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, 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

FPGA'19, February 23–25, 2019, Monterey, CA, USA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Oron Port and Yoav Etsion. 2019. Hardware Description Beyond Register-Transfer Level Languages. In *FPGA '19: International Symposium on Field-Programmable Gate Arrays, February 23–25, 2019, Monterey, CA.* ACM, New York, NY, USA, 8 pages. 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, the complexity of an RTL design implementing 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.

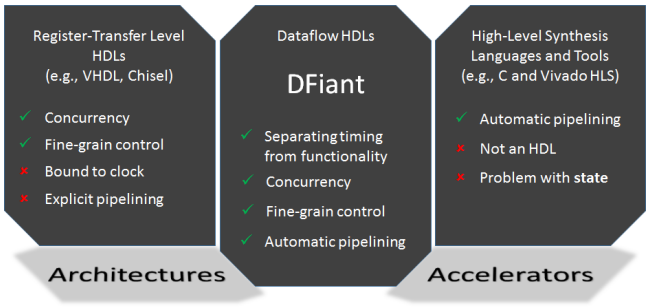

Figure 1: DFiant bridges the gap

Ongoing efforts to bridge this hardware programmability gap [11, 18, 25] 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 [26], Catapult [8], and others [12, 16]) 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 [27]. 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 [2], Bluespec [19], PyRTL [5], and others [4, 6, 10, 14, 15, 23]) 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"* [22] and are bound to specific timing and target constraints.

In this paper we propose dataflow-based HDL constructs that abstract away registers and clocks. We further introduce DFiant[1], a Scala-embedded HDL that utilizes these dataflow constructs to decouple functionality from implementation constraints. DFiant brings together constructs and semantics from dataflow[1, 9, 13, 24], 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 [20] and its MaxJ Java-based programming language, the OpenDF framework [3] which is based on the CAL actor language [7], and CAPH [21]. 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 a 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 which provides a general overview of the DFiant HDL language. Section 5 describes our evaluation of the DFiant language and compiler, and, finally, Section 6 concludes the paper.

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 a clockless design frontend, we want the synchronizers to be implicit. A dataflow HDL compiler needs to infer synchronizers according to

[1] A preliminary version of DFiant was first introduced as a poster. The reference was removed for blind review.

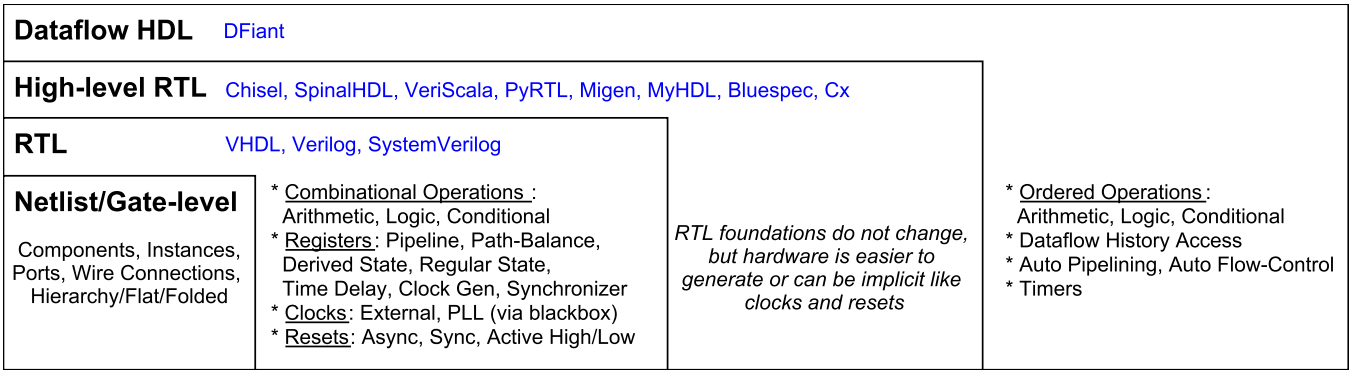

Figure 2: HDL abstraction layer summary (lowest=netlist, highest=dataflow)
Each layer subsumes the capabilities of the layer below it. Dataflow constructs replace RTL registers with their true functionality (e.g., state) or insert them implicitly (e.g., pipelining)

the design 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 clockless 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.

