Zuspec: Pythonic Model-Driven Hardware Development

Matthew Ballance

2025-09-21

Abstract

Designing and implementing hardware is challenging, and is getting more difficult each year as systems become more complex. Design and verification teams are looking to boost productivity as a way to keep pace, while also looking for ways to provide models of the design behavior to software teams earlier. The fragmented and esoteric nature of the languages and methodologies used for design and verification only make this more difficult. Zuspec is a unified, extensible Pythonic framework for multi-abstraction hardware modeling that simplifies the design and verification process and enables traditional and AI-driven automation.

The abstraction gap and model fragmentation are two key drivers of hardware design-flow complexity. The abstraction difference between a natural-language design specification and the register-transfer-level (RTL) model that implements it is enormous, and only increasing as designs become larger and more complex. It's natural to prioritize the RTL model, since that is required to get to implementation. However, because RTL models are detailed and execute slowly, they don't do a good job of meeting the needs of adjacent disciplines, such as firmware. This delays how early software teams obtain access to a representation of the design, and limits the platforms on which they can work to fast hardware emulation or prototyping environments.

Over time, new requirements have resulted in the development of new domain-specific languages and language-like class libraries. SystemVerilog, PSS, UPF, IP-XACT, SystemC, and others all provide features that address pain points of hardware development. Unfortunately, this has also led to a very fragmented ecosystem with loosely-integrated languages and methodologies and complex build flows.

Considering the Ecosystem

All of these innovations have primarily come from a hardware-design perspective, and focus on enabling hardware-design flows. That, in itself, is a challenge given

the relative sizes of the hardware- and software-engineering ecosystems. While it's difficult to find accurate detailed data, US Bureau of Labor Statistics reports a labor-market size of 76,800 for hardware engineers vs a labor-market size of 1,534,790 for software engineers (inclusive of all sub-disciplines in both cases).

Given that the collective challenge is a combined hardware and software system, getting buy-in from both disciplines is critical. It seems quite unlikely that a language created to serve the unique requirements of the ecosystem's minority can serve the combined needs and gain the acceptance of the majority.

Zuspec

Zuspec [1] adopts Python as its starting point, and embeds hardware semantics into that ecosystem. The result is a platform that is familiar to software engineers, offers high productivity for hardware engineering, and has the goal of increasing the ability to share artifacts across the disciplines.

Why Python?

Many factors are involved in selecting a language for any purpose: key language features, tool ecosystem, relevant libraries, as well as the community around the language. Applying an existing language to the semantics of another domains raises another factor to consider: flexibility of the language.

Popularity can be Self-Reinforcing

Python is a popular language overall, holding the top spot in many rankings for several consecutive years, and being ranked highly for many years before that. Language popularity may not seem relevant compared to the technical features of a language. Not only it is relevant, popularity has a direct bearing on language technical features. TIOBE [2], for example, measures language rank in terms of searches via a range of internet search engines [3]. This is, of course, a rough measure a the size of a language's community. Larger communities produce more ideas for using a language and, thus, a a larger library ecosystem. Larger communities more-rapidly produce and refine adjacent technologies, such as code development and package management tools, and new language features.

Popularity often builds upon itself, and there is evidence that AI is acting as a driver of Python's popularity. Specifically, AI assistants are often reported to be more effective with Python than with other languages [4]. This has the effect of drawing more developers to Python, which increases the available code in Python, which more-rapidly increases the quality of results with Python.

[1] https://zuspec.github.io/

[2] https://www.tiobe.com/tiobe-index/

[3] https://en.wikipedia.org/wiki/TIOBE_index

[4] https://www.perplexity.ai/page/ai-generates-up-to-30-of-micro-Iy6zscIfSy6miYtIqvrsMA

But, language popularity is only relevant for the set of languages that can be used to capture relevant domain semantics. Fortunately, Python measures up very well here again.

Technical Arguments for Python

There are strong technical arguments for the Python language as well. As a dynamic language, Python provides excellent facilities for introspecting and manipulating a Python description.

```
import zuspec.dataclasses as zdc

@zdc.dataclass
class SendPacket(zdc.Struct):
  sz : zdc.Bit[8] = zdc.rand()

  @zdc.constraint
  def valid_sz(self):
    self.sz in [1, 2, 4, 8, 16]
```

Python `decorators` can be used to annotate elements of the description, identifying specific semantics to be applied to an element or attaching special processing instructions. For example, the `constraint` decorator above marks the body of the `valid_sz` method as having constraint semantics.

Python also allows inheritance relationships to be inspected at any point in time. The base type controls capabilities and restrictions of the derived type.

While Python can be used as a purely dynamically-typed language, it also provides the ability to associate type "hints" with variables. The `sz` field in the example above specifies that it is an unsigned 8-bit field. This enables the modeler to control how data in the model will be represented in the implementation.

Finally, Python supports AST introspection and transformation. This capability allows tools to access the raw AST for code, such as the constraint method above, without needing to use tricks like operator overloading.

Statically-typed languages often provide some introspection facilities that are available during the compilation phase. In contrast, Python allows these facilities to be applied to a Python description at any point in time, providing much more flexibility in processing flows. Effectively, Python allows libraries like Zuspec to act as a compiler within the Python interpreter.

