Wire Sorts: A Language Abstraction for Safe Hardware Composition

Michael Christensen
Department of Computer Science
University of California, Santa Barbara, USA
mchristensen@cs.ucsb.edu

Timothy Sherwood
Department of Computer Science
University of California, Santa Barbara, USA
sherwood@cs.ucsb.edu

Jonathan Balkind
Department of Computer Science
University of California, Santa Barbara, USA
jbalkind@ucsb.edu

Ben Hardekopf
Department of Computer Science
University of California, Santa Barbara, USA
benh@cs.ucsb.edu

Abstract

Effective digital hardware design fundamentally requires decomposing a design into a set of interconnected modules, each a distinct unit of computation and state. However, naively connecting hardware modules leads to real-world pathological cases which are surprisingly far from obvious when looking at the interfaces alone and which are very difficult to debug after synthesis. We show for the first time that it is possible to soundly abstract even complex combinational dependencies of arbitrary hardware modules through the assignment of IO ports to one of four new sorts which we call: **to-sync**, **to-port**, **from-sync**, and **from-port**. This new taxonomy, and the reasoning it enables, facilitates modularity by escalating problematic aspects of module input/output interaction to the language-level interface specification. We formalize and prove the soundness of our new wire sorts, implement them in a practical hardware description language, and demonstrate they can be applied and even inferred automatically at scale. Through an examination of the BaseJump STL, the OpenPiton manycore research platform, and a complete RISC-V implementation, we find that even on our biggest design containing 1.5 million primitive gates, analysis takes less than 31 seconds; that across 172 unique modules analyzed, the inferred sorts are widely distributed across our taxonomy; and that by using wire sorts, our tool is 2.6–33.9x faster at finding loops than standard synthesis-time cycle detection.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454037

CCS Concepts: • **Hardware → Hardware description languages and compilation**; *Software tools for EDA*; *Semi-formal verification.*

Keywords: hardware description languages, modules, composition

ACM Reference Format:
Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. 2021. Wire Sorts: A Language Abstraction for Safe Hardware Composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454037

1 Introduction

In our current era of diminished transistor scaling, the need for higher levels of energy efficiency and performance is greater than ever. The quest to achieve these goals calls for more people to be able to participate in the creation of accelerators and other digital hardware designs. It has become common for hardware designers to utilize commercial libraries (known as Intellectual Property or IP catalogs) to get hold of the most efficient or performant hardware components. At the same time, open-source hardware has begun to emerge as a viable development strategy, drawing parallels to open-source software, due to the commercial benefits of exploiting free and open components. This new development paradigm raises questions of how hardware developers can best compose their components and treat their underlying implementations as opaque.

Modern high-level programming languages have many mechanisms that work in support of effective modularity and abstraction; for example, one might place requirements on data (e.g. arguments) at an interface (e.g. function call) through a type system. Most hardware description languages (HDLs), in contrast, have comparatively little support for these features. The interface of the primary unit of abstraction, a module, is typically described simply as "wires" which, in turn, may be refined as "input" or "output." However, we find experimentally across hundreds of designs that these

interfaces actually carry surprisingly complex requirements not just on how the data are to be used or interpreted but even on what compositions leads to well-defined digital designs. The goal of our work is to turn a programming language eye to this problem: to be mathematically precise in the definition of wired interfaces and ultimately give more support to hardware designers seeking modularity, abstraction, and better compositional guarantees at the HDL level.

We wish to support a scenario where (1) separate hardware designers can independently create a set of hardware modules according to some connection protocol using an HDL, and the HDL can *automatically* infer relevant properties about the input and output wires for each module in isolation; (2) a hardware designer can treat these modules as opaque components without knowledge of their internals, wiring them together into a circuit such that the HDL provide guarantees based on the properties of the modules' input and output wires; and (3) the number of design "surprises" discovered late in the development cycle due to intermodular incompatibilities is significantly reduced.

Such a scenario is increasingly not just desirable but strictly necessary. In the traditional design methodology where a whole chip may be designed by a single company or team who can agree on interfaces in advance and readily inspect modules' internals, establishing modules' compositionality was straightforward. However, this is a much stickier problem in a world of IP-driven design where the user of a module may have no knowledge of the module's internals, perhaps even working with an obfuscated or encrypted design [12]. IP catalog designers today lack any clear specification of the module-level connection properties needed to ensure well-composed designs. Thus, it is incredibly easy to create a design that assumes something about an up- or downstream interface which only becomes apparent after the *full design* has been completed at the RTL level. Discovering such an issue late in the process can be a serious issue because the exact cycle a data value is produced might need to change to accommodate a different interface. While this sounds easy in theory, traditional RTL design practices are fragile to timing changes, and fixing problems might mean significant surgery to control state machines, the recoordination of multiple producers or consumers, or even failure to meet a latency goal. As we ask a broader set of engineers to engage in the hardware design process, whether to understand tradeoffs in an AI accelerator design or deploy computation into an FPGA in the cloud, we need languages that help steer effort towards realizable designs and reduce the number of "surprises" (i.e. failures) typically only found at the very last stages of implementation (at synthesis time).

The specific property that we focus on in this work is what we are calling *well-connectedness*; we formally specify the property in Section 3.4 but informally, it implies that the final circuit does not contain any combinational loops.[1] Combinational loops are a sign of a broken design (except in certain rare circumstances) and must be avoided. Such loops are easy to spot once all components have been fully implemented and then synthesized into a netlist (one need only look for cycles in the netlist graph) but are hidden through the *entire* process of design at the HDL level, especially when they cross module boundaries and require reasoning about multiple modules' internal structures. This is a real problem we have encountered in our experiences writing digital hardware designs, motivating us to find better ways via programming language abstraction and enforcement.

This problem of avoiding combinational loops at the HDL module level is surprisingly subtle, requiring that designers reason about a number of non-obvious corner cases. Well-connectedness cannot always be guaranteed by looking at pair-wise module interconnections but is in fact a property of the entire circuit requiring information about all modules at once. Nevertheless, we show that it is possible to annotate module interfaces at the HDL level for each module independently such that the well-connectedness of a given combination of modules can be automatically proven by only looking at these interface annotations. We further show that if a full implementation of the design is already available, such as for legacy code, we can *automatically* infer annotations directly from the design. These annotations in turn radically lessen the number of interfaces where "surprises" might occur, allowing designers to focus their attention more effectively. The specific contributions of this paper are:

- We are the first to apply a modular static analysis to the problem of ensuring the correct compositionality of hardware modules in arbitrary RTL, via a global property which we define as *well-connectedness*.
- We prove this property is achievable in a *modular* way via a mathematical specification of wire dependencies, developing a novel taxonomy of sorts: **to-sync**, **to-port**, **from-sync**, and **from-port**.
- We embody these properties, and the analysis they enable, in a usable and scalable tool that completely prevents the late discovery of combinational loops. We further propose an extension to the analysis to protect synchronous memory semantics through composition.
- We analyze more than 500 parameterized hardware modules to quantify, for the first time, the diversity of expectations placed on module interfaces found in the wild. Across three independent projects (BaseJump STL, OpenPiton, and a RISC-V implementation) our analysis is able to automatically infer the correct wire sorts to enable composability in less than 31 seconds.

[1] This is a necessary but not sufficient condition for overall correctness. For example, we are not concerned in this paper about checking that a specific protocol is being correctly followed. Our techniques could potentially also reason about properties related to timing and circuit layout, but we leave these for future work.

Our analysis is 2.6–33.9x faster at finding intermodular loops than standard cycle detection during synthesis.

2 Motivation and Related Work

To demonstrate the problem, we use the example of a simple first-in first-out (FIFO) queue using the ready-valid protocol, as shown in Figure 1. The role of the FIFO queue is to accept input data from one module (at the consumer endpoint), buffer that data inside its internal state, and then send the data to another module (at the producer endpoint) upon request. The consumer endpoint consists of a set of wires: $data_{\text{in}}$ contains the data being sent to the FIFO; $valid_{\text{in}}$ determines whether the incoming signals on $data_{\text{in}}$ represent valid input from the connected module; and $ready_{\text{out}}$ is an outgoing signal indicating whether the FIFO is ready to accept input (i.e., it isn't full). Similarly, the producer endpoint consists of another set of wires: $data_{\text{out}}$ contains the data being produced by the FIFO and read from another connected module; $valid_{\text{out}}$ determines whether the outgoing signals on $data_{\text{out}}$ represent valid data from the FIFO (i.e., it isn't empty); and $ready_{\text{in}}$ is an incoming signal indicating whether the connected module is ready to receive data from this FIFO.

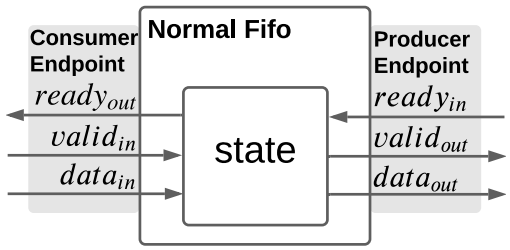


Figure 1. Normal FIFO Queue. The consumer endpoint receives data from one module, and the producer endpoint sends data to another module.

