2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Soumak

How rich descriptions enable early detection of hookup issues

Peter Birch (*peterb@graphcore.ai)* and Thomas Brown (*thomasb@graphcore.ai*)
Graphcore Ltd, Bristol, UK

***Abstract*—Complex system-on-chip assembly in an HDL such as SystemVerilog can be impractical due to the syntactical verbosity and inconsistent results between tools. Graphcore has developed Soumak, with a concise syntax that focuses on assembling complex hierarchies while clearly capturing the designer's 'intent'. Soumak's analysis and reporting flows can detect many abnormalities at an early stage, reducing the chance of a bug reaching production.**

Keywords—Python; SystemVerilog; RTL; CDC; Wiring; Top-Level

I. Introduction

A system-on-chip (SoC) often comprises many components and subsystems, arranged into deep and complex hierarchies and connected by an intricate web of wiring. As SoCs grow ever more complex, the task of constructing and verifying these systems dominates the development timeline, which is exacerbated by long compilation and simulation times. Exhaustive simulations, formal proof, and other verification strategies often become infeasible at this scale, and late detection of bugs can be costly and comes with an increased risk of an issue reaching production.

Manually instancing and connecting subsystems, pads, test instruments, and other critical components to form the top level of a SoC in a hardware description language (HDL) such as SystemVerilog is a time-consuming and error-prone endeavor. With thousands of connections required, the occasional mistake is to be expected. Gross issues such as mismatching bus widths may be simple to detect through lint or simulation, but more subtle issues such as incorrectly connecting two infrequently used one-bit strobe signals may take much longer to detect.

Abstraction can be used to reduce the complexity of forming connections, allowing entire interfaces (for example a data bus with associated sideband and flow control signals) to be routed as a single entity. While existing HDLs offer some limited support, the concept can be taken further, providing the opportunity to detect certain issues earlier in the development process (a 'shift-left' [1] approach). This can eliminate trivial hookup issues and allow verification to focus on finding deeper functional problems.

This paper will discuss Soumak, an internal tool developed by Graphcore's Logical Design team to assemble subsystems and SoC top levels using a concise syntax with support for complex signal types and rich annotated metadata. It will detail how such a tool is integrated into the design process and will explore how it enables early issue detection prior to lint or test. Applications include the generation of RTL, documentation, coarse-grained clock domain crossing (CDC) analysis, and guiding third-party tools such as static analysers.

II. Existing Solutions

The Accelera IP-XACT [2] standard defines a machine-readable format (based on XML) for describing a component's capabilities and its interface boundary. Tools such as Agnisys' IDesignSpec [3] or the open-source Kactus2 [4] parse these component descriptions and provide a graphical environment for subsystem design. However, when building complex and repetitive topologies such as rings, chains, and meshes, a code-based approach can be preferable. IP-XACT is cumbersome to work with without software assistance, and its allowance for vendor-specific extensions provide a hurdle to maintaining custom tooling.

At the other extreme, assembling a design using only a main-stream HDL can be a difficult task. While both SystemVerilog [5] and VHDL [6] support interface ports (which carry signals travelling both into and out of a block), commercial tool support is highly variable - for example the way in which names are transformed in synthesis can vary wildly between vendors, making it difficult to build a consistent back-end flow. Further to this, limitations of the syntax can make hookup labour intensive and error prone. Alternative HDLs such as the Scala-

based Chisel [7] or Python-based Amaranth [8] attempt to address some of these shortcomings by leveraging the capabilities of the modern scripting languages on which they are based. However, these HDLs can be difficult to partially adopt and can require complex shims at the boundary of legacy or vendor-provided IPs.

Tools such as EnjoyDigital's LiteX [9] and Blu Wireless' BLADE [10] have attempted to tread the 'middle-ground' of automating subsystem construction but leaving leaf node implementation to be handled by a mainstream HDL. However, LiteX is primarily intended for FPGA development and would need alternation to support ASIC design. By contrast, BLADE is intended for ASIC development, however its YAML-based input syntax is difficult to extend and has no intrinsic support for variables, loops, or conditionals. In Graphcore's experience, having a flexible and easily extendable design capture syntax is essential when developing a complex system, but some constraints may help to guide the designer towards producing high quality and maintainable code.

