

Hardware Description Beyond Register-Transfer Level Languages

Oron Port

soronpo@campus.technion.ac.il
Technion – Israel Institute of Technology
Haifa, Israel

Yoav Etsion

yetsion@technion.ac.il
Technion – Israel Institute of Technology
Haifa, Israel

ABSTRACT

Prevalent hardware description languages, e.g., Verilog and VHDL, employ register-transfer level (RTL) as their underlying programming model. A major downside of the RTL model is that it tightly couples design functionality with timing and device constraints. This coupling increases code complexity and yields code that is more verbose and less portable. Emerging high-level synthesis (HLS) tools decouple functionality from timing and design constraints by utilizing constructs from imperative programming languages. These constructs and their sequential semantics, however, impede construction of inherently parallel hardware and data scheduling, which is crucial in many design use-cases.

In this paper we propose to use constructs from dataflow programming languages as basis for hardware design. We present DFiant, a Scala-embedded HDL that leverages dataflow semantics to decouple functionality from implementation constraints. DFiant enables the timing-agnostic and device-agnostic hardware description by using the dataflow firing rule as a logical construct, coupled with modern software language features (e.g., inheritance, polymorphism) and classic HDL traits (e.g., bit-accuracy, input/output ports). Using DFiant we demonstrate how dataflow constructs can be used to write code that is substantially more portable and more compact than the equivalent design in RTL and HLS languages.

We implemented a compiler for DFiant that transforms DFiant code into a dataflow graph, optionally auto-pipelines the design to meet the target performance and device requirements, and maps the graph into synthesizable VHDL code.

CCS CONCEPTS

• **Hardware** → **Hardware description languages and compilation.**

KEYWORDS

HDL, HLS, Dataflow

ACM Reference Format:

Oron Port and Yoav Etsion. 2020. Hardware Description Beyond Register-Transfer Level Languages. In . ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/1122445.1122456>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-9999-9/18/06...\$15.00
<https://doi.org/10.1145/1122445.1122456>

1 INTRODUCTION

The register-transfer level (RTL) programming model paved the road for Verilog and VHDL to flourish as the leading hardware description languages (HDLs). That road, however, is steadily nearing its end as both hardware designs and devices become increasingly more complex. While the software world is striving for a “write once, run anywhere” programmability, RTL design complexity for a given functionality may vary greatly across different FPGA and ASIC devices that incorporate various technologies and core components. Moreover, minor requirement changes may lead to significant redesigns, since RTL abstraction tightly couples functionality with timing constraints. For example, registers serve various roles such as preserving a state, pipelining and balancing a data path, deriving timed signals from an input clock, and synchronizing an input signal. This coupling between functionality, timing constraints, and device constraints leads to verbose and unportable RTL designs.

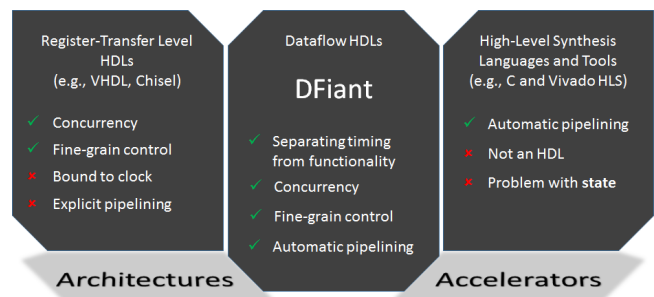


Figure 1: DFiant bridges the gap

Ongoing efforts to bridge this hardware programmability gap [16, 18, 25, 37] can be largely split into two classes: high-level synthesis (HLS) tools and high-level RTL (HL-RTL) languages. On the one hand, HLS tools (such as Vivado [38], Catapult [12], and others [17, 23]) rely on programming languages like C and incorporate auto-pipelining and optimization mechanisms to make hardware accelerators accessible for non-hardware engineers. While this approach is successful in algorithmic acceleration domains, such languages carry von Neumann sequential semantics and thus hinder construction of parallel hardware, which is crucial for hardware design [39]. Moreover, some trivial periodic hardware operations (like toggling a LED) are unbearably difficult to implement in HLS languages. On the other hand, HL-RTL languages (such as Chisel [5], Bluespec [26], PyRTL [8], and others [7, 9, 15, 20, 21, 35]) aim to enhance productivity by introducing new hardware generation

constructs and semantics but do not abstract away register-level description (even Bluespec, which uses concurrent guarded atomic actions, assumes rules complete within a single clock cycle). Therefore, HL-RTL designs are still subjected to the “*tyranny of the clock*” [34] and therefore bound to specific timing and target constraints.

In this paper we propose dataflow-based HDL abstractions for registers and clocks. We also introduce our extended work on DFiant, a Scala-embedded HDL that utilizes these dataflow abstractions to decouple functionality from implementation constraints. An early version of DFiant was first introduced by Port and Etsion [30] and mainly focused on semantic differences between RTL, HLS and dataflow languages. Our work completed important missing semantics and language constructs like expressing state, and further enhanced the language and compiler to support various use-cases. DFiant brings together constructs and semantics from dataflow [4, 13, 19, 36], hardware, and software programming languages to enable truly portable and composable hardware designs. The dataflow model offers implicit concurrency between independent paths while freeing the designer from explicit register placement that binds the design to fixed pipelined paths and timing constraints.

Recent related dataflow-for-hardware efforts are the Maxeler framework [29] and its MaxJ Java-based programming language, the OpenDF framework [6] which is based on the CAL actor language [11], and CAPH [33]. MaxJ indeed shares common traits with DFiant, but it is tailored for its target hardware framework and is not designed to be a general purpose HDL. Both OpenDF and CAPH share similar goals with our work, but they use actors and networks to describe hardware, which is completely different than a conventional HDL composition based on component instances and port connections.

This work focuses on applying dataflow principles through the DFiant language and compiler. DFiant is *not* an HLS language, nor is it an RTL language. Instead, DFiant is an HDL that provides abstractions beyond the RTL behavioral model, which reduce verbosity and maintain portable code. Since DFiant is implemented as a Scala library, it offers a rich type safe ecosystem alongside its own hardware-focused type system (e.g., bit-accurate dataflow types, input/output port types). The library performs two main tasks: first, the frontend compilation, which enforces the type-safe rule-system and constructs a dataflow dependency graph; and second, the backend compilation, which translates the graph into pipelined RTL code and a TCL constraints file. The resulting code can be synthesized using commercial tools.

