Registerless Hardware Description

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

Table 1: Hardware Programming Model Comparison. Dataflow HDLs can bridge the gap between HLS and RTL.

Abstraction	Languages/Tools	Hardware Design Fundamentals	Design Abstraction & Automation	Hardware Meta-Programming
High-Level Synthesis (HLS)	VivadoHLS, LegUp, Catapult, Synphony, HercuLeS, OpenCL	✗	✓	✗
Dataflow HDL (DF-HDL)	DFiant	✓	✓	✓
High-Level RTL (HL-RTL)	Chisel, SpinalHDL, PyRTL, nMigen, MyHDL, Bluespec, Cx	✓	✗	✓
RTL	VHDL, Verilog, SystemVerilog	✓	✗	✗

ABSTRACT

To bridge the programmability gap between HLS and RTL languages, we claim that hardware programming abstractions must cover most, if not all, of the numerous synthesizable uses of RTL constructs. Our proof of concept relies on a novel dataflow hardware description language (HDL) abstraction layer and implements the DFiant HDL and compiler.

1 INTRODUCTION

Most RTL-alternatives can be classified either as high-level synthesis (HLS) tools or high-level RTL (HL-RTL) languages. On the one hand, HLS tools (such as Vivado [27], and others [5, 10, 14, 15, 20, 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 [28]. 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 [3], and others [2, 4, 7–9, 13, 17–19, 25]) aim to enhance productivity by introducing new hardware generation constructs and semantics but do not abstract away register-level description (even Bluespec [21], which uses concurrent guarded atomic actions, until recently [11] assumed rules complete within a single clock cycle). Therefore, HL-RTL designs are still subjected to the *"tyranny of the clock"* [24] and are bound to specific timing and target constraints.

In this paper we claim that better HDLs must adhere to all known RTL design use-cases, yet still maintain enough abstraction to allow automatic pipelining and target-agnostic design. Therefore, we propose a novel dataflow HDL (DF-HDL) abstraction layer to abstract over registers and clocks. This concept is proven via DFiant[1] [22], a Scala-embedded DF-HDL that utilizes these dataflow abstractions to decouple functionality from implementation constraints. DFiant brings together constructs and semantics from dataflow [1, 6, 12, 16, 26], 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.

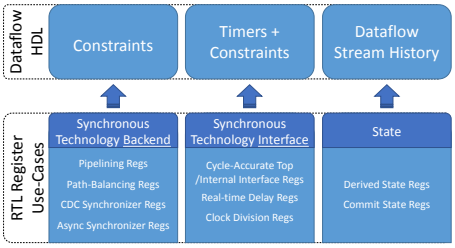


Figure 1: DF-HDL Register Abstraction
Registers use-cases are divided into three main categories, and abstracted accordingly in DF-HDLs.

Table 1 compares the main hardware programming models according to three categories: hardware design fundamentals, which include capabilities such as IO connectivity, hierarchies, and synchronous design; design abstraction & automation, which includes abstractions that enable automatic pipelining, path-balancing and flow control; and finally hardware meta-programming, that enables simple generation of complex hardware structures. The comparison indicates that DFiant bridges the programmability gap between HLS tools and RTL languages by enabling designers full control over the generated hardware whilst still enabling features like automatic pipelining. DFiant is *not* an HLS language, nor is it an RTL language. Instead, DFiant is an *asynchronous* dataflow HDL that provides abstractions beyond the RTL behavioral model, thereby reducing verbosity and maintaining portable code.

2 A DATAFLOW HDL ABSTRACTION

The basic notion of a DF-HDL 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

[1] https://dfianthdl.github.io

LATTE '21, April 15, 2021, Virtual, Earth

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 identified three main uses for registers in the RTL model: *synchronous technology backend*, *synchronous technology interface*, and *design functionality* (i.e., state). We summarized the various uses in Fig. 1, and we now turn to discuss them and how the dataflow model can derive the first two main 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. *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. *Path-balancing registers* are added to maintain design correctness when pipelining. *Synchronizers*, often composed of registers, are used to mitigate CDC and asynchronous metastability effects and bring the design to the proper reliability.

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 frequently only affect the interface and not the core design itself. 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.

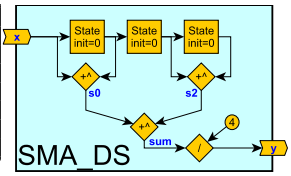
Real-time signals or *derivations of timed* signal inputs require timer constructs. For example, a design using a 100MHz clock may drive a UART stream at 1Mbps or toggle a led at 1Hz. Rather than directly using registers as clock dividers or employing clock generation components (e.g., PLLs), one can 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 consider all clocks and generate the proper clock tree based on the available device resources and other design constraints.

