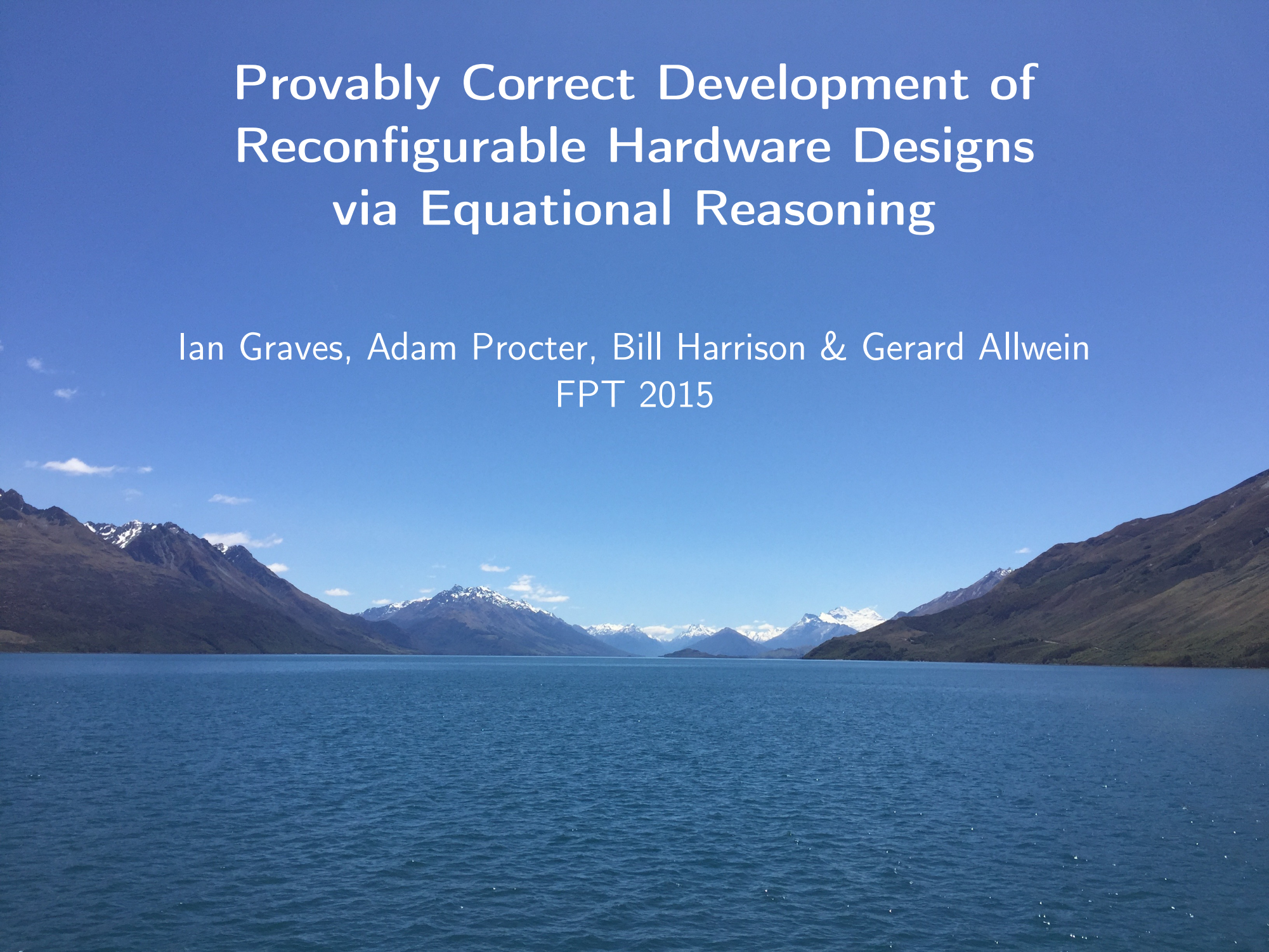
Provably Correct Development of Reconfigurable Hardware Designs via Equational Reasoning