The remainder of this paper is organized as follows. The next section details the motivation behind the dataflow HDL abstractions, and Section 3 provides a general overview of the DFiant HDL language. Section 4 briefly outlines the main DFiant compiler operations. Section 5 describes our evaluation of the DFiant language and compiler, and, finally, Section 6 concludes the paper and discusses future work.

2 A DATAFLOW HARDWARE DESCRIPTION ABSTRACTION

In this section we detail how dataflow abstractions help decouple the functionality from its constraints. We also overview what

dataflow HDL constructs are required to achieve maximum portable code. In the next section we demonstrate how these constructs are used in DFiant.

Fig. 2 summarizes the basic elements that make up HDLs at different abstraction layers, from a netlist up to the dataflow constructs presented in this paper. Each layer includes the expressive capabilities of the lowest layer (e.g., structural instance composition is possible in all HDLs). The layers are tagged with the relevant HDL names. Note that HLS languages and simulation constructs are not included in this summary.

The basic notion of a dataflow abstraction is that instead of wires and registers we have dataflow token streams. This key difference between RTL and dataflow abstractions reveals why the former is coupled to device and timing constraints, while the latter is agnostic to them. Primarily, *the RTL model requires designers to express what operations take place in each cycle, whereas the dataflow model only require the designer to order the operations based on their data dependencies*. More specifically, the RTL model utilizes combinational operations that must complete (their propagation delay) within a given cycle if fed to a register, while the dataflow abstraction only assumes order and not on which cycle operations begin or complete. By decoupling operations from fixed clock cycles the dataflow model enables the compilation toolchain to map operations to cycles and thereby independently pipeline the design.

Furthermore, the RTL model requires designers to use registers for a variety of uses and thus binds the design to specific timing conditions. Specifically, we find three main uses for registers in the RTL model: *synchronous technology backend*, *synchronous technology interface*, and *design functionality* (i.e., state). We now turn to discuss these different uses for registers and how the dataflow model can derive the first two uses without explicit user description.

2.1 Synchronous Technology Backend Registers

Registers are often required in a low-level design due to the underlying synchronous technology. Since they are unrelated to the functional requirement, a dataflow HDL can derive them automatically based on the functional requirements and design constraints. We differentiate between the following backend uses of registers:

2.1.1 Pipelining and Path-Balancing. Pipeline registers are inserted to split long combinational paths, and their placement is determined by designer-specified constraints, such as the maximum path cycle latency or the maximum propagation delay between registers. Pipelining increases the path cycle latency, and if the path converges with another path that requires no pipelining, then additional path-balancing registers are added to maintain correctness of the design. Because a balanced pipelining does not affect the design functionality, it can be automatically applied by the dataflow HDL compiler.

2.1.2 Synchronizers. Clock domain crossing (CDC) and asynchronous signals are exposed to metastability. Synchronizers, often composed of registers, are used to mitigate its effect and bring the design to the proper reliability. Since we wish to have a clockless design frontend, we want the synchronizers to be implicit. A dataflow HDL compiler needs to infer synchronizers according to the design

Dataflow HDL DFiant		
High-level RTL Chisel , SpinalHDL , VeriScala , PyRTL , Migen , MyHDL , Bluespec , Cx		
RTL VHDL , Verilog , SystemVerilog		
Netlist/Gate-level Components, Instances, Ports, Wire Connections, Hierarchy/Flat/Folded	* <u>Combinational Operations</u> : Arithmetic, Logic, Conditional * <u>Registers</u> : Pipeline, Path-Balance, Derived State, Regular State, Time Delay, Clock Gen, Synchronizer * <u>Clocks</u> : External, PLL (via blackbox) * <u>Resets</u> : Async, Sync, Active High/Low	* <u>Ordered Operations</u> : Arithmetic, Logic, Conditional * Dataflow History Access * Auto Pipelining, Auto Flow-Control * Timers

*RTL foundations do not change,
but hardware is easier to
generate or can be implicit like
clocks and resets*

Figure 2: HDL abstraction layer summary (lowest=netlist, highest=dataflow)

Each layer subsumes the capabilities of the layer below it. Dataflow constructs replace RTL registers with their true functionality (e.g., state) or inserts them implicitly (e.g., pipelining).

constraints without designer intervention. Note: our work currently focuses on single clock designs so the compiler we implemented does not yet support this feature.

2.2 Synchronous Technology Interface Registers

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 affect the interface and are unrelated to the design itself. To maximize design portability, we apply timed or legacy constructs *solely in the periphery*, while coding the design core with only clock-less dataflow constructs. We differentiate between the following synchronous signaling:

2.2.1 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). A dataflow HDL supports legacy RTL constructs to synchronously interface external IOs and instantiate blackboxes.

2.2.2 Timers. Timers are design constructs for generating real-time signals or creating derivations of timed signal inputs. For example, a design using a 100MHz clock may drive a UART stream at 10Mbps or toggle a led at 1Hz. Rather than directly using registers as clock dividers or employing clock generation components (e.g., PLLs), one can create functional representation of their timed use-cases. A dataflow HDL has timer constructs that generate tokens at a given or derived rate. The compiler can take all clocks into consideration and generate the proper clock tree based on the available device resources and other design constraints.

2.3 Design Functionality (State) Registers

Functional registers, or state, are needed when a design must access (previous) values that are no longer available on an input signal (e.g., cumulative sum or a state-machine's state). RTL designs invoke registers (behaviorally) to store the state. But, registers not only store the state, but also enforce specific cycle latencies. Furthermore,

typical RTL languages declare additional variables and place extra assignments just to save the state. A dataflow HDL overcomes all these issues by including a construct to reuse a token from the stream history. Additionally, a related construct should set a token history to be used at initialization time. We differentiate between two kinds of state: *derived state*, and *feedback state*.

2.3.1 Derived State. A derived state is a state whose current output value is *independent* of its previous and can thereby be deduced by the compiler. For example, checking if a dataflow stream value has changed requires reusing the previous token and comparing to the current token.

2.3.2 Feedback State. A feedback state is a state whose current output value is *dependent* on its previous state value. For example, the current cumulative sum value is dependent on the previous sum value. Therefore, a dataflow HDL requires not only to fetch previous token values, but also set the future state value. Addressable memory pools also hold feedback state (e.g., a processor register-file, memory blocks) and can be expressed as a large selectable state array or available dedicated memory components.

The two kinds of state differ heavily in performance improvement when the design is pipelined. A derived state path can produce a token for every clock tick, and pipelining a combination operation to reduce its cycle time will also increase its throughput. In contrast, a feedback state path is circular and cannot be pipelined as-is.