We have left the internals of the FIFO opaque (as they may realistically be to a user); the details do not matter for our purposes except to note that each FIFO endpoint is combinationally independent of the other. In other words, every path between the endpoints is interrupted by some state inside the FIFO, so that an action at one endpoint cannot affect the other endpoint within a single cycle.

A FIFO queue of this kind is often called a "universal interface" because it can be placed between any two modules without danger of ill effects due to timing issues. However, for various reasons (such as efficiency) a normal FIFO queue may not be appropriate. A *forwarding* FIFO improves efficiency by allowing data entering in one clock cycle to be immediately sent out in the same clock cycle if the FIFO is empty. An abstract depiction of this module is shown in Figure 2.

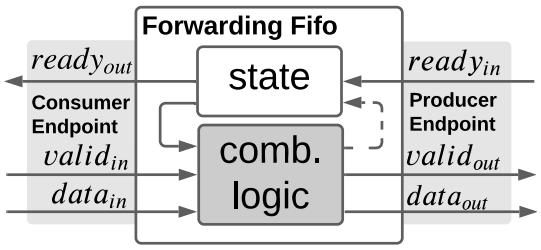


Figure 2. Forwarding FIFO Queue.

The important points for our purposes are that: (1) the module interface (i.e., the ready-valid endpoint specification) is unchanged from the normal FIFO, so that from a module connection standpoint the two are indistinguishable; and (2) the endpoints are no longer combinationally independent because there is a combinational path from one endpoint to the other, enabling the data forwarding that is the whole point of the new module. Here's a closer look at the combinational logic used for assigning to $valid_{\text{out}}$ across the two FIFO modules (where $count_{\text{reg}}$ is a register containing the number of enqueued elements):

- **Normal:**
  $valid_{\text{out}} ::= (count_{\text{reg}} > 0)$
- **Forwarding:**
  $valid_{\text{out}} ::= (count_{\text{reg}} > 0) \vee (valid_{\text{in}} \wedge ready_{\text{out}})$

This combinational dependence between the endpoints means that designers may inadvertently cause a combinational loop when they wire modules together. In fact, the problem may not even arise due to direct interactions between the queue and the modules connected to its endpoints, but rather due to indirect interactions mediated by yet other modules. We show an example of a problematic circuit in Figure 3.

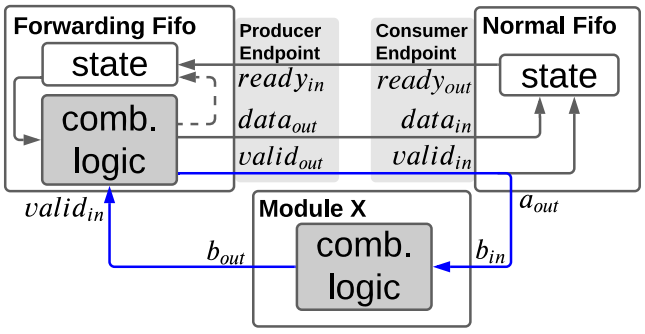


Figure 3. Forwarding FIFO connected to other modules causing a combinational loop (in blue). Only pertinent IO ports have been shown for each module.

Here we have three modules: a normal FIFO, a forwarding FIFO, and some module X. In this contrived example, the normal FIFO sends a signal to module X that is some combinational function of its $valid_{\texttt{in}}$ wire (here, a direct connection); module X sends some combinational function of its input to the forwarding FIFO's $valid_{\texttt{in}}$, which (as previously discussed) is a combinational input to the forwarding FIFO's $valid_{\texttt{out}}$, which in turn is wired to the normal FIFO's $valid_{\texttt{in}}$. If the forwarding FIFO were instead a normal FIFO (which at a module connection level looks the same) then this would be fine, but since it is not this circuit contains a combinational loop. Detecting and understanding the cause of the loop requires reasoning about the internal details of three different modules.

We note that detecting the existence of the combinational loop is simple once the HDL program has been synthesized to a netlist: simply perform a standard cycle detection algorithm. Verilog [40] synthesis tools such as the linter in Verilator [32] and the Yosys synthesis suite [42] and HDLs such as Chisel [2] and PyRTL [15, 17] can provide warnings about loops during synthesis. However, relying on loop detection at synthesis time has several drawbacks. First, gate-level netlists take a long time to produce and are significantly larger (47X larger in one example we studied), since high-level and multi-bit operations have been transformed into sets of simple 1-bit primitive gates. Second, these detection systems aren't infallible: under certain combinations of flags or optimizations, tools like Yosys fail to detect loops or silently delete them, "successfully" synthesizing the offending circuit. Third, once the loop is detected after synthesis, it is entirely up to the designer to trace the synthesized loop back to the relevant modules and interactions in the original HDL program.

The RTL, on the other hand, has fewer gate dependencies to analyze while still representing the same dataflow graph. Going up one level of abstraction, the behavioral level describes the same system algorithmically, making it even easier to take advantage of high-level constructs for determining dependency. Thus, our goal is to raise the level of abstraction for detecting loops up to the HDL module level in order to give the designer maximum information and context, to avoid loops more easily, and to detect loops sooner in the design process. An apt analogy is the difficulty in trying to determine the cause of an assembly-level link time error versus one presented at the source level; we aim to do the latter for HDLs.

There do exist HDL-level tools to check certain kinds of properties, for example SystemVerilog Assertions (SVA) [25], Property Specification Language (PSL)/Sugar [20, 24], and Open Verification Library (OVL) [23]. These frameworks facilitate the specification of temporal relationships between wires, which are checked via simulation or model checking rather than statically at design time. These tools can express properties about the relative order in which things occur but not the reasons why they occur. Since our analysis is concerned with the exact causes of events (i.e., combinational dependencies between wires), we believe from our experience using these tools that they are not suitable for our purpose.

There is additionally a long history of using higher-level abstractions to describe hardware formally [14, 31] and of using richer type systems [21] and functional programming techniques [5, 22, 28, 30]. DSLs like Murφ [18] and Dahlia [29] target specific use cases like protocol descriptions or improved accelerator design, while high-level synthesis (HLS) techniques [7, 16] translate subsets of C/C++ to RTL. Other HDLs [38] like PyMTL [26], Cλash [1], Pi-Ware [19], HardCaml [33], BlueSpec [6], and Kami [13] also use modern programming language techniques to overcome some of the issues that arise when writing in traditional HDLs [34, 35]; like many of them, we focus on improving the register-transfer level design process by creating better and more expressive abstractions.

2.1 BaseJump STL

The closest work to our own is BaseJump STL [37, 43]. Their work discusses the requirements for creating a library of hardware modules (analogous, in their words, to the C++ standard template library) and introduces some informal terminology to help describe module interfaces and promote properties such as well-connectedness. They draw upon the principles of latency-insensitive hardware design [8–11] but aim for a less restrictive model.

BaseJump STL informally defines the notions of **helpful** and **demanding** module interface endpoints (such as the ready-valid endpoints from the previous FIFO example). The distinction is based on whether an endpoint is able to offer up data without "waiting" for input. For the ready-valid protocol, a **helpful** producer offers $valid_{\texttt{out}}$ upfront while a **demanding** producer waits for $ready_{\texttt{in}}$ before computing and emitting $valid_{\texttt{out}}$. Similarly, a **helpful** consumer offers $ready_{\texttt{out}}$ upfront while a **demanding** consumer waits for $valid_{\texttt{in}}$ before computing and emitting $ready_{\texttt{out}}$. BaseJump STL creates a taxonomy of interface connections based on the various combinations of **helpful** and **demanding** endpoints. They note that the only unsafe combination is a **demanding-demanding** connection, which would directly lead to a combinational loop.

The problem with BaseJump STL's approach is that it considers module endpoint connections in isolation: the notion of dependence inherent in the **demanding** and **helpful** classifications only considers wires that directly participate in the connection. However, this isn't sufficient to guarantee detection of combinational loops, as we have shown with our previous example of a problematic circuit in Figure 3. In that example, the forwarding FIFO's producer endpoint is considered **helpful** because $valid_{\texttt{out}}$ is offered without needing to wait on $ready_{\texttt{in}}$. The normal FIFO's consumer endpoint is

considered **helpful** because $ready_{\texttt{out}}$ doesn't wait on $valid_{\texttt{in}}$. According to BaseJump STL's model, these modules have a **helpful-helpful** connection and are therefore safe. But as we have demonstrated, the design is actually faulty due to the third module in the circuit and how it interacts with the connection between the forwarding and normal FIFOs.