Ian Graves, Adam Procter, Bill Harrison & Gerard Allwein

FPT 2015



Provably Correct Development, Bird-Wadler Style

Reference Specification

```
fib :: Int -> Int
fib 0       = 0
fib 1       = 1
fib (n + 1) =
  fib(n − 1) + fib(n)
```

Provably Correct Development, Bird-Wadler Style

Reference Specification

```
fib :: Int -> Int
fib 0       = 0
fib 1       = 1
fib (n + 1) =
   fib(n − 1) + fib(n)
```

Implementation

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

Provably Correct Development, Bird-Wadler Style

Reference Specification

```
fib :: Int -> Int
fib 0       = 0
fib 1       = 1
fib (n + 1) =
  fib(n − 1) + fib(n)
```

Implementation

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

Linking Theorem

For all $n \geq 0$, `fib(n)` $=$ `fst (fib2(n))`

Equational Proof on the Code Itself

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

Proof by Induction.

n=0 Inspection.

n=k+1

$$
\begin{aligned}
&\texttt{fib2(k+1)} \\
&\quad = (\texttt{b}, \texttt{a}+\texttt{b}) \text{ where } (\texttt{a},\texttt{b}) = \texttt{fib2(k)} \\
&\quad = (\texttt{b}, \texttt{a}+\texttt{b}) \text{ where } (\texttt{a},\texttt{b}) = (\texttt{fib(k)}, \texttt{fib(k+1)} \\
&\quad = (\texttt{fib(k+1)}, \texttt{fib(k)} + \texttt{fib(k+1)}) \\
&\quad = (\texttt{fib(k+1)}, \texttt{fib(k+2)})
\end{aligned}
$$

□

Overview

Bridging the Semantic Gap

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

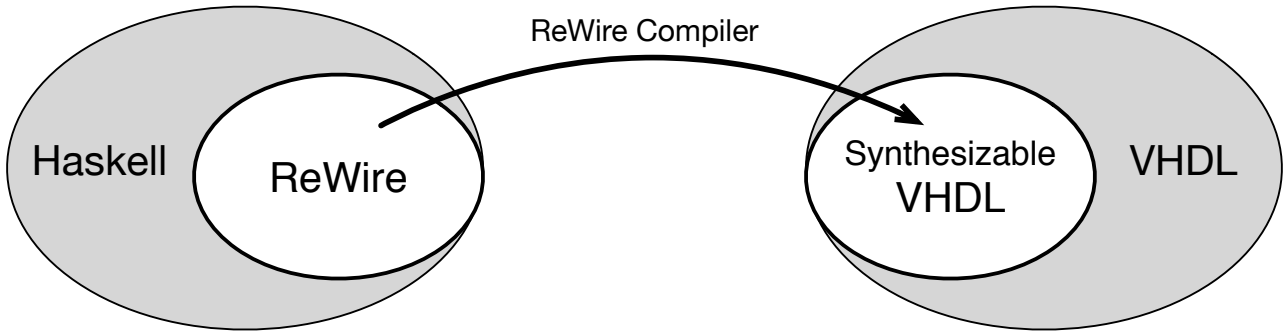
Experiment

- 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

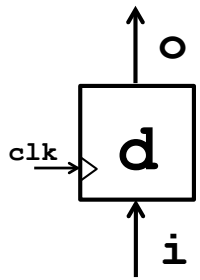
ReWire Functional Hardware Description Language



- 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

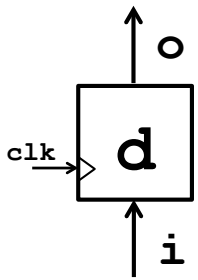
Expressing Diagrams in ReWire with *Connect Logic*

```
d :: Dev i o
```