III. A New Approach

There were several factors to why a new approach was necessary and not simply a case of 'reinventing the wheel'. The first was scale: Graphcore's IPU chip is among the largest and most complex single-die ASICs in history, containing thousands of subsystems and hundreds of thousands of connections - dealing with this complexity in a mainstream HDL would be impractical. By describing modules and interfaces in a richer, more abstracted way, the designer's intent could be better captured. Such descriptions could then be used to guide and automate connections leading to fewer mistakes and reducing the total engineering effort required.

The second factor was integration: Graphcore's internal IP is primarily written in SystemVerilog and all internal tooling is designed around this approach. Adopting a modern HDL such as Chisel could solve many problems, but it would require substantial cost in retraining the design team and then either porting legacy IPs or coping with difficult boundaries between the two ecosystems. The ideal solution would be to not disturb existing IP implementations while simplifying the process of subsystem and SoC construction. Being able to share constants, enumerations, and structured data types between the different descriptions could further improve the integration.

A third factor was focus: separating the design of detailed logic from the construction of the hierarchy enforces clean boundaries to be formed around each IP, meaning that flops and gates simply cannot be part of wiring layers. The syntax of each tool can then be tailored to the problem it addresses - for hierarchy construction this could allow common topologies (such as chains, rings, and meshes) to be formed in a simple statements, hiding the complexity of the supporting connections.

Finally, the solution needed to be flexible and able to adapt to rapidly evolving architectural requirements. Previous internal tools had been built around a fixed topology, which made it simple to construct a design which complied with the expected architecture but very difficult to form something radically different. Ideally the tool would provide robust primitives for describing and connecting hierarchies, but also offer simple ways to extend the functionality to support the formation of complex and repeated topologies.

IV. Introduction to the Syntax

Soumak's syntax is modelled after Python's dataclass library [11]. This provides a clear and concise way of defining constructs such as Blocks, Structs and Interfaces, with type-annotated fields specifying components such as ports, subblocks, struct and interface fields, and parameters.

Soumak has six *meta* types: **`block`**, **`interface`**, **`struct`**, **`union`**, **`package`** and **`enum`**. These are defined by using a class decorator as shown in Example 1.

```
@soumak.interface()
class Handshake:
  """ A simple handshake interface """
  valid : Request(width=1, desc="Forwards travelling component")
  ready : Response(width=1, desc="Backwards travelling component")
```

Example 1 Soumak definition of a simple handshake interface. '**`Request`**' components travel with the direction of the interface and '**`Response`**' components travel in the opposite direction. The optional docstring and descriptions add metadata to the components.

Interfaces are flattened out into a set of simple ports (maximizing tool compatibility), although there is the option of rendering them into SystemVerilog interfaces. On the other hand, structs, unions and enum types translate canonically into SystemVerilog. Example 2 shows the definition and translation of a simple struct.

```
@soumak.struct()
class ControlledData:
  " A Data bus accompanied by a valid bit "
  valid : Scalar()
  data  : Instance(Data)
```

```
// A Data bus accompanied by a valid bit
typedef struct packed {
  logic  valid;
  data_t data;
} controlled_data_t;
```

Example 2 Soumak definition of a struct, where '**Data**' is another type which could be a union, enum, typedef or another struct. The right-hand side shows the struct as rendered in SystemVerilog – the Python docstring helpfully appears as a comment.

Unlike interfaces, structs and unions do not carry direction, so components within them are either scalars or instances of other defined types.

The definition of packages and enum types are similar to standard dataclasses. Packages can contain **Constants** and **Typedefs**; enum types can either explicitly define the underlying value of each field or can have the tool enumerate them automatically.

