Accelerator Interfacing is Like an Onion*

Katie Lim
katielim@cs.washington.edu
University of Washington

Jonathan Balkind
jbalkind@ucsb.edu
UC Santa Barbara

ABSTRACT

Herein, we rant.

1 PROGRAMMING A NEW MACHINE FROM SCRATCH

There is a fundamental difference in how we go about building out infrastructure atop a CPU-centric software system versus an accelerator-focused system. Software systems are typically built up as layers of abstraction whereas accelerator-focused systems are built both monolithic and bespoke.

Imagine you're programming a new machine from scratch that has a CPU with a shiny new ISA. You have some architectural resources available to you and you want to build some abstractions. It's your responsibility to create those abstractions using the resources that you've been provided and some of the first abstractions that you're likely to want to create include software mainstays like stacks and functions. What begins as a calling convention quickly morphs into a more formal ABI. This ABI allows you to support multiple languages and abstractions that may otherwise have little relation to each other. You quickly solidify some of your resources as serving particular responsibilities, like keeping a stack pointer, argument registers, locals, a return address, and a return value. Soon, your users will consider these the basic atoms of your system and will be ossified beyond memory. These abstractions are relatively simple though - a single function call, a single stack to push or pop, maybe a little bit of recursion. To enable more flexibility, we need to coordinate across function calls and data structures by building additional abstractions atop our basic ones to enable functionality like sequencing and synchrony and asynchrony. Eventually, we find ourselves at the apex, coordinating calls with varying semantics, and exposing behaviours across our abstractions to a programmer. We've finally made it: The application programmer interface (API).

Now imagine you're programming a new machine from scratch. Except it's hardware made of accelerators. What is this bundle of wires I'm being given? Just when and how is data meant to flow again? Why does everything need to use AXI which only lets us do loads, stores, and maybe some atomics? Why are we attached to PCIe which only lets us do loads, stores, and maybe some atomics? What is the difference between CXL and PCIe? Where am I? When did I start writing this hardware design anyway? Did someone mention an abstraction?

Now imagine you're programming a new machine from scratch that is going to be all (or mostly) hardware accelerators. Do we have an abstraction stack like in software? We, too, can have resources: Maybe they're wires, registers, memories; most commonly for our purposes they are the ports on our modules. Upon these, we need to establish our calling convention or ABI: The first step here is timing details. We can, implicitly or explicitly, have tight timing requirements, like an initiation interval as often used in HLS or for DSPs, or more elegantly as expressed with a timeline type a la Filament. We can alternatively express timing more abstractly via handshakes like latency-insensitive interfaces. Here, we begin to be able to see equivalences to the calling convention or ABI that describe how data is communicated over physical resources. This is roughly the level at which we define interfaces like AXI. Likely PCIe and CXL too, based on most uses thereof. But how can we continue from here?

Imagine you're still programming a new machine from scratch. You've got a wonderful new hardware ABI - you maybe even have the ability to do data movement via a protocol like AXI or PCIe or CXL. What kinds of features are you likely to build on top? From here, you can probably build functions or streams that give semantic information to data on the wire or mechanisms like memcpy. Many projects have stopped here and exposed this level of abstraction in such a manner to software. The programmer can call a magical function inside their application, at which time an unknown library or driver or other contrived mechanism will move their bytes through the ether and provide functionality resembling a function. Similarly, this level can be exposed from one accelerator to another, but this relies on a variety of guarantees and ecosystem-specific conventions to keep straight, stymieing our ability to plug and play across languages and systems in the manner we do from software with features like foreign function interfaces.

Now imagine you're programming a new machine from scratch. Well, this time it's not so from scratch. You've got some lovely components written by a far-off collaborator who likes that one HDL you never wanted to read code in. But they told you that they're using common interfaces that they call a function or a stream or a DMA. You have to coordinate the use of these and incorporate higher-level semantic information. This is beginning to smell like an API, but at times it remains as bare as a function call itself, monomorphised in its types and lacking depth. This is roughly where we find ourselves today: AXI conventions that we call functions but which are just wires and latency-insensitive interfaces in a trench coat. Or PCIe-exposing drivers bundled in half-abstracted libraries requiring a kernel rebuild.

The API that specifies the coordination of components still eludes us and we lack the formalism or taxonomy to differentiate between these layers in a deep way. We are left with an onion or a parfait or something else that looks remarkably similar at each layer while it tries to convince us that its abstraction exists. In software, we can lie with types and convince ourselves of the abstraction, but down here in hardware, all complexity is laid bare as wires and gates and registers and time.