*We discovered the issues with BaseJump STL's notions of **helpful** and **demanding** endpoints when we attempted to formalize them and prove that they were adequate to detect combinational loops at the HDL module level of abstraction.* Our experience led us to conclude that in order to guarantee well-connectedness, we need to: (1) be able to reason about module endpoints based on wire dependencies between the input and output wires within a module; and (2) using only the resulting endpoint annotations, reason about an entire circuit at the module level to resolve possible loops introduced by interactions between multiple modules.

3 Wire Sorts and Well-Connectedness

In this section we define our notion of wire sorts, formalize the property of well-connectedness using these sorts, and prove a set of properties that can be used to demonstrate that a circuit composed of independently designed modules is well-connected. Finally, we show exactly how our definitions contrast to BaseJump STL's notions of **helpful** and **demanding** endpoints and how our approach avoids the problems that BaseJump STL encounters.

3.1 Defining Basic Domains

We formally define a set of basic domains that collectively comprise a circuit composed of independent modules, so that we can precisely define wire sorts and well-connectedness and prove that a well-connected circuit has no combinational loops. Our formalisms and techniques apply to synchronous digital designs, and we assume for simplicity that there is a single clock driving all stateful elements (both are properties of the most commonly found designs).

A *wire* is denoted by w_σ where $\sigma \in \{\texttt{const}, \texttt{reg}, \texttt{in}, \texttt{out}, \texttt{basic}\}$. A constant wire $w_{\texttt{const}}$ produces a 0 or 1, an input wire $w_{\texttt{in}}$ serves as input into a module, and an output wire $w_{\texttt{out}}$ serves as output from a module. Registers are stateful elements that are latched each cycle according to the same shared clock; the $w_{\texttt{reg}}$ wires represent the outputs of these registers. Basic wires $w_{\texttt{basic}}$ are used to connect or combine these wires together via nets. A *net* is a tuple $(\overrightarrow{w_\sigma}, w_\sigma, op)$ representing a gate, with multiple wires $\overrightarrow{w_\sigma}$ coming into the gate, a single wire w_σ coming out of the gate, and a bitwise logical operation op denoting the type of gate such that $w_\sigma = op(\overrightarrow{w_\sigma})$.

A *module* M is a tuple $(\overrightarrow{w_{\texttt{in}}}, \overrightarrow{w_{\texttt{out}}}, \overrightarrow{net})$ composed of sets of input wires, output wires, and nets representing a directed acyclic graph (DAG); in this DAG, the nets are nodes, and the outputs of the nets are the forward edges in the graph.

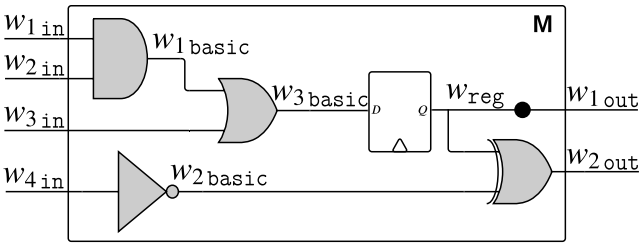


Figure 4. Example for computing the output-port-set and input-port-set of a module M. The output-port-set of input $w_{4\texttt{in}}$ is $\{w_{2\texttt{out}}\}$ and $\emptyset$ for the other inputs. The input-port-set of $w_{2\texttt{out}}$ is $\{w_{4\texttt{in}}\}$ and $\emptyset$ for $w_{1\texttt{out}}$.

The input and output wires form the module's external interface. Given a module $M = (\overrightarrow{w_{\texttt{in}}}, \overrightarrow{w_{\texttt{out}}}, \overrightarrow{net})$ we will use the shorthand $M.inputs$, $M.outputs$, and $M.nets$ to mean $\overrightarrow{w_{\texttt{in}}}$, $\overrightarrow{w_{\texttt{out}}}$, and $\overrightarrow{net}$, respectively.

A *circuit* C is a tuple $(\overrightarrow{M}, \overrightarrow{(w_{\texttt{out}}, w_{\texttt{in}})})$ composed of a set of modules $C.modules$ and the connections $C.conns$ between their inputs and outputs. Given $M_1, M_2 \in C.modules$ and two wires $w_{\texttt{out}} \in M_1.outputs$ and $w_{\texttt{in}} \in M_2.inputs$, we use $w_{\texttt{out}} \rightarrow_C w_{\texttt{in}}$ to mean that $w_{\texttt{out}}$ is directly connected to $w_{\texttt{in}}$, i.e., $(w_{\texttt{out}}, w_{\texttt{in}}) \in C.conns$. We define the function $\texttt{module}(w_{\texttt{in}}, C) = M$ iff $w_{\texttt{in}} \in M.inputs \wedge M \in C.modules$. Without loss of expressiveness, we assume that one module's outputs are always connected directly to another module's inputs.[2] Note that a circuit C and its set of modules $C.modules$ can essentially define a larger module composed of *submodules*. A circuit composed of many of these "supermodules" connected together in turn makes an even larger module, ad infinitum. Thus the intra- and intermodular analyses we discuss in the following sections are fully generalizable to the notion of submodules common in popular HDLs.

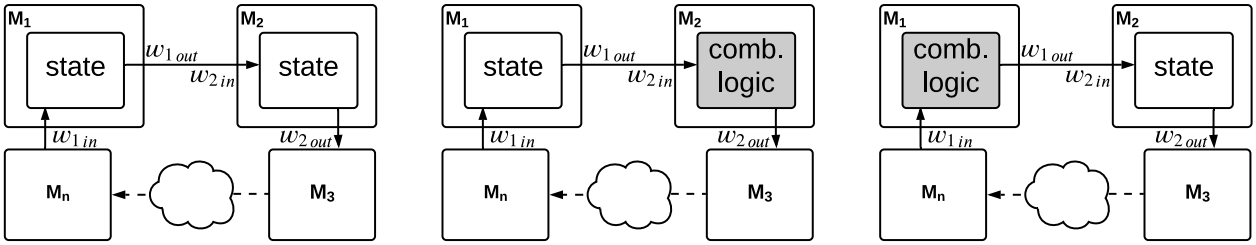
3.2 Defining Combinational Reachability

We define two different levels of combinational reachability: one intra-modular that can be computed for each module independently and one inter-modular that involves the entire circuit.

Given a module M containing a wire w_σ, we define the *combinationally reachable set* $\texttt{reachable}(M, w_\sigma)$ as the set of wires reachable from w_σ in $M.nets$ without going through any wire $w_{\texttt{reg}}$; in other words, the transitively reachable wires that don't go through any registers (state).

We can now define two terms that will be important for determining combinational reachability at the module level without needing the internal details of the relevant modules: **output-port-set** and **input-port-set**. The output-port-set is relevant for module inputs: given module M and input $w_{\texttt{in}}$, the output-port-set $\texttt{output-ports}(M, w_{\texttt{in}})$ is the set of

[2] If there is any extra-modular logic between modules, one can wrap that logic into its own module to trivially meet this condition.



(a) A **from-sync** wire $w_{1\texttt{out}}$ connected to a **to-sync** wire $w_{2\texttt{in}}$.

(b) A **from-sync** wire $w_{1\texttt{out}}$ connected to a **to-port** wire $w_{2\texttt{in}}$.

(c) A **from-port** wire $w_{1\texttt{out}}$ connected to a **to-sync** wire $w_{2\texttt{in}}$.

Figure 5. Connections between **to-sync** or **from-sync** wires cannot result in combinational loops.

module output wires that are combinationally reachable from that input wire. In other words, $\texttt{output-ports}(M, w_{\texttt{in}}) = \texttt{reachable}(M, w_{\texttt{in}}) \cap M.outputs$. Similarly, the input-port-set is relevant for module outputs: for an output wire $w_{\texttt{out}}$ of module M, the input-port-set $\texttt{input-ports}(M, w_{\texttt{out}})$ is the set of module input wires that combinationally reach that output wire. In other words, $\texttt{input-ports}(M, w_{\texttt{out}}) = \{w_{\texttt{in}} \mid w_{\texttt{in}} \in M.inputs, w_{\texttt{out}} \in \texttt{output-ports}(M, w_{\texttt{in}})\}$. These sets need only be computed once per module definition (regardless of how many instantiations are used in a circuit).

To illustrate these definitions consider the module diagram in Figure 4. In this module, which we'll call M, the output-port-set of input $w_{4\texttt{in}}$ is $\texttt{output-ports}(M, w_{4\texttt{in}}) = \{w_{2\texttt{out}}\}$, while the output-port-set of each of the inputs $w_{1\texttt{in}}, w_{2\texttt{in}}, w_{3\texttt{in}}$ is $\emptyset$. The input-port-set of output $w_{1\texttt{out}}$ is $\emptyset$, while the input-port-set of $w_{2\texttt{out}}$ is $\texttt{input-ports}(M, w_{2\texttt{out}}) = \{w_{4\texttt{in}}\}$.

