Improving Hardware Programmability By Exploring Beyond Register-Transfer-Level Languages

ABSTRACT

Today's dominant hardware description languages (HDLs), namely Verilog and VHDL, rely on limited register-transfer-level (RTL) constructs. These constructs tightly couple design functionality with timing requirements and target-device constraints. As hardware designs and device architectures become increasingly more complex, these dominant HDLs yield verbose and unportable code. To raise the level of abstraction, several high-level synthesis (HLS) tools were introduced, usually based on software languages such as C. Unfortunately, designing hardware with sequential software language semantics comes at a price; the designer loses the ability to control hardware construction and data scheduling, which is crucial in many design use-cases.

In this paper we further extend DFiant, a Scala-based HDL that uses the dataflow model to decouple functionality from implementation constraints. DFiant's frontend enables functional bit-accurate hardware description, while maintaining a complete timing-agnostic and device-agnostic code. DFiant bridges the gap between software programming and hardware construction, driving an intuitive functional object oriented code into a high-performance hardware implementation.

For a proof of concept, we implemented a compiler frontend for DFiant, which transforms DFiant code into a dataflow graph, and a preliminary auto-pipelining backend, which maps the graph into synthesizable Verilog code. We further implemented two test cases in DFiant: an Advanced Encryption Standard cipher block and an IEEE-754 floating point multiplier. We compared both test cases against modern design flows. Our results demonstrate that DFiant can greatly simplify hardware designs yet still maintain competitive performance.

KEYWORDS

FPGA, HDL, HLS, Dataflow

1 INTRODUCTION

Low-level hardware description languages (HDLs) such as Verilog and VHDL have been dominating the field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) domains for decades. These languages burden designers with explicitly clocked constructs that do not distinguish between design functionality and implementation constraints (e.g., timing, target device). For example, the register-transfer-level (RTL) constructs of both Verilog and VHDL require designers to explicitly place a register, regardless if it is part of the core functionality (e.g., a state-machine state register), an artifact of the timing constraints (e.g, a pipeline register), or an artifact of the target interface (e.g., a synchronous protocol cycle delay). These semantics narrow design correctness to specific timing restrictions, while vendor library component instances couple the design to a given target device. Evidently, formulating complex portable designs is difficult, if not impossible. Finally, these older languages do not support modern programming features that enhance productivity and correctness such as polymorphism and type safety.

High-level synthesis (HLS) tools such as Vivado HLS [29], and high-level HDLs such as Bluespec SystemVerilog [18] and Chisel [1] attempt to bridge the programmability gap. While these tools and languages tend to incorporate modern programming features, they still mix functionality with timing and device constraints, or lack hardware construction and timed synchronization control. For example, designs must be explicitly pipelined in Chisel or Bluespec, while a simple task as toggling a led at a given rate is impossible to describe with C++ constructs in Vivado HLS. Such tools and languages, therefore, fail to deliver a clean separation between functionality and implementation that can yield portable code, while providing general purpose HDL constructs. We explore these gaps further in Section 4.

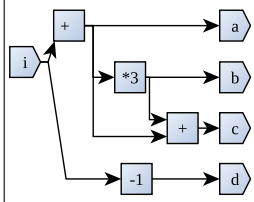
In this paper we further extend DFiant [23], a modern HDL whose goal is to improve hardware programmability and designer productivity by enabling designers to express truely portable and composable hardware designs. DFiant decouples functionality from timing constraints (in an effort to end the *"tyranny of the clock"* [26]). DFiant offers a clean model for hardware construction based on its core characteristics: (i) a clock-agnostic dataflow model that enables implicit parallel data and computation scheduling; and (ii) functional register/state constructs accompanied by an automatic pipelining process, which eliminate all explicit register placements along with their direct clock dependency. DFiant borrows and combines constructs and semantics from software, hardware and dataflow languages. Consequently, the DFiant programming model accommodates a middle-ground approach between low-level hardware description and high-level sequential programming.

DFiant is implemented as a Scala library and relies on Scala's strong, extensible, and polymorphic type system to provide its own hardware-focused type system (e.g., bit-accurate dataflow types,

FPGA'18, February 25-28, 2018, Monterey, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Table 1: Data Scheduling Semantics Example Function, f: Definition and Implementations

Formal Definition	Functional Drawing	C++ Impl.†	VHDL Impl.‡	DFiant Impl.
$f : (i_n)_{n\in\mathbb{N}} \to (a_n, b_n, c_n, d_n)_{n\in\mathbb{N}}$ $\triangleq \begin{cases} a_k = i_k + 5 \\ b_k = a_k * 3 \\ c_k = a_k + b_k \\ d_k = i_k - 1 \end{cases} k \geq 0$		`void f(int i, &a,&b,&c,&d){ a = i + 5; b = a * 3; c = a + b; d = i - 1; }`	`f : process(clk) begin if rising_edge(clk) begin a <= i + 5; b <= a * 3; â <= a;--cyc delay c <= â + b; d <= i - 1; end; end process;`	`def f(i : DFSInt[32]) = { val a = i + 5 val b = a * 3 val c = a + b val d = i - 1 (a,b,c,d) //tuple of } //four`

† Some type annotations were removed for brevity.
‡ **â** represents a clock cycle delay of **a**.

input/output port types). The library performs two main tasks: the frontend compilation, which translates dataflow variable interactions into a dependency graph; and the backend compilation, which translates the graph into a pipelined RTL code and a TCL constraints file, followed by a hardware synthesis process using commercial tools. Additionally, the graph can be simulated within the Scala integrated development environment (IDE).

This work focuses on the DFiant language and frontend compiler. DFiant is *not* an RTL language, nor is it a sequential language such as C. The following two sections highlight DFiant's unique semantics by comparing them against modern design language alternatives. For a proof of concept, we implemented a preliminary auto-pipelining backend compiler to compare DFiant and traditional HDL design flows in two test cases: an Advanced Encryption Standard [19] (AES) cipher block and an IEEE-754 [11] floating point multiplier (FPMul). Future work may delve further into the backend compiler and its HLS potential.

The paper is organized as follows. The next two sections describe DFiant's concurrency and state abstractions, followed by Section 4, which contrasts DFiant with related work. Section 5 details the DFiant type system. Section 6 provides a real-world DFiant code compilation example. Section 7 provides the proof of concept results. Finally, Section 8 concludes the paper.

