# Temporal Staging for Correct-by-Construction Cryptographic Hardware

Yakir Forman
*High Assurance Solutions Group*
*Two Six Technologies*
yakir.forman@twosixtech.com

William L. Harrison
*High Assurance Industrial Systems Group*
*Idaho National Laboratory*
william.lawrence.harrison@gmail.com

***Abstract*—There is a conceptual divide between the ways cryptographic algorithms are defined (i.e., informal imperative pseudocode) and commodity hardware design languages (e.g., Verilog). How does one even begin to compare a pseudocode to an HDL design that purports to implement it in hardware? Bridging this divide requires substantial manual intervention and, consequently, "shrinking the divide" can drastically reduce the cost of high-assurance cryptographic hardware by reducing the cost of formal verification. We present a correct-by-construction methodology for the functional hardware design language, ReWire, in which a reference cryptographic algorithm is transformed into a provably correct hardware design with a program transformation called *temporal staging*. We illustrate this methodology with case studies including one for the BLAKE2b cryptographic hash function. Because the reference algorithm, the temporal staging transformation, and the resulting implementation are all expressed in ReWire, formal verification can proceed immediately via a published ReWire semantics.**

***Index Terms*—Formal methods for hardware, Specification models and methodologies for hardware, Rapid-design approaches for application-specific processors.**

## I. Introduction

Correct-by-construction software methodologies—i.e., constructing correct implementations from abstract specifications by applying verifiable transformations—have been around for many decades [1]–[5], but, to the best knowledge of the authors, such techniques have never been applied to hardware languages. *Temporal staging* is a novel correct-by-construction (CbyC) methodology for cryptographic hardware developed by the authors and introduced here. The contributions of this article are: (1) we introduce the temporal staging methodology and transformation; and (2) we illustrate temporal staging with significant case studies including BLAKE2b. Temporal staging was successfully applied to produce a verified cryptographic component for the BLAKE2b hash function that is now included in an FHE ASIC in production.

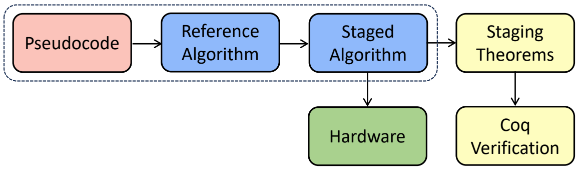


Fig. 1: High-level Overview of Temporal Staging Methodology. The focus of this article is on phases within the dotted line. Full explication of other phases (e.g., performance and formal verification) will be left to a follow-on publication.

Figure 1 presents the temporal staging methodology. The input is informal, imperative pseudocode defining a cryptographic function, a typical example of which is presented in Figure 2. Input pseudocode is formalized by transliterating it into ReWire (i.e., the "Reference Algorithm" in Figure 1) where it may be executed on test cases to gain confidence. The staging transformations (described in Section III) are then applied to obtain the "Staged Algorithm" in ReWire, that can be (1) compiled with the ReWire compiler to produce a performant implementation or (2) embedded into, and verified with, a theorem prover.

The genesis of this research is a project in high-assurance hardware design and implementation in which the authors were part of a team building a chip supporting fully homomorphic encryption (FHE) [6]. We were called upon to develop a performant, correct hardware implementation of the BLAKE2b cryptographic hash function. One approach would have been to repeat the methodology we had used previously on the project—i.e., let the hardware engineers produce Verilog for BLAKE2b and have the formal methods team verify its correctness using ReWire solely for formal modeling [7]. A second approach creates and verifies the BLAKE2b design in ReWire, and compiles the verified design to Verilog with the ReWire compiler. The second approach was attractive because it requires less time from both the hardware and formal methods teams. The first approach modeled hand-written Verilog produced by the hardware team with a meticulous, time-consuming manual process. The second approach came with risks, though, because the ReWire-produced BLAKE2b implementation had to have satisfactory performance to be used. The ReWire-produced implementation did, indeed, have satisfactory performance and is now part of an ASIC currently scheduled for production [6].

The remainder of this section introduces temporal staging. Section II presents the ReWire programming model and, in light of this model, Section III introduces temporal staging in ReWire. Section IV presents the first case study illustrating

This research was developed with funding from the Defense Advanced Research Projects Agency (DARPA) program *Data Protection in Virtual Environments* (DPRIVE).

```
FUNCTION G( v[0..15], a, b, c, d, x, y )
|
|    v[a] := (v[a] + v[b] + x) mod 2**w
|    v[d] := (v[d] ^ v[a]) >>> R1
|    v[c] := (v[c] + v[d])     mod 2**w
|    v[b] := (v[b] ^ v[c]) >>> R2
|    v[a] := (v[a] + v[b] + y) mod 2**w
|    v[d] := (v[d] ^ v[a]) >>> R3
|    v[c] := (v[c] + v[d])     mod 2**w
|    v[b] := (v[b] ^ v[c]) >>> R4
|
|    RETURN v[0..15]
|
END FUNCTION.
```

Fig. 2: Informal Pseudocode: This example (BLAKE2b Function G) taken directly from Saarinen and Aumasson [8].

temporal staging of a simple carry-save adder. This first case study is small enough to present in its entirety. Section V overviews the BLAKE2b development using temporal staging. This second case study is too large to present in its entirety here due to page restrictions; however, sufficient detail is presented to demonstrate that temporal staging scales up. Section VI reviews related work. Section VII summarizes this work and outlines future directions.