Soumak block definitions, like SystemVerilog modules, contain port definitions. If the block represents a wiring layer in the design hierarchy, it will also contain subblocks and hookup definitions – subblocks are declared using the **Instance** keyword and connections are specified with a **connect** method. Otherwise, it is – from Soumak's point of view at least – a leaf node and its implementation (sequential/combinational logic, etc) must be defined in a separate RTL template file.

```
@soumak.block()
class Parent:
  clk   : In(desc="Inbound clock")

  child : Instance(Child)

  def connect(self):
    self.link(self.clk, self.child.clk)
```

```
// A simple block with child
module parent (
  input logic i_clk  // Inbound clock
);

child u_child (
  .i_clk ( i_clk )
);

endmodule
```

Example 3 Soumak definition of a block with one input port '**clk**' and one subblock, '**child**', which we assume also has an input '**clk**' port.

NOTE: The Soumak definition of the '`clk`*' port in Example 3 does not include the prefix '*`i_`*'. This is because in general a port could contain a complex hierarchy of interfaces with signals travelling in opposing directions – Soumak is left to add the '*`i_`*', '*`o_`*' or '*`io_`*' prefixes to the individual flattened components automatically.*

The **connect** method has access to a special method, **link**, which forms point-to-point connections. This example forms a single-bit connection but in general a single **self.link(...)** statement can join ports or signals of any type, including interfaces – hence Soumak code is less verbose than the equivalent SystemVerilog.

If a block has multiple subblocks and connections between them, Soumak automatically declares and hooks up the necessary intermediate wires when rendering the block in SystemVerilog.

```
self.link(self.child_a.egress,
          self.child_b.ingress)
```

```
logic child_a_egress;

Child u_child_a (
  .egress ( child_a_egress )
);

Child u_child_b (
  .ingress ( child_a_egress )
);
```

Example 4 Connections in Soumak are less verbose than SystemVerilog, and to an even greater extent when connecting complex interfaces.

Blocks and Interfaces can also contain parameters, much like SystemVerilog modules. These parameters can define integers, strings or Booleans which can control, for example, signal widths, pipeline lengths and conditional inclusion of components.

```
@soumak.interface()
class ParameterizedData:
  dwidth   : Parameter(desc="data width") = Default(32)
  control  : Parameter(desc="include valid signal") = Default(True)
  valid    : control @ Request(desc="presence is determined by control parameter")
  data     : Request(width=dwidth, desc="width is determined by dwidth parameter")

@soumak.block()
class Pipeline:
  num_stages : Parameter(desc="number of pipe stages") = Default(1)
  stages     : num_stages * Instance(PipeStage, desc="a bundle of PipeStage intances")
```

Example 5 Soumak parameters in use

Example 5 introduces the conditional **@** notation, which includes the component only if the Boolean on the left-hand side is true; and the bundle notation *****, which declares an array of components. Like SystemVerilog, parameters must have default values.

V. A Worked Example

To demonstrate the benefits of Soumak when it comes to constructing complex hierarchies of blocks and interfaces, this section walks through the construction of an AMBA AXI4-Lite [12] interface and pipeline block hierarchy. AXI4-Lite is chosen over the full AXI interface for brevity – but even the AXI4-Lite interface contains nineteen individual signals.

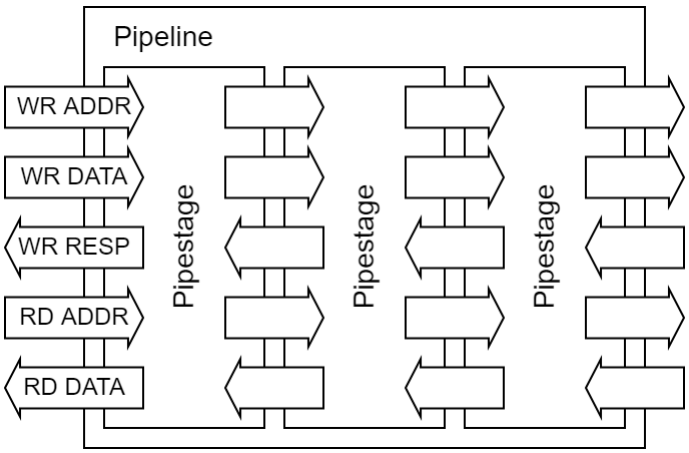


Figure 1 Showing the desired pipeline hierarchy and five AXI4-Lite channels.