Feedback state causes bottlenecks in many systems. For instance, a RISC-V processor program counter (PC) register manifests as a feedback 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 on dataflow abstractions that solve such problems functionally.

```

1 import DFiant._
2
3 trait MA4 extends DFDesign {
4   val a = DFSInt[16] <> IN init 0
5   val b = DFSInt[16] <> IN init 0
6   val c = DFSInt[16] <> IN init 0
7   val d = DFSInt[16] <> IN init 0
8   val o = DFSInt[16] <> OUT
9
10  def ma(src : DFSInt[16]) = {
11    val acc = DFSInt[18] init 0 //--Compiled to->
12    acc := acc - src.prev(4) + src //--Compiled to->
13    (acc / 4).toWidth(16)
14  }
15  def avg2(src1 : DFSInt[16], src2 : DFSInt[16]) =
16    ((src1 + src2).wc / 2).toWidth(16)
17    // (_ + _).wc is a with-carry addition
18
19  o := avg2(
20    avg2(ma(a), ma(b)), avg2(ma(c), ma(d))
21  )
22 }
23
24
25
26
27
28
29
30
31
32 object MA4App extends DFApp.VHDLCompiler[MA4]

```

Figure 3: The MA4 DFiant code
Concise and portable

```

...
signal acc : signed(17 downto 0);
signal acc_prev1 : signed(17 downto 0);
signal src_prev1 : signed(15 downto 0);
signal src_prev2 : signed(15 downto 0);
signal src_prev3 : signed(15 downto 0);
signal src_prev4 : signed(15 downto 0);
...
sync_proc : process (CLK, RSTn)
begin
  if RSTn = '0' then
    acc_prev1 <= 18d"0";
    src_prev1 <= 16d"0";
    src_prev2 <= 16d"0";
    src_prev3 <= 16d"0";
    src_prev4 <= 16d"0";
  elsif rising_edge(CLK) then
    acc_prev1 <= acc;
    src_prev1 <= src;
    src_prev2 <= src_prev1;
    src_prev3 <= src_prev2;
    src_prev4 <= src_prev3;
  end if;
end process sync_proc;
...
async_proc : process (all)
  variable v_acc : signed(17 downto 0);
begin
  v_acc := acc_prev1;
  v_acc := v_acc - src_prev4 + src;
  acc <= v_acc;
end process async_proc;

```

Figure 4: The MA4 DFiant lines 11-12 compiled to VHDL
The DFiant code is extremely compact in comparison.

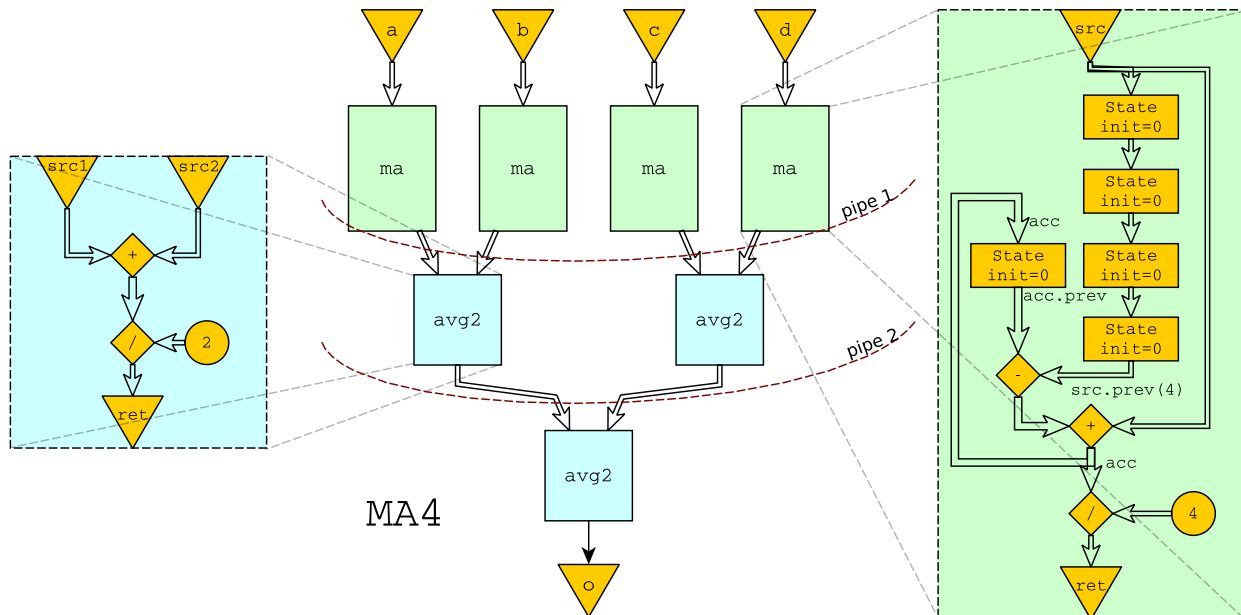


Figure 5: The MA4 dataflow graph (the inputs are **a**, **b**, **c**, **d** and the output is **o**)
The entire design is a composition of the **ma** and **avg** functions (detailed blowouts are depicted as well).
The compiler places pipeline tags to achieve the required performance and the backend inserts registers accordingly.
The concurrent construction is implied from a sequential composition thanks to the dataflow abstraction.

3 THE DFiant LANGUAGE OVERVIEW

DFiant is a Scala library and thus possesses various rich type safe language constructs. DFiant also incorporates unique language semantics that enable dataflow-based hardware description. Throughout this section we elaborate on these constructs and semantics via our running example, a four-by-four moving average (MA4) unit. The MA4 has four 16-bit integer input channels and is required to output the average of all channels, while each channel is averaged by a four-sample moving window continuously. The complete MA4 DFiant implementation and its dataflow graph are available in Fig. 3 and 5, respectively. Fig. 4 presents a subset of the DFiant-generated VHDL (2008) code derived from lines 11-12 in Fig. 3.

3.1 Hello DFiant World!

The DFiant code in Fig. 3 demonstrates the structure of any DFiant program: it imports the `DFiant` library (line 1), creates the top-level design by extending the `DFDesign` abstract class (lines 3–21), and creates a runnable application that instantiates the top design trait, and compiles it into a VHDL file (line 32).