### *Temporal Staging for CbyC Cryptographic Hardware*

Figure 3 illustrates the temporal staging methodology and the program transformation it is based on. The starting point for temporal staging is a cryptographic algorithm defined as imperative pseudocode—this is how such algorithms are frequently defined [8]–[10]. Figure 2 presents such an example—the pseudocode for the "mixing" function G of BLAKE2b as it appears in its official definition [8]. Given an imperative pseudocode for a cryptographic algorithm, it is transliterated into an executable reference algorithm in Haskell (Figure 3a) written in terms of a state monad (i.e., Storage as described below in Section II). (N.b., the reference algorithm in Figure 3a is generic and not intended to represent the G function from Figure 2.) The resulting reference algorithm is executable and can be evaluated against test cases that are typically provided in Haskell.

Figure 3b displays the results of the temporal staging transformation to the reference algorithm. The precise definitions of the staging functions, stagei and stage, are given below in Section III, but, intuitively for now, they are constructors for Mealy machines that create machine transitions from imperative actions (c1 x), (c2 y), and (c3 z). The codomain of staged reflects this Mealy construction. The reference algorithm returns its value a; the staged algorithm places it on the Mealy machine's output port with signal (Val a).

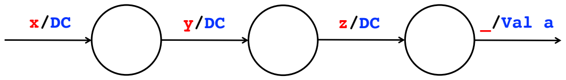
Reference algorithms (Figure 3a) define the correctness standard and conformance to this standard by the transformed algorithm (Figure 3b) is what correctness verification entails. Conformance here means that, given the arguments x, y, and z on successive clock cycles (Figure 3c), then the (Val a) is identical to the value returned by reference x y z. Conformance is demonstrated in Coq using properties of stage, etc., proved using ReWire's mechanized semantics [7].

```
reference :: i → i → i → Storage s a
reference x y z = do c1 x
                     c2 y
                     a ← c3 z
                     return a
```

(a) *Imperative Reference Algorithm* performs c1, c2, and c3 in sequence on inputs x, y, and z producing answer a.

```
data Ans a = DC | Val a -- don't care/valid
staged :: i → Mealy i s (Ans o) i
staged x = do y ← stagei (c1 x)
              z ← stagei (c2 y)
              a ← stage  (c3 z)
              signal (Val a)
```

(b) *Staged Algorithm in ReWire* performs these same steps, but now accepts inputs interactively with answer a signaled on output port. Changes from reference algorithm are highlighted in red.



(c) Compiling staged with ReWire compiler creates Mealy machine fragment. Applying staging functions stage and stagei schedule c1, c2, and c3 on successive clock cycles.

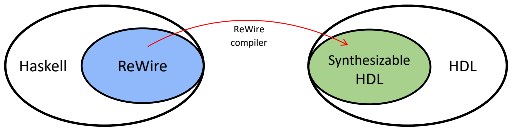
Fig. 3: Temporal Staging Transformations for Correct-by-Construction Cryptographic Hardware.

## II. Digital Design in ReWire

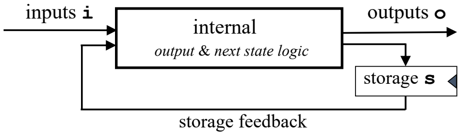
This section presents an overview of the ReWire high-level synthesis language. ReWire (Figure 4a) is a domain-specific language embedded within the Haskell functional language. Because every ReWire program is a Haskell program, ReWire programs may be evaluated just as any Haskell program using, for example, the GHCi interpreter. The ReWire compiler translates programs directly into synthesizable Verilog, and ReWire has a formal semantics mechanized in the Coq and Isabelle provers [7]. Of necessity, we assume basic familiarity with functional programming and Haskell.

The Mealy machine (Figure 4b) is a standard mental model of hardware for digital designers and it is also a motivating idea underlying the design of ReWire. Mealy machines have input, storage, and output lines i, s, and o, resp. At the beginning of any clock cycle, the combinational logic "internal" computes the next values on the s and o lines, which are latched at the end of the clock cycle. The ReWire programming model views these lines as individual types, i, s, and o. Type constructors representing Mealy machines in ReWire are (Storage s) and (Mealy i s o) (Figure 4c).

The *internal* combinational logic and *storage* s are encapsulated by (Storage s) and its associated operations, the type declarations of which are given in Figure 4c. In type (Storage s a), the parameter s signifies the type of *storage* and the a parameter is the *value type*. For example, a register file type with three eight bit words could be represented as a triple, (W 8,W 8,W 8), where W n is the built-in type of n-bit words (for any n), so that (Storage (W 8,W 8,W 8) a) represents a computation using those registers and returning a value of



(a) ReWire is a Functional High Level Synthesis (FHLS) language embedded in Haskell with a formalized semantics. ReWire's compiler translates programs into synthesizable HDL.