The AXI4-Lite interface is constructed as a hierarchy, starting first with the address channel – this interface is reused for both the read address and write address channels. The interface extends the **`Handshake`** interface defined in Example 1 - this means it picks up the **`valid`** and **`ready`** signals and needs only to add **`addr`** and **`prot`**.

```
@soumak.interface()
class Axi4LiteAddressChannel(Handshake):
  width : Parameter(desc="Address width") = Default(32)
  addr  : Request(width=width)
  prot  : Request(width=3)
```

The write data, write response and read data channels are defined similarly:

```
@soumak.interface()
class Axi4LiteWriteDataChannel(Handshake):
  width : Parameter(desc="Data width") = Default(32)
  data  : Request(width=width)
  strb  : Request(width=width / 8)

@soumak.interface()
class Axi4LiteWriteResponseChannel(Handshake):
  resp  : Request(width=2)

@soumak.interface()
class Axi4LiteReadDataChannel(Handshake):
  width : Parameter(desc="Data width") = Default(32)
  data  : Request(width=width)
  resp  : Request(width=2)
```

Finally, the AXI4-Lite interface instantiates all five required channels:

```
@soumak.interface()
class Axi4Lite:
  data_width : Parameter(desc="Data width"   ) = Default(32)
  addr_width : Parameter(desc="Address width") = Default(32)
  aw         : Request(Axi4LiteAddressChannel,   width=data_width)
  w          : Request(Axi4LiteWriteDataChannel, width=data_width)
  b          : Response(Axi4LiteWriteResponseChannel)
  ar         : Request(Axi4LiteAddressChannel,   width=addr_width)
  r          : Response(Axi4LiteReadDataChannel, width=data_width)
```

The block hierarchy contains two block definitions, **`Pipeline`** and **`Pipestage`**. **`Pipestage`** is a leaf node, meaning that the sequential logic forming its implementation is defined in a separate RTL file.

```
@soumak.block()
class PipeStage:
  clk     : In(Clock)
  ingress : In(Axi4Lite)
  egress  : Out(Axi4Lite)
```

`Pipeline` is a wiring layer so its entire definition is written in Soumak.

```
@soumak.block(traits=[Chain])
class Pipeline:
  clk     : In(Clock)
  ingress : In(Axi4Lite)
  egress  : Out(Axi4Lite)

  stages : 3 * Instance(PipeStage)

  def connect(self):
    self.link(self.ingress, self.stages[0].ingress)
    self.chain(self.stages.all.egress, self.stages.all.ingress)
    self.link(self.stages[-1].egress, self.egress)
```

The port list of **`Pipeline`** is identical to that of **`Pipestage`**, but the implementation differs. The pipeline is declared to be of length three but could easily be parameterized as shown in Example 5. The most interesting part of the connect method is the second line. Here a topology method **`chain`** is employed, which is supplied by the trait **`Chain`** (seen as an argument to the decorator). This is a powerful hookup tool which takes two lists and connects them as a chain, exactly as shown in Figure 1. Finally note the suffix of **`.all`** on **`self.stages`** – this is a special attribute that allows a collection of objects to be formed by expanding bundled components within a signal's hierarchy. Thus **`self.stages.all.egress`** is equivalent to **`[stage.egress for stage in self.stages]`** and likewise for **`self.stages.all.ingress`**.

Comparing the Soumak description for this Pipeline block with what would be required to define the equivalent SystemVerilog module demonstrates just how concise and powerful Soumak is as a hardware description language.

In addition, future alterations to the definition of one of the interfaces, the number of pipeline stages, or the topology of the wiring layer require a very small number of line changes compared to SystemVerilog.

