Provably Correct Development of Reconfigurable Hardware Designs via Equational Reasoning

Ian Graves, Adam Procter & William L. Harrison
Department of Computer Science, University of Missouri

Gerard Allwein
US Naval Research Laboratory, Washington, DC

***Abstract*—There is a semantic gap between the hardware definition languages used to design and implement hardware and the languages and logics used to formally specify and verify them. Bridging this gap—i.e., constructing formal models from existing hardware artifacts—can be costly, time-consuming, and error prone—and yet utterly necessary if formal verification is to proceed. This work demonstrates that this gap can be collapsed by starting in a pure functional language that is also a hardware description language, and that equational style verifications may be performed directly on the source text of a hardware design, thereby significantly lowering the verification cost for reconfigurable designs. When combined with an efficient compiler, this methodology achieves both good performance and low cost verification.**

I. Introduction

Reconfigurable computing emphasizes a "mix and match" approach to system construction, frequently involving specially tailored "one off" components. Formal methods can provide high confidence that systems obey critical properties (e.g., safety and security), but, by reputation, they can also involve a substantial investment of time and effort. Formal methods may, therefore, seem somewhat antithetical to reconfigurable computing. Can it make economic sense to invest the resources for formal methods on potentially "one off" reconfigurable systems?

The proposed methodology aims to make hardware verification cost effective for reconfigurable designs via a functional programming language that also serves as a hardware description language. The principal hypothesis of this research is that following this methodology can significantly reduce the effort of verifying hardware designs, thereby making formal verification cost effective for reconfigurable computing. The functional language—ReWire [1]—plays a dual rôle for both hardware description and formal specification. We support this hypothesis with a demonstration of the approach in which the stream cipher Salsa20 [2] is implemented efficiently in ReWire and verified using equational reasoning on the implementation source code.

In the functional programming community, equational reasoning about programs frequently goes by the moniker "Bird-Wadler style" (so named for the influential textbook [3]). Functional programmers reason about source programs in an equational style, by replacing equals for equals, making simplifications, induction and coinduction, etc. Equational reasoning is commonly used to justify, among other things, source-to-source transformations and program correctness. This is precisely what we use Bird-Wadler reasoning for in this paper, although, in ReWire, programs *are* hardware descriptions.

```
fib :: Int -> Int                 fib2 :: Int -> (Int,Int)
fib 0       = 0                   fib2 0 = (0,1)
fib 1       = 1                   fib2 n = (b, a + b)
fib (n + 1) =                       where
   fib(n - 1) + fib(n)                (a, b) = fib2(n)
```

Theorem (Fib)**.** *For all* $\mathtt{n} \geq 0$*,* $\mathtt{fib(n)} = \mathtt{fst\,(fib2(n))}$.

Fig. 1: Bird-Wadler Program Development

This research demonstrates that formal methods and reconfigurable systems are not antithetical to one another at all. **The contributions of this paper are as follows. (1)** We describe a methodology for developing high assurance, reconfigurable systems leveraging pure functional languages and equational reasoning. A standard practice in functional programming—Bird-Wadler reasoning—is repurposed to hardware design with this methodology. **(2)** We introduce an extension to ReWire called *Connect Logic*, which consists of domain specific language abstractions for hardware devices that support a mixture of functional and structural design styles. **(3)** Encapsulation of a pipelining structuring technique in Connect Logic is exhibited along with **(4)** several performant implementations of the Salsa20 stream cipher based on it.

Reconfigurable Salsa20 without ReWire: Consider the following experiment. A hardware designer decides to implement the Salsa20 cipher in hardware. There are a number of good reasons to do so, not the least of which is that reconfigurable hardware can increase the possible throughput compared to a software implementation. The hardware designer uses a tried and true hardware definition language (HDL) like VHDL or Verilog. The implementation path is straightforward—she implements Bernstein's defining equations [2] in terms of the HDL and performs her usual development process involving synthesis, simulation, and testing.

This first implementation is one step removed from Bernstein's high-level specification, and, furthermore, is expressed in a language without a formal semantics. So, how does she prove that the first implementation is correct? It becomes clear to the hardware engineer that the first implementation does not suffice: even implemented in the most optimized fashion, it contains too many gates for most FPGAs. So, the hardware engineer produces a second implementation structured in an explicitly pipelined form resulting in a circuit that fits on her FPGA.

Is she all done? Not if formal proof is required that the second implementation is correct. The second implementation is two steps removed from Bernstein's high level specification and it is written in a language without a formal semantics. To verify its correctness, where does she even start? She could attempt to verify the implementation by encoding it in the logic of a theorem prover, but, observe that this involves yet another translation—and one which is not straightforward. With this approach, how can we be sure that her logical specification faithfully relates Bernstein's high-level specification to a VHDL implementation?

Bird-Wadler Provably Correct Development: To illustrate the formal methodology we advocate for reconfigurable computing, consider first this classic example (p.131, [3]) of Bird-Wadler style equational reasoning in Fig. 1. On the left is the usual recursive definition of the Fibonacci function. It serves as a *reference specification* defining the meaning of the Fibonacci function, but it has terrible $O(2^n)$ performance. The other version of the Fibonacci function on the right is in an optimized, "accumulator-passing style" form with $O(n)$ performance.

The hallmark of Bird-Wadler development is that there is a reference specification (e.g., `fib`) and one or more transformations from it (e.g., into `fib2`) that give rise to an equational verification (e.g., the Fib theorem in Fig. 1). This verification justifies using the optimized version (i.e., replacing `fib(n)` with `fst(fib2(n))`).

Provably Correct Development of Salsa20 with ReWire: It is precisely the Bird-Wadler style of development that ReWire enables for reconfigurable computing. Fig. 4 presents the hash function from the Salsa20 stream cipher [2] represented in a Haskell-like syntax. We discuss this figure in some detail as well as explain the requisite Haskell syntax in subsequent sections. It suffices to say that Fig. 4 contains a functional program defining the Salsa20 hash function that also serves as the high-level reference specification in the Bird-Wadler development presented in our case study. To render it into a synthesizable form, we add some Connect Logic annotations to produce the ReWire code in Fig. 5. The ReWire compiler can now synthesize a circuit for Salsa20. This new implementation can now be measured in two ways: against standard performance metrics as in Table I or by verifying that it produces the same answers as the reference specification (Theorem 1). The first ReWire implementation is now rewritten using pipelining constructs also written in Connect Logic (the ten and twenty stage pipelines in Figures 6 and 7, resp.). The correctness of the pipelining transformation is given in Theorem 2.

