Provably Correct Development of Reconfigurable Hardware Designs via Equational Reasoning

Ian Graves, Adam Procter, Bill Harrison & Gerard Allwein
FPT 2015
Provably Correct Development, Bird-Wadler Style

Reference Specification

\[
\begin{aligned}
fib &:: \text{Int} \rightarrow \text{Int} \\
fib\ 0 &\ =\ 0 \\
fib\ 1 &\ =\ 1 \\
fib\ (n + 1) &\ = \\
&\ \quad \ fib(n - 1) + fib(n)
\end{aligned}
\]
**Reference Specification**

\[
\text{fib} :: \text{Int} \to \text{Int} \\
\text{fib} \ 0 = 0 \\
\text{fib} \ 1 = 1 \\
\text{fib} \ (n + 1) = \text{fib}(n - 1) + \text{fib}(n)
\]

**Implementation**

\[
\text{fib2} :: \text{Int} \to (\text{Int}, \text{Int}) \\
\text{fib2} \ 0 = (0, 1) \\
\text{fib2} \ n = (b, a + b) \\
\text{where} \\
(a, b) = \text{fib2} \ (n - 1)
\]
Provably Correct Development, Bird-Wadler Style

Reference Specification

\[
\begin{align*}
  \text{fib} :: \text{Int} & \rightarrow \text{Int} \\
  \text{fib} \ 0 & = 0 \\
  \text{fib} \ 1 & = 1 \\
  \text{fib} \ (n + 1) & = \\
  & \text{fib}(n - 1) + \text{fib}(n)
\end{align*}
\]

Implementation

\[
\begin{align*}
  \text{fib2} :: \text{Int} & \rightarrow (\text{Int}, \text{Int}) \\
  \text{fib2} \ 0 & = (0, 1) \\
  \text{fib2} \ n & = (b, a + b) \\
  \text{where} \\
  & (a, b) = \text{fib2} \ (n - 1)
\end{align*}
\]

Linking Theorem

For all \( n \geq 0 \), \( \text{fib}(n) = \text{fst} (\text{fib2}(n)) \)
Equational Proof on the Code Itself

Lemma. For all \( n \geq 0 \), \( \text{fib2}(n) = (\text{fib}(n), \text{fib}(n + 1)) \)

Proof by Induction.

\begin{align*}
\text{\textbf{n=0} Inspection.} \\
\text{\textbf{n=k+1}} \\
\text{\textbf{fib2}(k + 1)} \\
&= (b, a + b) \text{ where } (a, b) = \text{fib2}(k) \\
&= (b, a + b) \text{ where } (a, b) = (\text{fib}(k), \text{fib}(k + 1)) \\
&= (\text{fib}(k + 1), \text{fib}(k) + \text{fib}(k + 1)) \\
&= (\text{fib}(k + 1), \text{fib}(k + 2))
\end{align*}
Overview

Bridging the Semantic Gap

- Pure functional languages support verification, HDLs don’t.

Experiment

- Salsa20, stream cipher developed by Daniel Bernstein
  - ECRYPT ESTREAM portfolio of cryptographic ciphers
- Derive verified Salsa20 implementations a’ la Bird-Wadler in ReWire

Contributions

- Bird-Wadler Repurposed to HW Design
  - Pure Functional HDL ReWire supports equational reasoning
- Mixed functional/structural style with Connect Logic
  - E.g., pipeline structuring with Connect Logic
- Several performant implementations of Salsa20 stream cipher
ReWire Functional Hardware Description Language

- Inherits Haskell’s good qualities
  - Pure functions & types, monads, equational reasoning, etc.
  - Formal denotational semantics [HarrisonKieburtz05,Harrison05]
- Types & operators for HW abstractions (“connect logic”).
- Formalizing ReWire in Coq Theorem Proving System
  - Support proof checking & compiler verification
Expressing Diagrams in ReWire with *Connect Logic*

\[ d :: \text{Dev io} \]

\[ d \uparrow i \downarrow o \]

\[ d \rightarrow_{\text{clk}} \]

\[ d \rightarrow_{\text{i}} \]

\[ d \leftrightarrow_{\text{o}} \]

