LastLayer:Towards Hardware and Software Continuous Integration

Luis Vega, Jared Roesch, Joseph McMahan, Luis Ceze.
Paul G. Allen School of Computer Science & Engineering, University of Washington

Abstract—This paper presents LastLayer, an open-source tool[1] that enables hardware and software continuous integration and simulation. Compared to traditional testing approaches based on the register transfer level (RTL) abstraction, LastLayer provides a mechanism for testing Verilog designs with any programming language that supports the C foreign function interface (CFFI). Furthermore, it supports a generic C interface that allows external programs convenient access to storage resources such as registers and memories in the design as well as control over the hardware simulation. Moreover, LastLayer achieves this software integration without requiring any hardware modification and automatically generates language bindings for these storage resources according to user specification. Using LastLayer, we evaluated two representative integration examples: a hardware adder written in Verilog operating over NumPy arrays, and a ReLu vector-accelerator written in Chisel processing tensors from PyTorch.

***Index terms*—** hardware simulation, hardware language interoperability, agile hardware design

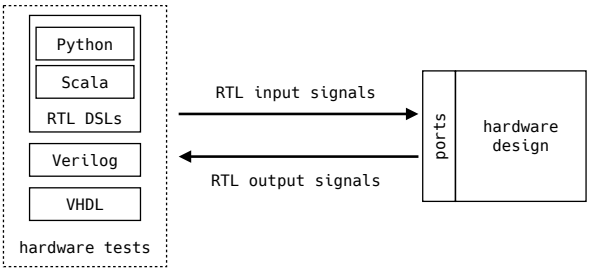
1 Introduction

From RISC-V [9] processors to domain-specific accelerators, both industry and academia have produced an outstanding number of open-source hardware projects. This progress has also fostered the creation of multiple hardware tools such as FPGA frameworks [16] and fast simulators [11]. However, the growing demand for highly productive hardware design presents numerous challenges.

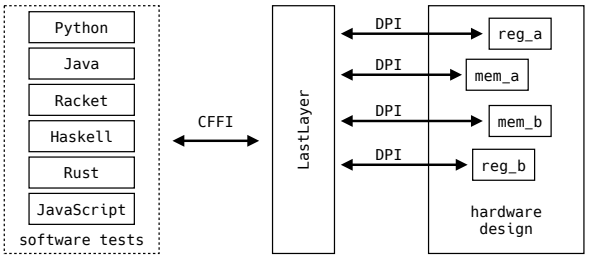
One overlooked challenge in hardware design is the limited language interoperability available during simulation. Most of the efforts in testing and verification are made around the register transfer level (RTL) abstraction, which is fundamental for building reliable hardware. In contrast to software languages, the language used for designing hardware is not used for production. In other words, once a hardware design is compiled down to an FPGA or silicon in the case of ASICs, RTL based languages cannot be used to perform regression tests.

For example, consider the hypothetical scenario of a hardware developer designing a scalar adder in Verilog that performs element-wise summation over two NumPy arrays. NumPy is an open-source and popular library written for Python for scientific computing [8]. Although writing the software and hardware for this system is fairly trivial, the integration of the two present several challenges for developers, including understanding the complex foreign function mechanism in Verilog, writing language bindings, and figuring out how to use a hardware simulator with other programming languages besides Verilog and VHDL.

1. https://github.com/uwsampl/lastlayer



(a) Traditional RTL simulation



(b) LastLayer simulation

Fig. 1: Comparison between traditional and LastLayer simulation approaches. The traditional approach is based on hardware tests driving ports written in RTL languages such as embedded DSLs and hardware languages. Alternatively, LastLayer enables software tests written in any language that supports CFFI to interact with the hardware design by accessing registers and memories, similarly to how hardware is managed once is prototyped.

The motivation for integrating software languages during hardware simulation lies not only in testing purposes but also for evaluating hardware under more realistic traces that would otherwise be difficult to generate with RTL languages. One example of using software integration for better testing traces is based on loading binaries into memory when simulating a processor [5]. Nevertheless, this software integration approach is not portable to other processor designs, because it is tightly coupled to this particular implementation.