Given a circuit composed of multiple modules along with the output-port-set and input-port-set for each input and output wire of each module, we can compute inter-modular combinational loops without needing to inspect the internals of any module. The transitive forward reachability of any output wire amounts to a fixpoint computation involving the output-port-sets of the modules in the circuit; while tracing a path from between wires, if a module input wire is encountered, skip over its module's internal logic by continuing with the output wires in its output-port-set. We use $w_1 \rightsquigarrow_C w_2$ to denote that wire w_1 transitively affects wire w_2 in circuit C and call $\rightsquigarrow_C$ the *TransitivelyAffects* relation.

3.3 Wire Sorts

We can now formally define the novel set of sorts for module input and output wires, a key contribution of this paper. An input wire $w_{\texttt{in}}$ is **to-sync** if $\texttt{output-ports}(M, w_{\texttt{in}}) = \emptyset$ and is **to-port** otherwise. An output wire $w_{\texttt{out}}$ is **from-sync** if $\texttt{input-ports}(M, w_{\texttt{out}}) = \emptyset$ and is **from-port** otherwise. The **to-sync**, **to-port**, **from-sync**, or **from-port** designation of a wire is its **sort**, and this set of sorts is sufficient to label all module ports. In Figure 4, the sort of input wires $w_{1\texttt{in}}, w_{2\texttt{in}}, w_{3\texttt{in}}$ is **to-sync** while the sort of $w_{4\texttt{in}}$ is **to-port**. Of the outputs, the sort of $w_{1\texttt{out}}$ is **from-sync** while the sort of $w_{2\texttt{out}}$ is **from-port**.

Note that an input wire of sort **to-sync** cannot be involved in a combinational loop, nor can an output wire of sort **from-sync**. By definition, these wires terminate or originate in some stateful or constant-valued element, and therefore module interface wires of these sorts can be freely connected to other modules safely without regard to the connected module's interface wire sorts or the rest of the circuit. This leads us to our first property.

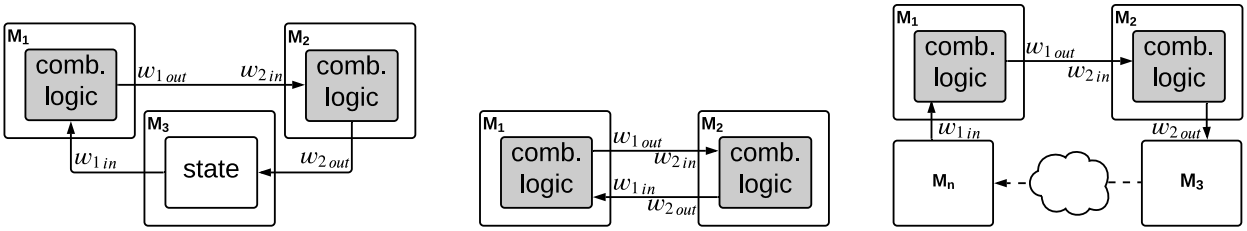
Property 1. *Two connected wires $w_{\texttt{out}}$ and $w_{\texttt{in}}$ cannot be involved in a combinational loop if $w_{\texttt{out}}$ is **from-sync** or $w_{\texttt{in}}$ is **to-sync**.*

Proof. Given a module M_1 such that $w_{\texttt{out}} \in M_1.outputs$, if $w_{\texttt{out}}$ is **from-sync**, then $\texttt{input-ports}(M_1, w_{\texttt{out}}) = \emptyset$, meaning it does not combinationally depend on any module input. Similarly, given a module M_2 such that $w_{\texttt{in}} \in M_2.inputs$, if $w_{\texttt{in}}$ is **to-sync**, then $\texttt{output-ports}(M_2, w_{\texttt{in}}) = \emptyset$, meaning it does not combinationally affect any module output. □

In Figure 5a, **from-sync** wire $w_{1\texttt{out}}$ is connected to **to-sync** wire $w_{2\texttt{in}}$, while in Figure 5b, it is connected to **to-port** wire $w_{2\texttt{in}}$. We can see that it doesn't matter what sort of input $w_{1\texttt{out}}$ connects to, since there is at least one stateful element shielding $w_{1\texttt{out}}$ from being fed into itself combinationally: the stateful elements of M_1 in both figures and additionally the stateful elements of M_2 in Figure 5a. In both cases, it doesn't matter what modules $M_3 \ldots M_n$ may do or any other output M_1 may have that could possibly feed into them. Similarly, in Figure 5c, because **from-port** wire $w_{1\texttt{out}}$ is connected to **to-sync** wire $w_{2\texttt{in}}$, we can know even without analyzing the entire circuit that this particular connection is safe.

3.4 Defining Well-Connectedness

There are cases, like our previous example of a forwarding FIFO queue in Section 2, where it doesn't make sense to require that module interface wires be only **to-sync** or **from-sync**. Relaxing this requirement means we cannot rely solely on Property 1 for establishing safety between wires, and so



(a) A **from-port** wire $w_{1\mathtt{out}}$ connected to a **to-port** wire $w_{2\mathtt{in}}$, which through interactions with other modules can be shown to not result in a combinational loop.

(b) A **from-port** wire $w_{1\mathtt{out}}$ connected to a **to-port** wire $w_{2\mathtt{in}}$ resulting in a combinational loop.

(c) A **from-port** wire $w_{1\mathtt{out}}$ connected to a **to-port** wire $w_{2\mathtt{in}}$, which cannot be determined to be well-connected until the entire circuit has been completed.

Figure 6. Connections between **from-port** or **to-port** wires might result in combinational loops.

we must more precisely define our notion of inter-wire safety as follows:

Definition 3.1 (Wire Well-Connectedness)**.** Given a circuit C and two modules $M_1, M_2 \in C.modules$ (where M_1 may be M_2), an output wire $w_{\mathtt{out}} \in M_1.outputs$, and an input wire $w_{\mathtt{in}} \in M_2.inputs$ such that $w_{\mathtt{out}} \rightarrow_C w_{\mathtt{in}}$, $w_{\mathtt{out}}$ is **well-connected** to $w_{\mathtt{in}}$ iff $\forall w_1 \in \mathtt{input\text{-}ports}(M_1, w_{\mathtt{out}}), \forall w_2 \in \mathtt{output\text{-}ports}(M_2, w_{\mathtt{in}})$, $w_2 \not\rightsquigarrow_C w_1$.

It is straightforward to show that it satisfies our desired safety property:

Property 2. *The connection between two wires $w_{\mathtt{out}}$ and $w_{\mathtt{in}}$ that are well-connected to one another does not introduce a combinational loop.*

Proof. By definition, all of the input wires w_1 in M_1 that combinationally affect $w_{\mathtt{out}}$ are present in its input-port-set. Likewise, by definition, all of the output wires w_2 in M_2 that are combinationally affected by $w_{\mathtt{in}}$ are in its output-port-set. If it is impossible to transitively trace any output wire w_2 through the nets it combinationally affects to any input wire w_1 that $w_{\mathtt{out}}$ is awaiting, then no combinational loop has been introduced by $w_{\mathtt{out}} \rightarrow w_{\mathtt{in}}$. □

We illustrate this property in Figure 7 below.

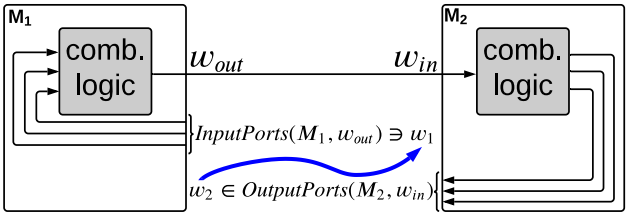


Figure 7. Illustration of the Wire Well-Connectedness definition. Given a circuit C, well-connectedness for a connection $(w_{\mathtt{out}}, w_{\mathtt{in}}) \in C.conns$ occurs when there does not exist an output port w_2 in $w_{\mathtt{in}}$'s output-port-set that is transitively connected ($\rightsquigarrow_C$) to any wire w_1 in $w_{\mathtt{out}}$'s input-port-set.

Any wires of sort **to-port** or **from-port** are potential problems, so we cannot in general determine safety without inspecting the entire circuit. For example, Figure 6a and Figure 6b both show two connected modules with a **from-port** output wire connected to a **to-port** input wire, but in the former case it does not result in a combinational loop while in the latter it does. Note, however, that we still do not need to inspect the internals of any modules as long as we know the sorts of their interface wires.

We can distinguish between the examples in Figure 6a and Figure 6b by defining a safe class of connections to **from-port** sorts, called **safely from-port**:

Definition 3.2 (Safely From-Port Wires)**.** A **from-port** output wire $w_{\mathtt{out}}$ connected to a **to-port** input wire $w_{\mathtt{in}}$ is called **safely from-port** with respect to $w_{\mathtt{in}}$ if $w_{\mathtt{out}}$ and $w_{\mathtt{in}}$ are well-connected according to Definition 3.1.