Bill Harrison

FPT 2015
Expressing Diagrams in ReWire with *Connect Logic*

\[ d :: \text{Dev} \, i \, o \]

\[ d = \text{iter} \, f \]

**Bill Harrison**  
FPT 2015
Expressing Diagrams in ReWire with \textit{Connect Logic}

\begin{align*}
\text{d} &::= \text{Dev i o} \\
\text{d} &= \text{iter f} \\
\text{d} &\quad\triangleleft \quad \text{d1} &\quad\triangleleft \quad \text{d2}
\end{align*}
Expressing Diagrams in ReWire with *Connect Logic*

\[ d \colon \text{Dev} \ i \ o \]

\[ d \ = \ \text{iter} \ f \]

\[ \text{d1 \ (\&) \ d2} \]

\[ \text{refold out conn} \ d \]
Expressing Diagrams in ReWire with Connect Logic

d :: Dev i o

d = iter f

refold out conn d

\[
\begin{align*}
\text{d1 } \langle \& \rangle \text{ d2} \\
\text{d1 } \rightsquigarrow \text{ d2}
\end{align*}
\]
**Salsa20 Hashing Algorithm**

| $R_1$ | 1 | $x[4] \oplus (x[0] \boxtimes x[12]) \ll 7$ | $x[9] \oplus (x[5] \boxtimes x[1]) \ll 7$ |
|       |   | $x[14] \oplus (x[10] \boxtimes x[6]) \ll 7$ | $x[3] \oplus (x[15] \boxtimes x[11]) \ll 7$ |
|       | 2 | $x[8] \oplus (x[4] \boxtimes x[0]) \ll 9$ | $x[13] \oplus (x[9] \boxtimes x[5]) \ll 9$ |
|       |   | $x[2] \oplus (x[1] \boxtimes x[10]) \ll 9$ | $x[7] \oplus (x[3] \boxtimes x[15]) \ll 9$ |
|       | 3 | $x[12] \oplus (x[8] \boxtimes x[4]) \ll 13$ | $x[1] \oplus (x[13] \boxtimes x[9]) \ll 13$ |
|       |   | $x[6] \oplus (x[2] \boxtimes x[14]) \ll 13$ | $x[11] \oplus (x[7] \boxtimes x[3]) \ll 13$ |
|       | 4 | $x[0] \oplus (x[12] \boxtimes x[8]) \ll 18$ | $x[5] \oplus (x[1] \boxtimes x[13]) \ll 18$ |
|       |   | $x[10] \oplus (x[6] \boxtimes x[2]) \ll 18$ | $x[15] \oplus (x[11] \boxtimes x[7]) \ll 18$ |
| $R_2$ | 5 | $x[1] \oplus (x[0] \boxtimes x[3]) \ll 7$ | $x[6] \oplus (x[5] \boxtimes x[4]) \ll 7$ |
|       |   | $x[11] \oplus (x[10] \boxtimes x[9]) \ll 7$ | $x[12] \oplus (x[15] \boxtimes x[14]) \ll 7$ |
|       | 6 | $x[2] \oplus (x[1] \boxtimes x[0]) \ll 9$ | $x[7] \oplus (x[6] \boxtimes x[5]) \ll 9$ |
|       |   | $x[8] \oplus (x[11] \boxtimes x[10]) \ll 9$ | $x[13] \oplus (x[12] \boxtimes x[15]) \ll 9$ |
|       | 7 | $x[3] \oplus (x[2] \boxtimes x[1]) \ll 13$ | $x[4] \oplus (x[7] \boxtimes x[6]) \ll 13$ |
|       |   | $x[9] \oplus (x[8] \boxtimes x[11]) \ll 13$ | $x[14] \oplus (x[13] \boxtimes x[12]) \ll 13$ |
|       | 8 | $x[0] \oplus (x[3] \boxtimes x[2]) \ll 18$ | $x[5] \oplus (x[4] \boxtimes x[7]) \ll 18$ |
|       |   | $x[10] \oplus (x[9] \boxtimes x[8]) \ll 18$ | $x[15] \oplus (x[14] \boxtimes x[13]) \ll 18$ |