Section II discusses related work, Section III introduces Connect Logic and the pipelining structuring technique applied and verified in Sections IV and V, resp. Section VI summarizes and concludes. Most of the subject matter in this paper relates to provably correct development of reconfigurable hardware rather than on more traditional areas of reconfigurable computing. The targeted audience for the paper is, however, the reconfigurable computing community and so considerable effort has been made to make the paper as self-contained as possible.

II. Related Work

There is a long history of formal methods being applied to hardware designs [4]. The general process involves encoding a hardware design in the logic of a theorem prover by hand[1] and then proving theorems about the encoding. There is an obvious danger that the encoding process—which one might call *semantic archaeology*—will introduce errors as well as a problem of soundness (i.e., how do you know a theorem about the encoding applies to the hardware device itself?).

"Semantic archaeology"—the process of developing a formal specification for an *existing* computing artifact—is the principal reason that formal methods can be so time-consuming and expensive. Sarkar et al. [5] describe the semantic archaeology process in the context of modeling the x86 multiprocessor instruction set architecture: "*The key difficulty was to go from the informal-prose vendor documentation, with its often-tantalising ambiguity, to a fully rigorous definition (mechanised in HOL) that one can be reasonably confident is an accurate reflection of the vendor architectures (Intel 64 and IA-32, and AMD64).*"

Cryptol [6] is a domain-specific language for specifying, verifying and implementing cryptographic algorithms. Given a cryptographic algorithm, one can specify it in Cryptol, run a number of automatic and semi-automatic proof tools over the specification, and ultimately generate C code implementing the algorithm itself. The current open source version of Cryptol (v.2) does not generate hardware implementations, although a previous proprietary version (v.1) did. ReWire, by contrast, is a subset of Haskell compilable to VHDL and is not restricted to cryptographic algorithms. Salsa20 has been specified in Cryptol v.2, but no effort has been made to backport this specification to Cryptol v.1 and synthesize it.

The usual standards for evaluating hardware architectures and design flows are performance-based metrics (e.g., time and space performance, power usage, etc.). Within the context of mission critical systems, formal analysis and verification are required evaluation modes as well. The Common Criteria for Information Technology Security Evaluation (a.k.a. Common Criteria or CC) is an international standard (ISO/IEC 15408) for computer security certification and the US Federal government mandates following the CC requirements for mission critical systems. The CC sets seven evaluation assurance levels (EAL). The most stringent such level is EAL7, which requires "*extensive formal analysis*" for applications in "*extremely high risk situations and/or where the high value of the assets justifies the higher costs*" ensuing from formal verification [7]. For reconfigurable computing to be applied in the space of mission critical systems, cost effective formal methods techniques must be developed. The current research is a step in this direction.

Previous work demonstrated the construction and verification of a secure many-core system in ReWire [1]. The present work, in contrast, demonstrates the expression of a common hardware design pattern (stall-free pipelining) in ReWire and its verification. The emphasis in the former was on the design and implementation of the ReWire language, while the current work focuses on ReWire as a vehicle for hardware verification.

[1]E.g., Isabelle/HOL (http://isabelle.in.tum.de), ACL2 (http://www.cs.utexas.edu/users/moore/acl2), and PVS (http://pvs.csl.sri.com), are the most commonly used provers for hardware verification.

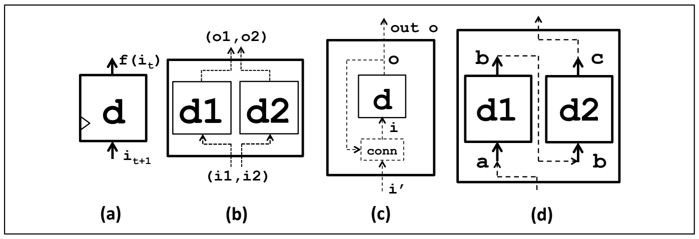


Fig. 2: Device Constructors

III. Connect Logic in ReWire

Connect Logic has operations for composing and connecting smaller devices to create larger ones. Sec. III-B below introduces Connect Logic at a high level; for reasons of space, a semantic treatment of Connect Logic is left for future work. We then illustrate the use of Connect Logic via the design of a pipelining transformation for ReWire. Section III-A gives background information on pure functional languages and equational verification.

A. Pure Functional Languages & Equational Verification

1) Primer on Haskell/ReWire Syntax: For the sake of being as self-contained as possible, this section presents a quick overview of Haskell—and, hence, ReWire—syntax necessary to understand this paper.

Haskell [8] is a strongly-typed, purely functional language. A Haskell program consists of a number of function and datatype declarations. The type of a function from type $\mathtt{a}$ to type $\mathtt{b}$ is written, $\mathtt{a} \mathbin{\texttt{->}} \mathtt{b}$. The type for a tuple with first and second components $\mathtt{a}$ and $\mathtt{b}$, resp., is written $(\mathtt{a}, \mathtt{b})$. The fact that a Haskell expression $\mathtt{e}$ has type $\mathtt{a}$ is written $\mathtt{e} :: \mathtt{a}$. Haskell has a built-in list type constructor: $[\mathtt{a}]$ is the type of all lists of elements of type $\mathtt{a}$. Because of Haskell's lazy evaluation strategy, lists can have an infinite number of elements—such lists are also called *streams*.

Below are a number of function declarations. The simplest function is the identity function, which takes its argument and simply returns it. Given two functions, $\mathtt{f}$ and $\mathtt{g}$, their composition is written, $\mathtt{f} \circ \mathtt{g}$. Function application is written either $\mathtt{g}(\mathtt{x})$ or by simple juxtaposition, $\mathtt{g}\,\mathtt{x}$. The function `map` takes two arguments, a function $\mathtt{f}$ and a list $\mathtt{l}$, and applies $\mathtt{f}$ to each element of $\mathtt{l}$, thereby creating a new list. The function `drop` takes a non-negative integer $\mathtt{n}$ and a list $\mathtt{l}$, and returns the list missing the first $\mathtt{n}$ elements from $\mathtt{l}$. Cons (:) takes an item $\mathtt{i}_0$ and a list of items and returns a new list with $\mathtt{i}_0$ on the front. N.b., it is important to distinguish (::)—"has type"—from list cons (:).