VI. Benefits of Rich Descriptions

Sections IV and V introduced the syntax for design capture and demonstrated how it could be used to compactly describe complex interfaces and block hierarchies. The generation of SystemVerilog wiring layers is only a single use case; this section explores how the rich syntax can be used to drive advanced analysis and reporting flows.

A. Early Issue Detection

Missing or incorrect connections in a big design can take a long time to find and resolve as testbenches will be complex and simulations may run very slowly. Therefore, detection of such issues early in the design process can save considerable effort. Certain types of bad connection may be easier to detect than others - for example linking two signals of different widths will be detected by most HDL compilers and lint tools, but detecting incorrect connection of valid and ready qualifiers to a network-on-chip is a far more subtle error to detect as the bad design can still be syntactically correct.

By capturing a better understanding of the designer's intent this can allow a tool to make more informed judgments as to what the correct connection should be. The use of structured interfaces (introduced in Section IV) is a large part of this since it allows connections between two incompatible ports to be quickly flagged. For example, as the syntax allows an entire interface to be connected in a single statement, this would make it impossible to cross-connect an AXI **`arvalid`** (read request valid) to an **`awvalid`** (write request valid) as the detailed component connections are outside of the designer's control.

Sometimes connections at a more detailed level are essential, and in these cases strict type checking can help to spot abnormalities. For example, there are often many signals of a single bit width such as clocks, resets, valid and accept qualifiers, and so on. In SystemVerilog such signals may all be declared as a one-bit **`logic`**, and an **`assign`** statement can easily link any of them together. Such problems may be detected by linting or other analysis tools, but are not banned by the language. By contrast, Soumak provides a primitive **`Scalar`** type (equivalent to **`logic`**) but encourages the use of more specific signal types such as **`Clock`**, **`Reset`**, or custom types defined by the designer. Even if the signals at either end of a connection are identical in width, if the types do not match then the connection is prohibited and an error is raised. For the rare case where a conversion between types is intended, an explicit cast is required to link the points together and unnecessary casts raise an error to prevent indiscriminate usage.

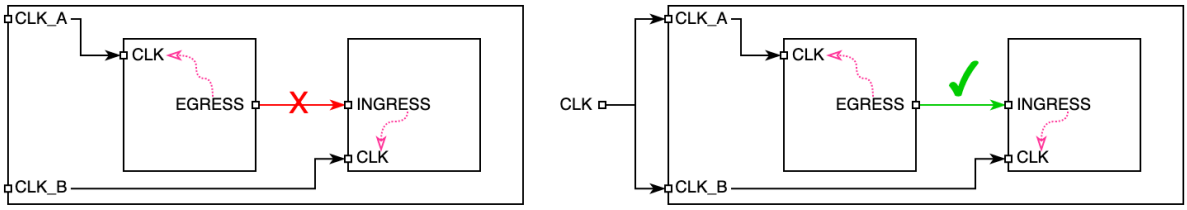


Figure 2 How annotations and connectivity tracing can detect crossings between clock domains

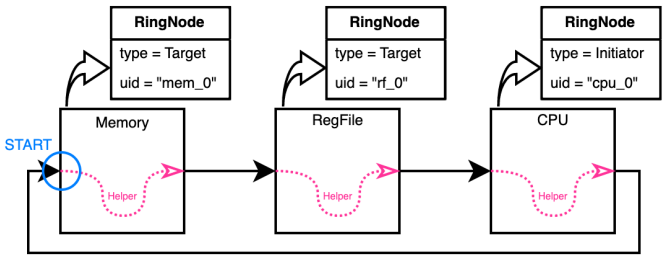
Certain CDC violations can be straightforward to detect if the clocks associated to the endpoints of a connection are known. As Soumak is not aware of the implementation of leaf nodes, this requires the clock and reset to be explicitly annotated onto the ports of each block. As shown in Figure 2, when combined with the connectivity information, clocks and resets can be traced back to determine whether they stem from a common source, with differing sources flagged as an error. By spending a small amount of time in maintaining these annotations, this approach can detect gross problems early in the design process. As a further benefit, these annotations can then be exported as a baseline for constraints for third-party CDC analysis tools, reducing redundant data entry.