**Remarks**

- Assignments 1-8 are *quarter rounds*.
- Double round $R_1; R_2$ repeated ten times,
- $x$ is 16-element array of 32 bit words.
Reference Specification for Salsa20 Hash Function

- Bernstein’s functional spec. using Haskell syntax
- Not practical to synthesize as-is

\[ \text{salsa20} :: \text{W128} \rightarrow \text{Hex W32} \]
\[ \text{salsa20 nonce} = \text{hash} (\text{initialize key}_0 \text{ key}_1 \text{ nonce}) \]

\[ \text{hash} :: \text{Hex W32} \rightarrow \text{Hex W32} \]
\[ \text{hash } x = x + \underbrace{\text{doubleround} (\cdots (\text{doubleround} (x)) \cdots )}_{10} \]

\[ \text{doubleround} :: \text{Hex W32} \rightarrow \text{Hex W32} \]
\[ \text{doubleround } x = \text{rowround} (\text{columnround } x) \]

\[ \text{quarterround} :: \text{Quad W32} \rightarrow \text{Quad W32} \]
\[ \text{quarterround} (y_0, y_1, y_2, y_3) = \ldots \]

\[ \text{rowround} :: \text{Hex W32} \rightarrow \text{Hex W32} \]
\[ \text{rowround} (y_0, \ldots, y_{15}) = \ldots \]

\[ \text{columnround} :: \text{Hex W32} \rightarrow \text{Hex W32} \]
\[ \text{columnround} (x_0, \ldots, x_{15}) = \ldots \]
Iterative Salsa20 Hashing Device

\[
\text{sls20dev} :: \text{Dev} \ (\text{Bit}, \text{W128}) \ (\text{Hex W32}) \\
\text{sls20dev} = \text{refold} \ \text{out} \ \text{conn} \ (\text{passthru} \ (&) \ \text{dblrd})
\]

\[
\text{dblrd} :: \text{Dev} \ (\text{Hex W32}) \ (\text{Hex W32}) \\
\text{dblrd} = \text{iter} \ \text{doubleround} \ (\text{doubleround} \ \text{zeros})
\]

\[
\text{passthru} :: \text{Dev} \ (\text{Hex W32}) \ (\text{Hex W32}) \\
\text{passthru} = \text{iter} \ \text{id} \ \text{zeros}
\]

\[
\text{zeros} :: \text{Hex W32} \\
\text{zeros} = \langle \ldots\text{sixteen all zero words}\ldots\rangle
\]

\[
\text{out} :: (\text{Hex W32}, \text{Hex W32}) \to \text{Hex W32} \\
\text{out} \ ((x_0, \ldots, x_{15}), (y_0, \ldots, y_{15})) = (x_0 + y_0, \ldots, x_{15} + y_{15})
\]

\[
\text{conn} :: (\text{Hex W32}, \text{Hex W32}) \to \\
(\text{Bit}, \text{W128}) \to (\text{Hex W32}, \text{Hex W32})
\]

\[
\text{conn} \ (o_1, o_2) \ (\text{Low}, \text{nonce}) = (o_1, o_2) \\
\text{conn} \ (o_1, o_2) \ (\text{High}, \text{nonce}) = (x, x)
\]

\[
\text{where} \\
x = \text{initialize key} _0 \ \text{key} _1 \ \text{nonce}
\]
Linking Theorem

Theorem (Correctness of Iterative Salsa20)

For all nonces $n, n_0, \ldots, n_9 :: W128$ and input streams is of the form $[(High, n), (Low, n_0), \ldots, (Low, n_9), \ldots]$, then:

$$salsa20_n = nth 10 \ (feed \ is \ sls20dev)$$
Automated Testing with QuickCheck