$$\begin{array}{ll}
\mathtt{id}\,\mathtt{x} & = \mathtt{x} \\
(\mathtt{f} \circ \mathtt{g})\,\mathtt{x} & = \mathtt{f}\ (\mathtt{g}(\mathtt{x})) \\
\mathtt{map}\,\mathtt{f}\,[\mathtt{i}_0, \mathtt{i}_1, \ldots] & = [\mathtt{f}(\mathtt{i}_0), \mathtt{f}(\mathtt{i}_1), \ldots] \\
\mathtt{drop}\,\mathtt{n}\,[\mathtt{i}_0, \ldots, \mathtt{i}_{\mathtt{n}-1}, \mathtt{i}_\mathtt{n}, \ldots] & = [\mathtt{i}_\mathtt{n}, \ldots] \\
\mathtt{i}_0\ :\ [\mathtt{i}_1, \ldots] & = [\mathtt{i}_0, \mathtt{i}_1, \ldots] \\
\mathtt{nth}\,\mathtt{j}\,[\mathtt{i}_0, \ldots, \mathtt{i}_\mathtt{j}, \ldots] & = \mathtt{i}_\mathtt{j} \\
\mathtt{fst}\,(\mathtt{a}, \mathtt{b}) & = \mathtt{a} \\
\mathtt{snd}\,(\mathtt{a}, \mathtt{b}) & = \mathtt{b}
\end{array}$$

We note without proof that, for two non negative integers, $\mathtt{n}$ and $\mathtt{m}$, it holds that:

$$\mathtt{drop}\,(\mathtt{n}+\mathtt{m})\,\mathtt{l} = \mathtt{drop}\,\mathtt{n}\,(\mathtt{drop}\,\mathtt{m}\,\mathtt{l}) \qquad (\dagger)$$

In Haskell/ReWire, we can introduce new datatypes with the `data` keyword. In the following declarations, `Quad` and `Hex` are *type constructors* that, given any type `a`, construct new types, `Quad a` and `Hex a`, resp. To construct a value of a datatype, apply a *data constructor*; the data constructors below are `Q` and `H`. For example, a value `Q 1 2 3 4` is of type `Quad Int`; we write this type declaration as `Q 1 2 3 4 :: Quad Int`. A `Bit` is either `High` or `Low`.

```
data Quad a = Q a a a a
data Hex  a = H a a a a a a a a a a a a a a a a
data Bit    = High | Low
```

ReWire has built-in types for words. A 32-bit (128-bit) word belongs to the type `W32` (`W128`). For example, a value of type (`Quad W32`) has the form (`Q w1 w2 w3 w4`), which is nothing more than four 32-bit words.

2) Purity and Equational Verification: Haskell (and, hence, ReWire) is a pure language, which is a critical foundation for equational reasoning. Purity means that the type of a Haskell program faithfully represents its value and behavior. If a Haskell function has type `Int -> Int`, then the function takes an `Int` as input and produces an `Int` as output. Furthermore, we can conclude that the function possesses no side effects whatsoever because, in Haskell, side effects are reflected accurately in the types. The expression `(print "Hello World")`, for instance, prints out `Hello World` to the prompt and, therefore, `(print "Hello World") :: IO ()`—it produces the value nil, (), which is tagged in its type with `IO`, meaning it performs input/output in some form.

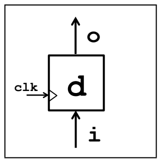
To prove an equation, $\mathtt{e} = \mathtt{e}'$, one starts from $\mathtt{e}$ and "replaces equals for equals" until $\mathtt{e}'$ is reached. In symbols, this proof is $\mathtt{e} = \mathtt{e}_1 = \mathtt{e}_2 = \cdots = \mathtt{e}_\mathtt{n} = \mathtt{e}'$ in which each step is justified by a known equation x=y—as in "replace $\mathtt{x}$ in $\mathtt{e}_\mathtt{i}$ by $\mathtt{y}$ to obtain $\mathtt{e}_{\mathtt{i}+1}$". Purity supports this style of reasoning because, being all Haskell expressions are side effect free, they cannot interact unpredictably with the expressions in which they are substituted.

B. Extending ReWire with Connect Logic

This section presents the ReWire operators for the compositional construction of devices from other devices. We refer to these particular operators as "Connect Logic". Connect Logic enables two or more existing devices to be composed in parallel and connected together. Connect Logic supports a compositional style of hardware design akin to structural VHDL. Formulating the design of a hardware device may be accomplished as in previous work [1] (i.e., without Connect Logic), or, existing devices may be composed with Connect Logic operations into bigger devices.

There is a type constructor `Dev` for synchronous devices in ReWire. There are three basic architectural constructors that Connect Logic adds to the ReWire language. The first, `iter`, constructs a synchronous device from a pure function from inputs to outputs. The second, $\langle\&\rangle$, composes two devices in parallel. The third, `refold`, is a recursion operator that is used to interconnect devices and/or express feedback loops (i.e., feed back device outputs to inputs).

1) Types for Devices: There is one basic unit of Connect Logic, devices, for which we introduce the following type: `Dev i o` for any types `i` and `o`. A term of type, `Dev i o`, represents a clocked computation that, for each clock cycle, takes an input of type `i`, produces an output of type `o`, and may possess internal storage. We eschew the formal definition of `Dev` as it is unnecessary to understanding Connect Logic and its uses. Device `d` is clocked, as illustrated in the inset figure. The clock is represented by the underlying structure of `Dev i o`, rather than as an explicit parameter. A device is created in Connect Logic by either iterating a function or through composition of existing devices. We introduce operators for constructing devices and composing them into larger, interconnected devices. All Connect Logic operations are *constructors* for `Dev`, meaning that they are functions producing `Dev i o` values for some `i` and `o` types.



`d :: Dev i o`

2) Iteration: The most basic Connect Logic constructor, `iter`, iterates a pure function of type `i -> o`, producing an output corresponding to the input at each clock cycle. The Haskell definition of `iter` is as follows:

```
iter :: (i -> o) -> o -> Dev i o
iter f o = do i <- signal o
              iter f (f i)
```

Fig. 2**(a)** illustrates the device created with the `iter` operation. The type declaration above means that `iter` is a device constructor that takes a function from inputs `i` to outputs `o` and an initial output value and constructs a corresponding device. The device (`iter f o`) will, at the first clock cycle, return output `o` and, in the next clock cycle after consuming an input `i`, will produce a new output, (`f i`). This pattern repeats recursively ad infinitum. The (`signal o`) operator outputs its argument `o` and returns the next input. The definition of the (`iter f o`) constructor above may be read as (1) output `o` (i.e., `signal o`), (2) receive the next input (i.e., `do i <− signal o`), and then (3) repeat the pattern with new "initial" output (`f i`).