2 CONCURRENCY AND DATA SCHEDULING ABSTRACTIONS

Concurrency and data scheduling abstractions rely heavily on language semantics. In this section we explore semantics of three distinctively different languages: C++[1], VHDL, and DFiant.

Consider a function f and its implementations, as detailed in Table 1. Despite similar code appearance, the semantics are very different, as depicted in Fig. 1. The following subsections qualify these semantics.

2.1 C++ Semantics

Sequential programming models, such as C++, do not possess concurrent semantics that allow the independent parts of `f` to run concurrently. Data scheduling order is set by the *code statement order* and cannot be pipelined. `f` can be executed in parallel by more than one thread, but each thread runs `f` sequentially and as a whole[2]. These concurrent semantics are important both for fine-grain parallelism and flexible concurrent code composition, as we demonstrate later in Section 5. HLS utilities extend sequential languages with `pragma` directives that change semantics. We observe the C++ `f` implementation as follows:

(1) All statements are variable assignments.
(2) `d` is independent of `a`, `b`, and `c` but cannot be scheduled concurrently. Additionally, `a` cannot be safely read until `f` finishes. Proper pragmas allow dataflow analysis and function inlining to overcome these limitations.

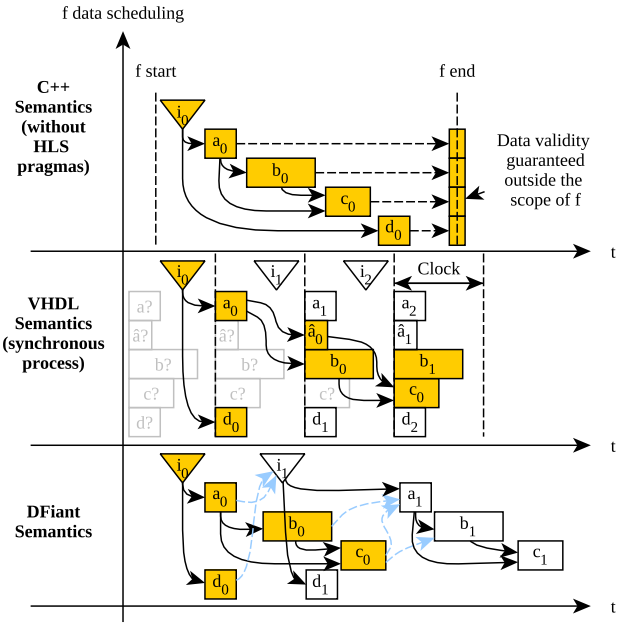


Figure 1: f data scheduling semantics in C++, VHDL, and DFiant

[1] C++ is required because reference `&` variables are not available in C. More advanced C++ capabilities are not always synthesizable, thus rarely used for hardware description.

[2] We only observe language semantics. Out-of-order or multi-processor executions may still apply.

(3) Time between/of the data operations is unconstrained. The code does not restrict the functional requirement and will maintain correctness for every hardware synthesis fitting its semantics.

2.2 VHDL Semantics

The RTL programming model is concurrent. Data scheduling is manual and clock-bound, while the order is set by the *assignment cycle-time*. VHDL process semantics are different for *signals* and *variables*: signals are updated when the process ends, while variables are updated instantly. When embedded in a signal edge-detection conditional construct, both signals and variables can be interpreted as registers, depending on the context. We observe the VHDL `f` implementation as follows:

(1) All statements are synchronous signal assignments with an explicit single-clock dependency. Clocked `f` imposes time restrictions to f. Although this implementation does not contradict the formal definition of f, its correctness is guaranteed solely under these restrictions.
(2) A latency balancing register added to maintain correctness of the `c` assignment pipeline[3].
(3) Data is scheduled for every clock cycle, thus creating a pipeline. Each output signal is valid at a different time. Invalid outputs may be accessed, since VHDL has no implicit *guard* semantics. More hardware is required to match the output cycle-latencies, and implement explicit guards.
(4) The implementation is very fragile and has limited reusability. Foremost, VHDL process construct alone is not reusable and requires an *entity-architecture* encapsulation for structural instantiation. Additionally, `f` is tightly-coupled to `clk` timing and logic propagation delay. The slightest change in requirements or target device can lead to a painful redesign.

2.3 DFiant Semantics

DFiant has a dataflow programming model. Data scheduling order, or *token-flow*, is set by the *data dependency*. Essentially, the DFiant semantics schedules all independent dataflow expressions concurrently, while dependent operations are synthesized into a guarded FIFO-styled pipeline. Dataflow branches are implicitly forked and joined. Semantically, unused nodes, always consume tokens and are discarded during compilation. We observe the DFiant `f` implementation as follows:

(1) All expressions are dataflow variable declarations.
(2) Concurrency is implicit. Function `f` is coded intuitively in a sequential manner, since dataflow dependencies are oblivious to statement order.
(3) Data scheduling is implicitly guarded by data dependencies. For example, `a` is forked into both `b` and `c` operations, while `c` joins branches from `a` and `b`. It is impossible to read an invalid result or an old result (without extending semantics further).
(4) DFiant semantics are intuitive: data is consumed only when it is ready and can be accepted by all receiving nodes, while back-pressure prevents data loss.
(5) A literal value (e.g., `5`) is a token generator that can produce infinite tokens (its production rate is set by the consumer).

[3]We can use VHDL variable to avoid latency balancing, by forming a combinational circuit.

2.4 Comparing Semantics

When comparing DFiant and VHDL, it is evident that DFiant is less verbose and has better semantics for code reuse. The DFiant compiler generates a hardware description that respects the design, timing, and target device constraints, in contrast to the given VHDL implementation which is equivalent to a singular possible DFiant code compilation for a given set of constraints. DFiant prevents `f` users from reading invalid values, while in VHDL it must be programmed explicitly. Bluespec and Chisel have similar semantics to VHDL, thus suffer from related limitations (e.g., explicit pipelining). Fortunately, they both can provide guarded types that prevent invalid data use.

When comparing DFiant and C++, we observe that C++ HLS tools rely on code analysis and pragma directives to change the semantics of their sequential code, while DFiant has its own dataflow type system that guarantees its seamless concurrent semantics. Consequently, C++ HLS tools limit language constructs and hierarchies which are not supported by the analysis algorithms (e.g., recursion), in contrary to DFiant which supports all finite Scala constructs (e.g., finite generation loops and recursions).