Python also offers a strong set of system-programming features. Combined with Python's dynamic language features, these dramatically simplify the process of integrating external tools and systems.

Python also specifies a package specification and provides tools for producing, discovering, and consuming packages.

Together, these capabilities make Python a very compelling platform for developing, verifying, and publishing hardware models.

What does this look like?

All languages are a combination of syntax and semantics. Syntax governs the lexical aspects of a language: the keywords, operators, and legal ways of arranging them. Semantics governs the meanings of those statements – for example, whether `a = b` changes the value of the `a` variable, changes the `a` variable to reference the `b` variable, or something entirely different. Zuspec proposes a way to adopt full Python syntax, while identifying key regions in which different semantics apply to this syntax. These new semantics are always more restrictive than native Python semantics, allowing existing code checkers (eg mypy) to work unmodified.

Zuspec divides a Python description into two core region kinds:

- Pure Python regions
- Python regions with domain-specific semantics

Even a simple binary counter highlights several unique semantics that a hardware description must capture:

- Bit width of the `count` field
- When the count should be incremented
- Hardware-specific notions like reset

Zuspec identifies regions with these semantics using a combination of Python `decorators` and base classes. The example below shows a 32-bit counter modeled using the Zuspec library.

```
import zuspec.dataclasses as zdc

@zdc.dataclass
class Counter(zdc.Component):
  clock : zdc.Bit = zdc.input()
  reset : zdc.Bit = zdc.input()
  count : zdc.Bit[32] = zdc.output()

  @zdc.sync(clock=lambda s:s.clock, reset=lambda s:s.reset)
  def inc(self):
    if self.reset:
      self.count = 0
    else:
      self.count += 1
```

The `Counter` example above illustrates these two regions. By default, pure Python semantics are used. Consequently, the `import` statement at the top uses

pure Python semantics.

The `Counter` class inherits from the Zuspec `Component` class, which designates it as a class with specific capabilities. The `inc` method is decorated with the `sync` decorator. This marks it as a method that is automatically evaluated on the active edge of the specified clock or reset signals, and a method where deferred assignment is used. It also marks it as a method that may not be invoked directly.

A class domain with special semantics is a model of implementation, and cannot be used directly. Instead, a `Transformer` class must first be used to create an implementation. In this case, the implementation could be pure-Python, Verilog, or something entirely different like documentation.

```
import asyncio
from zuspec.be.py import ComponentFactory
import zuspec.dataclasses as zdc

@zdc.dataclass
class CountTB(zdc.Component):
  clkrst : zdc.ClockReset = zdc.field(init={period=10})
  counter : Counter = zdc.field(
      bind=lambda s:{
          s.counter.clock : s.clkrst.clock,
          s.counter.reset : s.clkrst.reset
      })

def test_smoke(self):
  tb = ComponentFactory(CountTB)
  asyncio.run(tb.clkrst.do_reset(count=10))
  assert tb.counter.count == 0
  asyncio.run(tb.clkrst.wait(count=10))
  assert tb.counter.count == 10
```

The example above shows a small testbench around the `Counter` component with a simple `Pytest` unit test. The type transformer creates a Python object that is used to dynamically evaluate the model. While the interface is Python, the implementation may or may not be Python. For example, the transformer might, instead, transform the model to Verilog and create a Verilator [5] simulator executable that evaluates the model much faster than a pure-Python implementation, while still exposing a Python interface to the signals.

```
module Counter(
    input            clock,
    input            reset,
    output reg[31:0] count);
```

[5] https://www.veripool.org/verilator/

```
    always @(posedge clock or posedge reset) begin
      if (reset) begin
        count <= {32{1'b0}};
      end else begin
        count <= count + 1;
      end
    end
endmodule
```

Another transformer might convert the model to the synthesizable Verilog shown above to be used as input to existing synthesis or simulation flows.

The User-Extensible Language

The `Counter` example is quite simple, and at the register-transfer level (RTL). Zuspec is designed to be able to capture a broad range of hardware-centric semantics that cover:

- Verification
  - Constrained randomization and functional coverage
  - Portable test and stimulus (PSS)
  - Assertions (SVA)
- Hardware design
  - RTL
  - Medium-level synthesis
  - Transaction-level modeling (TLM)
  - Register modeling
  - Clock and power domains
  - Physical design
- Firmware
  - Register interface
  - Specialized memory management

Zuspec brings a new approach to domain-specific languages that leverages the popularity and ecosystem of Python, and embeds domain-specific semantics from hardware design, verification, and firmware. The big "bet" is that educating humans and LLMs on the delta between a Python and a set of domain-specific semantics is far easier than doing the same with a completely new language.

While Zuspec models are captured in Python, processing tools may transform Zuspec models into appropriate non-Python implementations such as embedded C or synthesizable Verilog. Zuspec encourages a Pythonic development process for hardware that is code-centric, AI-friendly, and nimble. And, most importantly, Zuspec provides togetherness: a common environment in which new language innovations can be explored along with integrated existing technologies.