To solve these software integration challenges, we develop a tool called LastLayer that takes Verilog designs and transforms them into a C shared library that can be dynamically hosted by any programming language supporting CFFI. Moreover, the interface of this library is generic and independent of the design being

tested, which is convenient for software layers built on top of it. Concretely, the benefits of using LastLayer include:

- Enabling external programming languages to host and manage a hardware design, regardless of its complexity.
- Enhancing the quality of hardware simulation with testing traces produced by real programs.
- Lowering the barrier to entry when simulating hardware with external languages.

2 Hardware Interoperability Challenges

2.1 Traditional hardware simulation

Figure 1a describes the conventional approach used for simulating hardware, which consists of writing tests at the same level of abstraction used for designing a hardware module. A hardware developer would normally create unit tests by generating input signals that drive a hardware module and asserting that its output signals match a specification. These hardware tests are often referred as "testbenches" and are executed with commercial simulators, such as Synopsys VCS [10], Mentor ModelSim [7], among others. Currently, there is increasing adoption of open-source hardware simulators, including Verilator [11] and Icarus Verilog [6].

Independent of the simulator used, the role of the hardware engineer is writing multiple testbenches with signal traces that test the design under different scenarios and corner cases [18]. The accuracy of these test cases is dependant on the hardware models used by the designer and not the simulator itself. For example, a design can be synthesized from a behavioral description into a structural gate-level netlist and still be tested with the same testbench written at the behavioral level. The cost of a more detailed simulation is time, as modeling detailed circuit interactions is more computationally expensive than higher level descriptions. Similarly, complex interactions with external devices such as interconnects e.g., PCIe can be also simulated at this level with vendor-provided RTL models. In fact, some hardware simulators even provide support for testing hardware languages, e.g., Verilog, together with analog circuit models defined in SPICE. This type of simulation is known as mixed-signal simulation.

However, accurate testing is not the only motivation for performing hardware simulation. Certain hardware designs require performance estimation metrics after running testing traces. For example, a hardware engineer may want to assess the number of mispredicted branches or cache misses in a processor implementation while executing a particular program. In this particular case, writing tests in RTL to recreate the inputs produced by a program written in a higher language is an error prone and unproductive task. Instead, developers rely on language bindings such as foreign function interfaces (FFI) available in the hardware language to integrate software programs that interact with the hardware design during simulation. The foreign function interface available in Verilog is called direct programming interface (DPI), and it is based on the C programming language[2]. One example of this hardware and software simulation approach can be found in the Ibex RISC-V core [5] testing infrastructure, which uses DPI to load ELF binaries to memory for simulating programs running in the processor.

2. DPI was included in the SystemVerilog 3.1 standard in April 2003

2.2 Software integration challenges

While handwriting DPI programs to interact with hardware designs seems to solve some software integration challenges, we strongly argue that there are still opportunities for improvement. Specifically, we found three important features missing in current hardware and software simulation approaches.

Automatic generation of DPI programs. The current approach for enabling hardware and software interaction during simulation is to write a fair amount of boilerplate DPI code. For example, take the task of simulating a processor described earlier, which uses DPI to model memory. What happens if a hardware developer wants to test *N* different RISC-V implementations with different memory interfaces?. In order to make this possible today, the developer would have to write *N* different DPI programs, one for each case. This is extremely inefficient and economically expensive to justify.

Flexible simulator interface. Regardless of whether the hardware design is a processor or an accelerator, hardware simulators already internally compile hardware languages into software counterparts like C++ for performing cycle-accurate simulation. Nevertheless, the simulator interface is opaque and hard to use with foreign programming languages. Both commercial and open-source simulators suffer from this limitation, since they were built mainly for pure RTL simulation. A flexible simulator interface should provide control to external languages in a similar fashion to how a device is managed by an operating system. Foreign programming languages should be able to launch, pause, and terminate simulation at any point in time, in addition to reading and writing data to the design during simulation. More importantly, an ideal simulator interface should be generic and independent of any particular design.