Contrarily, tandem operations are described more naturally in C++, and loops are utilized to describe repetitive dependent tasks. With proper pragmas, C++ loop iterations can run concurrently, but since they can also run sequentially, loops, and nested loops especially, may be semantically confusing. For this reason, DFiant does not support loops, same as VHDL (hardware generation loops are supported), and opts for state machine semantics to describe sequential operations.

3 REGISTER AND STATE ABSTRACTION

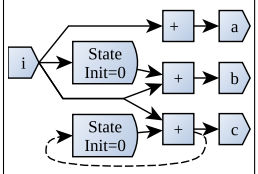
In the previous section we compared implementations of a pure (stateless) function, f. Typical designs, however, possess a state that is implemented via clocked registers. Registers have other functional roles, such as pipelining a data path, deriving timed signals from an input clock, and synchronizing an input signal. We believe that by avoiding explicit register use, we can design without, or very little, clock dependency.

In this section we classify various register use-cases, and present their DFiant functional counterpart. The classifications are divided into three main categories: *synchronous technology backend*, *synchronous technology interface*, and *state*. For the state classification, we also explore an impure function, g, and compare its implementation in VHDL, C++, and DFiant (see Table 2).

3.1 Synchronous Technology Backend

Registers are often forced upon the design due to a synchronous technology choice. Since they are unrelated to the functional requirement, DFiant has no constructs to express them, and relies on its compiler to implement them properly based on the functional requirements and design constraints. We differentiate between the following backend register uses:

Table 2: State Semantics Example Function, g: Definition and Implementations

Formal Definition	Functional Drawing	C++ Impl.[‡]	VHDL Impl.[‡]	DFiant Impl.
$g : (i_n)_{n\in\mathbb{N}} \to (a_n, b_n, c_n)_{n\in\mathbb{N}}$ $\triangleq \begin{cases} a_k = i_k + 5 & k \geq 0 \\ b_k = i_k + i_{k-1} & k > 0 \\ b_k = i_k + 0 & k = 0 \\ c_k = i_k + c_{k-1} & k > 0 \\ c_k = i_k + 0 & k = 0 \end{cases}$		`void g(int i, &a,&b,&c){ static int î=0; static int C=0; a = i + 5; b = i + î; î = i; c = i + C; C = c; }`	`g : process(clk) variable î : integer:=0; begin if rising_edge(clk) begin a <= i + 5; b <= i + î; î := i; c <= i + c; end; end process;`	`def g(i : DFSInt[32]) = { val a = i + 5 val b = i + i.init(0).prev val c = DFSInt[32] c := i + c //prev optional (a,b,c) //tuple of three }`

† Some type annotations were removed for brevity.
‡ **î** and **C** represent previous state values of **i** and **c**, respectively.

3.1.1 Pipelining. DFiant auto-pipelines the design by inserting registers to split long combinational paths. The amount of pipelining is determined by designer-specified constraints, such as the maximum path cycle latency, or the maximum propagation delay between registers.

3.1.2 Synchronizers. Sampling clock domain crossing (CDC) or asynchronous signals is exposed to metastability. Synchronizers, often composed of registers, are used to mitigate its effect and bring the design to the proper reliability. Since we aspire for a clockless design frontend, we want the synchronizers to be implicit. Currently, DFiant only supports a single clock backend, and does not require synchronizers. Further research may explore other backend options.

3.2 Synchronous Technology Interface

Functional design requirements are often accompanied by synchronous input/output (IO) timing constraints such as clocked protocol interfaces or real-time restrictions. However, these constraints only effect the interface, and are unrelated to the design core. To maximize design portability, we apply legacy constructs solely in the periphery, while keeping the design core coded in dataflow. DFiant exposes a frontend bridge across legacy RTL constructs and its dataflow types. We differentiate between the following synchronous signaling:

3.2.1 External IO and Blackbox Interfaces. External IOs that are exposed to the top design hierarchy, or blackboxes that are exposed to the internal design core, may impose synchronous protocols (e.g., data is valid one clock cycle after address is set). DFiant supports legacy RTL constructs to synchronously interface external IOs and instantiate blackboxes.

3.2.2 Timers. Timers are design constructs for outputting real-time signals, or creating derivations of timed signal inputs. For example, a target device is fed by a 100MHz clock and we want to output a UART stream at 10Mbps or toggle a led at 1Hz. Instead of directly applying registers or clock generation components, we can create functional representation of their timed use-cases. Currently, DFiant supports timers with legacy RTL constructs. This work may be expanded to include functional timers.

3.3 State

State occurs when we require access to (previous) values which are no longer available on a function's inputs (e.g., cumulative sum or a state-machine's state). Table 2 introduces a state function, g, and its implementation in C++, VHDL, and DFiant. The C++ implementation uses the `static` keyword to create variables that maintain the history of `i` and `c`. Because a static variable saves its value for every call of `g`, the C++ implementation cannot be used in the same design more than once. The VHDL implementation invokes registers (behaviorally) to save the state. Unfortunately, registers not only save the state, but also enforce specific cycle latencies. Furthermore, both C++ and VHDL declare additional variables and place extra assignments just to save the state. DFiant overcomes all these issues and in a less cumbersome way.

The DFiant state abstraction is achievable via the `.prev` construct, to summon the previous dataflow variable value, and also the construct `.init(value)`, to create an initialized dataflow variable. The `:=` assignment operation is available for a mutable dataflow variable such as `c` (see Section 5.2). The creation of `c` carries within it an implicit assignment `c := c.init(0).prev`, which makes the next assignment of `c`, `c := i + c`, equivalent to `c := i + c.init(0).prev`. This is possible due to the following semantics: *previous values change at the end of the DFiant code* (similarly to signal update semantics in VHDL processes). One advantage is that the code resembles its RTL counterparts, but less verbose. Another advantage is that any type of state component, like the Muller C-element[15], can be applied with DFiant as a frontend (note that the functional drawing in Table 2 has no register drawn).

We differentiate between two kinds of state: *derived state*, and *regular state*. Addressable memory pools also hold state, but we currently classify them as blackboxes.

3.3.1 Derived State. A derived state is a state whose current output value is *independent* of its previous value. For example, calculating output `b` of function `g` requires summoning previous value of `i`.