3) Parallelism: Parallelism is expressed with the device constructor, ⟨&⟩, that composes two existing devices, `d1` and `d2`, into a single device, `d1` ⟨&⟩ `d2`, in which both devices operate in parallel and in isolation from one another. N.b., we are assuming, here and elsewhere, that both arguments `d1` and `d2` are non-terminating. The type declaration of ⟨&⟩ is:

```
⟨&⟩ :: Dev i₁ o₁ ->
           Dev i₂ o₂ ->
              Dev (i₁,i₂) (o₁,o₂)
```

We omit its Haskell definition as doing so would require an unnecessary excursion into Haskell's syntax and semantics. Fig. 2**(b)** presents a pictorial version of `d1` ⟨&⟩ `d2`. The type signature of ⟨&⟩ means that the input and output types of constructed device `d1` ⟨&⟩ `d2` are pairs of the inputs and outputs of `d1` and `d2`, resp. Both subdevices `d1` and `d2` are isolated from one another in `d1` ⟨&⟩ `d2`—i.e., there is no intercommunication or shared state between them. Such interaction may be added explicitly using the `refold` operator described below. The parallelism operator may be generalized to arbitrary numbers of devices (i.e., beyond two), but, for lack of space, we only present the simplest case.

$$
\begin{array}{ll}
R1 & \left[\begin{array}{l}
1 \left[\begin{array}{ll} x[4] \oplus\!=(x[0] \boxplus x[12]) \lll 7 & x[9] \oplus\!=(x[5] \boxplus x[1]) \lll 7 \\ x[14] \oplus\!=(x[10] \boxplus x[6]) \lll 7 & x[3] \oplus\!=(x[15] \boxplus x[11]) \lll 7 \end{array}\right. \\
2 \left[\begin{array}{ll} x[8] \oplus\!=(x[4] \boxplus x[0]) \lll 9 & x[13] \oplus\!=(x[9] \boxplus x[5]) \lll 9 \\ x[2] \oplus\!=(x[14] \boxplus x[10]) \lll 9 & x[7] \oplus\!=(x[3] \boxplus x[15]) \lll 9 \end{array}\right. \\
3 \left[\begin{array}{ll} x[12] \oplus\!=(x[8] \boxplus x[4]) \lll 13 & x[1] \oplus\!=(x[13] \boxplus x[9]) \lll 13 \\ x[6] \oplus\!=(x[2] \boxplus x[14]) \lll 13 & x[11] \oplus\!=(x[7] \boxplus x[3]) \lll 13 \end{array}\right. \\
4 \left[\begin{array}{ll} x[0] \oplus\!=(x[12] \boxplus x[8]) \lll 18 & x[5] \oplus\!=(x[1] \boxplus x[13]) \lll 18 \\ x[10] \oplus\!=(x[6] \boxplus x[2]) \lll 18 & x[15] \oplus\!=(x[11] \boxplus x[7]) \lll 18 \end{array}\right.
\end{array}\right. \\
R2 & \left[\begin{array}{l}
5 \left[\begin{array}{ll} x[1] \oplus\!=(x[0] \boxplus x[3]) \lll 7 & x[6] \oplus\!=(x[5] \boxplus x[4]) \lll 7 \\ x[11] \oplus\!=(x[10] \boxplus x[9]) \lll 7 & x[12] \oplus\!=(x[15] \boxplus x[14]) \lll 7 \end{array}\right. \\
6 \left[\begin{array}{ll} x[2] \oplus\!=(x[1] \boxplus x[0]) \lll 9 & x[7] \oplus\!=(x[6] \boxplus x[5]) \lll 9 \\ x[8] \oplus\!=(x[11] \boxplus x[10]) \lll 9 & x[13] \oplus\!=(x[12] \boxplus x[15]) \lll 9 \end{array}\right. \\
7 \left[\begin{array}{ll} x[3] \oplus\!=(x[2] \boxplus x[1]) \lll 13 & x[4] \oplus\!=(x[7] \boxplus x[6]) \lll 13 \\ x[9] \oplus\!=(x[8] \boxplus x[11]) \lll 13 & x[14] \oplus\!=(x[13] \boxplus x[12]) \lll 13 \end{array}\right. \\
8 \left[\begin{array}{ll} x[0] \oplus\!=(x[3] \boxplus x[2]) \lll 18 & x[5] \oplus\!=(x[4] \boxplus x[7]) \lll 18 \\ x[10] \oplus\!=(x[9] \boxplus x[8]) \lll 18 & x[15] \oplus\!=(x[14] \boxplus x[13]) \lll 18 \end{array}\right.
\end{array}\right.
\end{array}
$$

Fig. 3: Salsa20 Hashing Algorithm [9]. Operation $\oplus$ is bitwise exclusive OR and $\boxplus$ is addition modulo 2^{32}, and $\lll$ is left rotate. Each set of four assignments numbered 1–8 is a *quarter round*, and each *round*, $R1$ and $R2$, consists of four quarter rounds each. The algorithm consists of repeating each *double round* ($R1; R2$) ten times in succession. Argument x is a 16 element array of 32 bit words.

4) Interdevice Communication & Feedback: Making interconnections between devices occurs using another device level operator, `refold`. The `refold` operator can be used to connect sub-devices within its third argument and to hide internal connections as well. The use of `refold` is illustrated in Fig. 2**(c)**. Given a device `d` :: `Dev` $\mathtt{i_1}$ $\mathtt{o_1}$, and two pure functions, `out` :: $\mathtt{o_1}$ `->` $\mathtt{o_2}$ and `conn` :: ($\mathtt{o_1}$ `->` $\mathtt{i_2}$ `->` $\mathtt{i_1}$), `refold out conn d` is a new device with the following behavior. Given an external input $\mathtt{i}'$ and current value output `o` by internal device `d`, the new input to `d` is `conn o` $\mathtt{i}'$ and the new external output is `out o`. The type of `refold` is:

```
refold :: (o₁ -> o₂)        ->
          (o₁ -> i₂ -> i₁)  ->
          Dev i₁ o₁         ->
          Dev i₂ o₂
```

5) Defining a Pipeline: The form of pipeline we consider is a simple one, namely stall-free pipelines, in which the output from a stage flows directly into the input of the next stage. It is possible to define more complex pipelines (e.g., instruction pipelines that stall, etc.) with Connect Logic, but we leave that subject for a follow-on publication.

