Formalized High Level Synthesis with Applications to Cryptographic Hardware*

Bill Harrison Ian Blumenfeld Eric Bond
Chris Hathhorn Paul Li May Torrence Jared Ziegler

High Assurance Solutions
Two Six Technologies, Inc.
Arlington Virginia

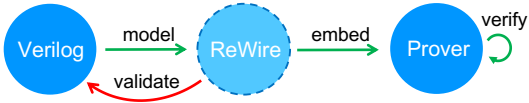
* This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) program *Data Protection in Virtual Environments* (DPRIVE). The views, opinions and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

The Story

- Hardware Engineering team creates $\mathcal{D}$ to implement cryptographic algorithm $\mathcal{A}$
  - $\mathcal{D}$: Verilog highly optimized by-hand for performance
  - $\mathcal{A}$: Specified with Math &/or informal pseudocode
- FM team to verify $\mathcal{D}$
  - Correctness of design $\mathcal{D}$ $\equiv$ Conformance to the algorithm $\mathcal{A}$
- Challenge: establish a precise connection between $\mathcal{A}$ and $\mathcal{D}$
  - $\mathcal{D}$ not created with verification in mind,
  - Complicated by the lack of formal specifications for both $\mathcal{A}$ and $\mathcal{D}$ and,
  - Disparity between the computational models of HDL and pseudocode (i.e., synchronous parallelism vs. imperative loop code).

Model Validation

Use Formalized HLS Language as a Bridge for $\mathcal{D}$ and $\mathcal{A}$

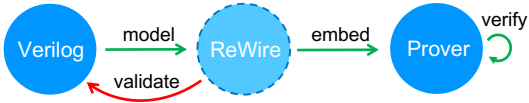


Model Validation Steps

1. **model** creates ReWire model $\mathcal{M}$ that "mimics" $\mathcal{D}$
2. **embed** translates $\mathcal{M}$ to ITP logic
3. **verify** proves conformance of $\mathcal{M}$ to $\mathcal{A}$
4. **validates** demonstrates cycle equivalence of compiled $\mathcal{M}$ to $\mathcal{D}$

Model Validation

Use Formalized HLS Language as a Bridge for $\mathcal{D}$ and $\mathcal{A}$



Model Validation

(model ; embed ; verify)

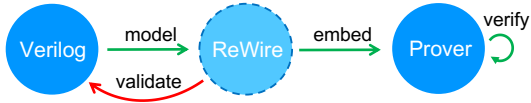
- Factors "semantic archaeology" through FHLS

(model ; validate)

- Checks faithfulness of $\mathcal{M}$ to $\mathcal{D}$

Model Validation

Use Formalized HLS Language as a Bridge for $\mathcal{D}$ and $\mathcal{A}$

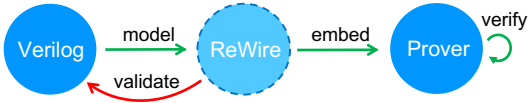


Model Validation Case Studies

1. Pipelined Barrett Modular Multipliers (word sizes = 64, 128, 256, 512, 1024).
2. Iterative Montgomery Modular Multipliers, (word sizes including 4096).

Model Validation

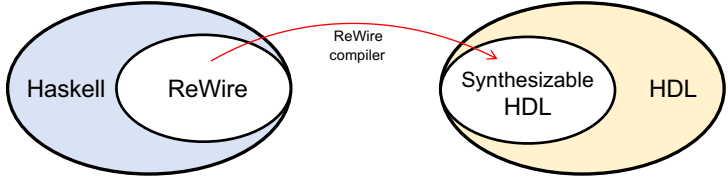
Use Formalized HLS Language as a Bridge for $\mathcal{D}$ and $\mathcal{A}$



Focus Today: embed

1. Mechanized Semantics for ReWire in multiple ITP systems: Isabelle, Coq, and Agda;
2. *Definitional Interpreters for Higher-Order Programming Languages* [Reynolds72]
   - ReWire formalized as computational λ-calculus [Moggi91]
   - Monads of *Reactive Resumption over State* defined as streams of "snapshots"