3.3.2 Regular State. A regular state is a state whose current output value is *dependent* of its previous state value. For example,

the cumulative sum output, `c`, of function `g` is dependent on the old sum value.

The two types of state differ heavily in performance improvement when the design is pipelined. A path from `i` to `b` can produce a token for every clock tick, and if we pipeline the addition operation to increase the maximum frequency, the maximum throughput will increase as well. Contrarily, a path from `i` to `c` also depends on the previous value of `c`, and if we pipeline the addition operation of that path, the extra latency may even decrease the throughput (multi-cycle path). Furthermore, `i` is forked into several paths, and abides by the slowest path throughput.

Regular state causes bottlenecks in many systems. For instance, a processor's program counter (PC) register manifests as a regular state. The processor pipeline can only be improved thanks to a speculative mechanism that predicts the next PC value to prefetch instructions (e.g., PC+4 for a branch-not-taken prediction). In case of a miss-prediction, other mechanisms take place. Further research may expand DFiant's abstractions, and solve such problems functionally.

4 RELATED WORK

Recent studies [12, 17, 28] surveyed a variety of HDLs and HLS tools. Neither survey had explicit conclusion which tool or language should be used for hardware design. Earlier, we focused on comparing DFiant to VHDL and C++-based HLS. We rely on the surveys conclusions, and contrast DFiant to a few key hardware design languages and tools.

Chisel, SpinalHDL, and VeriScala. Chisel [1], SpinalHDL [3], and VeriScala [13] are Scala-based libraries that provide advanced HDL constructs. SpinalHDL focuses on a more accurate hardware description (e.g., multiple clock domains), while Chisel focuses on providing cycle accurate simulation alongside its HDL constructs (via C++ test code generation), and VeriScala focuses on IDE-interactive FPGA application testing. When compared to DFiant, all three HDL libraries utilize RTL constructs by implicitly or explicitly invoking clocked registers, and do not auto-pipeline designs. Moreover, DFiant is an early-adopter of new Scala features such as literal types [21] and operations [24], which further improve type safety (e.g., a `DFBits[5].bits(Hi,Lo)` bit selection is compile-time-constrained within the 5-bits vector width confines).

Synflow Cx. Synflow developed Cx [27] as a designer-oriented RTL with new language semantics that better fit hardware design than the classic C syntax. However, the concurrency in Cx limits dataflow description flexibility. A `fence` statement is required to force a new cycle. This statement affects all variables within a `task`. To avoid this, separate tasks are required, which limits functional clustering in a single task. Moreover, Cx is not object-oriented and has a limited type-system.

MyHDL and PyRTL. MyHDL [7] and PyRTL [4] are Python-based RTLs. MyHDL favors verification capabilities over purely synthesizable hardware constructs, in contrary to our approach in DFiant. PyRTL provides a large primitives library and a complete tool-chain. Both MyHDL and PyRTL carry RTL constructs and therefore do not support automatic pipelining. Moreover, as Python-based libraries, they also lack type-safety.

Bluespec. Bluespec uses concurrent guarded atomic actions to create rules that derive hardware construction. Bluespec's rules are atomic and execute within a single clock cycle. Consequently, the rule semantics bound the design to the clock, and if the design does not meet timing constraints, the rules system must be modified.

Vivado HLS. Vivado HLS [29] is a mature tool that helps achieve high productivity in some domains. Nevertheless, it is not accepted as a general purpose HDL, since its C/C++ semantics are unfitting [30] and its SystemC synthesizable constructs provide roughly identical capabilities of traditional RTLs [9].

Maxeler. The Maxeler framework [22] and its MaxJ Java-based programming language take part in acceleration systems. MaxJ is dataflow-centric, same as DFiant, and provides constructs to fetch previous values from a token stream. However, MaxJ is tailored for its target hardware framework and does not fit as a general purpose HDL. Furthermore, while MaxJ possesses a limited set of counters for general loop control, DFiant has no such limitations thanks to its `prev` semantics which enable expressing any required control state.

OpenDF. The OpenDF framework [2] which is based on the CAL actor language [8] is a dataflow hardware description framework. Its paper provides similar motivation to use the dataflow programming model for hardware description instead of sequential languages such as C. However, using actors and networks is completely different than conventional hardware and software programming languages. We already demonstrated that DFiant can achieve the same dataflow description capability, and in a more intuitive and familiar fashion.

5 THE DFIANT TYPE SYSTEM

DFiant is a Scala library, hence it inherently supports type safe and rich language constructs. DFiant brings type driven development concepts to hardware design, by creating an extensible dataflow class hierarchy, with the trait `DFAny` at its head (similar concept to Scala's Unified Types hierarchy). `DFAny` contains all properties that are common to every dataflow variable (e.g., `.width` represents the number of bits contained by the variable). Fig. 2 illustrates a simplified inheritance diagram of DFiant's dataflow types. Further explanation is given in Section 5.2.

5.1 Bit-Accurate Operations, Type Inference, and Data Structures

All DFiant's dataflow types are bit-accurate and structurally static, with their bit-width set upon construction (e.g., `DFBits[5]` is a 5-bit vector). Operations between dataflow variables produce a bit-accurate result with the proper type inference. For example, an addition between an unsigned 5-bit variable (`DFUInt[5]`) and a signed 10-bit variable (`DFSInt[10]`) produces an adder that can be implicitly converted to a 10-bit signed variable, if carry is not required, or an 11-bit signed variable by explicitly invoking `.wc` from the addition.

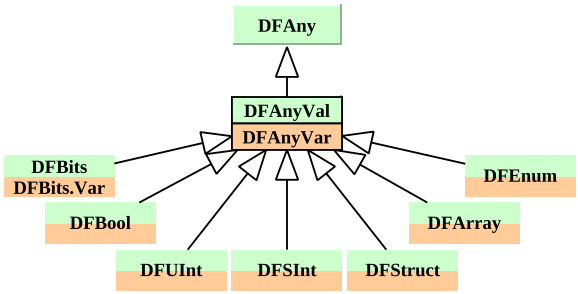


Figure 2: DFiant dataflow types: simplified inheritance diagram

DFiant also allows operations between dataflow types and their corresponding Scala numeric types, by treating the Scala numeric types as constants (e.g., addition between `DFSInt` and `Integer` variables). A constant in the dataflow graph is a node that can produce infinite tokens of the same value.

