DFiant: A Dataflow Hardware Description Language

Oron Port[1] Yoav Etsion[1,2]
Electrical Engineering[1] Computer Science[2]
Technion – Israel Institute of Technology
soronpo@campus.technion.ac.il yetsion@technion.ac.il

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

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

I. Introduction

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

Emerging high-level synthesis (HLSs) tools such as Vivado HLS [1], Bluespec SystemVerilog [2], and Chisel [3] attempt to bridge the programmability gap. While HLSs tend to incorporate modern programming features, they still mix functionality with timing and device constraints, or lack hardware construction and timed synchronization control. For example, designs must be explicitly pipelined in Chisel or Bluespec, while a simple task as toggling a led at a given rate is impossible to describe with C++ constructs in Vivado HLS. Emerging HLSs, therefore, still fail to deliver a clean separation between functionality and implementation that can yield portable code, while providing general purpose HDL constructs. We explore these gaps further in Section III.

In this paper, we introduce DFiant, a modern HDL whose goal is to allow designers to express portable hardware designs. DFiant continues our previous work [4] to decouple functionality from timing constraints (in an effort to end the *"tyranny of the clock"* [5]). DFiant offers a clean model for hardware construction based on its core characteristics: (i) a clock-agnostic dataflow model that enables implicit parallel data and computation scheduling; and (ii) functional register/state constructs, accompanied by an automatic pipelining process, which eliminate all explicit register placements along with their direct clock dependency. DFiant borrows and combines constructs and semantics from software, hardware and dataflow languages. Consequently, the DFiant programming model accommodates a middle-ground approach between low-level hardware description and high-level sequential programming.

DFiant is implemented as a Scala library, and relies on Scala's strong, extensible, and polymorphic type system to provide its own hardware-focused type system (e.g., bit-accurate dataflow types, input/output port types). The interactions between DFiant dataflow types create a dependency graph that can be simulated in the Scala integrated development environment (IDE), or compiled to an RTL top design file and a TCL constraints file, followed by a hardware synthesis process using vendor tools.

II. Concurrency and Data Scheduling Abstractions

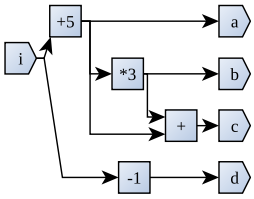
Concurrency and data scheduling abstractions rely heavily on language semantics. In this section, we explore semantics of three distinctively different languages: C++[1], VHDL, and DFiant.

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

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

TABLE I
DATA SCHEDULING SEMANTICS EXAMPLE FUNCTION, f: DEFINITION AND IMPLEMENTATIONS

Formal Definition	Functional Drawing	C++ Impl.†	VHDL Impl.‡	DFiant Impl.

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

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

A. C++ Semantics

Sequential programming models, such as C++, do not have concurrent semantics. Data scheduling order is set by *code statement order* and cannot be pipelined[2]. HLS utilities extends these languages with `pragma` directives that change semantics. We observe the C++ `f` implementation as follows:

1) All statements are variable assignments.
2) `d` is independent of `a`, `b`, and `c` but cannot be scheduled concurrently. Additionally, `a` cannot be safely read until `f` finishes. Proper pragmas allow dataflow analysis and function inlining to overcome these limitations.
3) Time between/of the data operations is unconstrained. The code does not restrict the functional requirement and will maintain correctness for every hardware synthesis fitting its semantics.