The MA4 design is fairly straightforward. Lines 4–8 generate the signed dataflow ports and include a `0` value initialization (see Section 3.3). Lines 10–14 define the function `ma` that generates a single four-sample moving average, while lines 15–16 define the function `avg2` that generates a two-input average unit. Finally, lines 18–20 compose `avg2` and `ma` to generate the entire MA4 functionality and assign it to the output port `o`. We elaborate on the unique DFiant constructs and semantics in the next sections.

3.2 Dataflow Variable Semantics

DFiant code is expressed in a sequential manner yet employs an asynchronous dataflow programming model to enable an intuitive concurrent hardware description. For this purpose, DFiant applies the following rules:

3.2.1 Concurrency and Execution Order. Concurrency is implicit and the data scheduling order, or *token-flow*, is set by the *data dependency*. DFiant schedules all independent dataflow expressions concurrently, while dependent operations are synthesized into a guarded FIFO-styled pipeline. The MA4 dataflow graph in Fig. 5 demonstrates the concurrent paths constructed from the dataflow dependency.

3.2.2 Basic Operations. Each application of an arithmetic/logic operator is translated into the appropriate hardware construction and applies a dataflow *join* on their arguments. The arguments require a valid token for consumption to produce a new token generated from the operations. For example, `+` in `avg2` joins `src1` and `src2` and requires a token from both to produce the token `src1 + src2`.

3.2.3 Path Divergence. Diverging paths are implicitly *forked*, so token production is possible if all target nodes are ready to consume the token. For example, `acc` result in `ma` is forked into a division operation and the state feedback. It is impossible to consume an invalid token and once a token is consumed it is invalidated.

3.2.4 Constants. Any Scala primitive value is considered as a constant when applied as an argument to a dataflow operation. For example, the value `2` in `avg2` is a primitive `Int` and is considered a constant in the division operation. Semantically, a constant is an infinite token generator that produces a new token with the same initial value each time the token is consumed.

3.2.5 Pruning. Unused nodes always consume tokens and are discarded during compilation.

3.2.6 Bit-Accurate Operations and Type Inference. DFiant supports various basic dataflow types such as `DFBool`, `DFBits`, and others. All DFiant's dataflow types are bit-accurate and structurally static, with their bit-width set upon construction (e.g., at line 11 of `MA4` the variable `acc` is an 18-bit signed integer). Operations between dataflow variables produce a bit-accurate result with the proper type inference. For example, at line 12 of `MA4`, a 16-bit value is subtracted from an 18-bit value which results in an 18-bit value. In line 16 the addition between `src1` and `src2` returns a carried 17-bit result due to invocation of `.wc`. DFiant also employs various type-safe restrictions to protect designers from unexpected or surprising behavior (e.g., it is unclear what should occur at `DFUInt[4] - 1000`).

3.2.7 Assignment and Mutability. DFiant supports dataflow variable "mutation" via the `:=` operator. This not an actual mutation but only a change of the variable reference to a different dataflow stream (hardware *construction* is immutable). Assignments are also used inside conditional constructs (Section 3.4) and to set the input of a feedback state (Section 3.3.5).

Dataflow variables can be either mutable (accept `:=`) or immutable (compiler error for `:=`). The mutability trait depends on the type and sometimes also on the context within they are applied. For example, we cannot assign to a the dataflow stream returned at line 13 of `MA4`. The only mutable variables are the ones explicitly constructed or output ports as seen at lines 11 and 8 of `MA4`. Mutation must be local to the context of assignment operation (e.g., cannot assign to variable outside its design). Note: do not confuse with Scala-level mutation which is enabled by using `var` instead of `val` (the DFiant compiler disallows constructing DFiant objects under a `var` reference).

3.2.8 Read Access. All dataflow values can be read (yes, even output ports). The only exception exist in an output port that is neither assigned or initialized because in this situation the reader will only consume stall bubble tokens and possibly lead to a system deadlock.

3.2.9 Bit Aliasing and Casting. In general, aliasing is a mechanism to reference a cluster of one or more dataflow variable parts and cast it to any dataflow variable type that has the same bit width. The most trivial aliasing construct is `.bits(hi, lo)` that selects part of any dataflow variable from bit `hi` downto to bit `lo` and casts it to a bit vector. The aliasing mechanism implements bit concatenation, selection, shifting, reversal, and inversion. Aliasing preserves the mutability property: an alias of an immutable variable is immutable, while an alias of a mutable variable is mutable. Mutating an alias is equivalent to mutating the original aliased

variable(s) at the relevant bit indexes. Aliasing of an alias is also possible, while maintaining relative bits indexing.

3.3 State Constructs and Semantics

In contrary to RTL languages, DFiant does not directly expose register and wire constructs. Instead, DFiant assumes every dataflow variable is a stream and provides constructs to initialize the token history via the `.init` construct, reuse tokens via the `.prev` construct, and update the state via the `:=` construct. Lines 11-12 in Fig. 3 along with their compiled VHDL representation in Fig. 4 demonstrate the state semantics as follows:

3.3.1 Initialization. The `.init` construct is accompanied by one or more token values and only sets the initial state history. For example, line 11 constructs a dataflow variable and initializes all of its history to zero value tokens. The history sequence is typed from left to right and sets tokens from newest to oldest, respectively. If the sequence is accessed beyond the oldest token then the oldest token is repeatedly produced. If the history is empty (not initialized) then stall bubble tokens are produced when accessed (see Section 3.3.4). It is also possible to directly place bubble tokens in the history by writing either `φ` or `Bubble`.

3.3.2 History Access. The `.prev` construct reuses the previous state token. The very first "reused" token is the one set via `.init`. It is also possible to call `.prev(step)` with a step number argument to reuse older stream values. For any dataflow value `a` and a given positive integer `step`, a call to `a.prev(step)` is equivalent to repeated `<step>` applications of `a.prev.prev...prev`. For example, in line 12 we reuse a `src` token from four steps ago. If the `src` token stream is "1, 2, 3, 4, ..." with a 0 initialization, then the `src.prev(4)` token stream is "0, 0, 0, 0, 1, 2, 3, ...".

3.3.3 Distributive Property. The history access via `.prev` is distributive over all the basic DFiant operations. For example, the distributed `a.prev + b.prev` is equivalent to `(a + b).prev`. This means that the initialization history is also respectively affected from these basic operations.

3.3.4 Stall Bubbles. Invoking `.prev` on an uninitialized dataflow variable generates a stall bubble. Stall bubbles are consumed and produced like any other token, yet a basic operation with a stall bubble token must produce a stall bubble token. Additionally, stall bubbles do not affect a feedback state. The backend compiler is responsible to generate the additional logic required for existing design stalls.