5.2 Mutability

DFiant supports dataflow variables mutability via the `:=` operator. Do not confuse with Scala-level mutability which is enabled by using `var` instead of `val`. Each dataflow class has two variations: an immutable class, which inherits from `DFAny`**`Val`** and a mutable class, which inherits from `DFAny`**`Var`** and accepts `:=`. The difference between the types enforces an immutable right-hand-side (RHS), where required, and a mutable variable creation. Consider, for instance, the DFiant implementation of `g` in Table 2: `a` is immutable because it is a RHS addition between the dataflow variable `i` and a literal value `5`. Contrarily, `c` is mutable, since it is a dataflow variable constructor (`.init` constructs a new initialized variable, while preserving the mutability trait).

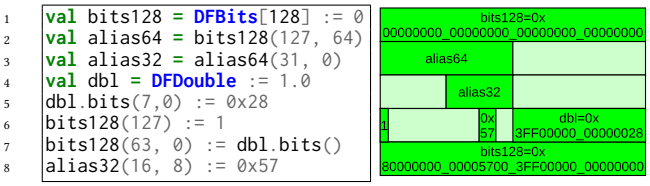
Fig. 2 demonstrates a dual class definition for every type (immutable and mutable). The naming convention helps to reason about the mutability. For example, `DFBits` and `DFBits.Var` are immutable and mutable classes, respectively. Constructing a new variable via `DFBits` (e.g, `val a = DFBits[5]`) returns the mutable `DFBits.Var[5]`. Usually, we either receive or return an immutable type, hence we do not require annotating a type with its mutable variation. In cases where we want to return a mutable type, we annotate it as an output port (see Section 5.5).

5.3 Bit Aliasing and Casting

Aliasing in DFiant enables referencing a part of a dataflow variable, by invoking `.bits(hiIdx, loIdx)`, which creates a bits vector alias that references the original variable at the given index parameters. Every change of a dataflow variable affects its alias and vice versa (similar to VHDL's signal aliasing). Since this function also casts the variable as `DFBits`, this feature is used as a raw-data cast between different dataflow types. Aliasing of an alias is also possible, while maintaining relative bits indexing. Aliasing preserves the mutability trait: an alias of an immutable variable is immutable, while an alias of a mutable variable is mutable.

Fig. 4 demonstrates aliasing code and its effect on the contents of a dataflow variable (`bits128`). Each line code does as follows:

(1) Constructs a new 128-bit vector, `bits128`, and clears it.
(2) Creates a new alias, `alias64`, which references the most significant 64 bits of `bits128`. Since `bits128` is a `DFBits` variable, there is no need to invoke `.bits()`, and we can apply the required indexes directly.
(3) Creates a new alias, `alias32`, which references the least significant 32 bits of `alias64`, which reference bits 64 to 95 of `bits128`.
(4) Constructs a new double precision floating point dataflow variable, `dbl`, and initialize its value as `1.0` (hexadecimal value of `0x3FF00...0`).
(5) Modifies the least significant byte of `dbl`.
(6) Sets the most significant bit of `bits128`.
(7) Assigns `dbl` to the least significant 64 bits of `bits128` through casting. All the bits of `dbl` are selected because `.bits()` is invoked without index parameters.
(8) Modifies a byte of `bits128`.



(a) DFiant code **(b) Contents of `bits128`**

Figure 4: Bit aliasing and casting example

5.4 Structural Composition and Generation

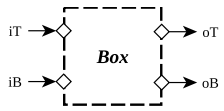
DFiant expands traditional structural composition capabilities by utilizing Scala's object oriented features such as inheritance and polymorphism, as well as finite loops and recursive composition. The hierarchical compositions provide the scope and dependencies for the dataflow variables. The hierarchy itself is transparent to the dataflow graph, as if the entire design is flattened, inlined, and unrolled. Therefore, hierarchies in DFiant are synthesizable, highly reusable, and do not affect the design performance (may affect compilation time). Different composition examples are available in Table 3.

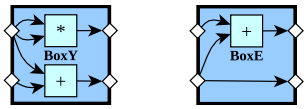
5.5 IO Ports

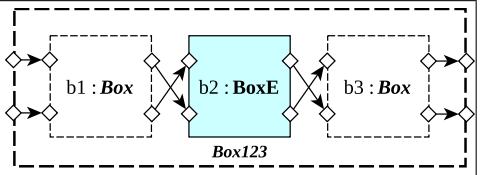
The class *Box* from Table 3 can also be coded as demonstrated in Fig. 5. The annotation `DFVar <> Dir` controls *DFVar*'s access by encapsulating the variable with the dataflow port class, `DFPort`: an `IN` port can only be read (immutable), while an `OUT` port can only be modified (unreadable). DFiant has implicit conversions in place that selectively converts between `DFPort` and `DFAny` instances, without breaking mutability rules and type safety. The

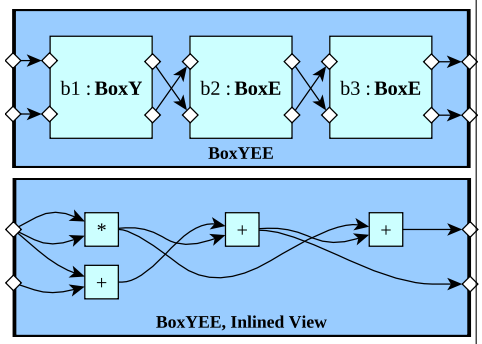
Table 3: DFiant Hierarchy Example: Inheritance, Polymorphism, Recursive Composition, and Inlined View

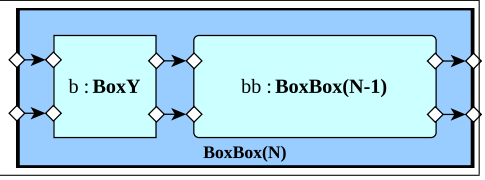
Description	DFiant Code	Functional Drawing

Abstract base class, *Box* (defines only an interface)	`type DFB = DFUInt[32] //Type alias, to save code space`
	`abstract class Box(iT: DFB, iB: DFB) { //T=Top, B=Bottom`
	`  val oT: DFB`
	`  val oB: DFB`
	`}`
		