B. VHDL Semantics

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

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

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

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

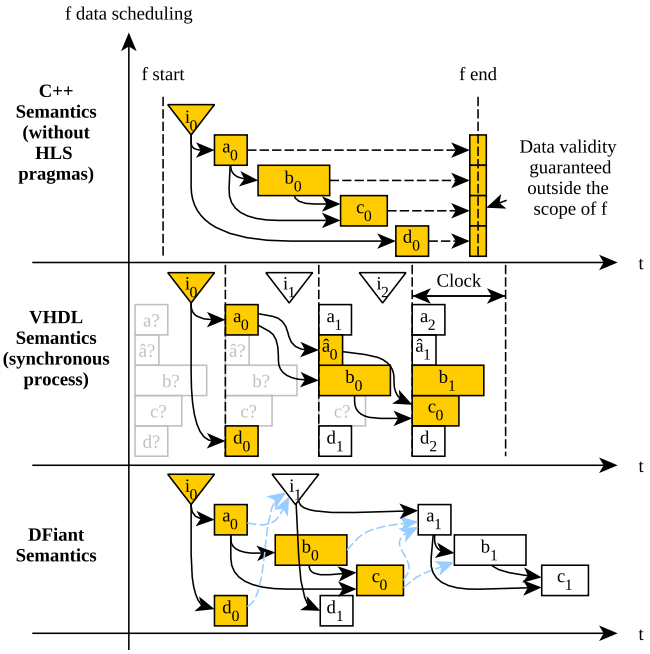


Fig. 1. f data scheduling semantics in C++, VHDL, and DFiant

C. DFiant Semantics

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

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

D. Comparing Semantics

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

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

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

III. Related Work

Recent studies [6] [7] [8] surveyed a variety of HDLs and HLS tools. Neither survey had explicit conclusion which tool or language should be used for hardware design. Earlier, we focused on comparing DFiant to VHDL and C++-based HLS. In this section, we further contrast DFiant to a few key hardware design languages and tools.

Chisel, SpinalHDL, and VeriScala: Chisel [3], SpinalHDL [9], and VeriScala [10] are Scala-based libraries that provide advanced HDL constructs. When compared to DFiant, all three DSL libraries resemble RTL semantics by implicitly or explicitly acknowledging existence of clocked registers, and do not auto-pipeline designs. Moreover, DFiant is an early-adopter of new Scala features such as literal types [11] and operations [12], which further improve type safety (e.g., a `DFBits[5].bits(Hi,Lo)` bit selection is compile-time-constrained within the 5-bits vector width confines).

Synflow Cx: Synflow developed Cx [13] as a designer-oriented HDL with new language semantics that better fit hardware design than the classic C syntax. However, the concurrency in Cx limits dataflow description flexibility. A `fence` statement is required to force a new cycle. This statement affects all variables within a `task`. To avoid this, separate tasks are required, which limits functional clustering in a single task. Moreover, Cx is not object-oriented and has a limited type-system.

MyHDL: MyHDL [14] is a Python-based HDL. MyHDL favors verification capabilities over purely synthesizable hardware constructs, in contrary to our approach in DFiant. Since MyHDL is based on Python, it also lacks type-safety. MyHDL does not support automatic pipelining.

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

Vivado HLS: Vivado HLS [1] is a mature tool that helps achieve high productivity in some domains. Nevertheless, it is not accepted as a general purpose HDL, since its C/C++ semantics are unfitting [15] and its SystemC synthesizable constructs provide roughly identical capabilities of traditional HDLs [16].

Maxeler: The Maxeler framework [17] and its MaxJ Java-based programming language take part in acceleration systems. MaxJ is dataflow-centric, same as DFiant, but is tailored for its target use-case and does not fit as a general purpose HDL.

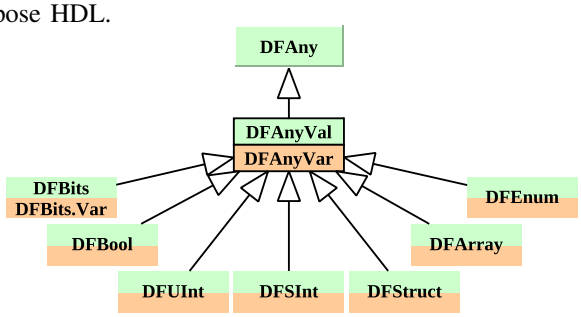


Fig. 2. DFiant dataflow types: simplified inheritance diagram

IV. The DFiant Type System

DFiant is a Scala library, hence it inherently supports type safe and rich language constructs. DFiant brings type driven development concepts to hardware design, by creating an extensible dataflow class hierarchy, with the trait `DFAny` at its head. `DFAny` contains all properties that are common to every dataflow variable. Fig. 2 illustrates a simplified inheritance diagram of DFiant's dataflow types.

```
class AES_DFKeySchedule(Nk: Int, Nb: Int, Nr : Int)
  extends AES_DFWords(Nb*(Nr+1)){
  val Rcon = Array[Int](0x00000000, ...)
  def KeyExpansion(key : AES_DFKey): Unit = {
    val temp = AES_DFWord()

    for (i <- 0 until Nk)
      this(i) := key(i)



    for (i <- Nk until Nb*(Nr+1)) {
      temp := this(i-1)
      if (i % Nk == 0)
        temp := temp.RotWord().SubWord() ^ Rcon(i / Nk)
      else if ((Nk > 6) && (i % Nk == 4))
        temp := temp.SubWord()

      this(i) := this(i-Nk) ^ temp

    }
} }
```