A **safely from-port** output wire combinationally depends on some module input wires (and hence its value is not valid until those inputs have propagated to the output wire later in the clock cycle) but still guarantees the absence of combinational loops with respect to certain connected wires. In Figure 6a, the dependent output wire $w_{1\mathtt{out}}$ is **safely from-port** with respect to $w_{2\mathtt{in}}$, and hence the overall circuit is well-connected since $w_{1\mathtt{out}}$ is not connected to anything else. In contrast, in Figure 6b the **from-port** output wire $w_{1\mathtt{out}}$ is *not* **safely from-port** and hence the overall circuit is *not* well-connected.

Determining whether a wire is **safely from-port** or not requires the complete circuit in order to compute the *TransitivelyAffects* relation. Figure 6c demonstrates this fact. We define a circuit composed of a set of modules such that all module interface wires are connected to be a **complete circuit**. A **well-connected circuit** is a complete circuit that has no combinational loops. This definition brings us to our final property:

Property 3 (Circuit Well-Connectedness)**.** *A complete circuit is well-connected if and only if all **from-port** output wires*

*in the circuit are **safely from-port** with respect to the **to-port** input wires to which they are connected.*

Proof. The forward implication is that in a complete, well-connected circuit C, all **from-port** output wires are **safely from-port**. By definition, a well-connected circuit does not contain any combinational loops. If there exists some module M_1's **from-port** output wire $w_{\texttt{out}}$ that is *not* **safely from-port**, then by the definition of **safely from-port** (Definition 3.2) either:

1. Wire $w_{\texttt{out}}$ is not connected to any other wire. But this contradicts the fact that the circuit must be complete.
2. Wire $w_{\texttt{out}}$ is connected to wire $w_{\texttt{in}}$ of some module M_2 and there exist wires $w_1 \in \texttt{input-ports}(M_1, w_{\texttt{out}})$, $w_2 \in \texttt{output-ports}(M_2, w_{\texttt{in}})$ such that $w_2 \rightsquigarrow_C w_1$. By the definition of $\rightsquigarrow_C$ this means that there is a combinational loop in the circuit. But this contradicts that the circuit is well-connected.

Therefore by contradiction the forward implication holds. The reverse implication is that if all **from-port** output wires are **safely from-port**, then the complete circuit is well-connected. Since the circuit is complete, every input and output wire is connected to some output or input wire, respectively. For a given connection, if either the output wire is **from-sync** or the input wire is **to-sync** then they cannot be part of a combinational loop. So the only case that we need to worry about is if the output wire $w_{\texttt{out}}$ is **from-port** and the input wire $w_{\texttt{in}}$ is **to-port**. Assuming that $w_{\texttt{out}}$ is **safely from-port**, this means that by Definition 3.2 it must be true that $w_{\texttt{out}}$ and $w_{\texttt{in}}$ are well-connected according to Definition 3.1. This property directly implies that these wires cannot be part of a combinational loop. Therefore the forward direction holds. □

3.5 Putting It All Together

Given the definitions and properties stated above, we can divide checking a circuit for well-connectedness into three stages:

- **Stage 1.** At the time each module is designed, automatically compute the sort of each input and output wire. Annotate each wire with its sort and, for a **from-port** or **to-port** wire, its input-port-set and output-port-set, respectively.
- **Stage 2.** When modules are connected during circuit design, any connections involving a **from-sync** or **to-sync** wire can be marked as safe immediately.
- **Stage 3.** Either periodically during circuit construction (useful when using interactive HDLs with a tight feedback loop) or only once when the circuit is completed: for each **from-port** output wire connected to a **to-port** input wire, check whether the output wire is **safely from-port** with respect to the input wire.

This process neatly encapsulates the necessary information about the module's internal details into its interface and allows for checking well-connectedness in the final circuit while treating each module as a opaque.

3.6 Comparison to BaseJump STL

We can relate the informal notions given by BaseJump STL (described in Section 2) to our more precise definitions given here and thereby pin down exactly where the BaseJump STL notions become problematic. BaseJump STL says that an endpoint is **demanding** if it needs the other endpoint's input signal ($valid_{\texttt{in}}$ for the consumer endpoint, $ready_{\texttt{in}}$ for the producer endpoint) before computing its own output signal ($ready_{\texttt{out}}$ for the consumer endpoint, $valid_{\texttt{out}}$ for the producer endpoint) and is **helpful** otherwise.

Using our definitions, we can formulate these notions precisely. We are given a module M with producer endpoint ($ready_{\texttt{in}}$, $valid_{\texttt{out}}$, $data_{\texttt{out}}$) and consumer endpoint ($ready_{\texttt{out}}$, $valid_{\texttt{in}}$, $data_{\texttt{in}}$). The producer endpoint is **helpful** iff $ready_{\texttt{in}} \notin \texttt{input-ports}(M, valid_{\texttt{out}})$, otherwise it is **demanding**. This says nothing about the presence or absence of M's other inputs in $\texttt{input-ports}(M, valid_{\texttt{out}})$, meaning $valid_{\texttt{out}}$ could be **from-port** and thus potentially cause a loop due to other module connections. The consumer endpoint is **helpful** iff $valid_{\texttt{in}} \notin \texttt{input-ports}(M, ready_{\texttt{out}})$, otherwise it is **demanding**; again, this does not preclude $ready_{\texttt{out}}$ from being **from-port**.

According to the BaseJump STL work, the only potentially problematic connection is between two **demanding** endpoints; all other types of connections are safe while **demanding-demanding** connections should be forbidden. However, according to our analysis above this is not a sufficient condition for correctness. It is possible (as demonstrated in Section 2) for a **helpful-helpful** connection to create a combinational loop; this is because the **helpful** and **demanding** endpoint classifications focus only on direct connections between two modules and do not consider the possibility of combinational loops via other modules not directly involved in the connection.

3.7 Extension to Synchronous Memory Reads

The basic set of domains described in Section 3.1 omits mention of memories. Memories are a special case in digital logic; their semantics partially depend upon whether they are synchronous or asynchronous. Synchronous memories are often preferable in order to be able to synthesize a design into efficient hardware, but using them imposes additional conditions on the design. For example, one class of synchronous memories requires that the read operations are able to start at the *beginning of the clock cycle*. What this often means is that the designer must make sure that the read address port, $raddr_{\texttt{in}}$, is fed directly from a register.

Take as an example the module-memory interconnection in Figure 8. At first glance, this condition requires that any external module's output wire $w_{\texttt{out}}$ connected to $raddr_{\texttt{in}}$ be **from-sync**. However, this still doesn't meet the required

conditions for synchronous memories; our definition of **from-sync** allows combinational logic to exist between the source register from which the **from-sync** data originates and its destination. In order for the data on $w_{\texttt{out}}$ to be available immediately at the beginning of the clock cycle, it must not go through any combinational logic at all (since all gates have propagation delay), and so we find that we must create a **from-sync** subsort, which we'll call **from-sync-direct**.

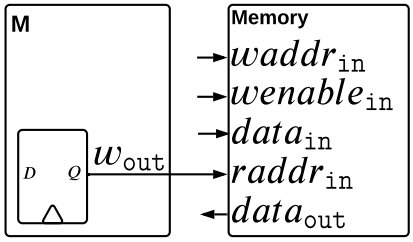


Figure 8. The read address line $raddr_{\texttt{in}}$ of certain synchronous memories must be connected to **from-sync-direct** wires like $w_{\texttt{out}}$.

A **from-sync-direct** output wire $w_{\texttt{out}}$ is simply one where $\texttt{reachable}(M, w_{\texttt{out}}) = \emptyset$. By our definition of `reachable` in Section 3.2, this means that $w_{\texttt{out}}$ is connected only directly to registers, with no intermediate combinational logic. In Figure 4, wire $w_{1\texttt{out}}$ could thus be labelled **from-sync-direct** and qualify as being able to be connected to a synchronous memory's input wires. Its data is available at the start of the clock cycle because its signal doesn't need to propagate through any attached combinational logic. A **from-sync** wire that isn't **from-sync-direct** is said to be **from-sync-indirect**.

There are other forms of memories where synchronous requirements are placed on certain *outputs*, rather than inputs. In these memories, the designer must ensure that the $data_{\texttt{out}}$ wire is fed directly into a register. This naturally leads to an input subsort for describing such conditions, which we call **to-sync-direct**; a **to-sync-direct** input wire $w_{\texttt{in}}$ is one where $\texttt{reachable}(M, w_{\texttt{in}}) = \emptyset$. A **to-sync** wire that isn't **to-sync-direct** is said to be **to-sync-indirect**.

By providing these additional sorts, designers can communicate the interface requirements of modules using synchronous memories, making libraries of hardware components more accessible and easier to use. Thus, this sort taxonomy, now at for inputs: **to-sync** (with its subsorts **to-sync-direct** and **to-sync-indirect**) and **to-port**; and for outputs: **from-sync** (with its subsorts **from-sync-direct** and **from-sync-indirect**) and **from-port**, has a wide range of applications and can be potentially expanded even further.