3.3.5 State Update. The new updated token is pushed into a dataflow stream by using the `:=` assignment operator. There can be more than one assignment to same variable, however only the last assignment updates the state and occurs when all dependent dataflow firing rules are satisfied. This rule is similar to signal update semantics in VHDL processes.

3.3.6 Default Self-Generation. Any dataflow variable `a` has an implicit self-assignment `a := a.prev` that comes immediately after the variable construction. This creates an equivalent reference

between `a` and `a.prev` which leads to a more intuitive programming. For example, in line 12 we used `acc - ...` and not `acc.prev - ...`, since both expressions are equivalent.

Another example that illustrates the benefits of the DFiant state constructs is the 32-bit Fibonacci series generator in Fig. 6. The dataflow version is shorter than its VHDL counterpart [3] and greatly resembles the formal Fibonacci definition: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

```

1  trait FibGen extends DFDesign {
2    val o = DFUInt[32] <> OUT
3    val f = DFUInt[32] init (1, 0)
4    f := f.prev + f.prev(2)
5    o := f.prev(2) //output from 0
6  }

```

Figure 6: A DFiant 32-bit Fibonacci series generator
Great resemblance to the formal Fibonacci definition

Both Fig. 4 and Fig. 6 emphasize the advantages of DFiant state constructs over RTL registers and wires. One advantage is that the DFiant code resembles its RTL counterparts, but is also very concise since state elements are automatically constructed when a stream history is accessed. Another advantage is portability, because state elements are not registers and therefore any type of state component is applicable. Our first synchronous backend indeed maps state elements to registers, but even an asynchronous backend can compile the same code and apply the Muller C-element [24] as a state element.

3.4 Conditional Constructs and Semantics

DFiant has `ifdf` and `matchdf` conditional constructs which logically resemble the Scala `if` and `match`, respectively. These dataflow conditional constructs are also very similar to their RTL counterparts and yield multiplexer hardware. However, their semantics follow the same dataflow *join* and *fork* rules we described in Sections 3.2.2 and 3.2.3, and therefore all dataflow values used within the conditional constructs are joined together and all modified dataflow variables which are referenced outside the conditional constructs are forked as well. Of course, if there is no need for this additional flow control logic, the default compiler yields a simple combinational structure.

Fig. 7 demonstrates a simple DFiant finite state-match (FSM) that utilizes these conditional constructs to detect the sequence "1001". The VHDL [2] implementation is very similar but also a bit longer due to the split between the sequential and combination parts, which is applied automatically by the DFiant compiler. Note we did not need to directly refer to `state.prev` at line 8, thanks to the default self generation described in Section 3.3.6.

3.5 Hierarchy and Connectivity

So far, all examples demonstrated a single dataflow design without any hierarchies. The MA4 design managed to create structural encapsulations and composition via Scala definitions. Earlier, in Section 2, we stressed the importance of any HDL to support hierarchies via constructs that describe components, ports and their


```

1 trait SeqDet extends DFDesign {
2   val seqIn = DFBool() <> IN
3   val detOut = DFBool() <> OUT
4   object State extends Enum.Auto {
5     val S0, S1, S10, S100, S1001 = Entry
6   }
7   val state = DFEnum(State) init State.S0
8   matchdf(state)
9   .casedf(State.S0) {
10     detOut := 0
11     ifdf (seqIn) {state := State.S1}
12     .elsedf {state := State.S0}
13   }.casedf(State.S1) {
14     detOut := 0
15     ifdf (seqIn) {state := State.S1}
16     .elsedf {state := State.S10}
17   }.casedf(State.S10) {
18     detOut := 0
19     ifdf (seqIn) {state := State.S1}
20     .elsedf {state := State.S100}
21   }.casedf(State.S100) {
22     detOut := 0
23     ifdf (seqIn) {state := State.S1001}
24     .elsedf {state := State.S0}
25   }.casedf(State.S1001) {
26     detOut := 1
27     ifdf (seqIn) {state := State.S1}
28     .elsedf {state := State.S10}
29   }
30 }

```

Figure 7: A DFiant "1001" sequence detector FSM
Very similar to its VHDL counterpart

connections. These DFiant features surpass the synthesizable capabilities of traditional HDLs, and are noticeably absent from HLS languages such as C++.

3.5.1 Dataflow IO Ports. Dataflow IO ports act in many ways just like dataflow variables. Ports are used either to connect a parent design to its child or connect two child sibling designs. The operator `<>` is used both to construct dataflow ports (from dataflow variables) and create connections between ports. Dataflow ports can be either `IN` or `OUT` as seen at lines 7–8 of `MA4` and can also be initialized via `.init`. Input ports are immutable. Output ports are mutable, but only within the context of their owner design. There are various connectivity rules and there are important differences between a connection `<>` and an assignment `:=`. See Table 1 for a brief comparison.

3.5.2 Dataflow Designs. Dataflow designs are the basic hierarchy abstraction in DFiant. By extending the abstract class `DFDesign` the designer populates it with IO ports that makeup its interface and with dataflow variables and operations that makeup its implementation. 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 difference between a Scala hierarchy and a DFiant hierarchy itself is that pure Scala constructs are transparent to the dataflow graph, as if the entire design is flattened, inlined, and unrolled.

Figures 8, 9, and 10 showcase the flexibility when applying both DFiant hierarchies and Scala inheritance. Indeed these capabilities

are no strangers to HL-RTLs such as Chisel, and may even be expressible by native RTLs in some cumbersome fashion, but all these languages still couple the design to its specific timing and target constraints. For example, `NWParCross` should have larger PD than `NWParDirect`. On the one hand, pipelining inside the `NWLeft` and `NWRight` may result in over-pipelining, while on the other hand, pipelining outside, in their parent designs, may cause the same. DFiant helps us describe a truly agnostic network and *optionally* leave the hard work to the compiler.