B. Tracing & Annotations

In a complex hierarchy manually tracing signals becomes tiresome and error-prone. By leveraging rich connectivity descriptions, Soumak can report ports and blocks encountered while following a chain of connections

from a given starting point – this is called 'connection tracing'. However, as Soumak is unaware of the implementation and internal connectivity of leaf nodes, the trace will terminate prematurely. To avoid this, the designer provides 'helper' methods that yield each pathway exiting a block from a given entrypoint, thus allowing the tool to complete the trace.

Annotations are another mechanism that Soumak provides, allowing arbitrarily formatted data to be associated to any object in the design. Annotations are created after construction of each layer of the design and so they may inspect any attribute of the constructed object including finalized parameter values. This allows a unique annotation to be created for every instance of a given interface, block, or other type. If an object presents multiple annotations then it may associate them to particular attributes (such as a port), which can then be used by the tracer to filter out irrelevant information.



Trace Report		
Block	**Port**	**Annotated**
Memory	Ingress	✓
Memory	Egress	
RegFile	Ingress	✓
RegFile	Egress	
CPU	Ingress	✓
CPU	Egress	

Figure 3 Tracing connections around a simple ring of three nodes

The combination of connection tracing and object annotation enables automated reporting processes. Consider the ring topology in Figure 3 that contains both initiators and targets - each node in the ring has a pair of ingress and egress ports that transport a packetized protocol called **`RingBus`**. A helper function assists connection tracing by accepting the ingress port of an initiator or target and returning the matching egress port, allowing the tracer to complete the loop around the ring. A **`RingNode`** annotation is associated to each node's ingress port, specifying the block's type (initiator or target) and its relation to an architectural specification via a unique ID. The output from tracing the connections from a given starting point can then be fed into a simple tool that captures the annotation object for each node encountered and produces human and machine-readable reports.

C. Shared Definitions

In any project involving both hardware and software development, repeated declaration of information can lead to inconsistencies and bugs. Being able to share common descriptions reduces the chance of inconsistencies and can save a team considerable effort.

One such example is microarchitectural documentation. As a design matures its hierarchy, interfaces, and data structures can change frequently. Where documentation is manually maintained, it can quickly fall out of step with what has been implemented, which can be both unhelpful and potentially misleading. This problem can be partly solved by using Soumak to define constants, interfaces and data types and then using these to generate sections of the documentation. Diagrams may also be generated from the assembled design to show the hierarchy or interesting connectivity, such as the clock tree or the aggregation and distribution of a particular data bus.

In a similar fashion, header files for verification or software development can be generated to allow firmware running on embedded CPUs to easily interact with peripherals – sharing constants, enumerated values, and data structures. These source files can be automatically regenerated and released each time updates are made to the hardware, keeping firmware in step with the changing functionality.

Detailed information can be derived from tracing connections and collecting annotations, such as generating unique maps of complex address spaces for each initiator in a subsystem as shown in Figure 4. By annotating each point in the design where streams of data are either arbitrated or distributed with a mapping from target address to egress port, the address map can be progressively accumulated and will automatically adapt as the design matures.

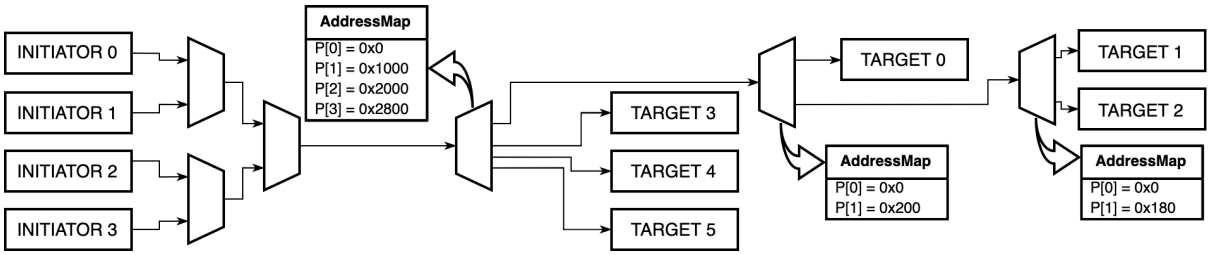


Figure 4 How address maps can be progressively derived from tracing connectivity and extracting annotations