Concrete *Box* implementation examples	`case class BoxY(iT: DFB, iB: DFB) extends Box(iT, iB) {`
	`  val (oT, oB) = (iT * iT, iT + iB)`
	`}`
	`case class BoxE(iT: DFB, iB: DFB) extends Box(iT, iB) {`
	`  val (oT, oB) = (iT + iB, iB)`
	`}`
		

Box123, an abstract polymorphic composition of three *Box* instances	`abstract class Box123(iT: DFB, iB: DFB) extends Box(iT, iB){`
	`  def b1Bld(iT: DFB, iB: DFB) : Box`
	`  def b3Bld(iT: DFB, iB: DFB) : Box`
	`  val b1 = b1Bld(iT,  iB)`
	`  val b2 = BoxE(b1.oB,  b1.oT)`
	`  val b3 = b3Bld(b2.oB,  b2.oT)`
	`  val (oT, oB) = (b3.oT, b3.oB)`
	`}`
		

BoxYEE, a concrete polymorphic composition of three *Box* instances
+
An inlined view of *BoxYEE*
	`case class BoxYEE(iT: DFB, iB: DFB) extends Box123(iT, iB) {`
	`  def b1Bld(iT: DFB, iB: DFB) = BoxY(iT, iB)`
	`  def b3Bld(iT: DFB, iB: DFB) = BoxE(iT, iB)`
	`}`
		

Finite recursive composition example	`case class BoxBox(N: Int)(iT: DFB, iB: DFB)`
	`extends Box(iT, iB) {`
	`  val b = BoxY(iT, iB)`
	`  val bb : Box = if (N > 0) BoxBox(N - 1)(b.oT, b.oB)`
	`  else b`
	`  val (oT, oB) = (bb.oT, bb.oB)`
	`}`
		

port annotations match the capabilities of traditional HDLs, and are noticeably absent from HLS languages such as C++.

```
abstract class Box() {
  val iT: DFB <> IN
  val iB: DFB <> IN
  val oT: DFB <> OUT
  val oB: DFB <> OUT
}
```

Figure 5: IO port annotation DFiant code example

6 LEADING-ZERO-COUNTER AND SHIFTER (LZCS) COMPILATION EXAMPLE

In this section we provide an LZCS DFiant implementation and describe its compilation process. We begin with a pseudo code reference implementation[16] as depicted in Fig. 6. The DFiant LZCS implementation[4] is very similar to the pseudo code reference, as can be seen in Fig. 7.

[4]This code was also utilized to implement FPMul in DFiant

Combined leading-zero counting and shifting

$k \leftarrow \lceil \log_2 n \rceil$
$x_k \leftarrow x$
for $i = k - 1$ downto 0 **do**
  if there are 2^i leading zeros in x_{i+1} **then**
    $d_i \leftarrow 1$
    $x_i \leftarrow x_{i+1}$, shifted left by 2^i
  else
    $d_i \leftarrow 0$
    $x_i \leftarrow x_{i+1}$
  end if
end for
return $(d, x0)$

Figure 6: LZCS pseudo code reference

When we apply `lzcs` on an 8-bit vector `num`, the DFiant frontend compiler unrolls the `for` loop and converts the `ifdf` control

```
def lzcs[N](num : DFBits[N]) = {
  val k = log2Up(num.width)
  val x = DFBits(num.width) := num
  val d = DFBits(k) := 0

  for (i <- k-1 downto 0) {
    ifdf (x.msbits(2~^i) == 0) {
      d(i) := 1
      x := x << 2~^i
    }
  }
  (d, x)
}
```

Figure 7: LZCS DFiant code

statements into multiplexer nodes. An equivalent unrolled code is given in Fig. 8. This code is similar to the frontend compiler's static single assignment (SSA) intermediate representation (IR) form. The DFiant frontend IR is a dataflow dependency graph as illustrated in Fig. 9. We can optimize the IR design independently of the target device. For example, we can derive `d_i` directly from `c_i` and remove unnecessary multiplexer nodes.

```
val x_3 = DFBits[8] := num;    val d_3 = DFBits[3] := 0
val c_2 = x_3(7,4)==0;         val X_3 = x_3 << 4
val x_2 = mux(c_2, X_3, x_3); val D_3 = d_3(2):=1
val d_2 = mux(c_2, D_3, d_3)
val c_1 = x_2(7,6)==0;         val X_2 = x_2 << 2
val x_1 = mux(c_1, X_2, x_2); val D_2 = d_2(1):=1
val d_1 = mux(c_1, D_2, d_2)
val c_0 = x_1(7,7)==0;         val X_1 = x_1 << 1
val x_0 = mux(c_0, X_1, x_1); val D_1 = d_1(0):=1
val d_0 = mux(c_0, D_1, d_1)
return (d_0, x_0)
```

Figure 8: LZCS unrolled code for an 8-bit input

The backend compiler converts the dataflow graph to Verilog and adds pipeline registers as required to achieve the target performance. Not all dataflow nodes cost logic and therefore do not affect performance. Bit referencing, concatenation, shifting, and assignment do not cost any hardware resources (software backend compilation can incur performance penalty).

The preliminary auto-pipelining backend compiler we implemented relies on an estimation database that is used to calculate each node's potential propagation delay cost. The compiler makes sure no path propagation delay surpasses the target clock period, including a fixed safe margin for additional wiring delay that may occur during placement and routing. Strategic register placement brakes down long paths but increases the path clock cycle latency. All converging paths must possess the same latency to maintain correctness (we can rely on token exchange signals to provide dependency correctness, but to achieve optimal performance we need to balance the paths' latency nonetheless).

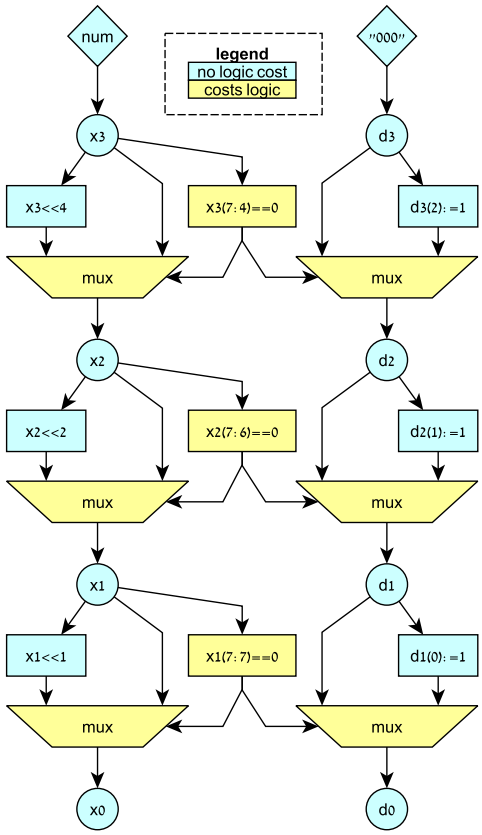


Figure 9: LZCS IR graph for an 8-bit input

7 PROOF OF CONCEPT

In this section we present a working proof of concept by implementing two case studies: an AES cypher, and a double precision FPMul. We compare both test cases against traditional designs, and demonstrate competing performance while simplifying code verbosity significantly.