3.6 Simulation Constructs

Unlike VHDL and Verilog which were developed for simulation and only later adopted for synthesis, DFiant was developed with a synthesizable language focus in mind. Nonetheless, to test our designs we added some basic simulation-only constructs, and to clearly separate them from the rest of the language, we placed them under a separate name space – `sim`. Furthermore, all simulation constructs exist only within a simulation context which is formed only if the toplevel design is actually a simulation design, `DFSimulation`. The `DFSimulation` construct typically holds both the device-under-test (DUT) and the testing logic. Currently, the DFiant compiler only supports a handful of dedicated simulation constructs. Fig. 11 demonstrates basic testing methods in DFiant, like using `.prev` to inject token sequences (lines 2,3,5) and easily constructing strings via string interpolation (line 6).

```

1 trait SeqDetTest extends DFSimulator {
2   val TestSeq = Seq(1, 1, 0, 1, 0, 0, 1, 0, 1)
3   val seqIn = DFBool() init TestSeq.reverse
4   val dut = new SeqDet {}
5   dut.seqIn <> seqIn.prev(TestSeq.length)
6   sim.report(dfs"det: ${dut.detOut}")
7 }

```

Figure 11: Simulation wrapper for our sequence detector

4 THE DFIANT COMPILER

Like many compilers, the DFiant compiler is built from a frontend compiler and a backend compiler. The frontend compiler applies all language rules and semantics to produce a workable context that represents the design. This context can then be used by the backend compiler for various purposes. Currently we implemented only two backend compilers. One recompiles the context into DFiant code, for debugging purposes, and the other compiles the context into VHDL code.

4.1 The Frontend Compiler

The DFiant frontend compiler relies mostly on the Scala compiler since DFiant is embedded as Scala library, and therefore has strong type-safe protection. DFiant is an early-adopter of new Scala features such as literal types [27] and operations [31], 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). After the Scala compiler finishes compilation and produces a runnable executable, the main program loads the DFiant

```

1 //All networks have the same interface
2 trait Network extends DFDesign {
3   val iT = DFSInt[16] <> IN //Top
4   val iB = DFSInt[16] <> IN //Bottom
5   val oT = DFSInt[16] <> OUT //Top
6   val oB = DFSInt[16] <> OUT //Bottom
7 }
8 //State in Top Connection
9 trait NWLeft extends Network {
10   iT <> oT //Connection
11   oB := iB * 3 //Assignment
12 }
13 //State in Bottom Assignment
14 trait NWRight extends Network {
15   iT <> oT + 5 //Connection
16   oB := iB.prev //Assignment
17 }
18 //Parent box includes two sibling boxes
19 trait NWParent extends Network {
20   iT init (5, 7) //Initializing history
21   iB init (2, 6) //Initializing history
22   val nwL = new NWLeft {}
23   val nwR = new NWRight {}
24   nwL.iT <> iT
25   nwL.iB <> iB
26   nwR.oT <> oT
27   nwR.oB <> oB
28 }
29 //Direct connections between siblings
30 trait NWParDirect extends NWParent {
31   nwL.oT <> nwR.iT
32   nwL.oB <> nwR.iB
33 }
34 //Cross connections between siblings
35 trait NWParCross extends NWParent {
36   nwL.oT <> nwR.iB
37   nwL.oB <> nwR.iT
38 }

```

Figure 8: Various network designs
Featuring hierarchies and inheritance

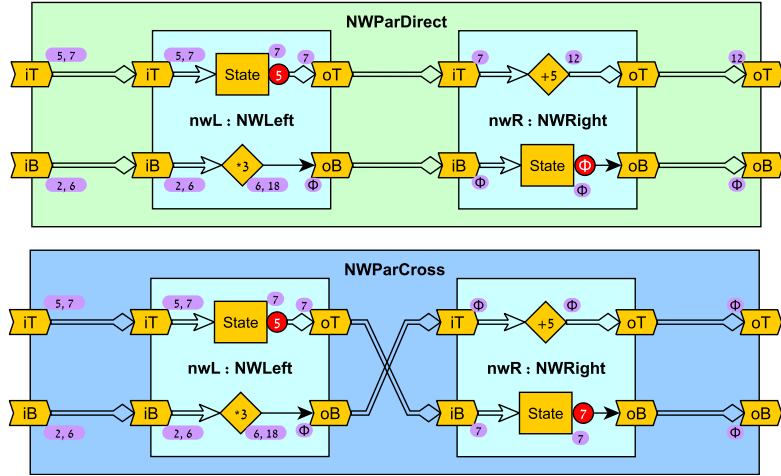


Figure 9: Hierarchical drawing of NWParDirect and NWParCross
The purple numbers mark the initialization history as it is propagated according to the semantic rules of DFiant. The red numbers mark initial tokens generated by the state elements.

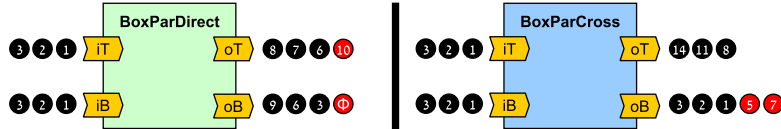
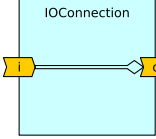
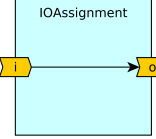


Figure 10: Input/Output token flow example to/from the two parent boxes (the rightmost tokens are the oldest).

Table 1: Connection <> and Assignment := Operator Comparison

Criteria	Connection <>	Assignment :=
Code	<pre> trait IOConnection extends DFDesign { val i = DFUInt[8] <> IN val o = DFUInt[8] <> OUT o <> i } </pre>	<pre> trait IOAssignment extends DFDesign { val i = DFUInt[8] <> IN val o = DFUInt[8] <> OUT o := i } </pre>
Functional Diagram	 <p>A double line arrow indicates a dataflow dependency with an initial condition dependency.</p>	 <p>A single line arrow indicates a dataflow dependency without affecting initial conditions of the consumer.</p>
Directionality and Commutativity	The operator is commutative, meaning $a <> b$ is equivalent to $b <> a$. One argument is the <i>producer</i> , while the other is the <i>consumer</i> . The dataflow direction is sensitive to the context in which the operator is applied.	The operator is non-commutative, meaning $a := b$ determines that b is the <i>producer</i> , transferring data to the <i>consumer</i> a .
Initialization	Initialization is transferred to the consumer. If the consumer has both initialization via <code>.init</code> and connection, then the <code>.init</code> is the one that takes effect.	The consumer initialization is not affected.
Mutation	A consumer can only be connected once at each bit.	The same bit in a consumer can be assigned to more than once.
Statement Order	Connections statements can be placed in any order.	Assignments affect the scope reference of the modified dataflow variable, and therefore their order is highly important.

compiler to continue the rest of the compilation process (e.g., connectivity rules are enforced during this phase). After this process DFiant holds the top-level design context and is ready to initiate a backend compilation phase.