Stall-free pipelines—henceforth simply "pipelines"—have the flavor of functional composition, and the architectural combinators of ReWire allow the formalization of this intuition. For functions, $\mathtt{f_j}$, of appropriate type, the composition, $\mathtt{f_n} \circ \cdots \circ \mathtt{f_1}$, resembles a pipeline. Of course, this ignores the timing aspect of a pipeline. In ReWire, we can express this pipeline, along with its timing, as the following:

$$\mathtt{iter}\,\mathtt{f_1}\,\mathtt{o_1} \rightsquigarrow \cdots \rightsquigarrow \mathtt{iter}\,\mathtt{f_n}\,\mathtt{o_n}$$

where $\mathtt{f_j}$ :: $\mathtt{a_j}$ `->` $\mathtt{a_{j+1}}$ are pure functions from input of type $\mathtt{a_j}$ to output of type $\mathtt{a_{j+1}}$ and each $\mathtt{o_j}$ :: $\mathtt{a_{j+1}}$ is the initial output value produced by pipeline stage $\mathtt{iter}\,\mathtt{f_j}\,\mathtt{o_j}$. The $\rightsquigarrow$ combinator chains each stage together, connecting the

```
salsa20 :: W128 -> Hex W32
salsa20 nonce = hash (initialize key0 key1 nonce)

hash :: Hex W32 -> Hex W32
hash x = x + doubleround(··· (doubleround(x)) ··· )
             \_________________________________/
                             10

doubleround :: Hex W32 -> Hex W32
doubleround x = rowround (columnround x)

quarterround :: Quad W32 -> Quad W32
quarterround (y0, y1, y2, y3) = (z0, z1, z2, z3)
  where
    z1 = y1 ⊕ (y0 + y3) ⋘ 7
    z2 = y2 ⊕ (z1 + y0) ⋘ 9
    z3 = y3 ⊕ (z2 + z1) ⋘ 13
    z0 = y0 ⊕ (z3 + z2) ⋘ 18

rowround :: Hex W32 -> Hex W32
rowround (y0, ..., y15) = (z0, ..., z15)
  where
    (z0, z1, z2, z3)     = quarterround (y0, y1, y2, y3)
    (z5, z6, z7, z4)     = quarterround (y5, y6, y7, y4)
    (z10, z11, z8, z9)   = quarterround (y10, y11, y8, y9)
    (z15, z12, z13, z14) = quarterround (y15, y12, y13, y14)

columnround :: Hex W32 -> Hex W32
columnround (x0, ..., x15) = (y0, ..., y15)
  where
    (y0, y4, y8, y12)  = quarterround (x0, x4, x8, x12)
    (y5, y9, y13, y1)  = quarterround (x5, x9, x13, x1)
    (y10, y14, y2, y6) = quarterround (x10, x14, x2, x6)
    (y15, y3, y7, y11) = quarterround (x15, x3, x7, x11)
```

Fig. 4: Reference Specification of Salsa20 Hash Function [2], which plays the rôle of reference specification in our case study. Operation $\oplus$ is bitwise exclusive OR, $+$ is addition modulo 2^{32}, and $\lll$ is left rotate.

output of the $\mathtt{j}^{\mathtt{th}}$ stage to the input of the $\mathtt{j{+}1}^{\mathtt{th}}$ stage. The combinators for pipelining, etc., are defined below.

Note that $\rightsquigarrow$ is not syntactic sugar for function composition. For example, while it is true that $\mathtt{id} \circ \mathtt{f} = \mathtt{f}$, it is also the case that $\mathtt{iter\ id\ o_1} \rightsquigarrow \mathtt{iter\ f\ o_2} \neq \mathtt{iter\ f\ o_2}$. The LHS of this inequality is a two stage pipeline while the RHS is a one stage pipeline. The outputs both pipelines produce will be related, of course.

Given two devices, d1 and d2, the ReWire code for connecting them in pipelined sequence is below. This construction is illustrated in Fig. 2(**d**). The two devices are first placed unconnected in parallel (i.e., d1 <&> d2 :: Dev (a, b) (b, c)) and, in this context, both devices operate in isolation. The combined device consumes a single input of type (a, b) and produces a single output of type (b, c). The output type for (d1 $\rightsquigarrow$ d2) is c; i.e., the second component of the output tuple of d1 <&> d2. The external input (of type a) to (d1 $\rightsquigarrow$ d2) is passed to the subdevice d1 and the output of d1 to the input of d2; thus the routing function pipe is as defined below:

```
(⇝) :: Dev a b -> Dev b c -> Dev a c
d1 ⇝ d2 = refold snd pipe (d1 ⟨&⟩ d2)
  where
    pipe (b, c) a = (a, b)
```

```
sls20dev :: Dev (Bit, W128) (Hex W32)
sls20dev = refold out conn (passthru ⟨&⟩ dblrd)

zeros    :: Hex W32
zeros    = ⟨...sixteen all zero words...⟩

dblrd :: Dev (Hex W32) (Hex W32)
dblrd = iter doubleround (doubleround zeros)

passthru :: Dev (Hex W32) (Hex W32)
passthru = iter id zeros

out      :: (Hex W32, Hex W32) -> Hex W32
out ((x0, ..., x15), (y0, ..., y15)) = (x0+y0, ..., x15+y15)

conn :: (Hex W32, Hex W32) ->
          (Bit, W128) -> (Hex W32, Hex W32)
conn (o1, o2) (Low, nonce)     = (o1, o2)
conn (o1, o2) (High, nonce))   = (x, x)
  where
    x = initialize key0 key1 nonce
```

Fig. 5: Iterative Salsa20 Device in ReWire.

6) Compiling Connect Logic: A pure function f in ReWire will be compiled into a combinational circuit of fixed depth that, in turn, determines a fixed delay. If $\mathtt{f} = \mathtt{f_n} \circ \cdots \circ \mathtt{f_1}$, then its depth is additive as is its delay. Composition of pure functions exposes an opportunity for a pipelining optimization to reduce the average propagation delay of the entire circuit.

The two operators ⟨&⟩ and refold are treated as primitives in the ReWire compilation process. These operations correspond directly to structural features in generated VHDL. The ⟨&⟩ operation is compiled to a single VHDL entity that handles the combined IO of two ReWire devices, and port maps it accordingly. The refold operator is a single entity with included functions to manipulate the IO of a device in the manner prescribed by the type of the refold function.

IV. Provably Correct Development of Salsa20 Devices in ReWire and Connect Logic

Salsa20 is a stream cipher developed by Bernstein [9] and is part of the ECRYPT ESTREAM [10] portfolio of cryptographic ciphers. Salsa20 was originally intended for software implementation, but can also be synthesized on an FPGA with careful consideration given to space and mapping constraints. Fig. 3 presents the Salsa20 hashing algorithm, which is the heart of the Salsa20 algorithm itself and where the bulk of its computation occurs. The inputs to the algorithm include a 16-element array of 32 bit words, called x in the figure.