Interoperability with other languages. Most of the hardware testing today is written around the RTL abstraction, which is still a necessary and valuable task in the entire hardware design process. However, traditional RTL tests cannot be reused once hardware languages are compiled down into silicon by either prototyping an ASIC or mapping it down to an FPGA. There is no fundamental reason for not allowing programs written in external languages to interact with hardware compiled for cycle-accurate simulation. Any programming language supporting CFFI should be able to interact with hardware, including Racket, Rust, Python, Haskell, Java, and JavaScript, among other languages. This feature will increase productivity, because hardware developers can reuse libraries written in these higher-level languages to test designs with more realistic traces — or even the actual programs to be used in production.

3 LastLayer

We developed LastLayer in Rust. The tool addresses current hardware and software integration challenges by generating DPI functions from user specification, provides a flexible simulation interface in C, and packages the hardware design as a shared library that external languages can dynamically host. Verilog sources are compiled down to C++ with Verilator [11]. Figure 1b describes at a high level what kind of hardware and software simulation LastLayer enables compared to traditional RTL-based testing approaches.

One key difference with traditional approaches is the fact that tests can be written in external languages and leverage existing domain-specific libraries that the hardware design will use after getting manufactured. For example, a hardware accelerator can be tested, using cycle-accurate simulation, with images produced

by a computer vision library written in a higher level language. Compared to traditional testing, LastLayer does not require any modifications in the hardware source code to accommodate the integration with software. The tool uses the hierarchical path of registers and memories to read and write values with automatically generated DPI functions. Another benefit of this type of integration is that tests are not coupled with the interface of the hardware module, allowing reuse of test programs between designs with similar functionality. For instance, an accelerator template that supports different interconnect protocols, such as Intel Avalon [2] or ARM AXI [1], can reuse software tests when simulating each protocol implementation.

```
use lastlayer::Build;

// add_register(id, path, width)
// add_memory(id, path, width)

fn main() {
    Build::new()
        .out_dir("out")
        .top_module("top_name")
        .verilog_include_dir("one_header.vh")
        .verilog_include_dir("another_header.vh")
        .verilog_file("one_file.v")
        .verilog_file("another_file.v")
        .add_register(0, "top_name.reg_0", 32)
        .add_register(1, "top_name.reg_1", 32)
        .add_memory(0, "top_name.mem_0", 128)
        .add_memory(1, "top_name.mem_1", 64)
        .compile("lib_name");
}
```

Listing 1: Building a library from a hardware design in LastLayer

```
/* device handle */
typedef void* LastLayerHandle;

/* allocate device */
LastLayerHandle LastLayerAlloc();

/* deallocate device */
void LastLayerDealloc(LastLayerHandle handle);

/* read from a register */
int LastLayerReadReg(LastLayerHandle handle,
    int hid, int sel);

/* write to a register */
void LastLayerWriteReg(LastLayerHandle handle,
    int hid, int sel, int value);

/* read from a memory */
int LastLayerReadMem(LastLayerHandle handle,
    int hid, int addr, int sel);

/* write to a memory */
void LastLayerWriteMem(LastLayerHandle handle,
    int hid, int addr, int sel, int value);

/* reset for n clock cycles */
void LastLayerReset(LastLayerHandle handle, int n);

/* run simulation for n clock cycles or until
 * $finish system task is invoked in Verilog
 */
void LastLayerRun(LastLayerHandle handle, int n);
```

Listing 2: Generated LastLayer C simulator interface

3.1 Building a device library from a hardware design

The Rust program described in Listing 1 shows how to use LastLayer for a given design. After running this program, LastLayer will produce a shared library located in the `out` directory. This library is described in Listing 2 and is based on the C programming language. This interface is the same for every hardware design processed by LastLayer, regardless of its complexity. Therefore, the software tests built on top are independent of the hardware interface.

Additionally, the LastLayer builder (`Build`) provides methods for adding hardware source and header files, named `verilog_file()` and `verilog_include_dir()` respectively. It also supports other methods for adding the registers and memories that external languages can have access to. These methods are named as `add_register()` and `add_memory()`. Information about these registers and memories must be provided by the user including an unique identifier, the hierarchical path, and the bit width.

Before generating DPI functions from this information, LastLayer checks that every register and memory has unique identifiers (`id`) and adds runtime assertions for checking that these identifiers are valid when external languages invoke these functions.