7.1 Methodology

We implemented both test cases in DFiant, constrained them by a variance of minimum frequency requirements, and compiled them to RTL. The DFiant compiler automatically pipelined the design to achieve the required minimum frequency, and generated an RTL verilog file and a TCL constraints file. For a baseline we obtained equivalent open-source RTL cores and Vivado HLS implementations, where possible. We disabled the DFiant backend support for pipelined valid/ready signaling and a blocking back-pressure, since the RTL cores did not support this capability.

We chose the following comparison metrics: the maximum clock frequency, clock cycle latency, utilizations of both look-up tables (LUTs) and flip-flop registers (FFs), and lines of code (LoC). Digital signal processing (DSP) block utilization was zero for AES and equivalent for FPMul across all designs, thus neglected from the table. a We used Xilinx Vivado to synthesize and implement the

RTL design for a Virtex-7 FPGA, part number: xc7vx485tffg1761-2. The tool was configured to use default strategy for both synthesis and implementation processes. For each design, we recorded the maximum clock frequency, LUTs, and FFs. We recorded the design latency as reported by the DFiant and Vivado HLS compilers, and the RTL cores documentation. Finally, we automatically counted the LoC [5], applied standard score normalization (0-100) to all metrics, and assured higher values indicate better score for all metrics. Mean score of all metrics is presented as well.

7.2 Case Study: AES Cypher

For baseline comparison we used three AES cypher RTL designs from opencores.org: Das core [6], Hsing core [10] and Salah core [25]. Additionally, we obtained a Vivado C++ HLS design [20]. All these designs are fully pipelined, meaning that in every cycle the design accepts new key and data inputs, and emits an encrypted data output, delayed by a fixed design latency.

We compiled the DFiant and Vivado HLS designs with three target clock frequencies: 200 MHz, 300 MHz, and 450 MHz. We named the designs accordingly (e.g., DFiant_200 is the 200MHz target design). We collected the results in Table 4, and displayed their normalized standard score in Fig. 10. We added the supported key types quantity as a metric, since some designs support 128bit keys and do not include 192bit and 256bit keys as well. The table also includes an 'SBox BRAM Use' column, since some designs do not use memory to implement the AES SBox function.

The Hsing core clearly has the best performance among the different designs (but the lowest score on LUTs utilization). The primary reason it achieved this is because it uses LUTs instead of BRAMs. This enables the synthesizer to optimize the AES SBox function, and even pipeline it. DFiant uses its `lookupTable` library function to implement SBox, and we have yet to enable such an option for DFiant.

Although the DFiant-generated RTL performance is not optimal, it can still be improved without modifying the DFiant AES code, if the DFiant compiler is optimized. Moreover, this code has an adaptive pipeline, while the RTL cores pipelines are fixed. The Vivado implementation enjoys the same advantages as DFiant, and has even less LoC. However, the Vivado code does not support all possible keys and its maximum performance is far from optimum (we did not attempt to improve the HLS pragma directives).

If we assume all metrics have the same weight, the mean score places the DFiant solutions at the top. It is difficult to determine what is truly the best solution, but DFiant clearly has the best potential for further improvement without any modification to the application code.

7.3 Case Study: Double Precision FPMul

We compared our FPMul with the open IEEE-754 compatible Lundgren core [14] (the only IEEE-754 fully compatible FPMul RTL design we had access to). Since the core is a complete floating point unit, we disabled the unnecessary parts, reducing it to only an FPMul, for a fair comparison with DFiant's code. The DFiant code was written by using the Lundgren VHDL code as a reference design. The designs are very similar in their structure, except that DFiant is considerably less verbose, and has no explicit pipeline.

We had no access to an open Vivado HLS FPMul for comparison. We could have directly invoked a *double* multiplication, but an inspection of the generated RTL revealed that Vivado HLS just instantiates an RTL floating point blackbox core. DFiant can choose to use this core as well, and achieve identical performance to Vivado HLS. Furthermore, the Vivado HLS floating point implementation is not fully compatible with IEEE-754 (e.g, does not support denormalized numbers).

Similarly to the AES case study, we collected the results in Table 5, and displayed their normalized standard score in Fig. 11. In comparison with the Lundgren core, DFiant_350 is better in every criteria, aside from FFs utilization. Ultimately, DFiant out-performs its reference design of FPMul, and demonstrates its ability to provide different RTL designs for design space exploration.

8 CONCLUSION

In this paper we presented our extension of DFiant, a dataflow HDL, and exposed its advantageous semantics, when compared to modern RTLs and C++-based HLS tools, such as VHDL and Vivado HLS, respectively. DFiant provides a seamless concurrent programming approach, and yet it still facilitates a versatile compositional and hierarchical expressiveness. We evaluated DFiant in two computational-heavy case studies, and demonstrated its competing performance alongside its code simplification.

So far, we demonstrated how DFiant covers static one-to-one token transfer functions. Notwithstanding, functionality may require upsampling (e.g., duplicate each token), downsampling (e.g., drop every third token), token arrival time dependency (e.g., priority round-robin arbiter), or token value dependency (e.g., filter out odd-valued tokens). Future work may explore expanding control over token generation and consumption.