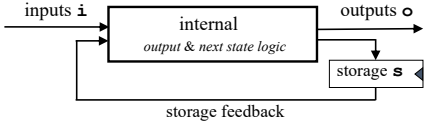
ReWire Language & Toolchain



- Inherits Haskell's good qualities
  - Pure functions, strong types, monads, equational reasoning, etc.
- ReWire compiler produces Verilog, VHDL, or FIRRTL
- Freely Available: `https://github.com/twosixlabs/rewire`
- Today: ReWire Formalization in ITP Systems (Isabelle, Coq, Agda)

Mealy Machines

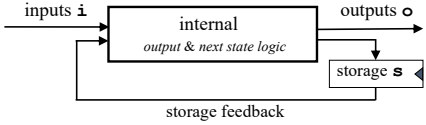
Diagrammatically and as ReWire Design Template




```
internal   :: i → ST s o
internal i = ...
onecycle   :: i → Re i s o i
onecycle i = lift (internal i) >>= λo. signal o
mealy      :: i → Re i s o ()
mealy i    = onecycle i >>= mealy
```

Mealy Machines

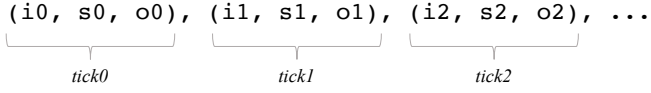
Diagrammatically and as ReWire Design Template




```
internal  :: i → ST s o
internal i = ...
onecycle  :: i → Re i s o i
onecycle i = lift (internal i) >>= λo. signal o
mealy     :: i → Re i s o ()
mealy i   = onecycle i >>= mealy
```

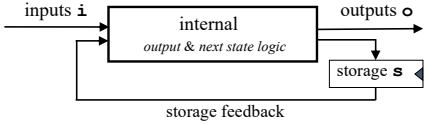
Extensional Model of Mealy Machine

Stream of "snapshots" of type $(\mathtt{i} \times \mathtt{s} \times \mathtt{o})$:



Mealy Machines

Diagrammatically and as ReWire Design Template



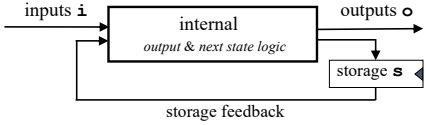

```
internal   :: i → ST s o
internal i = ...
onecycle   :: i → Re i s o i
onecycle i = lift (internal i) >>= λo. signal o
mealy      :: i → Re i s o ()
mealy i    = onecycle i >>= mealy
```

ReWire Effect Types (0, +, and ∞) track termination

Effect Typing	Terminates?	Signals?
`lift o internal : i → Re`^{0} `i s o o`	always	never
`onecycle : i → Re`$^{+}$ `i s o i`	always	at least once
`mealy : i → Re`$^{\infty}$ `i s o`	never	infinitely

Mealy Machines

Diagrammatically and as ReWire Design Template




```
internal   :: i → ST s o
internal i = ...
onecycle   :: i → Re i s o i
onecycle i = lift (internal i) >>= λo. signal o
mealy      :: i → Re i s o ()
mealy i    = onecycle i >>= mealy
```

Extensional Model of *Device* (i.e., term of type $\texttt{Re}^{\infty}\ \texttt{i s o}$)

$$(\texttt{i} \times \texttt{s} \times \texttt{o}) \rightarrow \texttt{Stream i} \rightarrow \texttt{Stream}\ (\texttt{i} \times \texttt{s} \times \texttt{o})$$

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

Running in GHCi

```
ghci> f 40 25 20
      (48,37)
ghci> f 41 25 20
      (50,36)
```

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = signal (f a b c) >>= csa
```

Carry-Save Adders in ReWire

Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = signal (f a b c) >>= csa
```