Test Harness

test :: W128 -> Bool

\[
\text{test } n = \text{reference } == \text{iterative}
\]

\[
\text{where}
\]

\[
\text{reference} = \text{salsa20 } n \\
\text{iterative} = \text{nth 10 (feed is s1s20dev)} \\
\text{is} = (\text{High}, n) : \text{repeat (Low, undefined)}
\]

Running QuickCheck

GHCi, version 7.10.1.
*Salsa20> quickCheck test
+++ OK, passed 100 tests.
*Salsa20>
Pipelining Salsa20

10 Stage Pipelined Salsa20

pipe10 :: Dev W128 (Hex W32)
pipe10 = refold out inpt tenstage
where
  tenstage = stage \rightarrow \cdots \rightarrow stage
  \underbrace{10}_{\text{10 stages}}
  stage = passthru \langle \& \rangle dblrd
20 Stage Pipelined Salsa20

\[
\text{crstage} = \text{passthru}\langle&\rangle \text{crdev}
\]
\[\text{where}\]
\[
\text{crdev} = \text{iter} \text{columnround}(\text{columnround zeros})
\]

\[
\text{rrstage} = \text{passthru}\langle&\rangle \text{rrdev}
\]
\[\text{where}\]
\[
\text{rrdev} = \text{iter} \text{rowround}(\text{rowround zeros})
\]

\[
\text{pipe}^{20} = \begin{pmatrix}
\text{crstage} \rightsquigarrow \text{rrstage} \rightsquigarrow \\
\vdots \\
\text{crstage} \rightsquigarrow \text{rrstage} \rightsquigarrow \\
\text{crstage} \rightsquigarrow \text{rrstage}
\end{pmatrix} (\times 10)
\]
Correctness of Pipelining

Theorem (Correctness of Pipelining)

Assuming $f = f_1 \circ \cdots \circ f_n$ and $l$ is an infinite stream, then:

$$\text{map } f \ l = \text{drop } n \left( \text{feed } l \left( \text{iter } f_n \circ_n \leadsto \cdots \leadsto \text{iter } f_1 \circ_1 \right) \right)$$

Remarks

- Correctness of 10- and 20-stage pipelined versions of Salsa20 are direct consequences of this theorem.
Resource usage, Fmax, and throughput

<table>
<thead>
<tr>
<th></th>
<th>LUTs</th>
<th>Slices</th>
<th>Fmax (MHz)</th>
<th>T (Gbit/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterative</td>
<td>3459</td>
<td>651</td>
<td>99.4</td>
<td>5.1</td>
</tr>
<tr>
<td>10 Stage</td>
<td>22840</td>
<td>6019</td>
<td>97.5</td>
<td>49.9</td>
</tr>
<tr>
<td>20 Stage</td>
<td>25519</td>
<td>12309</td>
<td>167.4</td>
<td>85.7</td>
</tr>
</tbody>
</table>

Remarks

- Using XiLinx ISE, targeting Kintex 7 FPGA
- Compares favorably with published hand-crafted Salsa20 VHDL implementation [Sugier 2013].
Related Work

- HW Synthesis from DSLs
  - Delite [Olukotun, Ienne, et al.]
  - DSLs and Language Virtualization
  - The “Three P’s” + Provability

- Functional HDLs
  - Chisel, Bluespec, Lava
  - ReWire design motivated by formal methods & security

- [Procter et al., 2015] produce a verified secure dual-core processor in ReWire

- Cryptol
Summary, Conclusions & Future Work

- ReWire artifacts verified as ordinary functional programs
  - Traditional HW verification “handcrafts” formal system models
  - “Bird-Wadler” style eliminates this requirement
    - Enabled by functional HDL ReWire
- Approach relies on semantically-faithful compiler
  - Mechanization in Coq; Compiler Verification
- Rewire is open source:
  https://github.com/mu-chaco/ReWire
THANKS!

* This research supported by the US National Science Foundation CAREER Award #00017806 and the US Naval Research Laboratory.