3.2 Hardware simulation lifetime

The LastLayer C interface allows external languages to host and manage the hardware design and simulation as if it were an external device, regardless of how close to completion the design under test is. One interesting feature of the simulator interface is the fact that clock only advances when `LastLayerReset()` and `LastLayerRun()` are called. Figure 2 describes one possible hardware simulation lifetime for a sample design. The simulation begins by a host program allocating the simulation object, followed by asserting reset for a determined number of clock cycles. Then, the host program writes a register and runs the simulation for another number of clock cycles. Later, the host program reads a register to evaluate a particular expression that is used by other procedures in the host program. The host program is free to execute other programs between function calls. This is particularly useful for simulating co-processing and context switching scenarios, without the requirement of restarting the entire simulation. After the host program finishes running other programs, the hardware simulation is restarted one more time for another number of clock cycles followed by reading another register in the design. Fully interactive tests can step the simulation one cycle at a time to check values after each step. Finally, the simulation is terminated once the host program deallocates the object.

4 Evaluation

4.1 Integrating a hardware adder with NumPy

We implemented a hardware adder in Verilog to characterize the productivity of using LastLayer for integrating and simulating hardware that processes data produced by the NumPy library. The adder is described behaviorally and is connected to three registers, two for storing operands and one for saving the result. The adder operates over unsigned integers of 8-bit, which correspond to the `uint8` data type object (`dtype`) in NumPy. We used the LastLayer builder described in Section 3 for generating a shared library, called `libadder.so`, for this particular hardware design.

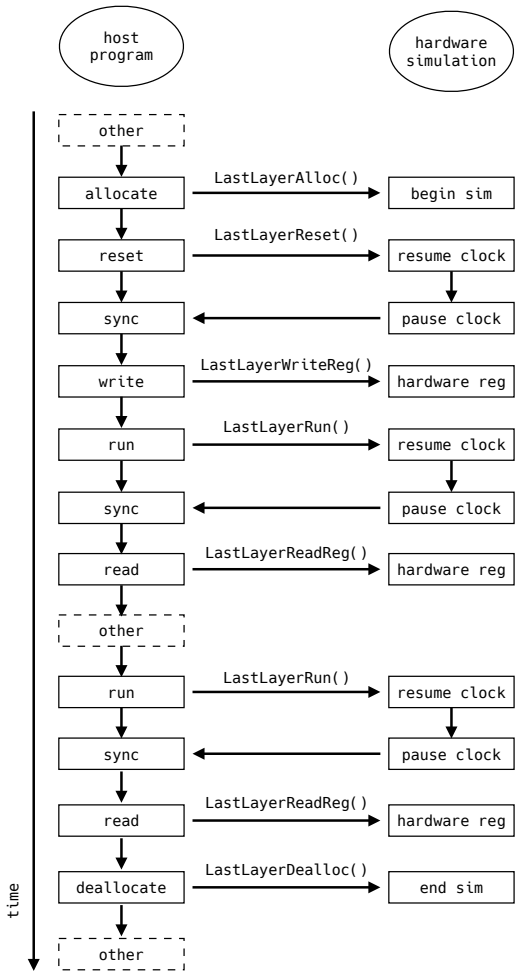


Fig. 2: Example of hardware simulation lifetime supported by LastLayer

```
import numpy as np
from device import Device

dev = Device("libadder.so") # lastlayer lib
dev.reset(10) # reset for 10 cycles

n = 16
maxv = 64

a = np.random.randint(maxv, size=n, dtype="uint8")
b = np.random.randint(maxv, size=n, dtype="uint8")
c = np.zeros(n, dtype="uint8")

# c = a + b
for i, d in enumerate(zip(a, b)):
    dev.write_reg_a(d[0])    # write operand a
    dev.write_reg_b(d[1])    # write operand b
    dev.run(3)               # run for 3 cycles
    c[i] = dev.read_reg_y() # read result c

y = np.add(a, b) # generate expected value
np.testing.assert_array_equal(c, y) # compare
```

Listing 3: Simulating a hardware adder with LastLayer and NumPy

On the Python side, we developed a `Device` class as part of the `device` module described in Listing 3. The class constructor for `Device` dynamically loads the shared library using the `ctypes` library and it wraps the LastLayer C interface. The class provides access to the three registers available in the hardware module.