A. Salsa20 Reference Specification

Fig. 4 contains the reference specification for Salsa20. This specification simply recasts Bernstein's functional specification [2] using Haskell syntax. The function hash formulates the original specification from Fig. 3 and the function salsa20 is the entry point for the whole algorithm. There are certain details which we have left out of this code for the sake of brevity and comprehensibility; these include routines to change endianness, to reformat words as sequences of bytes, and

$$
\begin{array}{l}
\texttt{pipe10} :: \texttt{Dev W128 (Hex W32)} \\
\texttt{pipe10} = \texttt{refold out inpt tenstage} \\
\quad \textbf{where} \\
\qquad \texttt{tenstage} = \underbrace{\texttt{stage} \rightsquigarrow \cdots \rightsquigarrow \texttt{stage}}_{10} \\
\qquad \texttt{stage} \quad\; = \texttt{passthru} \langle\&\rangle\, \texttt{dblrd}
\end{array}
$$

Fig. 6: Ten Stage Pipeline

similar such routines. The function `initialize` sets up the initial input; its definition is omitted as well.

B. Salsa20 Iterative Implementation

Fig. 5 contains the additional ReWire code to create an iterative version of Salsa20. Two devices are created, `dblrd` and `passthru`, using the `iter` constructor in Connect Logic. A diagrammatic view of the circuit produced is found in Fig. 8(a). Synthesis estimates of resource usage and FMax for `sls20dev` are in Table I.

There is one functional unit performing the `doubleround` operation. This operates ten cycles to produce on answer. When the inputs to the device `sls20dev` are $[(\texttt{High}, \texttt{n}), (\texttt{Low}, \texttt{n}_0), \ldots, (\texttt{Low}, \texttt{n}_9), \ldots]$, then, on the cycle with input $(\texttt{Low}, \texttt{n}_9)$, the output will be `salsa20 n`. The `High` bit signifies that the device should start hashing `n`. The $(\texttt{Low}, \texttt{n}')$ input signifies that $\texttt{n}'$ should be ignored and that the iteration should continue.

C. Pipelining Salsa20

The numbers for the iterative device are reasonable, but the structure of the cipher algorithm would indicate that there is room for improvement. There is an apparent performance gap with this approach: nine cycles of the device do not yield useful output. Pipelining our base components together gives us a way to keep our performance characteristics with respect to clock speed roughly the same while enabling our device to be productive on every clock cycle. We do so by placing ten different `passthru` ⟨&⟩ `dblrd` devices in sequence, connecting their inputs and outputs together to obtain `pipe10` in Fig. 6.

A twenty stage pipeline may be created by increasing the granularity of each stage. Now, instead of staging each `doubleround` as before, each component `columnround` and `rowround` is staged (see Fig. 7).

V. Evaluating Provably Correct Salsa20 Devices

This section evaluates the devices created in the previous section according to two modes: performance and verification. The devices synthesized by the ReWire compiler exhibit performance comparable to a previously published, hand optimized design [11] We sketch the verification of general theorem which characterizes the correctness of the pipelining transformation applied in Section IV-C.

In this section, we sketch the verification of the pipelining transformation defined in Section IV. There is a function of the following type that serves to run a device on stream of inputs: `feed` :: [i] -> Dev i o -> [o]. For a stream of inputs

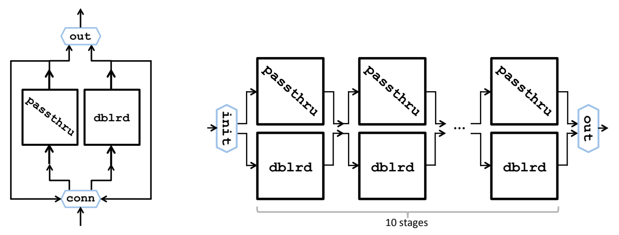
$$
\begin{array}{l}
\texttt{crstage} = \texttt{passthru} \langle\&\rangle\, \texttt{crdev} \\
\quad \textbf{where} \\
\qquad \texttt{crdev} = \texttt{iter columnround (columnround zeros)} \\
\\
\texttt{rrstage} = \texttt{passthru} \langle\&\rangle\, \texttt{rrdev} \\
\quad \textbf{where} \\
\qquad \texttt{rrdev} = \texttt{iter rowround (rowround zeros)} \\
\\
\texttt{pipe20} = \left(\begin{array}{c}
\texttt{crstage} \rightsquigarrow \texttt{rrstage} \rightsquigarrow \\
\vdots \\
\texttt{crstage} \rightsquigarrow \texttt{rrstage} \rightsquigarrow \\
\texttt{crstage} \rightsquigarrow \texttt{rrstage}
\end{array} \right) \quad (10)
\end{array}
$$

Fig. 7: Twenty Stage Pipeline.

`is` :: [i] and a device `d` :: `Dev i o`, `feed is d` is the stream of outputs created by running the device `d` on `is`. N.b., `feed` preserves the order of outputs with respect to inputs; i.e., if `i` is the n^{th} input in `is`, then the $(n+1)^{st}$ item in `feed is d` was produced by `d` on `i`. We omit the definition of `feed`.

A. Performance

We evaluated the performance of the VHDL generated from our high level specifications by synthesizing it using Xilinx ISE targeting a Kintex 7 FPGA (xc7k160t-3fbg676). The synthesis results detailed in Table I show an increase in throughput and resource utilization as we pipeline that is in line with intuitive expectations. The 10-stage pipeline and the iterative implementation are the same design core replicated tenfold. We observe a nearly tenfold increase of flip-flop usage and a notable increase in LUT usage (likely impacted by optimizations in the synthesis tools). In the 20-stage pipeline, we divide our basic unit into separate `rowRound` and `columnRound` pipeline stages. This introduces some addtional LUT usage, but doubles flip-flop (slice) usage because the number of stages in the pipeline are doubled. The maximum frequency of the 20-stage pipeline increases by approximately 1.7 times which indicates a doubling effect from doubling the pipeline with a moderate amount of overhead. These numbers demonstrate that our approach is competitive with similar work in the area of synthesizing Salsa20 [11] on modern FPGAs.



(a) Iterative Salsa20 (b) Ten Stage Pipelined Salsa20 Device

Fig. 8: Diagrammatic views of circuits produced by ReWire in Figs. 5 and 6. The `conn`, `out` and `init` blocks are combinational logic produced by compiling the functions of the same name. Code for `init` has been omitted.