*"Onions have layers." - Shrek

LATTE '25, March 30, 2025, Rotterdam, NL

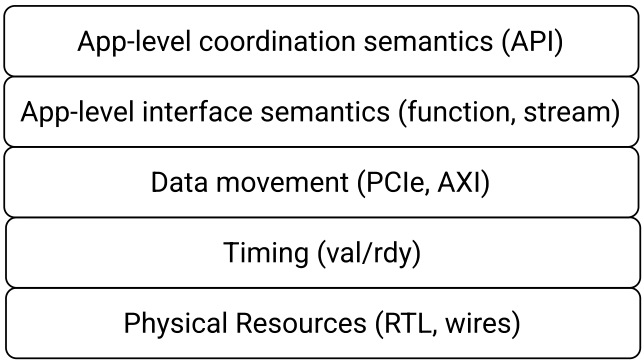

Figure 1: A possible parfait of hardware interaction layers

2 WHY DO WE CARE?

The monolithic nature of how we develop hardware makes it difficult to develop and express intra and inter-accelerator interactions accurately. Names of protocols and interfaces may be unhelpfully or erroneously broad, leading to everyone "Tower of Babel"-ing at each other. An example of a name that is unhelpfully broad is PCIe. PCIe is actually a standard that covers everything from the physical signaling on wires all the way up to how addressing and routing is done for devices attached to a PCIe bus. This broadness becomes an issue when we bring in CXL, which runs "over" PCIe. What does it mean to run over PCIe? Does it use just the physical layer? Or is it using PCIe addressing too? By not recognizing that PCIe actually refers to several layers, it is difficult to ascertain what CXL is actually doing.

AXI is also often used erroneously broadly. We (your entertainers) have heard "AXI transaction" used to refer to everything from wires, to bursts, all the way to a "function invocation". However, AXI transaction is not adequate to capture the full details of a function invocation. Information about what data is passed over the AXI bus and in what format is all necessary semantic information. Instead of in software where this semantic information is often conveyed via types, AXI provides no specification of how to convey this information. Instead, the typical approach is "ha ha we have kind of agreed on this sequence of bytes to be passed over the bus".

Two half-baked representations of our thoughts on layers in hardware are shown in Figure 1 and as follows:

- Physical resources: what are the wires and the ports and memory?
- Timing: when should signals take certain values?
- Data movement: addresses, bytes, reads, and writes
- App-level interface semantics: I'm a function! (What does the data on the wire mean)
- App-level coordination semantics: API (how do we coordinate the system)

3 NETWORKS ON CHIP

Network protocols give us structure and formalism that we can leverage. We know that the data which comes first is a header and belongs to the outermost layer of protocols that we might care about right now. Whatever is in the payload can be worried about later. Perhaps network protocols can give us some of the formalism and naming that we actually need for these other purposes. And if we can re-cast these other layers and behaviours in networking terms, we can begin to understand them and name them better? That is, AXI is itself a protocol, and we can in many cases carry it over another protocol (some of us have unfortunately seen 1-bit AXI over SPI or some other cursed arrangement). Upon networks, we built RPCs and API concepts and much more. While we didn't set out to argue for NoCs, perhaps there is something more general to learn from them.

Protocols also show us a similar set of Ouroboros-style layer mangling. IP-in-IP is common, and generic routing encapsulation (GRE) can carry any layer inside another layer. Why is this useful? Well, we need a stack to think of some things as being more or less abstract. We also get explicit indication of whether something belongs to a lower or upper layer and require the recipient to process the messages accordingly, including encapsulation or de-encapsulation.

4 TYPES AND ABSTRACTIONS

To advance toward the hardware systems programming language equivalent to Rust, we must consider how types and abstractions can be built and checked in a deeper way than we have yet managed. Given that Rust took several decades of core programming languages research before it became conceivable, we probably ought to build a few more languages and tools before claiming to embark on a simmilar endeavour. One could argue that SystemVerilog interfaces and structs give us the most basic naming mechanism upon which other abstractions could be built, however SystemVerilog interfaces are near universally derided because tool implementation is apparently too difficult when contemplating the full complexity of the rest of the language's specification. How can we then consider the application of typed semantic information on bundles of directed wires (input/output ports for example) to be represented when we end up in the inevitable IR that SystemVerilog has become? How can we build high-level abstractions which can simultaneously be solidly built upon and yet also be permeable in enabling cross-layer introspection? What does it mean for such an "abstraction" to exist?

5 OPEN QUESTIONS

- What is an API in hardware?
- Can network protocols and NoCs lead us to sanity?
- What is an API in hardware?
- If I bless my wires or registers or timing details with the gift of semantics, how can I express and retain those semantics?
- What is a reasonable way to enable expressing concrete abstractions while also enabling the breaking of those abstractions?