Behavior

```
((40,25,20),(),(0,0)), ((41,25,20),(),(48,37)), ((40,25,20),(),(50,36)), ...
```

tick0 *tick1* *tick2*

Carry-Save Adders in ReWire

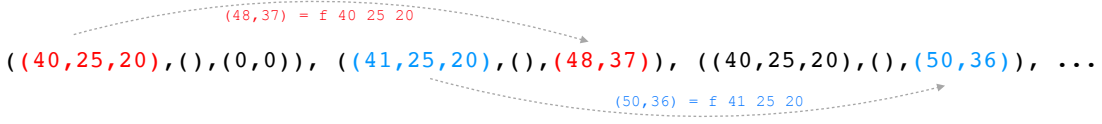
Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

CSA Device in ReWire

```
csa :: (W8, W8, W8) → Re (W8, W8, W8) () (W8, W8) ()
csa (a, b, c) = signal (f a b c) >>= csa
```

Behavior



Carry-Save Adders in ReWire

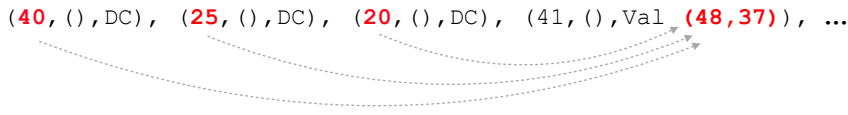
Carry-Save Addition (CSA) as Pure Function

```
f :: W8 → W8 → W8 → (W8, W8)
f a b c    = ( ((a & b) || (a & c) || (b & c) ) << '0' , a ⊕ b ⊕ c )
```

Pipelined CSA

```
data Ans a = DC | Val a    -- "don't care" and "valid"
pcsa :: W8 → Re W8 () (Ans (W8, W8)) ()
pcsa a     = signal DC >>= λb.
             signal DC >>= λc.
             signal (Val (f a b c)) >>= pcsa
```

Behavior



Type-Effect System for ReWire Calculus

Haskell, but not ReWire

```
bad :: i → Re i i o ()        -- Nope ; not signal-productive
bad i = lift (set i) >>= bad
```

Type-Effect System for ReWire Calculus

Haskell, but not ReWire

```
bad :: i → Re i i o ()          -- Nope ; not signal-productive
bad i = lift (set i) >>= bad
```

Effect Typing Rules

$$\frac{\Gamma \vdash x : \mathtt{ST}\ s\ a}{\Gamma \vdash \mathtt{lift}\ x : \mathtt{Re}^{0}\ i\ s\ o\ a}$$

$$\frac{\Gamma \vdash f : a \to \mathtt{Re}^{+}\ i\ s\ o\ a}{\Gamma \vdash \mathtt{iterRe}\ f : a \to \mathtt{Re}^{\infty}\ i\ s\ o\ a}$$

$$\frac{}{\Gamma \vdash \mathtt{signal} : i \to \mathtt{Re}^{+}\ i\ s\ o\ i}$$

Model

Input Verilog for BMM

```
module BMM (CLK, A_IN, B_IN, M_IN
          , mu_IN, km3_IN, Z_OUT);
 parameter N     = 128;
 parameter LOG_N = 7;
 input                CLK;
 input  [N-1     : 0] A_IN, B_IN, M_IN;
 input  [N+2     : 0] mu_IN;
 input  [LOG_N-1 : 0] km3_IN;
 output [N-1     : 0] Z_OUT;
 reg [2*N-1 : 0] stage0_XY_reg;
 reg [N+2   : 0] stage0_mu_reg;
                 .
                 .
                 .
```

Corresponding ReWire Mimic

```
type Inp = ( BV(N)        -- A_IN
           , BV(N)        -- B_IN
           , BV(N)        -- M_IN
           , BV(N + 3)    -- mu_IN
           , BV(LOG_N) )  -- km3_IN
type Out = BV(N)          -- Z_OUT

bmm :: Inp → Re Inp Reg Out ()
bmm i = do lift (internal i)
           i' ← signal (obs reg)
           bmm i'
    where
        internal :: Inp → ST Reg Out
        internal i = do r ← get
                        put (trans i r)
                        return_ST (obs r)
        trans :: Inp → Reg → Reg
        trans i r = ...
        obs :: Reg → Out
        obs r     = ...
```