B. Testing the Iterative Salsa20 Device Automatically

We used the QuickCheck tool [12] to test the putative correctness of the relationship between the reference specification `salsa20` and the iterative ReWire definition `sls20dev` (from Figs. 4 and 5, resp.). Below is a `Bool`-valued function, `test`, that takes a `W128` nonce `n` as input and computes an equation. Note that the value of input stream `is` is of the form $[(\mathtt{High}, \mathtt{n}), (\mathtt{Low}, \mathtt{undefined}), (\mathtt{Low}, \mathtt{undefined}), \cdots]$ where `undefined` is a special "don't care" constant built-in to Haskell.

```
test :: W128 -> Bool
test n = reference == iterative
   where
       reference = salsa20 n
       iterative = nth 10 (feed is sls20dev)
       is = (High,n) : repeat (Low,undefined)
```

QuickCheck can generate random inputs to `test` and, if `test` returns `True` for each input, then QuickCheck remarks that the tests were passed; below is a transcript of running QuickCheck on this correctness condition for `sls20dev`:

```
GHCi, version 7.10.1: http://www.haskell.org/
    ghc/  :? for help
[1 of 1] Compiling Salsa20          ( Salsa20.
    hs, interpreted )
Ok, modules loaded: Salsa20.
*Salsa20> quickCheck test
+++ OK, passed 100 tests.
*Salsa20>
```

The correctness condition is neatly summed up in the following theorem (stated without proof):

Theorem 1 (Correctness of Iterative Salsa20)**.** *For all nonces* $\mathtt{n}, \mathtt{n}_0, \ldots, \mathtt{n}_9 :: \mathtt{W128}$*, assume input stream* `is` *has the form* $[(\mathtt{High}, \mathtt{n}), (\mathtt{Low}, \mathtt{n}_0), \cdots, (\mathtt{Low}, \mathtt{n}_9), \ldots]$*. Then, the following equation holds:* $\mathtt{salsa20\,n} = \mathtt{nth}\,10\,(\mathtt{feed\,is\,sls20dev})$.

C. Verification of Pipelining

1) Lemmas: This section states the Lemmas used in proving the correctness of pipelining (Theorem 2 below). Each lemma is left unproven, although we describe the intuitive meaning of each.

Lemma 1 says that the pipelining operator is associative. The associativity of $\rightsquigarrow$ allows for "parentheses to be dropped"; i.e., $(\mathtt{f} \rightsquigarrow \mathtt{g} \rightsquigarrow \mathtt{h})$ can stand for either the right- or left-hand sides of the equation in the lemma.

Lemma 1 (Associativity)**.** *The* $\rightsquigarrow$ *operation is associative.*

$$\mathtt{f} \rightsquigarrow (\mathtt{g} \rightsquigarrow \mathtt{h}) = (\mathtt{f} \rightsquigarrow \mathtt{g}) \rightsquigarrow \mathtt{h}$$

	LUTs	Slices	Fmax (MHz)	T (Gbit/s)
Iterative	3459	651	99.4	5.1
10 Stage	22840	6019	97.5	49.9
20 Stage	25519	12309	167.4	85.7

TABLE I: Resource usage, Fmax, and throughput (T) of the Salsa20 algorithm as implemented and compiled in ReWire.

Lemma 2 relates stages in a pipeline of devices created with `iter`. The LHS below performs `f` and `g` in succession. The RHS performs `f` and `g` in the first stage and the identity function in the second stage. N.b., the RHS is *not* identical to $\mathtt{iter}\,(\mathtt{g} \circ \mathtt{f})\,(\mathtt{g}\,\mathtt{o}_2)$ because the former has two stages while the latter has one.

Lemma 2. *Let* $\mathtt{g} :: \mathtt{b} \to \mathtt{c}$*,* $\mathtt{f} :: \mathtt{a} \to \mathtt{b}$*,* $\mathtt{o}_1 :: \mathtt{c}$*, and* $\mathtt{o}_2 :: \mathtt{b}$*. Then, we have:*

$$\begin{aligned} &\mathtt{iter\,f\,o_2} \rightsquigarrow \mathtt{iter\,g\,o_1} \\ &\quad = \mathtt{iter}\,(\mathtt{g} \circ \mathtt{f})\,(\mathtt{g}\,\mathtt{o}_2) \rightsquigarrow \mathtt{iter\,id\,o_1} \end{aligned}$$

Lemma 3 relates `feed l` with $\rightsquigarrow$ in terms of infinite streams. It gives a condition under which the pipeline may be reduced by one stage.

Lemma 3. *Let* `l` *be an infinite stream and* $\varphi :: \mathtt{Dev\,i\,o}$*, then:* $\mathtt{feed\,l}\,(\varphi \rightsquigarrow \mathtt{iter\,id\,o}) = \mathtt{o} : \mathtt{feed\,l}\,\varphi$

Lemma 4 characterizes the interaction of `feed` and `iter` in terms of a stream recording the outputs of device argument to `feed`. The first is just the initial output of one stage pipeline device $(\mathtt{iter\,f\,o})$ and the rest are simply `f` mapped onto `l`.

Lemma 4. *Let* $\mathtt{l} :: [\mathtt{i}]$ *be an infinite stream and* $\mathtt{f} :: \mathtt{i} \to \mathtt{o}$*. Then,* $\mathtt{feed\,l}\,(\mathtt{iter\,f\,o}) = \mathtt{o} : \mathtt{map\,f\,l}$*.*

2) Correctness Theorem: The following theorem says that `feed`ing an `n`-stage pipeline a stream of inputs is the same as mapping a composite function across those inputs, as long as the first `n` outputs are ignored.

Theorem 2 (Correctness of Pipelining)**.** *Assuming that* $\mathtt{f} = \mathtt{f}_1 \circ \cdots \circ \mathtt{f}_n$ *and that* `l` *is an infinite stream, then:*

$$\begin{aligned} &\mathtt{map\,f\,l} \\ &\quad = \mathtt{drop\,n}\,(\mathtt{feed\,l}\,(\mathtt{iter\,f_n\,o_n} \rightsquigarrow \cdots \rightsquigarrow \mathtt{iter\,f_1\,o_1})) \end{aligned}$$

□

Proof:

First, define: $\mathtt{F}_0 = \mathtt{id}$ and $\mathtt{F}_{i+1} = \mathtt{F}_i \circ \mathtt{f}_{i+1}$. Observe that, by Lemmas 1 and 2 (n−1 times),

$$\begin{aligned} &\mathtt{iter\,f_n\,o_n} \rightsquigarrow \cdots \rightsquigarrow \mathtt{iter\,f_1\,o_1} \\ &\quad = \mathtt{iter\,F_n}\,(\mathtt{F_{n-1}\,o_n}) \\ &\qquad\quad \rightsquigarrow \mathtt{iter\,id}\,(\mathtt{F_{n-2}\,o_{n-1}}) \\ &\qquad\quad \rightsquigarrow \cdots \\ &\qquad\quad \rightsquigarrow \mathtt{iter\,id}\,(\mathtt{F_0\,o_1}) \\ &\{f = F_n, F_0 = \mathtt{id}\} \\ &\quad = \mathtt{iter\,f}\,(\mathtt{F_{n-1}\,o_n}) \\ &\qquad\quad \rightsquigarrow \mathtt{iter\,id}\,(\mathtt{F_{n-2}\,o_{n-1}}) \qquad (\ddagger) \\ &\qquad\quad \rightsquigarrow \cdots \\ &\qquad\quad \rightsquigarrow \mathtt{iter\,id\,o_1} \end{aligned}$$