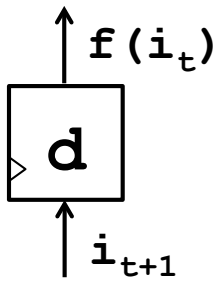


Expressing Diagrams in ReWire with *Connect Logic*

```
d :: Dev i o
```

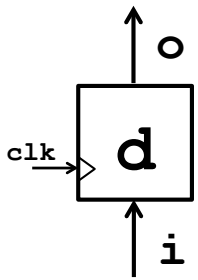



```
d = iter f
```

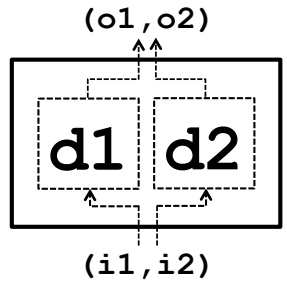


Expressing Diagrams in ReWire with *Connect Logic*

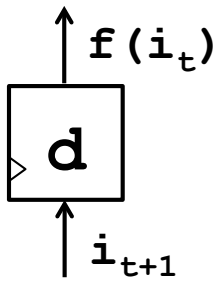
```
d :: Dev i o
```




```
d1 ⟨&⟩ d2
```

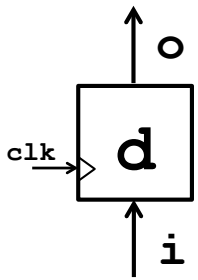



```
d = iter f
```

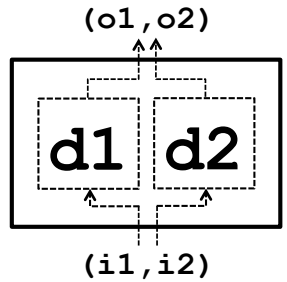


Expressing Diagrams in ReWire with *Connect Logic*

```
d :: Dev i o
```

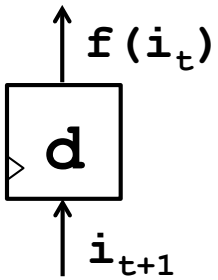


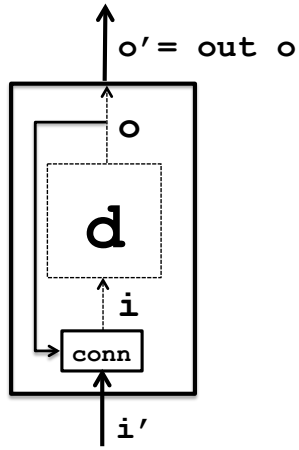

```
d1 ⟨&⟩ d2
```




```
refold out conn d
```