Table 4: AES Cypher RTL Designs Comparison (the numbering on the left associates configurations with Fig. 10)

	RTL Design	Latency [Cyc]	Max Freq. [MHz]	LUTs [#]	FFs [#]	LoC [#]	Key Support [#]	Sbox BRAM Use
1	DFiant_200	13	293	6070	3931	334	3	Yes
2	DFiant_300	25	311	3956	8196	334	3	Yes
3	DFiant_450	38	442	3782	11437	334	3	Yes
1	Vivado_200	20	231	2658	3811	161	1	Yes
2	Vivado_300	30	308	2624	1777	161	1	Yes
3	Vivado_450	39	350	2567	3957	161	1	Yes
1	Das Core	32	451	10369	11927	340	1	Yes
2	Hsing Core	21	454	11103	5386	922	3	No
3	Salah Core	41	410	9204	9687	618	1	No

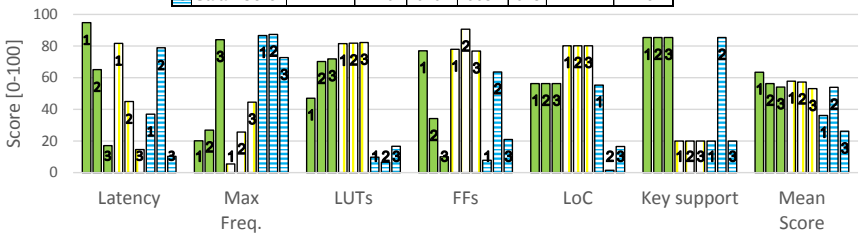


Figure 10: AES cypher RTL designs score comparison (higher = better)

Table 5: FP Mult. RTL Designs Comparison (the numbering on the left associates configurations with Fig. 11)

	RTL Design	Latency [Cyc]	Max Freq. [MHz]	LUTs [#]	FFs [#]	LoC [#]
1	DFiant_100	8	111	4635	3798	180
2	DFiant_200	15	229	2294	1633	180
3	DFiant_350	23	354	3733	2246	180
1	Lundgren Core	24	322	3744	1633	340

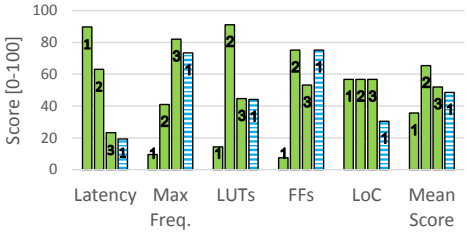


Figure 11: FP multiplication RTL designs score comparison (higher = better)

REFERENCES

[1] 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 *ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[2] Shuvra S Bhattacharyya, Gordon Brebner, Jörn W Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, and Mickaël Raulet. 2008. OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. *Computer Architecture News* 36, 5 (2008), 29–35.

[3] Papon Charles. 2016. SpinalHDL. (2016). http://spinalhdl.github.io/SpinalDoc

[4] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sh. 2017. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. In *Intl. Symp. on Field Programmable Gate Arrays*.

[5] Al Danial. 2009. Cloc–count lines of code. (2009). https://github.com/AlDanial/cloc

[6] Subhasis Das. 2010. Fully Pipelined AES Core. (2010). https://opencores.org/project,aes_pipe

[7] Jan Decaluwe. 2004. MyHDL: a python-based hardware description language. *Linux Journal* 127 (2004).

[8] Johan Eker and Jorn Janneck. 2003. *CAL language report.* Technical Report. Tech. Rep. ERL Technical Memo UCB/ERL.

[9] Dan Gajski, Todd Austin, and Steve Svoboda. 2010. What input-language is the best choice for high level synthesis (HLS)?. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.

[10] Homer Hsing. 2013. AES Core Specification. (2013). http://opencores.org/usercontent,doc,1354351714

[11] IEEE. 2008. *754-2008 - IEEE Standard for Floating-Point Arithmetic.*

[12] Nachiket Kapre and Samuel Bayliss. 2016. Survey of domain-specific languages for FPGA computing. In *Intl. Conf. on Field Programmable Logic and Applications*.

[13] Yanqiang Liu, Yao Li, Weilun Xiong, Meng Lai, Cheng Chen, Zhengwei Qi, and Haibing Guan. 2017. Scala Based FPGA Design Flow (Abstract Only). In *Intl. Symp. on Field Programmable Gate Arrays*.

[14] David Lundgren. 2014. Double Precision Floating Point Core VHDL. (2014). http://opencores.org/project,fpu_double

[15] DE Muller and WS Bartky. 1957. A theory of asynchronous circuits II. *Digital Computer Laboratory* 78 (1957).

[16] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. 2009. *Handbook of floating-point arithmetic.* Springer Science & Business Media.

[17] Razvan Nane, Vlad Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (2016), 1591–1604.

[18] Rishiyur Nikhil. 2004. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *ACM/IEEE Intl. Conf. on Formal Methods and Models for Co-Design*.

[19] NIST. 2001. Advanced Encryption Standard (AES). *Federal Information Processing Standards Publication* 197, 441 (2001).

[20] Colin O'Flynn. 2014. Rapid FPGA Design in C Using High-Level Synthesis. *Circuit Cellar* 283 (2014), 46–53.

[21] Erik Osheim, George Leontiev, Jon Pretty, Lars Hupel, Mike O'Connor, Miles Sabin, and Tom Switzer. 2017. Typelevel Scala. (2017). https://github.com/typelevel/scala

[22] Oliver Pell and Oskar Mencer. 2011. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News* 39, 4 (2011).

[23] Oron Port and Yoav Etsion. 2017. DFiant: A Dataflow Hardware Description Language. In *Intl. Conf. on Field Programmable Logic and Applications*.

[24] Frank S. Thomas, Matthew Pocock, Naoki Aoyama, and Oron Port. 2017. singleton-ops library. (2017). https://github.com/fthomas/singleton-ops

[25] Amr Salah. 2013. 128 bit AES Pipelined Cipher. (2013). http://opencores.org/usercontent,doc,1378852274

[26] I Sutherland. 2012. The tyranny of the clock. *Comm. ACM* 55, 10 (2012), 35–36.

[27] Synflow. 2014. Cx Language. (2014). http://cx-lang.org/

[28] Skyler Windh, Xiaoyin Ma, Robert J. Halstead, Prerna Budhkar, Zabdiel Luna, Omar Hussaini, and Walid A. Najjar. 2015. High-level language tools for reconfigurable computing. *Proc. of the IEEE* 103, 3 (2015), 390–408.

[29] Xilinx. 2015. Vivado High Level Synthesis User Guide. (2015).

[30] Zhipeng Zhao. 2017. Using Vivado-HLS for Structural Design : a NoC Case Study. In *Intl. Symp. on Field Programmable Gate Arrays*.