Working from the RHS of the theorem statement:

$$
\begin{aligned}
& \mathtt{drop\,n\,(feed\,l\,(iter\,f_n\,o_n} \rightsquigarrow \cdots \rightsquigarrow \mathtt{iter\,f_1\,o_1))} \\
& \{\text{By } (\ddagger)\} \\
& = \mathtt{drop\,n\,(feed\,l} \left(\begin{array}{l} \mathtt{iter\,f\,(F_{n-1}\,o_n)} \\ \quad \rightsquigarrow \mathtt{iter\,id\,(F_{n-2}\,o_{n-1})} \\ \quad \rightsquigarrow \cdots \\ \quad \rightsquigarrow \mathtt{iter\,id\,o_1} \end{array} \right) \mathtt{)} \\
& \{ \text{ Lemma 3, } \mathtt{n}{-}1 \text{ times } \} \\
& = \mathtt{drop\,n\,(o_1 : \cdots : F_{n-2}\,o_{n-1} : feed\,l\,(iter\,f\,(F_{n-1}\,o_n)))} \\
& \{ \text{ } (\dagger) \text{, Section III-A1 } \} \\
& = \mathtt{drop\,1\,(feed\,l\,(iter\,f\,(F_{n-1}\,o_n)))} \\
& \{ \text{ Lemma 4 } \} \\
& = \mathtt{drop\,1\,(F_{n-1}\,o_n : map\,f\,l)} \\
& \{ \text{ Defn. } \mathtt{drop} \text{ } \} \\
& = \mathtt{map\,f\,l}
\end{aligned}
$$

■

VI. Summary and Conclusions

This paper considered the provably correct development of several reconfigurable designs and implementations of the Salsa20 stream cipher. The vehicle for this development is the ReWire language. ReWire is a sublanguage of the pure, functional language Haskell, and, as such, possesses a rigorous semantics that supports formal verification. Functional languages are generally quite expressive, and, consequently, the Salsa20 specifications in ReWire were quickly produced, concise and comprehensible, and elegant. Connect Logic—a previously unpublished part of ReWire—supports a structural style of development in a functional HDL. Connect Logic was key to rapidly prototyping Salsa20 in ReWire, especially in the introduction of pipelining optimizations to the specifications.

It is commonplace for hardware engineers to "think in diagrams". Any circuit or device specification will include a diagram depicting the high-level structure of the device. This diagram domain abstraction is used as an informal guide for comprehending the design. But how do we express such structural notions in a functional language-based HDL like ReWire? To this end, we introduced an extension to ReWire called Connect Logic, that encapsulates the diagrammatic style directly in the syntax of ReWire. This paper defines Connect Logic and illustrates its use with a case study of the construction of an efficient, pipelined hardware design and implementation of the Salsa20 stream cipher. Furthermore, and more to the point, we verify the correctness of this device through equational reasoning on the ReWire source text.

New language abstractions are not typically cost free. There is usually some trade-off with respect to performance and language implementers attempt to minimize such overheads. Furthermore, new abstractions tend to be more useful in some situations than in others. The Salsa20 cipher was chosen as a test for ReWire to evaluate (1) how well cryptographic algorithms might be expressed in ReWire and (2) what performance trade-off, if any, might arise with respect to carefully hand-optimized implementations? The performance of the synthesized ReWire devices (as shown in Table I) was quite good and, although there are not any published numbers on hand-optimized implementations of Salsa20 that afford direct comparison with our results, the achieved performance was in line with the only relevant publication in the area [11]. Question (1) concerns what is, admittedly, more of an aesthetic issue than a measurable quantity. Still, it is safe to say that the Salsa20 specifications in ReWire would be readily comprehensible to those with experience in functional programming.

More importantly, a clear advantage of the ReWire methodology is that the artifacts we produced were verified in the manner of ordinary functional programs directly on the text of the design. This is a point worth emphasizing: verification of ReWire programs takes place *on the program itself*. Because VHDL has no mathematical semantics, artifacts produced in VHDL (or in Verilog for that matter) would require an additional step in which the formal specification of the device would encoded by hand in the logic of a theorem prover [4]. This hand-encoding is fraught with the potential for error as well as being quite time-consuming.

References

[1] A. Procter, W. L. Harrison, I. Graves, M. Becchi, and G. Allwein, "Semantics driven hardware design, implementation, and verification with ReWire," in *ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, 2015.

[2] D. J. Bernstein, "Salsa20 specification," 2005, http://cr.yp.to/snuffle/spec.pdf.

[3] R. Bird and P. Wadler, *Introduction to Functional Programming.* Prentice Hall, 1988.

[4] T. Melham, *Higher Order Logic and Hardware Verification*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1993, vol. 31.

[5] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave, "The semantics of x86-CC multiprocessor machine code," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09, 2009, pp. 379–391.

[6] L. Erkök, D. McNamee, J. Kiniry, I. Diatchki, and J. Launchbury, *Programming Cryptol.* Galois Inc., 2014.

[7] NIAP-CCEVS, "Common criteria for information technology security evaluation part 3: Security assurance components," National Information Assurance Partnership, Tech. Rep. CCMB-2012-09-003, September 2012, https://www.niap-ccevs.org/.

[8] S. Peyton Jones, Ed., *Haskell 98 Language and Libraries, the Revised Report.* Cambridge University Press, 2003.

[9] D. J. Bernstein, "New stream cipher designs," M. Robshaw and O. Billet, Eds., 2008, ch. The Salsa20 Family of Stream Ciphers, pp. 84–97.

[10] ——, "The eSTREAM project - eSTREAM phase 3 - Salsa20 (portfolio profile 1)," 2005, retrieved November 11, 2014. [Online]. Available: http://www.ecrypt.eu.org/stream/salsa20pf.html

[11] J. Sugier, "Low-cost hardware implementations of salsa20 stream cipher in programmable devices," *Journal of Polish Safety and Reliability Association Summer Safety and Reliability Seminars*, vol. 4, no. 1, 2013.

[12] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, 2000, pp. 268–279.