Although this sample design seems trivial, it is a representative testing pattern found today when designing domain-specific accelerators. For example, a hardware developer can start by designing a scalar accelerator that processes multi-dimensional data arrays generated by NumPy. In the next design iteration, the accelerator can be upgraded with a vector unit and still reuse the same arrays used for testing the scalar version. More architectural optimizations can be added and tested over time until the design is finally completed. The productivity of the hardware developer is increased under this environment, because architectural optimizations are evaluated with production data on every iteration.

4.2 Integrating a hardware ReLu with PyTorch

Another representative integration example to showcase the advantages of using LastLayer is based on a domain-specific accelerator written in Chisel and simulated with Pytorch. Pytorch is a deep learning framework developed in Python.

We developed a linear rectifier (ReLu) vector accelerator using Chisel [12], which is based on a vector unit that performs the mathematical function $f(x) = max(0,x)$. This a common activation function found in neural networks. The vector unit is connected to two data memories, storing the input and output tensor. Furthermore, the accelerator has control registers for specifying input and output memory addresses, the total number of vector operations, and when the accelerator is launched.

Regarding the PyTorch integration, we relied on the TorchScript interface normally used for extending PyTorch with custom C++ operators. This interface only supports the following types: `torch::Tensor`, `torch::Scalar`, `double`, `int64_t`, and `std::vector`. We developed a small C++ driver layer that adapts the TorchScript interface to the LastLayer simulator interface. Moreover, we used the LastLayer's builder to include additional Torch libraries needed by this interface. Finally, LastLayer packages this driver together with the hardware design into single shared library that can be loaded directly in PyTorch. These steps would also be necessary for the real hardware design, and so do not represent extra work involved with using LastLayer.

We evaluated multiple configurations of the ReLu accelerator, i.e., using different vector lengths, while processing PyTorch tensor data. Figure 3 shows the results from this evaluation. We measured the simulation time spent by each configuration operating over a one-dimensional tensor declared in PyTorch. The data type of the tensor is `int8` and the number of elements used for each run was 16,384. We performed these measurements sixteen times for each configuration on a machine with an Intel Core i7 2.6 GHz and 16 GB of DRAM. The takeaway from this evaluation is that the cost of performing cycle-accurate simulation on a hardware design integrated with the PyTorch framework is relatively low, even though simulation optimization like the multi-threaded option in Verilator and TorchScript serialization were not used.

5 Related Work

Embedded hardware languages. There are increasing numbers of modern hardware languages [12], [13], [15] aiming to improve productivity compared to traditional hardware languages such as Verilog and VHDL. These new languages leverage features available in the host language, such as including better type systems, collections, and decorators to improve the hardware design experience. Interestingly, an important factor in this experience is testing, which these languages have heavily invested in. For

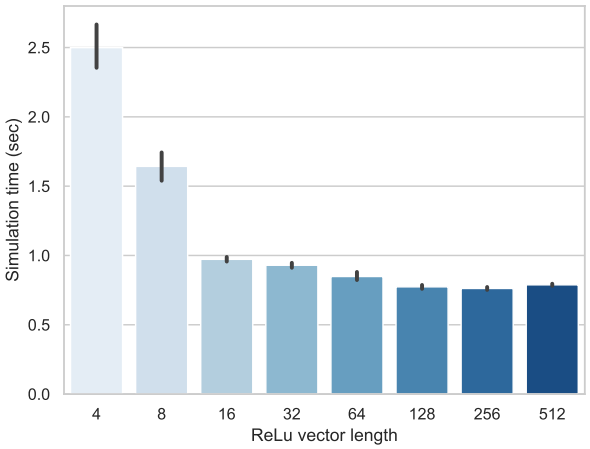


Fig. 3: Simulation time spent by different configurations of the ReLu vector-accelerator implemented in Chisel while processing a tensor of 16,384 elements in PyTorch.