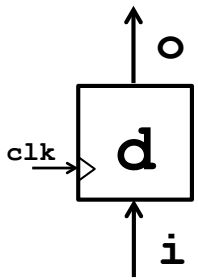
```
d = iter f
```



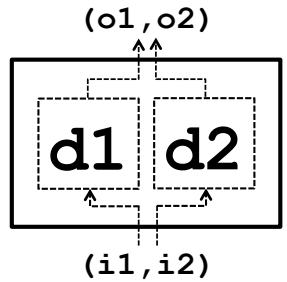


Expressing Diagrams in ReWire with *Connect Logic*

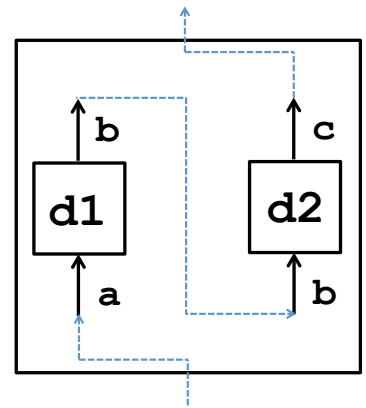
d :: Dev i o



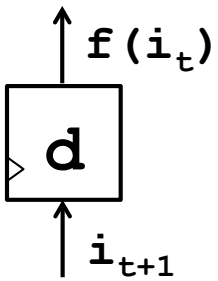
d1 ⟨&⟩ d2



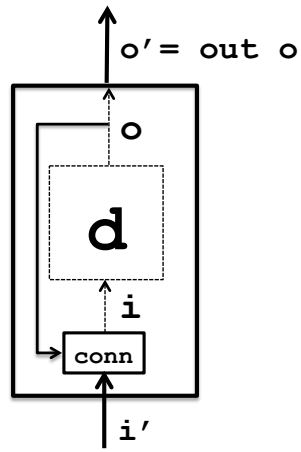
d1 ⇝ d2



d = iter f



refold out conn d



Salsa20 Hashing Algorithm

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

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

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

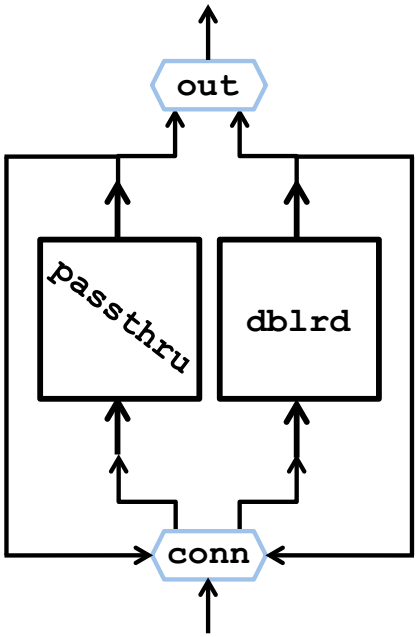
```
salsa20 :: W128 -> Hex W32
salsa20 nonce = hash (initialize key0 key1 nonce)

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) = ...
rowround :: Hex W32 -> Hex W32
rowround (y0, ..., y15) = ...
columnround :: Hex W32 -> Hex W32
columnround (x0, ..., x15) = ...
```

Iterative Salsa20 Hashing Device




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

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

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

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

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
```

Linking Theorem

Theorem (Correctness of Iterative Salsa20)

For all nonces $\mathtt{n}, \mathtt{n}_0, \ldots, \mathtt{n}_9 :: \mathtt{W128}$ *and input streams* $\mathtt{is}$ *of the form* $[(\mathtt{High}, \mathtt{n}), (\mathtt{Low}, \mathtt{n}_0), \cdots, (\mathtt{Low}, \mathtt{n}_9), \ldots]$*, then:*

$$\mathbf{salsa20}\ \mathtt{n} = \mathtt{nth}\ 10\ (\mathtt{feed}\ \mathtt{is}\ \mathbf{sls20dev})$$

Automated Testing with QuickCheck

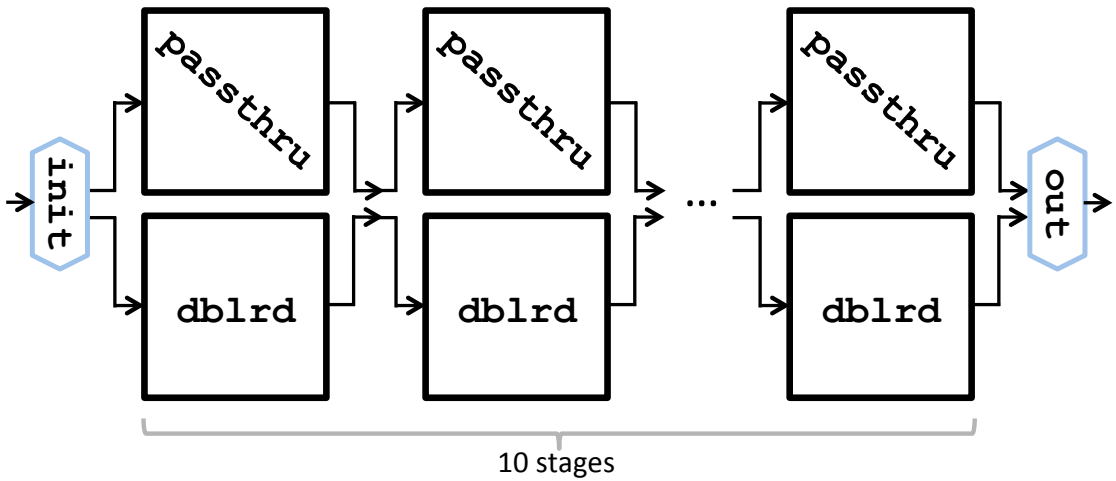
Test Harness

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

Running QuickCheck

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

10 Stage Pipelined Salsa20