(a) DFiant code

```
//comment line for alignment
//comment line for alignment
Rcon = [00000000, ...]
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk) begin
  word temp
  i = 0
  while (i < Nk)
    w[i] = word(key[4*i],key[4*i+1],key[4*i+2],key[4*i+3])
    i = i+1
  end while
  i = Nk
  while (i < Nb * (Nr+1)]
    temp = w[i-1]
    if (i mod Nk = 0)
      temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
    else if (Nk > 6 and i mod Nk = 4)
      temp = SubWord(temp)
    end if
    w[i] = w[i-Nk] xor temp
    i = i + 1
  end while
end
```

(b) AES spec. pseudo code reference

Fig. 3. AES KeyExpansion code example

Fig. 3 depicts part of our DFiant Advanced Encryption Standard [18] (AES) cipher implementation alongside its specification pseudo code reference. The DFiant code is similar or even simpler in comparison and does not employ global functions. We compared the complete DFiant AES code to three RTL designs [19] [20] [21]. DFiant provides the same functionality with 33-50% lines of code. Furthermore, the DFiant code is timing-agnostic and device-agnostic, thus tasking the compiler to construct the hardware fitting the target device and non-functional requirements (e.g., throughput, latency). When constrained by the appropriate target throughput, the DFiant compiler generated an RTL design that acheived better performance than the cited RTL designs.

V. Conclusion

In this paper, we presented DFiant, a dataflow HDL, and exposed its advantageous semantics compared to modern RTLs and C++-based HLS tools (e.g., VHDL and Vivado HLS). DFiant provides a seamless concurrent programming approach, and yet it still facilitates a versatile compositional and hierarchical expressiveness.

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

References

[1] Xilinx, "Vivado High Level Synthesis User Guide," 2015.

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

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2012.

[4] O. Port, "CAFEO: A Dataflow, Device-agnostic, Synthesizable Hardware Description Language," Master's thesis, 2015. [Online]. Available: http://library.technion.ac.il/thesis/ele/2619627.pdf

[5] I. Sutherland, "The tyranny of the clock," *Comm. ACM*, vol. 55, no. 10, pp. 35–36, 2012.

[6] N. Kapre and S. Bayliss, "Survey of domain-specific languages for FPGA computing," in *Intl. Conf. on Field Programmable Logic and Applications*, 2016.

[7] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[8] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar, "High-level language tools for reconfigurable computing," *Proc. of the IEEE*, vol. 103, no. 3, pp. 390–408, 2015.

[9] P. Charles, "SpinalHDL," 2016. [Online]. Available: http://spinalhdl.github.io/SpinalDoc

[10] Y. Liu, Y. Li, W. Xiong, M. Lai, C. Chen, Z. Qi, and H. Guan, "Scala Based FPGA Design Flow (Abstract Only)," in *Intl. Symp. on Field Programmable Gate Arrays*, 2017.

[11] E. Osheim, G. Leontiev, J. Pretty, L. Hupel, M. O'Connor, M. Sabin, and T. Switzer, "Typelevel Scala," 2017. [Online]. Available: https://github.com/typelevel/scala

[12] F. S. Thomas, M. Pocock, N. Aoyama, and O. Port, "singleton-ops library," 2017. [Online]. Available: https://github.com/fthomas/singleton-ops

[13] Synflow, "Cx Language," 2014. [Online]. Available: http://cx-lang.org/

[14] J. Decaluwe, "MyHDL: a python-based hardware description language," *Linux Journal*, vol. 2004, no. 127, 2004.

[15] Z. Zhao, "Using Vivado-HLS for Structural Design : a NoC Case Study," in *Intl. Symp. on Field Programmable Gate Arrays*, 2017.

[16] D. Gajski, T. Austin, and S. Svoboda, "What input-language is the best choice for high level synthesis (HLS)?" in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2010.

[17] O. Pell and O. Mencer, "Surviving the end of frequency scaling with reconfigurable dataflow computing," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 4, 2011.

[18] NIST, "Advanced Encryption Standard (AES)," *Federal Information Processing Standards Publication*, vol. 197, no. 441, 2001.

[19] S. Das, "Fully Pipelined AES Core," 2010. [Online]. Available: https://opencores.org/project,aes_pipe

[20] H. Hsing, "AES Core Specification," 2013. [Online]. Available: http://opencores.org/usercontent,doc,1354351714

[21] A. Salah, "128 bit AES Pipelined Cipher," 2013. [Online]. Available: http://opencores.org/usercontent,doc,1378852274