2.3 State Registers

State registers 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 initialize and reuse the stream

```
@df class SMA_DS extends DFDesign {
  val x   = DFSInt(16) <> IN init 0
  val y   = DFSInt(16) <> OUT
  val s0  = x +^ x.prev
  val s2  = x.prev(2) +^ x.prev(3)
  val sum = s0 +^ s2
  y       := (sum / 4).resize(16)
}
```

(a) DS code (`.prev` accesses the history)



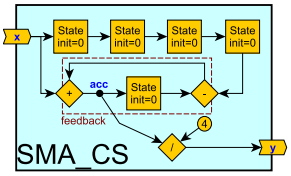
(b) DS drawing

$$y_k = \frac{1}{4}(x_k + x_{k-1} + x_{k-2} + x_{k-3}); \quad x_{k<0} = 0$$

```
@df class SMA_CS extends DFDesign {
  val x   = DFSInt(16) <> IN init 0
  val y   = DFSInt(16) <> OUT
  val acc = DFSInt(18) <> VAR init 0
  acc := acc - x.prev(4) + x
  y   := (acc / 4).resize(16)

}
```

(c) CS code (default access to `acc.prev`)



(d) CS drawing

$$y_k = y_{k-1} + \frac{x_k - x_{k-4}}{4}; \quad x_{k<0} = 0$$

Figure 2: Derived state (DS) and commit state (CS) SMA DFiant implementation codes and drawings. The state lines are highlighted in the code.

history. A *derived (feedforward) state* is a state whose current output value is *independent* of its previous value (e.g., detecting if an input has changed). A *commit (feedback) state* is a state whose current output value is *dependent* on its previous state value (e.g., the new cumulative sum value is dependent on its previous sum value). 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 can therefore be pipelined to reduce its cycle time and increase its throughput. In contrast, a commit state path is circular and cannot be pipelined as-is.

The basic DFiant example given in Fig. 2 provides two implementations of a simple moving average (SMA) unit; both have a 4-tap average window with 16-bit integer input and output, and compose the average arithmetic from the `+^` carry-addition and other operations. The difference is that `SMA_DS` has only derived state and can therefore be automatically pipelined by the DFiant compiler, while `SMA_CS` has a commit state and cannot be pipelined as-is. The dataflow history is accessed by invoking `.prev` with the required step parameter. The `SMA_CS` accumulation variable `acc` depends on itself and forms a commit state. Reading from `acc` before it is assigned manifests as a default read from `acc.prev`. This basic example does not cover various DFiant semantics and capabilities which include automatic stall and flow control, FSM meta-programming, combining dynamic and static dataflow, time abstraction, legacy RTL code integration, and more.

3 CONCLUSION

Modern RTL alternatives abstract either too little or too much. The numerous register use-cases must meet worthy abstractions to allow portable designs and minimize verbosity, yet without losing hardware design expressiveness. To achieve this we use a unique dataflow HDL abstraction that obfuscates registers and clocks.

ACKNOWLEDGMENTS

This work has been supported by EU H2020 ICT project LEGaTO, contract #780681.

REFERENCES

[1] Rishiyur S Nikhil Arvind. 1992. Id: A language with implicit parallelism. In *A Comparative Study of Parallel Programming Languages*.

[2] Christiaan Baaij. 2009. *Cλash: From haskell to hardware.* Master's thesis. University of Twente.

[3] 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)*.

[4] Peter Bellows and Brad Hutchings. 1998. JHDL-an HDL for reconfigurable systems. In *Intl. Symp. on Field-Programmable Custom Computing Machines*.

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An Open-source High-level Synthesis Tool for FPGA-based Processor/Accelerator Systems. *ACM Trans. on Computer Systems* 13 (2013).

[6] P Caspi, D Pilaud, N Halbwachs, and JA Plaice. 1987. LUSTRE: a declarative language for real-time programming. *Symp. on Principles of Programming Languages (POPL)* (1987).

[7] Papon Charles. 2016. SpinalHDL. http://spinalhdl.github.io/SpinalDoc

[8] John Clow, Georgios Tzimpragos, Deeksha Dangwal, Sammy Guo, Joseph McMahan, and Timothy Sherwood. 2017. A pythonic approach for rapid hardware prototyping and instrumentation. In *Intl. Conf. on Field Programmable Logic and Applications*.

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

[10] Mentor Graphics. 2008. Catapult C synthesis. *Website: http://www. mentor. com* (2008).

[11] David J Greaves. 2019. Further sub-cycle and multi-cycle schedulling support for Bluespec Verilog. In *ACM/IEEE Intl. Conf. on Formal Methods and Models for Co-Design*.

[12] John R. Gurd, Chris C. Kirkham, and Ian Watson. 1985. The Manchester prototype dataflow computer. *Comm. ACM* 28, 1 (1985).

[13] 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)*.

[14] Nikolaos Kavvadias and Kostas Masselos. 2013. Hardware design space exploration using HercuLeS HLS. *Panhellenic Conference on Informatics (PCI* (2013).

[15] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. 2018. Spatial: A language and compiler for application accelerators. In *Intl. Conf. on Programming Language Design and Impl. (PLDI)*.

[16] 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).

[17] 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*.

[18] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A unified framework for vertically integrated computer architecture research. In *Intl. Symp. on Microarchitecture (MICRO)*.

[19] Nick Matthijssen. 2020. Wyre: An ergonomic hardware definition language that compiles to Verilog. https://github.com/nickmqb/wyre

[20] Microsemi. 2015. Synphony Model Compiler ME. (2015).

[21] 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*.

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

[23] François Serre and Markus Püschel. 2019. DSL-Based Hardware Generation with Scala: Example Fast Fourier Transforms and Sorting Networks. *ACM Trans. on Reconfigurable Technology and Systems (TRETS)* 13 (2019).

[24] I Sutherland. 2012. The tyranny of the clock. *Comm. ACM* 55, 10 (2012).

[25] Synflow. 2014. Cx Language. http://cx-lang.org/

[26] Ghislaine Thuau and Daniel Pilaud. 1991. Using the declarative language Lustre for circuit verification. In *Designing Correct Circuits*. Springer.

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

[28] Zhipeng Zhao and James C Hoe. 2017. Using Vivado-HLS for structural design: a NoC case study. *Intl. Symp. on Field Programmable Gate Arrays* (2017).