4 Implementation of Modular Well-Connectedness Checks

We augmented the PyRTL HDL [15] to implement lightweight annotations and design-time checks according to the formal properties that we have described of the original four wire sorts (**to-sync**, **to-port**, **from-sync**, and **from-port**). PyRTL does not natively support a module abstraction, so we first modified the language by adding a `Module` class that isolates a modular design and exposes an interface consisting of input and output wires.[3]

Our formalism made two simplifying assumptions. First, it assumed that all logic is contained inside modules. For developer convenience, we eased this restriction to allow for arbitrary logic to exist between modules. We tweaked the *TransitivelyAffects* relationship ($\rightsquigarrow_C$) to account for combinational paths through this extra-modular logic. Second, it treated all wires as one bit in width. At the HDL level, it is much more convenient to group related one-bit wires, especially input and output ports, into single n-bit wire vectors. For native PyRTL designs (but not BLIF import), the output-port-set or input-port-set of each port wire vector becomes the union of the output-port-set or input-port-set of its constituent wires; thus we are overly conservative because single-bit dependencies are not tracked, but maintain soundness by continuing to be able to detect all combinational loops.

The well-connectedness implementation itself consists of (1) a sort inferencer that automatically computes the sorts of a module's input and output wires at module design time; (2) lightweight syntactic annotations that allow a designer to (optionally) specify what they believe the sorts should actually be; and (3) a whole-circuit checker that automatically triggers when needed to verify that a circuit composed of multiple modules is well-connected.

The computed sorts are then checked against any existing designer annotations to ensure that the computed sorts match the designer's expectations; any unascribed ports are labeled with their computed sorts. We require sort ascriptions, where the output-port-set or input-port-set are fully specified by the user, for all the ports of opaque modules, since there is no internal logic to use for calculating these sorts. Once a module has its wire sorts, these sorts make it quicker to determine intermodular connections because they facilitate re-use: every instantiation of the same module in the larger design reuses the same wire sort information.

An interesting question during circuit design, as modules are being composed, is *when* exactly to check well-connectedness. We would like to highlight problems as early as possible instead of waiting until the entire circuit is complete. However, we also want to minimize the cost of constantly checking well-connectedness during the design process. As such, our tool can either check for well-connectedness after all modules have been connected or instead whenever a newly formed connection between two modules meets the following condition: the connection's forward combinational

[3] Our PyRTL modifications and the complete implementation of our tool are available at https://github.com/pllab/PyRTL/tree/pldi-2021.

reachability set includes a **to-port** input, and its backward combinational reachability set contains a **from-port** output. This condition is cheap to track by saving and updating information about each wire's reachability as wires are added to the design, and it guarantees that (1) a check is never done unless a problem could potentially be found; and (2) an actual problem is found as soon as possible.

5 Evaluation

We evaluate our tool in five parts: (1) an application to a number of SystemVerilog modules provided by the BaseJump STL; (2) an application to several components from the OpenPiton Design Benchmark; (3) a case study applying the tool to the design of a multithreaded RISC-V CPU; (4) a comparison of our tool to standard cycle detection during synthesis; and (5) a discussion of the scalability and asymptotic complexity of the tool.

5.1 BaseJump STL Modules

We begin with an evaluation of a number of the SystemVerilog modules provided by the industrial-strength BaseJump STL library. This evaluation serves as a baseline sanity check allowing us to verify that we can successfully assign the correct sorts to module interfaces.

We ran our annotation framework successfully on 144 unique modules from the BaseJump STL as found in the BSG Micro Designs repository [36], a repository containing a large number of BaseJump STL modules parameterized and converted into Verilog. Each module was instantiated one to four times to test various combinations of its parameters (e.g. data bit width, address width, queue size, etc.), so that we analyzed 533 modules in total. Since our technique is currently only applicable to synchronous, single-clock designs, we were unable to analyze 5 modules that relied on asynchronous or multi-clock constructs.

We converted each top-level module and their submodules into the flattened BLIF format[4] via Yosys version 0.9 and imported the result into PyRTL. The average size of each module in BLIF was 1.7 MB, with an average number of primitive gates of 19,981, an average number of inputs and outputs per module of 6, and an average time for inferring all of the interface sorts for each module of 361 milliseconds. We ran all of these experiments using PyRTL on a computer with a 1.9 GHz Intel Xeon E5-2420 processor and 32 GB 1333 MT/s DDR3 memory.

A representative subset of these BaseJump STL modules is shown in Table 1: a first-in first-out queue, a parallel-in serial-out shift register, a serial-in parallel-out shift register, and cache DMA. Information on the sizes, wire sort annotations, and annotation time for all 533 modules is found in the supplementary material.

We highlight in particular the parallel-in serial-out shift register (PISO) as an interesting case. Three of its four inputs are **to-sync**, while `yumi_i` is **to-port** (specifically, its output-port-set contains the output `ready_o`). We can see the details by looking at the logic for output `ready_o`:

$$\begin{aligned} ready\_o_{\texttt{out}} ::= {} & (state_{\texttt{reg}} = \texttt{stateRcv}) \vee \\ & ((state_{\texttt{reg}} = \texttt{stateTsmt}) \wedge \\ & (shiftCtr_{\texttt{reg}} = \texttt{nSlots} - 1) \wedge yumi\_i_{\texttt{in}}) \end{aligned}$$

According to the BaseJump STL paper, the consumer endpoint of this module (to which $ready\_o_{\texttt{out}}$ belongs) is **helpful** because $ready\_o_{\texttt{out}}$ does not combinationally depend on $valid\_i_{\texttt{in}}$ (an input wire in the consumer endpoint). Thus, according to them, this module can safely be connected to any other module. However, our own analysis more precisely shows that while it is true that $ready\_o_{\texttt{out}}$ doesn't depend on other consumer endpoint wires, it does require other module input (in particular $yumi\_i_{\texttt{in}}$, which is part of the producer endpoint). This fact means that the PISO module connections may or may not be safe depending on the sorts of the interfaces to which it is directly or transitively connected.

Notably, after personal correspondence in which we reported the issue, the authors of the BaseJump STL PISO module updated it so that, according to our terms, $yumi\_i_{\texttt{in}}$ is now **to-sync** and $ready\_o_{\texttt{out}}$ is now **from-sync**.[4] This shows that designers care about the precise behavior of these interfaces and that an analysis that annotates wire sorts and verifies their interconnections is a useful thing to have.

5.2 OpenPiton Modules

We also used our analysis on a completely separate body of work: the OpenPiton Design Benchmark (OPDB), based on the OpenPiton manycore research platform [3, 39]. OPDB is interesting because it provides modules of a variety of scales and with different configuration options pre-generated per module. We were also interested in these OpenPiton designs due to anecdotes from the developers of OpenPiton related to issues they experienced with compositionality. In one instance, developing a like-for-like replacement for an existing component led to combinational loops that went undetected until final integration and synthesis due to minor mismatches in interfaces and test configurations. In another, a hardware generator produced combinational loops for only particular values of a parameter designed to change the size of a module, and those loops would require the composition of as many as seven modules to come into existence.

To process the OPDB designs, we followed the same Yosys Verilog-to-BLIF synthesis step as with the BaseJump STL designs, excluding some with asynchronous or multi-clock constructs. Our selected OPDB designs include a floating-point unit, network-on-chip router, and two caches, among others. Table 2 shows the OPDB designs we selected, their sizes in number of primitive gates, the time taken to infer

[4] See https://github.com/bespoke-silicon-group/basejump_stl/commit/67830f05ffce1333c7b790600530da0681af74fe

Table 1. Wire sorts of module ports for a subset of BaseJump STL; TS = **to-sync**, TP = **to-port**, FS = **from-sync**, FP = **from-port**. Every module also has a `reset` input wire whose sort is **to-sync**. The time listed is cumulative time to annotate all the wire sorts.

Module	Prim. Gates	Time (s)	Inputs			Outputs		
			Wire Name	Sort	Output Port Set	Wire Name	Sort	Input Port Set
First-In First-Out Queue	148,272	2.669	data_i	TS	∅	data_o	FS	∅
			yumi_i	TS	∅	ready_o	FS	∅
			v_i	TS	∅	v_o	FS	∅
Parallel-In Serial-Out Shift Reg.	53,637	0.606	valid_i	TS	∅	valid_o	FS	∅
			data_i	TS	∅	data_o	FS	∅
			yumi_i	TP	{ready_o}	ready_o	FP	{yumi_i}
Serial-In Parallel-Out SR	1,617,698	18.752	yumi_cnt_i	TS	∅	ready_o	FS	∅
			valid_i	TP	{valid_o}	valid_o	FP	{valid_i}
			data_i	TP	{data_o}	data_o	FP	{data_i}
Cache DMA	4,440	0.051	data_mem_data_i	TS	∅	data_mem_data_o	FS	∅
			dma_data_i	TS	∅	dma_data_o	FS	∅
			dma_data_v_i	TS	∅	dma_data_v_o	FS	∅
			dma_data_yumi_i	TS	∅	dma_data_ready_o	FS	∅
			dma_pkt_yumi_i	TP	{done_o}	dma_pkt_v_o	FP	{dma_cmd_i}
			dma_way_i	TP	{data_mem_w_mask_o}	data_mem_addr_o	FP	{dma_addr_i}
			dma_addr_i	TP	{data_mem_addr_o, dma_pkt_o}	data_mem_v_o	FP	{dma_cmd_i}
						data_mem_w_mask_o	FP	{dma_way_i}
			dma_cmd_i	TP	{done_o, dma_pkt_o, dma_pkt_v_o, data_mem_v_o}	dma_pkt_o	FP	{dma_addr_i, dma_cmd_i}
						done_o	FP	{dma_cmd_i, dma_pkt_yumi_i}
						data_mem_w_o	FS	∅
						dma_evict_o	FS	∅
						snoop_word_o	FS	∅

Table 2. Size (in primitive gates), wire sort inference time (in seconds), and number of IO ports of 17 OPDB modules.