4.2 The Backend Compiler

The dataflow abstraction enables designers to describe hardware without explicitly pipelining the design. The DFiant backend compiler can optionally pipeline the design by itself and place registers to split long combinational paths. The compiler works with a propagation delay (PD) estimation database that can be tailored for any target device and technology. With this information and a target clock constraint the compiler tags the dataflow graph with the additional pipe stages required before producing the RTL code. One possible tagging is depicted in Fig. 5, in which two pipe stages were added between the large operations. Depending on the availability of DSP blocks in the target device, it is also possible to break the basic operations to multiple cycles by instantiating the proper vendor IP (e.g., a long multiplication operation should require several cycles). All of these target-specific adaptations are done without designer intervention and thus make any DFiant design highly portable.

To maintain design correctness the compiler adds path-balancing registers when pipeline registers are added and different-latency paths converge. Since these two features are separate, we can allow designers to explicitly place pipe stages in critical junctions should our PD estimation fail. The `.pipe` construct adds a pipe stage at a specific node and the compiler will balance the rest of the converging paths. While both `.pipe` and `.prev` constructs appear similar, the `.prev` construct does not affect the path-balancing mechanism. For example, `x - x.prev` creates a derivation circuit while `x - x.pipe` results in a constant zero since the path-balancing applied at the subtraction input arguments manifests as a `x.pipe - x.pipe` operation.

The asynchronous nature of DFiant means the compiler can adapt the design to any FIFO ready-valid signaling for automatic flow-control. For example, the MA4 RTL interface can have ready-valid signaling to each of its input and output ports to allow back-pressure. To achieve this manually in RTL is extremely error-prone, while the DFiant generated RTL code stays true to the original dataflow description, and therefore, correct.

In Section 2 we discussed the problem when attempting to pipeline a feedback state. The `ma` function creates the feedback state referenced via the `acc` variable. The `ma` blowout in Fig. 5 exposes this problem where a circular feedback updates the `acc` state. This feedback cannot be pipelined as-is because path-balancing will never be able to satisfy the balancing rule due to circular path dependency. It is only possible to increase the clock rate in a feedback circuitry by applying multi-cycle or speculative logic (e.g., a RISC-V processor core contains several feedback junctions like the PC update and therefore has single-clock, multi-cycle and speculation-based pipelined implementations).

5 EVALUATION

DFiant aims to greatly improve designer productivity. To evaluate possible productivity gains we chose various open-source use-cases and implemented them in DFiant: an AES cypher [14], an IEEE-754 double precision floating point multiplier [22], a two-stage RISC-V core [32], a bitonic-sort network [1], a cyclic redundancy check (CRC) generator/checker [10], and other simple examples [2, 3, 28]. We compared the use-cases and our implementations via the following metrics: the maximum clock frequency, pipeline latency, utilizations of both look-up tables (LUTs) and flip-flop registers (FFs), and lines of code (LoC). We documented all results in Table 2. Apart from the LoC, all other metrics were obtained from a Vivado synthesis report. For maximum frequency, we introduced additional IO registers and recorded the longest path between registers. These extra registers effect was removed from the latency and FFs columns.

Also included in Table 2 are the pipelining methods we chose¹ and if our design is also hardware proven. The pipelining criteria is as follows: *None*, if the design is pure combinational; *Not Applicable*, if the design cannot be pipelined any further due to feedback; *Manual*, if the design is pipelined using the `.pipe` construct; and finally *Auto*, if the design is pipelined automatically by running the compiler with a maximum frequency constraint. From inspecting the results and the code we deduce as follows:

Conciseness The DFiant code is often significantly more compact than the RTL code (from 50% to 70% less code), because DFiant semantically implies much of the additional information required by the RTL description. The only difference occurs at the RISC-V implementation. The reason is that the RTL code design is flat, while we invested substantial code in hierarchical abstractions. We already came up with a solution to reduce this verbose code significantly, but have yet to implement it.

Portability, Performance, and Correctness An RTL implementation for a given functionality is but one of many possible RTL representations that vary when different device and performance constraints are required and therefore can be considered *correct* only when those conditions are true, whereas the DFiant design remains unchanged and can be considered always correct once verified.

Debugging and Legibility A DFiant design generated RTL code maintains a readable, true to the source, format, by maintaining the intended names and structure expressed in the original DFiant design, unlike cryptic codes generated by many HLS and high-level design tools. However, it is still harder to debug RTL code generated by DFiant than RTL code written by a human designer.

6 CONCLUSION

In this paper we proposed a new dataflow hardware description abstraction layer that goes beyond RTL to free designs from being coupled to specific device or timing constraints. This abstraction layer tosses aside the register and wire RTL foundation blocks in favor of dataflow principles discussed in Section 2. We also evolved the DFiant HDL and compiler to support dataflow state and many

¹ We did not include results of pipelining designs further than their RTL counterparts. We believe this comparison, although interesting, is out of scope for this paper which focuses on the language and not the compiler.

Table 2: Comparing various RTL codes with equivalent DFiant codes in terms of performance, utilization, and LoCs
The DFiant implementation is usually more concise and yet achieves roughly the same performance

Use Case	Design Source	Pipeline Latency [Cyc]	Max Freq. [MHz]	LUTs [#]	FFs [#]	LoC [#]	LoC Reduction [%]	DFiant Pipelining Method	Hardware Proven	Comment
AES Cypher	DFiant	38	442	3782	11437	334	64	Automatic	Yes	Three key widths supported. Compared at 128-bit.
	Hsing Core	21	454	11103	5386	922				
FP Multiplier	DFiant	23	354	3733	2246	180	47	Automatic	Yes	
	Lundgren Core	24	322	3744	1633	340				
Two-Stage RISC-V	DFiant	2	117	1345	1025	557	-79	Manual	Yes	The RTL code registers are inferred differently and are mapped to LUTRAM.
	Samsoniuk Core	2	103	1163	161	311				
CRC	DFiant	NA	217	23	17	41	60	Not Applicable	No	Compared with the parallel CRC module at 16-bit polynomial and 16-bit data input.
	Drange Core	NA	217	23	17	103				
Fibonacci Gen	DFiant	NA	339	32	30	8	74	Not Applicable	No	
	ExampleProblems	NA	339	32	30	31				
Sequence Detector	DFiant	NA	567	5	3	32	56	Not Applicable	No	Change is due different FSM state encoding inference (one-hot or not)
	FPGA4Student	NA	577	6	6	73				
Moving Average 4x4	DFiant	2	226	217	299	19	74	Manual	No	
	Our RTL	2	226	217	299	72				
Bitonic Sort Network	DFiant	0	33	3424	0	36	60	None	No	
	VLSICoding	0	33	3424	0	90				
Priority Encoder	DFiant	0	178	154	0	10	52	None	No	Compared at 128-bit input
	Kaufmann Core	0	178	154	0	21				

other useful constructs. DFiant provides a seamless concurrent programming approach, and yet it still facilitates a versatile compositional and hierarchical expressiveness.