VII. Conclusions

This paper introduced Soumak, an internal tool developed by Graphcore's Logical Design team to assist in the assembly of complex subsystems and systems-on-chip. The current state of commercial and opensource tooling was explored along with Graphcore's motivations for developing a new tool, this highlighted that no existing solution quite addressed the design's required scale nor the need to mix different HDLs.

Soumak's syntax, styled on Python's dataclass library, can be used to concisely construct complex, deeply hierarchical systems. Class inheritance allows common attributes and behaviours to be shared between related interfaces and subsystems, reducing the amount of repetition in the design. Comparing Soumak to SystemVerilog highlights how the tailored syntax can result in far fewer lines of code for the same design, helping to keep the designer's intent clear and reducing the chance of a bug.

Rich hardware descriptions offer numerous benefits, such as detecting unintentional connections at an early stage in the design process and producing rich reports of a design's structure and connectivity. Concise hardware descriptions can significantly reduce the effort required in maintaining a design, in some cases making it as simple as swapping two lines to reposition and reconnect all nodes within a topology such as a ring.

Developing and maintaining a tool as capable and complex as Soumak is a big investment for a silicon team, but it is Graphcore's belief that this investment in concise connectivity, early issue detection, and rich reporting is more than paid back during the development of a chip as complex as the IPU. The team will continue to build on this robust foundation as our silicon product offerings continue to advance and push technological barriers.

VIII. References

[1] "What it Means to "Shift Left" in Software Testing," SmartBear Software, [Online]. Available: https://smartbear.com/learn/automated-testing/shifting-left-in-testing/. [Accessed 20 August 2022].

[2] "IEEE Standard for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tool Flows," *IEEE Std 1685-2014 (Revision of IEEE Std 1685-2009),* 2014.

[3] "IDesignSpec - UVM Register Generator, SystemRDL Compiler," Agnisys, [Online]. Available: https://www.agnisys.com/products/idesignspec-uvm-register-generator/. [Accessed 20 August 2022].

[4] "Kactus2," Tampere University System-on-Chip Research Group, [Online]. Available: https://research.tuni.fi/system-on-chip/tools/. [Accessed 20 August 2022].

[5] S. Sutherland and D. Mills, "Synthesizing SystemVerilog - Busting the Myth that SystemVerilog is only for Verification," in *SNUG Silicon Valley 2013*, 2013.

[6] B. Brosens, "VHDL 2019: Interfaces," Sigasi, 11 June 2020. [Online]. Available: https://insights.sigasi.com/tech/what-is-new-in-vhdl-2019-part2/. [Accessed 20 August 2022].

[7] "Chisel/FIRRTL Hardware Compiler Framework," [Online]. Available: https://www.chisel-lang.org. [Accessed 20 August 2022].

[8] "Amaranth HDL toolchain," [Online]. Available: https://amaranth-lang.org/docs/amaranth/latest/. [Accessed 20 August 2022].

[9] "LiteX," Enjoy Digital, [Online]. Available: https://github.com/enjoy-digital/litex. [Accessed 20 August 2022].

[10] "BLADE," Blu Wireless Technology Limited, [Online]. Available: https://github.com/bluwireless/blade. [Accessed 20 August 2022].

[11] "Data Classes," Python Software Foundation, [Online]. Available: docs.python.org/3/library/dataclasses.html. [Accessed 20 August 2022].

[12] "AMBA AXI and ACE Protocol Specification Version E," Arm Limited, 22 February 2013. [Online]. Available: https://developer.arm.com/documentation/ihi0022/e/AMBA-AXI4-Lite-Interface-Specification. [Accessed 20 August 2022].