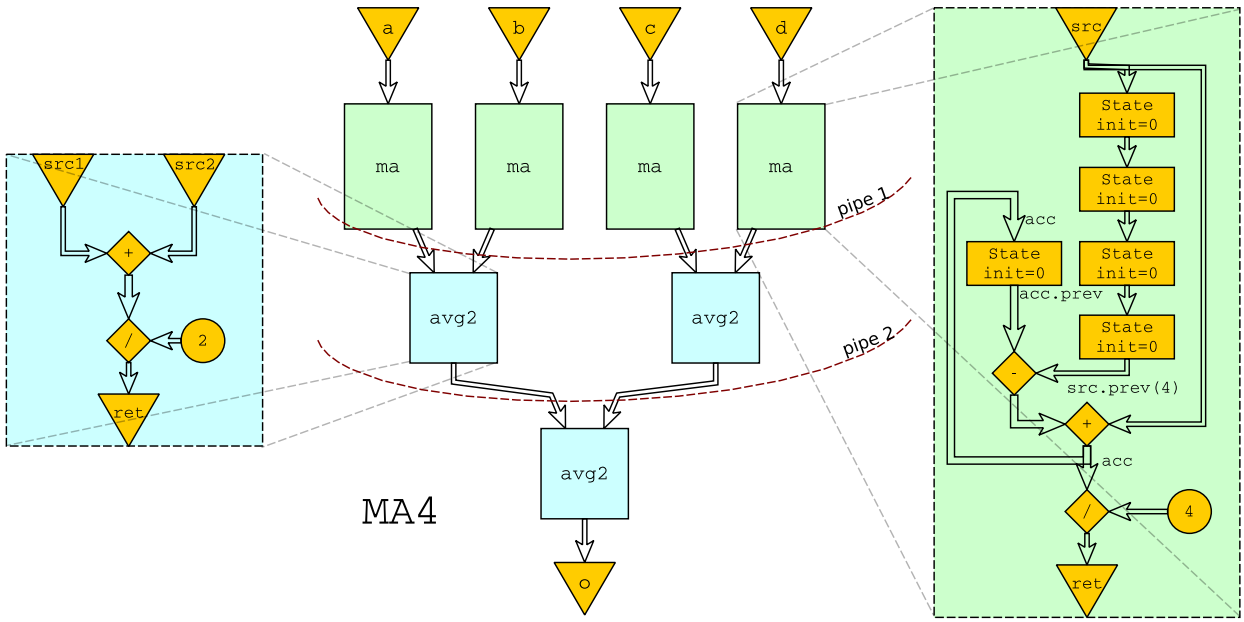

Figure 5: The MA4 dataflow graph (the inputs are `a`, `b`, `c`, `d` and the output is `o`)
The entire design is a composition of the `ma` and `avg` functions (detailed blowouts are depicted as well).
The compiler places pipeline tags to achieve the required performance and the backend inserts registers 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 equivalent 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 MA4 DFiant code in Fig. 3 demonstrates the basics of any DFiant compilation program: it imports the `DFiant` library (line 1), creates the top-level design by extending the `DFDesign` library trait (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 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.

3.3 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 zero value tokens.

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 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 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.

3.3.4 State Update Scheduling. The new updated token is pushed into a dataflow stream by using the `:=` assignment construct. There can be more then 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.

3.3.5 Default Self-Generation. Any dataflow `a` variable 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.

Fig. 4 emphasizes 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 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[17] as a state element.

3.4 Hierarchy and Connectivity

3.5 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).

3.6 Bit-Accurate Operations and Type Inference

DFiant supports various basic dataflow types such as `DFBool`, `DFBits`, `DFUInt`, and `DFSInt`. 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. 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 `Int` variables).

3.7 Simulation Constructs

The DFiant language constructs are synthesizable language. Unlike VHDL and Verilog which were developed for simulation and only later were adapted for synthesis. To clearly separate pure simulation constructs from the rest of the language, all these constructs fall under a separate name space – `sim`. All simulation constructs exist only within a simulation context. A simulation context is formed within a derivative of a dataflow design called `DFSimulation`. The `DFSimulation` construct is typically a toplevel instance that holds both our device-under-test (DUT) and the testing logic. Currently, the DFiant compiler only supports a handful of dedicated simulation constructs: `sim.assert(cond, msg, severity)` to check ; `sim.report(msg)` ; and finally `sim.finish` to finish the simulation. All these constructs are very similar to their VHDL counterparts and in fact are

4 THE DFIANT COMPILER

4.1 Automatic Pipelining, Path-Balancing and Flow-Control

The dataflow abstraction enables designers to describe hardware without explicitly pipelining the design. The DFiant backend compiler automatically pipelines the design and places registers to split long combinational paths. The compiler has 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 affect the path-balancing mechanism. For example, `x - x.prev` create a derivation circuit while `x - x.pipe` will result in a constant zero since path-balancing applied at the subtraction input arguments results in 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 by having a circular feedback that 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 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 several open-source RTL projects and implemented them in DFiant: an AES cypher, an IEEE-754 double precision floating point multiplier, a RISC-V single-cycle

```
trait Box extends DFDesign {
  val iT = DFSInt[16] <> IN
  val iB = DFSInt[16] <> IN
  val oT = DFSInt[16] <> OUT
  val oB = DFSInt[16] <> OUT
  iT.prev <> oT //connection
  oB := iB.prev //assignment
}



trait BoxTop extends DFDesign {
  val iT = DFSInt[16] <> IN init(5, 7)
  val iB = DFSInt[16] <> IN init(2, 6)
  val oT = DFSInt[16] <> OUT
  val oB = DFSInt[16] <> OUT
  val boxL = new Box {}
  val boxR = new Box {}
  boxL.iT <> iT
  boxL.iB <> iB
  boxL.oT <> boxR.iT
  boxL.oB <> boxR.iB
  boxR.oT <> oT
  boxR.oB <> oB
}
```