Embed

RITE (Rewire Isabelle Tool for Embeddings) translates ReWire into Isabelle

Before (ReWire):

```
bmm :: Inp → Re Inp Reg Out ()
bmm i = do lift (internal i)
           i' ← signal (obs reg)
           bmm i'
    where
        ...
```

After (Isabelle):

```
type_synonym ('i,'s,'o) DomRe_INF =
   "('i × 's × 'o) ⇒ 'i stream ⇒ (('i × 's × 'o) stream)"

definition ⟦bmm⟧ :: "Inp ⇒ (Inp, Reg, Out) Dom_Re_INF"
   where "⟦bmm⟧ i = iterRe body (i)"
        ...
```

Verify

Correctness Specification

```
fun compute_bmm :: "128 word ⇒ 128 word ⇒ 128 word ⇒ 131 word ⇒ 7 word ⇒ 128 word"
 where
  "compute_bmm a b m mu km3 =
    π3 (stake 5 (⟦bmm⟧ (a,b,m,mu,km3) (i0,s0,o0)  (repeat (a,b,m,mu,km3))) ! 4)"

theorem embedding_eq : "compute_bmm a b m mu km3 = bmm_abstract a b m mu km3"
```

Intuition

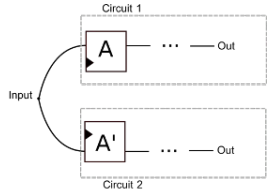
$$(i_0, s_0, o_0), (\mathbf{in}, s_1, o_1), \ldots, (\mathbf{in}, s_5, \mathbf{out})$$

$= \mathbf{bmm\_abstract\ in}$

Validate

Satisfaction Solver (SAT) Based Equivalence

- Final step uses sequential equivalence checking through YoSys/ABC to prove handwritten register transfer language (RTL) equivalent to ReWire Compiler (RWC) generated RTL
- YoSys proves circuit equivalence by temporal (k-)induction over clock cycles using a SAT solver (MiniSAT by default)



Summary, Conclusions, and Future Work

BMM Performance: Compiled ReWire vs. Handwritten Verilog

	Maximum Frequency (GHz)			Area (μm^2)		
Width	ReWire	Original	Δ%	ReWire	Original	Δ%
64	1.588	2.127	+25%	13399	12126	+10%
128	1.357	2.134	+36%	42970	41650	+3%
256	1.229	1.952	+37%	150463	157214	-4%
512	1.074	1.789	+40%	554612	578506	-4%
1024	0.954	1.473	+35%	2109037	2106714	+0.1%

Summary, Conclusions, and Future Work

BMM Performance: Compiled ReWire vs. Handwritten Verilog

	Maximum Frequency (GHz)			Area (μm^2)		
Width	ReWire	Original	Δ%	ReWire	Original	Δ%
64	1.588	2.127	+25%	13399	12126	+10%
128	1.357	2.134	+36%	42970	41650	+3%
256	1.229	1.952	+37%	150463	157214	-4%
512	1.074	1.789	+40%	554612	578506	-4%
1024	0.954	1.473	+35%	2109037	2106714	+0.1%

Hypothesis: "Boxing/unboxing" inputs in ReWire code generation

- Inputs passed as single bit-string; individual inputs projected
- E.g., `f :: Re∞ (i1, i2) s o`

Summary, Conclusions, and Future Work (cont'd)

- Accelerating the model step; some ideas:
  - Recent work [Zeng20,Zeng21] recovers update functions of type $\mathtt{i} \rightarrow \mathtt{s} \rightarrow (\mathtt{o} \times \mathtt{s})$ directly from Verilog designs
  - Compile pseudocode directly to ReWire
- Scalability
  - Isabelle automation critical to our approach
  - Fully automated approach seems unlikely to scale to, e.g., 4096-bit Montgomery Modular Multiplier

THANKS!