example, PyRTL [13] provides a cycle-accurate simulator hosted entirely in Python, which performs just-in-time (JIT) compilation of the circuit design into a straight-line Python program that can be quickly evaluated to perform simulation. Every cycle, the user fully specifies the design inputs; each cycle performs one call to this circuit. In addition, PyRTL can transform the design into a C program that is compiled and brought back in as library, providing even faster simulation at the expense of compile time (this is similar to LastLayer's approach, but tied only to Python and PyRTL). Chisel [12], on the other hand, provides a library based on peek and poke operations that can test hardware at different levels, i.e. FIRRTL [14] and Verilog, without leaving the Scala language. Similarly, PyMTL [15] supports multiple levels of hardware simulation from functional to cycle-accurate simulation. Alternatively, LastLayer software integration and simulation features are based on a higher level of abstraction compared to these RTL oriented testing solutions. One notable difference is that our testing approach is not based on driving hardware interfaces. Instead, LastLayer allows programs written in external languages to test the hardware design using a fixed device-like interface, while still supporting cycle-accurate simulation. Also crucial to note is that LastLayer is compatible with any hardware language that can emit Verilog, such as PyRTL or Chisel.

Simulation libraries. There are tools today focusing solely on the task of testing and verifying hardware. One of these tools is *cocotb* [3], which is used for simulating designs written in VHDL and Verilog and supports a wide variety of commercial and open-source simulators. Furthermore, it uses co-routines available in Python to drive hardware interfaces, while benefiting from the libraries available in this language. For example, there are use cases available in *cocotb* for testing hardware using network libraries written in Python. However, *cocotb* is still based on the RTL abstraction. Users have to write tests that drive hardware interfaces, including clocks, resets, buses, etc. Additionally, *cocotb* uses the verilog procedural interface (VPI) for interfacing with hardware, while LastLayer is based on the direct programming interface (DPI). VPI is a more complex interface [17] and slower [4] compared to DPI.

6 Conclusion

LastLayer is a tool that enables hardware and software simulation regardless of how polished or finalized the hardware design is. More importantly, LastLayer enables this simulation without requiring any hardware modification. This is accomplished by generating DPI functions that access resources such as memories and registers available in the design, according to user specification. It can be used with any hardware language that emits Verilog. Moreover, the simulation interface provided by LastLayer is generic, enabling easy integration with other programming languages while still performing cycle-accurate simulation. Finally, the software tests written for LastLayer can be reused once hardware is fabricated or emulated, because the testing traces used in the simulation are produced by real programs.

7 Acknowledgements

We would like to thank members of Sampa and SAMPL groups at the Allen School for their feedback on the work and manuscript. This work was supported in part by NSF under grant CCF-1518703; by CRISP, a center in JUMP, a Semiconductor Research Corporation program sponsored by DARPA; by gifts from Xilinx, Intel (under CAPA), Oracle, Amazon, Qualcomm, Facebook, Futurewei, and other anonymous sources.

References

[1] AMBA AXI and ACE Protocol Specification. https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf.
[2] Avalon Interface Specification. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf.
[3] Cocotb, a co-routine based co-simulation library for writing VHDL and Verilog testbenches in Python. https://github.com/cocotb/cocotb.
[4] DPI, Direct Programming Interface. http://www.sugawara-systems.com/veritak_sv_tutorial/dpi.htm.
[5] Ibex 32 bit RISC-V CPU core. https://github.com/lowRISC/ibex.
[6] Icarus Verilog. http://iverilog.icarus.com/.
[7] Mentor ModelSim. https://www.mentor.com/products/fv/modelsim/.
[8] NumPy. https://numpy.org/.
[9] RISC-V foundation. https://riscv.org/.
[10] Synopsys VCS. https://www.synopsys.com/verification/simulation/vcs.html.
[11] Verilator. https://www.veripool.org/wiki/verilator.
[12] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA, 2012. ACM.
[13] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–7, 2017.
[14] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216, 2017.
[15] D. Lockhart, G. Zibrat, and C. Batten. Pymtl: A unified framework for vertically integrated computer architecture research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 280–292, 2014.
[16] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic. Yosys+nextpnr: An open source framework from verilog to bitstream for commercial fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–4, 2019.
[17] Stuart Sutherland. The Verilog PLI is dead (maybe)...long live the SystemVerilog DPI. https://sutherland-hdl.com/papers/2004-SNUG-presentation_Verilog_PLI_versus_SystemVerilog_DPI.pdf.
[18] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 18(4):36–45, 2001.