Figure 6: Bla Bla
What we see

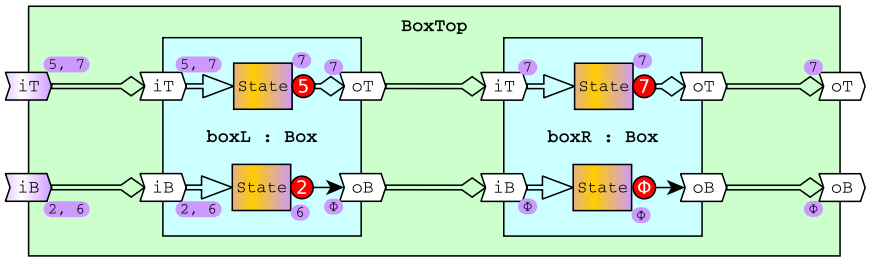

Figure 7: Bla Bla
As can be seen

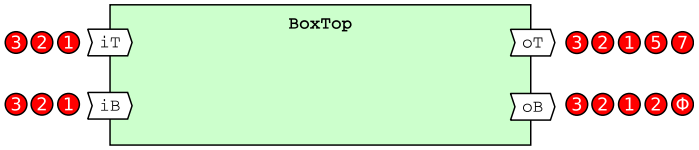

Figure 8: Bla Bla
As can be seen

core, a bitonic-sort network, and other simple examples. We compared the DFiant implementations with their RTL counterparts as follows:

(1) *Conciseness* The DFiant code is 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. Most prominently, the RTL code differentiates between registers and wires, requires explicit pipeline register placement, and adds stall control logic.
(2) *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.
(3) *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 an RTL code written by a human designer.

6 CONCLUSION

In this paper we proposed new dataflow hardware description constructs that abstract away registers and wires to achieve device-agnostic and timing-agnostic designs. We also presented DFiant, a dataflow HDL, and exposed its advantageous semantics and constructs when compared to RTL HDLs. DFiant provides a seamless concurrent programming approach, and yet it still facilitates a versatile compositional and hierarchical expressiveness. Future work may explore expanding the dataflow language constructs to simplify resource sharing by merging and splitting dataflow paths (e.g., using the same multiplier for different paths) as well as upsampling (e.g., duplicate each token) downsampling (e.g., drop every third token) a dataflow path.

REFERENCES

[1] Rishiyur S Nikhil Arvind. 1992. Id: A language with implicit parallelism. In *A Comparative Study of Parallel Programming Languages*. Elsevier, 169–215.
[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*.
[3] 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.
[4] Papon Charles. 2016. SpinalHDL. http://spinalhdl.github.io/SpinalDoc
[5] 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*.
[6] Jan Decaluwe. 2004. MyHDL: a python-based hardware description language. *Linux Journal* 127 (2004).
[7] Johan Eker and Jorn Janneck. 2003. *CAL language report*. Technical Report. Tech. Rep. ERL Technical Memo UCB/ERL.
[8] Mentor Graphics. 2008. Catapult C synthesis. *Website: http://www. mentor. com* (2008).
[9] John R. Gurd, Chris C. Kirkham, and Ian Watson. 1985. The Manchester prototype dataflow computer. *Commun. ACM* 28, 1 (1985), 34–52.
[10] 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)*.
[11] Nachiket Kapre and Samuel Bayliss. 2016. Survey of domain-specific languages for FPGA computing. In *Intl. Conf. on Field Programmable Logic and Applications*.
[12] 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

[13] 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.
[14] 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*.
[15] 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
[16] Microsemi. 2015. Synphony Model Compiler ME. (2015).
[17] DE Muller and WS Bartky. 1957. A theory of asynchronous circuits II. *Digital Computer Laboratory* 78 (1957).
[18] 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.
[19] 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*.
[20] Oliver Pell and Oskar Mencer. 2011. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News* 39, 4 (2011).
[21] 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.
[22] I Sutherland. 2012. The tyranny of the clock. *Comm. ACM* 55, 10 (2012), 35–36.
[23] Synflow. 2014. Cx Language. http://cx-lang.org/
[24] 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
[25] 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.
[26] Xilinx. 2015. Vivado High Level Synthesis User Guide. (2015).
[27] Zhipeng Zhao. 2017. Using Vivado-HLS for Structural Design : a NoC Case Study. In *Intl. Symp. on Field Programmable Gate Arrays*.