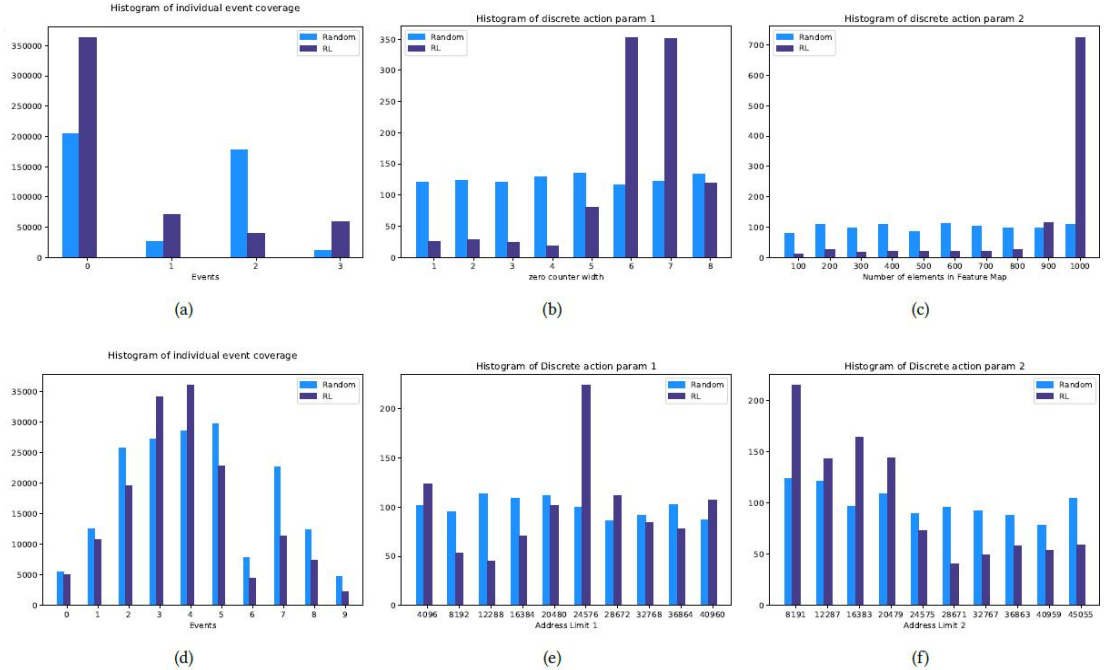


Fig. 3. Comparison of coverage and actions chosen during the 1000 iterations. RLE Compressor - Figure 3(a) shows the event coverage that is tracked when event e_3 is rewarded. Figure 3(b) shows the histogram of `count_width` values suggested by the RL agent. Figure 3(c) shows the histogram of sequence length suggested by the RL agent. AXI Crossbar - Figure 3(d) shows the event coverage that is tracked when event e_4 is rewarded. Figure 3(e) and Figure 3(f) shows the histograms of the choices for the lower and upper limit of the address ranges

Source: "VeRLPy: Python Library for Verification of Digital Designs with Reinforcement Learning", Aebel Joe Shibu et al, 2021.

III. Project Cocotbll

What is it

- Under development (to be open-sourced in 2024)

  Next generation Cocotb from us by re-design and re-implement it in various ways, and come with the enhancements that mentioned in section II.

III. Project Cocotbll

1) Cocotb with emerging Python-based DSLs

- The key ideas are in another paper **"Python-based emerging DSLs for FOSS EDA"** of mine @ DVCon China 2023.

III. Project Cocotbll

2) Cocotb with Dlang

- Rewrite the native code in Cocotb(mainly C++) with  , and more…

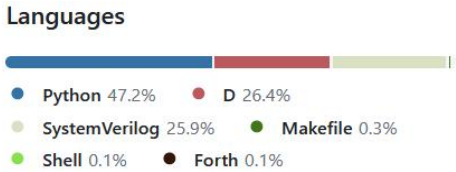
III. Project Cocotbll

2.1 RISCV-DV

- **https://github.com/chipsalliance/riscv-dv**

RISCV-DV is a SV/UVM based open-source instruction generator for RISC-V processor verification. It currently supports the following features:

- Supported instruction set: RV32IMAFDC, RV64IMAFDC
- Supported privileged mode: machine mode, supervisor mode, user mode
- Page table randomization and exception
- Privileged CSR setup randomization
- Privileged CSR test suite
- Trap/interrupt handling
- Test suite to stress test MMU
- Sub-program generation and random program calls
- Illegal instruction and HINT instruction generation
- Random forward/backward branch instructions
- Supports mixing directed instructions with random instruction stream
- Debug mode support, with fully randomized debug ROM
- Instruction generation coverage model
- Handshake communication with testbench
- Support handcoded assembly test
- Co-simulation with multiple ISS : spike, riscv-ovpsim, whisper, sail-riscv



III. Project Cocotbll

eUVM

- https://github.com/chipsalliance/riscv-dv/blob/master/euvm/README.md

About the RISCV-DV eUVM port

The RISCV-DV eUVM port is a line-by-line translation of the RISCV-DV SystemVerilog implementation. Except for functional coverage (a work in progress), all other RISCV-DV features have been implemented in eUVM port.

- https://github.com/coverify/euvm/releases

Jun 12

puneet

v1.0-beta27

f62139c

Compare

EUVM v1.0-beta27 Release Latest

New in this Release:

- Support for Verilator 5.0
- Support for Concurrency in Fork

Installation:

```
tar xf euvm-1.0-beta27.tar.xz
cd euvm-1.0-beta27
./utils/setup.sh
```

Assets 4

euvm-1.0-beta27-centos7.tar.xz	117 MB	Jun 12
euvm-1.0-beta27.tar.xz	117 MB	Jun 12
Source code (zip)		Nov 30, 2022
Source code (tar.gz)		Nov 30, 2022

For more details of **ESDL** and **eUVM**, you may refer to our report

"D-based Next Generation Verification Language" and its update.

III. Project Cocotbll

3) Cocotb with AI

- Extend Cocotb to support extensions for **AI** and **Distributed Computing**, and a customized project **Ray** (you may find it in another paper **"Python-based emerging DSLs for FOSS EDA"** of mine @ DVCon China 2023.
- …

III. Project Cocotbll

4) Chisel Verification with Cocotb?

- A long-term goal…

IV. Wrap-up

- More **software** development methods together with **AI** are being introduced into hardware design verification and deeply **integrated**.
- Our effort to try to enhance **Python-based** verification framework such like **Cocotb** in various ways shows that it still has great potential for further improvement and may be brought to more FOSS EDA projects in the near future.
- It is not surprisingly that **Python-centric** one-stop toolchain will gradually becoming mainstream in FOSS EDA.
- Is it time to thank **UVM** and say goodbye?

Source: https://olofkindgren.blogspot.com/2022/10/its-time-to-to-thank-uvm-and-say-goodbye.html

鲜卑拓跋枫

扫一扫上面的二维码图案，加我微信