Module	Prim. Gates	Time (s)	Ports
dynamic_node	29,918	0.759	35
fpu	168,525	1.456	16
ifu_esl	15,602	1.362	40
ifu_esl_counter	310	0.001	5
ifu_esl_fsm	2,299	0.040	34
ifu_esl_htsm	524	0.012	30
ifu_esl_lfsr	213	0.001	6
ifu_esl_rtsm	170	0.005	24
ifu_esl_shiftreg	208	0.001	4
ifu_esl_stsm	267	0.016	26
l2	1,088,384	15.128	16
l15	1,518,073	30.176	71
pico	36,479	0.245	24
sparc_ffu	104,966	0.723	77
sparc_mul	20,702	0.260	7
space_exu	320,397	10.203	132
sparc_tlu	650,364	8.753	214

the wire sorts of the design, and the number of input/output ports. Of the 17 designs we processed, the average number of gates was 232,788, while the smallest (`ifu_esl_rtsm`) had just 170 gates and the largest (`l15`) had more than 1.5 million gates. The designs had an average of 44 ports with the fewest ports (`ifu_esl_stsm`) being just 4, while the design with the most (`sparc_tlu`) had 214. The larger scale of these designs also skews to a longer average wire sort inference time, at 4.067 seconds, with a minimum of 0.001s and a maximum of 30.176s. We describe the asymptotic complexity of this operation in Section 5.5.

5.3 RISC-V CPU

For a more holistic case study, we implemented a multi-threaded single-cycle RISC-V [27, 41] CPU (RV32I base integer instruction set) in PyRTL. The CPU consists of 11 modules in total; the total number of primitive gates for the entire design, configured for five threads and five pipeline stages, is 229,011 gates. Our tool spent an average of 13.5 milliseconds on each module inferring its interface sorts; it took on average 162.7 milliseconds to determine all of the sorts, with a lower bound of 148.9 milliseconds and an upper bound of 194.2 milliseconds, at a rate of 298 nanoseconds per primitive gate. Once all the modules were connected, it was able to correctly check all the inter-module connections in an average of 67.1 milliseconds, with a lower bound of 62.5 milliseconds

Table 3. A comparison of cycle detection during synthesis (Yosys) versus our tool using wire sorts on large OPDB designs. Each unique module type only needs to be analyzed once; additional (non-unique) instantiations reuse the calculated sorts. The number of primitive gate differs from Table 2 because these are unflattened, and thus unoptimized, designs.

Module	Prim. gates (hier. BLIF)	Cycle det. time (s)		Speedup	Sort infer. time (s)	Submodules	
		Yosys	Ours			Total	Unique
`fpu`	189,452	46.42	3.11	14.92x	0.845	3530	118
`sparc_ffu`	105,688	11.30	1.00	11.30x	0.397	208	51
`sparc_exu`	331,452	22.81	8.65	2.63x	0.989	737	92
`sparc_tlu`	761,538	108.54	5.82	18.64x	0.813	777	128
`l2`	1,176,219	361.04	10.64	33.93x	13.80	157	45
`l15`	1,549,475	643.45	20.81	30.92x	7.86	68	26

and an upper bound of 77.3 milliseconds. Information on the sizes, wire sort annotations, and annotation times for the RISC-V submodules can be found in the supplementary material.

5.4 Comparison to Loop Detection During Synthesis

In our final analysis, we compared the efficiency of doing cycle detection at the HDL level via wire sorts versus at the netlist level during synthesis. Finding broken designs in the wild is difficult because most designers don't publish broken designs. So instead, we altered the OPDB Verilog designs slightly by introducing multi-module loops, importing the largest of them in their hierarchical BLIF format into PyRTL where our intermodular analysis is done. We then timed how long (1) Yosys takes to find the cycle during synthesis, (2) our tool takes to determine all interface sorts, and (3) our tool takes to check for intermodular loops given these sorts. We found that Yosys took longer to synthesize and find loops than our tool. It was also not straightforward to get Yosys to tell us these loops exist: depending on the options given, it would optimize them out or convert them to something else entirely without warning. Our results are found Table 3.

In actual use, we expect the user to write their designs in a modular fashion in a high-level HDL that can be analysed directly to begin with and to provide wire sort ascriptions if wanted. This experiment favored synthesis over our technique because it relied on importing a BLIF file, which has a few downsides. The Verilog-to-BLIF process converts N-bit ports into N 1-bit ports, meaning the number of ports increased by a factor equal to the average port bitwidth. The conversion also creates a module instance for each unique set of parameters used; since BLIF doesn't offer information that a module instantiation differs from another only by some parameter, those count as additional unique modules whose sorts must be calculated.

Despite this, annotating all modules with their I/O sorts was relatively quick, and detecting loops via intermodular connections using these sorts was **2.6–33.9x** faster than trying to find them during synthesis at the pure netlist level. We expect that by analyzing the design in its original form (e.g. Verilog or PyRTL), where the wires stay bundled together and parameterized module instances can be abstracted over, this speedup would increase significantly. This is exemplified by our RISC-V case study mentioned in Section 5.3, which was written entirely in PyRTL.

5.5 Complexity and Scalability

We describe the asymptotic complexity of the two analysis phases in order to demonstrate their scalability.

5.5.1 Module Wire Sort Inference. Sort inference takes place once per module definition. For a given module $M = (\overrightarrow{w_{\texttt{in}}}, \overrightarrow{w_{\texttt{out}}}, \overrightarrow{net})$, we must compute the transitive closure of combinationally reachable output wires for each $w_{\texttt{in}} \in \overrightarrow{w_{\texttt{in}}}$. Thus the total complexity of computing the sorts for all input wire sorts is $O(|\ \overrightarrow{w_{\texttt{in}}}\ | \cdot |\ edges\ |)$, where $edges = \bigcup_{net \in M.nets}\{\overrightarrow{w_\sigma} \mid (\overrightarrow{w_\sigma}, w_\sigma, op) = net\} \cup M.outputs$. Since the **to-port**/**from-port** relationship is symmetric, the wire sorts for outputs can be computed using the previously computed input wire sorts without traversing the module's internal wires again.

5.5.2 Circuit Well-Connectedness. The phase to check circuit well-connectedness uses the wire sorts computed by the module wire sort inference, and it operates only on the module interfaces without caring how large or complex any individual module might be. It only needs to be run once, after the circuit is complete. The algorithm iterates over each pair of inter-module input-output connections checking them against the *TransitivelyAffects* relationship ($\rightsquigarrow_C$).

Since each input port is connected to only one incoming output wire from another module, the number of connections is equal to the total number of input ports across the circuit. Given a circuit C and arbitrary wires $w_{\texttt{out}}$, $w_{\texttt{in}}$ in the circuit, the worst-case scenario is when the path from $w_{\texttt{out}}$ to $w_{\texttt{in}}$ traces through every inter-module connection before finally reaching the combinational loop. Thus, the

Table 4. The number of annotations per sort. TS = To-Sync, TP = To-Port, FS = From-Sync, FP = From-Port.

Source	Modules	Inputs		Outputs	
		TS	TP	FS	FP
BaseJump STL	144	233	211	178	197
OpenPiton DB	17	347	113	245	56
RISC-V	11	14	33	3	33
Total	**172**	**594**	**357**	**426**	**286**

TransitivelyAffects computation has a worst-case complexity of $O(|\ C.conns\ |)$. Since we do this check for *each* connection pair in $C.conns$, the total worst-case complexity is $O(|\ C.conns\ |^2)$.

5.5.3 Distribution of Wire Sorts. We found that sort annotations that our tool assigned to the module ports were widely distributed, as shown in Table 4. Across all 172 modules, **to-sync** inputs make up 62.5% of module inputs, compared to 37.5% for **to-port** inputs. **from-sync** outputs make up 59.8% of module outputs, compared to 40.2% for **from-port** outputs.