To evaluate the DFiant language and compiler, we reimplemented various RTL designs in DFiant and compared their performance, utilization, and LoCs. We demonstrated that most DFiant designs have equivalent performance and utilization to their RTL counterparts, yet manage to save between 50% to 70% LoCs. Evidently, the potential to increase designer productivity is significant, but notwithstanding, the greatest potential of DFiant is laid not in its concise syntax, but in its agnostic hardware description.

Future work may explore dedicated simulation backend, an asynchronous logic backend, support for dynamic dataflow, or the missing abstractions like *timers* (see Section 2).

REFERENCES

- [1] 2016. VHDL Code for Bitonic Sorter. <https://vlsicoding.blogspot.com/2016/01/vhdl-code-for-bitonic-sorter.html>
- [2] 2017. Full VHDL code for Moore FSM Sequence Detector. <https://www.fpga4student.com/2017/09/vhdl-code-for-moore-fsm-sequence-detector.html>
- [3] 2019. VHSIC hardware description language. http://www.exampleproblems.com/wiki/index.php/VHSIC_hardware_description_language
- [4] Rishiyur S Nikhil Arvind. 1992. Id: A language with implicit parallelism. In *A Comparative Study of Parallel Programming Languages*. Elsevier, 169–215.
- [5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzyniec, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.
- [6] 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.
- [7] Papon Charles. 2016. SpinalHDL. <http://spinalhdl.github.io/SpinalDoc>
- [8] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McManahan, and Timothy Sh. 2017. A Pythonic Approach for Rapid Hardware Prototyping and Instrumentation. In *Intl. Symp. on Field Programmable Gate Arrays*.
- [9] Jan Decaluwe. 2004. MyHDL: a python-based hardware description language. *Linux Journal* 127 (2004).
- [10] Geir Drange. 2016. Ultimate CRC. https://opencores.org/projects/ultimate_crc
- [11] Johan Eker and Jörn Janneck. 2003. *CAL language report*. Technical Report. Tech. Rep. ERL Technical Memo UCB/ERL.
- [12] Mentor Graphics. 2008. Catapult C synthesis. *Website: http://www.mentor.com* (2008).
- [13] John R. Gurd, Chris C. Kirkham, and Ian Watson. 1985. The Manchester prototype dataflow computer. *Commun. ACM* 28, 1 (1985), 34–52.
- [14] Homer Hsing. 2013. AES Core Specification. <http://opencores.org/usercontent/doc,1354351714>
- [15] Shunning Jiang, Berkin Ilbeyi, and Christopher Batten. 2018. Mamba: closing the performance gap in productive hardware development frameworks. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.
- [16] Nachiket Kapre and Samuel Bayliss. 2016. Survey of domain-specific languages for FPGA computing. In *Intl. Conf. on Field Programmable Logic and Applications*.
- [17] Nikolaos Kavvadias and Kostas Masselos. 2013. Hardware design space exploration using HerculeS HLS. *Proceedings of the 17th Panhellenic Conference on Informatics - PCI '13* (2013), 195. <https://doi.org/10.1145/2491845.2491865>
- [18] Sakari Lahti, Panu Sjöval, Jarno Vanne, and Timo D. Hamalainen. 2019. Are We There Yet? A Study on the State of High-Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (may 2019), 898–911. <https://doi.org/10.1109/TCAD.2018.2834439>
- [19] Paul Le Guernic, Albert Benveniste, Patricia Bournai, and Thierry Gautier. 1986. Signal-A data flow-oriented language for signal processing. *IEEE Trans. on Acoustics, Speech, and Signal Processing* 34, 2 (1986), 362–374.
- [20] 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*.
- [21] D Lockhart, G Zibrat, and C Batten. 2014. Pymtl: A unified framework for vertically integrated computer architecture research. ... (MICRO), 2014 47th Annual ... (2014). http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7011395
- [22] David Lundgren. 2014. Double Precision Floating Point Core VHDL. http://opencores.org/project,fpu_double
- [23] Microsemi. 2015. Symphony Model Compiler ME. (2015).
- [24] DE Muller and WS Bartky. 1957. A theory of asynchronous circuits II. *Digital Computer Laboratory* 78 (1957).
- [25] 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.
- [26] 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*.
- [27] Erik Osheim, George Leontiev, Jon Pretty, Lars Hupel, Mike O'Connor, Miles Sabin, and Tom Switzer. 2017. Typelevel Scala. <https://github.com/typelevel/scala>
- [28] Volnei A Pedroni. 2008. *Generic Priority Encoder from Digital electronics and design with VHDL*. Morgan Kaufmann.
- [29] Oliver Pell and Oskar Mencer. 2011. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News* 39, 4 (2011).
- [30] Oron Port and Yoav Etsion. 2017. DFiant: A Dataflow Hardware Description Language. In *Intl. Conf. on Field Programmable Logic and Applications*.
- [31] Frank S. Thomas, Matthew Pocock, Naoki Aoyama, and Oron Port. 2017. singleton-ops library. <https://github.com/fthomas/singleton-ops>

- [32] Marcelo Samsoniuk. 2019. DarkRISCV: Opensource RISC-V implemented from scratch in one night! <https://github.com/darklife/darkriscv>
- [33] Jocelyn Serot, Francois Berry, and Sameer Ahmed. 2011. Implementing stream-processing applications on fpgas: A dsl-based approach. In *Intl. Symp. on Field Programmable Gate Arrays*. IEEE.
- [34] I Sutherland. 2012. The tyranny of the clock. *Comm. ACM* 55, 10 (2012), 35–36.
- [35] Synflow. 2014. Cx Language. <http://cx-lang.org/>
- [36] Ghislaine Thuau and Daniel Pilaud. 1991. Using the Declarative Language LUSTRE for Circuit Verification. Springer London, 313–331. https://doi.org/10.1007/978-1-4471-3544-9_17
- [37] 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.
- [38] Xilinx. 2015. Vivado High Level Synthesis User Guide. (2015).
- [39] Zhipeng Zhao. 2017. Using Vivado-HLS for Structural Design : a NoC Case Study. In *Intl. Symp. on Field Programmable Gate Arrays*.