```
pipe10 :: Dev W128 (Hex W32)
pipe10 = refold out inpt tenstage
  where
    tenstage = stage ⇝ ··· ⇝ stage
                     (10)
    stage    = passthru ⟨&⟩ dblrd
```

20 Stage Pipelined Salsa20

```
crstage = passthru ⟨&⟩ crdev
  where
    crdev = iter columnround (columnround zeros)

rrstage = passthru ⟨&⟩ rrdev
  where
    rrdev = iter rowround (rowround zeros)
```

$$\texttt{pipe20} = \left(\begin{array}{l} \texttt{crstage} \rightsquigarrow \texttt{rrstage} \rightsquigarrow \\ \vdots \\ \texttt{crstage} \rightsquigarrow \texttt{rrstage} \rightsquigarrow \\ \texttt{crstage} \rightsquigarrow \texttt{rrstage} \end{array} \right) (\times 10)$$

Correctness of Pipelining

Theorem (Correctness of Pipelining)

Assuming $\mathtt{f} = \mathtt{f}_1 \circ \cdots \circ \mathtt{f}_n$ *and* $\mathtt{l}$ *is an infinite stream, then:*

$$\begin{aligned}&\mathtt{map}\ \mathtt{f}\ \mathtt{l}\\ &\quad = \mathtt{drop}\ \mathtt{n}\ (\mathtt{feed}\ \mathtt{l}\ (\mathtt{iter}\ \mathtt{f}_n\ \mathtt{o}_n \rightsquigarrow \cdots \rightsquigarrow \mathtt{iter}\ \mathtt{f}_1\ \mathtt{o}_1))\end{aligned}$$

□

Remarks

- Correctness of 10- and 20-stage pipelined versions of Salsa20 are direct consequences of this theorem.

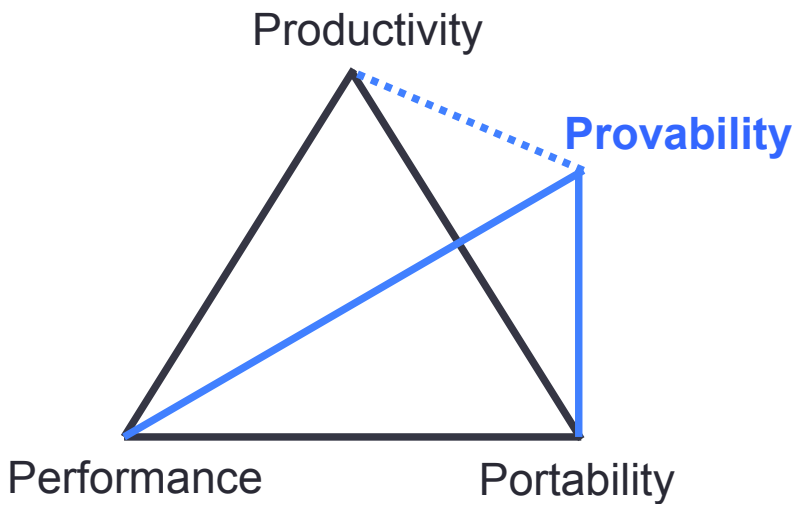
Resource usage, Fmax, and throughput

	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

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`

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