The foremost goal of this work was to reduce the number of "late surprises" in the design process. In these designs, 38.7% of the ports raise the possibility of a "late surprise" loops because they are **to-port** or **from-port**. For the remaining 61.3%, our technique has the additional advantage of making the checking process faster, by eliminating individual wires, or in the case of modules with entirely **to-sync**/**from-sync** IO, entire modules that need to be included in the cycle detection analysis.

6 Conclusion

We have presented an approach to creating hardware modules in isolation while tracking enough information to make checking their well-connectedness in an entire design feasible and user-friendly. BaseJump STL's informal approach of commenting ready-valid endpoints as **helpful** or **demanding** is a step in the right direction at classifying modules with information to help in connecting them at circuit design time in a plug-and-play fashion, but as we show it falls short in being able to prevent combinational loops.

Our solution is to provide wire-level information via a taxonomy of sorts: **to-sync**, **to-port**, **from-sync**, and **from-port**, allowing for modules to be written in isolation effectively and still safely connected without knowing their internals. We implemented our approach in a hardware description language and analyzed real-world designs (BaseJump STL and the OpenPiton Design Benchmark) as well as a multithreaded RISC-V CPU implementation, showing that our approach is feasible, effective, and efficient.

Acknowledgments

We thank our shepherd, Adam Chlipala, and the anonymous reviewers for their excellent suggestions on improving the paper. This material is based upon work supported by the National Science Foundation under Grants No. 1763699 and 1717779.

References

[1] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, Lille, France, 714–721. https://doi.org/10.1109/DSD.2010.21

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference* (San Francisco, California) *(DAC '12)*. Association for Computing Machinery, New York, NY, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[3] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 217–232. https://doi.org/10.1145/2980024.2872414

[4] UC Berkeley. 1992. Berkeley logic interchange format (BLIF). *Oct Tools Distribution* 2 (1992), 197–247.

[5] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: Hardware Design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/289423.289440

[6] Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. 2020. The Essence of Bluespec: A Core Language for Rule-Based Hardware Design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 243–257. https://doi.org/10.1145/3385412.3385965

[7] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. 2011. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '11)*. Association for Computing Machinery, New York, NY, USA, 33–36. https://doi.org/10.1145/1950413.1950423

[8] B. Cao, K. A. Ross, M. A. Kim, and S. A. Edwards. 2015. Implementing latency-insensitive dataflow blocks. In *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. IEEE, Austin, TX, USA, 179–187. https://doi.org/10.1109/MEMCOD.2015.7340485

[9] L. P. Carloni. 2015. From Latency-Insensitive Design to Communication-Based System-Level Design. *Proc. IEEE* 103, 11 (2015), 2133–2151. https://doi.org/10.1109/JPROC.2015.2480849

[10] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 1999. Latency Insensitive Protocols. In *Computer Aided Verification* (Berlin, Heidelberg, 1999) *(Lecture Notes in Computer Science)*, Nicolas Halbwachs and Doron Peled (Eds.). Springer, Berlin, Heidelberg, 123–133. https://doi.org/10.1007/3-540-48683-6_13

[11] Luca P. Carloni, Kenneth L. McMillan, and Alberto L. Sangiovanni-Vincentelli. 2006. Theory of Latency-Insensitive Design. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 20, 9 (Nov. 2006), 1059–1076. https://doi.org/10.1109/43.945302

[12] R. S. Chakraborty and S. Bhunia. 2009. HARPOON: An Obfuscation-Based SoC Design Methodology for Hardware Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 10 (2009), 1493–1502. https://doi.org/10.1109/TCAD.2009.2028166

[13] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proc. ACM Program. Lang.* ICFP, Article 24 (Aug. 2017), 30 pages. https://doi.org/10.1145/3110268

[14] Koen Claessen. 2001. *Embedded Languages for Describing and Verifying Hardware.* Ph.D. Dissertation. Chalmers University of Technology and Göteborg University.

[15] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL).* IEEE, Ghent, Belgium, 1–7. https://doi.org/10.23919/FPL.2017.8056860

[16] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[17] Deeksha Dangwal, Georgios Tzimpragos, and Timothy Sherwood. 2020. Agile Hardware Development and Instrumentation With PyRTL. *IEEE Micro* 40, 4 (2020), 76–84. https://doi.org/10.1109/MM.2020.2997704

[18] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. 1992. Protocol verification as a hardware design aid. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors.* IEEE, Cambridge, MA, USA, 522–525. https://doi.org/10.1109/ICCD.1992.276232

[19] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. 2018. Pi-Ware: Hardware Description and Verification in Agda. In *21st International Conference on Types for Proofs and Programs (TYPES 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:27. https://doi.org/10.4230/LIPIcs.TYPES.2015.9

[20] Daniel Geist. 2003. The PSL/Sugar Specification Language A Language for all Seasons. In *Correct Hardware Design and Verification Methods*, Daniel Geist and Enrico Tronci (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–3. https://doi.org/10.1007/978-3-540-39724-3_3

[21] Andy Gill, Tristan Bull, Andrew Farmer, Garrin Kimmell, and Ed Komp. 2011. Types and Type Families for Hardware Simulation and Synthesis. In *Trends in Functional Programming*, Rex Page, Zoltán Horváth, and Viktória Zsók (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 118–133. https://doi.org/10.1007/978-3-642-22941-1_8

[22] Andy Gill, Tristan Bull, Garrin Kimmell, Erik Perrins, Ed Komp, and Brett Werling. 2010. Introducing Kansas Lava. In *Implementation and Application of Functional Languages*, Marco T. Morazán and Sven-Bodo Scholz (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–35. https://doi.org/10.1007/978-3-642-16478-1_2

[23] OVL Working Group. 2014. *Accellera Standard OVL V2 Library Reference Manual.* Technical Report.

[24] 2010. IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (2010), 1–182. https://doi.org/10.1109/IEEESTD.2010.5446004

[25] 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)* (2018), 1–1315. https://doi.org/10.1109/IEEESTD.2018.8299595

[26] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture.* IEEE, Cambridge, United Kingdom, 280–292. https://doi.org/10.1109/MICRO.2014.50

[27] Jason Lowe-Power and Christopher Nitta. 2019. The Davis In-Order (DINO) CPU: A Teaching-Focused RISC-V CPU Design. In *Proceedings of the Workshop on Computer Architecture Education* (Phoenix, AZ, USA) *(WCAE'19).* Association for Computing Machinery, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/3338698.3338892

[28] Alan Mycroft and Richard Sharp. 2003. Higher-level techniques for hardware description and synthesis. *International Journal on Software Tools for Technology Transfer* 4, 3 (2003), 271–297. https://doi.org/10.1007/s10009-002-0086-1

[29] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable Accelerator Design with Time-Sensitive Affine Types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020).* Association for Computing Machinery, New York, NY, USA, 393–407. https://doi.org/10.1145/3385412.3385974

[30] J.T. O'Donnell. 2006. Overview of Hydra: a concurrent language for synchronous digital circuit design. *International Journal of Information* 9, 2 (March 2006), 249–264. https://doi.org/10.1109/IPDPS.2002.1016653

[31] Mary Sheeran. 1984. MuFP, a Language for VLSI Design. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA) *(LFP '84).* Association for Computing Machinery, New York, NY, USA, 104–112. https://doi.org/10.1145/800055.802026

[32] Wilson Snyder, Paul Wasson, and Duane Galbi. 2020. Verilator-convert Verilog code to C++/SystemC. http://www.veripool.org/wiki/verilator.

[33] Jane Street. [n.d.]. HardCaml: Register Transfer Level Hardware Design in OCaml. https://github.com/janestreet/hardcaml.

[34] Stuart Sutherland and Don Mills. 2006. Standard Gotchas Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know.

[35] Stuart Sutherland, Don Mills, and Chris Spear. 2007. Gotcha Again: More Subtleties in the Verilog and SystemVerilog Standards That Every Engineer Should Know.

[36] Lingshu Tang and Scott Davidson. 2019. BSG Micro Designs. https://github.com/bsg-idea/bsg_micro_designs.

[37] Michael Bedford Taylor. 2018. BaseJump STL: SystemVerilog Needs a Standard Template Library for Hardware Design. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18).* Association for Computing Machinery, New York, NY, USA, Article 73, 6 pages. https://doi.org/10.1145/3195970.3199848

[38] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Providence, RI, USA, 7:1–7:21. https://doi.org/10.4230/LIPIcs.SNAPL.2019.7

[39] Princeton University. 2020. OpenPiton Design Benchmark. https://github.com/PrincetonUniversity/OPDB.

[40] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

[41] Andrew Waterman, Yunsup Lee, David Patterson, Krste Asanović, and CS Division. 2016. The RISC-V Instruction Set Manual Volume I: User-Level ISA.

[42] Claire Wolf. [n.d.]. Yosys Open SYnthesis Suite. http://www.clifford.at/yosys/.

[43] Shaolin Xie and Michael Bedford Taylor. 2018. The BaseJump Manycore Accelerator Network. arXiv:1808.00650 [cs.AR]