(b) The Mealy machine is a standard mental model of synchronous hardware in digital design [11], [12].

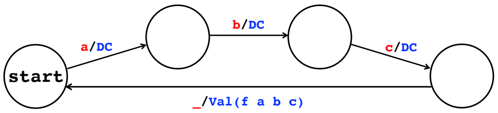
```
type Storage s
runST  :: Storage s a → s → (a , s)
return :: a → Storage s a
get    :: Storage s s
put    :: s → Storage s ()

type Mealy i s o
lift   :: Storage s a → Mealy i s o a
signal :: o → Mealy i s o i
```

(c) ReWire Types & Operators for Constructing Mealy Machines.

Fig. 4: ReWire in a Nutshell

```
purecsa :: W 8 → W 8 → W 8 → (W 8 , W 8)
purecsa a b c =
  let anb  = a .&. b in
  let anc  = a .&. c in
  let bnc  = b .&. c in
  let tmp1 = anb .|. anb .|. bnc <<. lit 1 in
  let tmp2 = (a ^ b) ^ c
  in
      (tmp1 , tmp2)
```

(a) Carry-Save Addition function



(b) Synchronous Hardware as a Mealy Machine

```
data Ans a = DC | Val a
csa :: Mealy (W 8) (W 8 , W 8 , W 8) (Ans (W 8, W 8)) ()
csa = do a ← signal DC
         b ← signal DC
         c ← signal DC
         _ ← signal (Val (purecsa a b c))
         csa
```

(c) Simple Carry-Save Adder.

Fig. 5: Carry-Save Addition in ReWire (explained in text)

type `a`. Given term `x` of type `(Storage s a)` and a store `st`, `runST x st` runs `x` with store `st`, returning a value and new store, `(a , st')`. For a value `a`, `return a` expresses a trivial computation that just returns `a` without updating the storage. The `get` operation reads the current storage and, hence, returns the current storage as its value without updating the storage; for the above example, `get` would return current values of the three registers. The `put` operation sets the storage to its argument; for the above register file type, `put (0x0,0x0,0x0)` zeroes out the storage.

Computations in `Storage` and `Mealy` are chained together using Haskell's `do` notation, examples of which are given below. Mealy machines are constructed by combining lifted `Storage` computations with input/output signals. `(signal o)` completes a clock cycle, placing `o` on the output port.

Figure 5 presents a ReWire specification of a simple carry-save adder, where the function `purecsa` (Figure 5a) is a ReWire translation of the definition from the carry-save adder Wikipedia entry. Function `purecsa` is instantiated for words of width 8 (i.e., built-in ReWire type `W 8`); "`lit`" is a convenience that constructs a bit-word from an integer. The symbols `.&.`, `.|.`, `^`, and `<<.` are ReWire built-in operations representing bitwise *and*, *or*, *xor*, and left shift, resp.

Figure 5b shows the input/output behavior as a Mealy machine and Figure 5c displays one possible ReWire implementation of that machine. The constructors for the `Ans` data type stand for "Don't Care" and "Valid". The `csa` device takes inputs `a`, `b`, and `c` as inputs over three successive clock cycles, each time putting `DC` on the output port to signify that no value has yet been computed. Given all the inputs, `(purecsa a b c)` is computed and placed on the output port while the input for this cycle is ignored (e.g., bound to the wildcard pattern "_").

*Technical Aside:* ReWire is a monadic language, meaning that ReWire programs are written in terms of a particular family of concurrency monads called *reactive resumption monads over state*. No previous knowledge of these monads nor of monadic semantics generally is required to understand the results presented in this paper, as we have taken pains to explain ReWire's constructs using intuitive examples. For readers interested in these technical underpinnings, please consult Harrison et al. [7] (and, in particular, its appendix) and the codebase [13]. Both `Storage` and `Mealy` are monads that would be defined in Haskell as follows using monad transformers `StateT` and `ReacT`:

```
type Storage s    = StateT s Identity
type Mealy i s o = ReacT i o (Storage s)
```

## III. Temporal Staging Transformations

This section defines the temporal staging functions in ReWire and outlines their properties. There are equational theorems that precisely characterize the behavior of the staging functions and one of them (Theorem 1 below) is discussed at the end of this section. The discussion is necessarily high-level and informal because these theorems are expressed in terms of ReWire's mechanized semantics [7].

Figure 6 presents the ReWire definitions of the staging functions. Each function can be viewed as a Mealy machine constructor, taking an action `x` of type `(Storage s a)` and creating a Mealy machine transition. `(stage x)` is a transition that computes a value `v`, signals `DC` on the output port, ignores the next input, and returns the value `v`. `(stagei x)` is a transition that performs `x` ignoring its computed value, signals

```
data Ans a = DC | Val a
stage :: Storage s a → Mealy i s (Ans o) a
stage x         = do
                   v ← lift x
                   signal DC
                   return v
stagei :: Storage s a → Mealy i s (Ans o) i
stagei x        = do
                   lift x
                   i ← signal DC
                   return i
stage_ :: Storage s a → Mealy i s (Ans o) ()
stage_ x        = do
                   lift x
                   signal DC
                   return ()
```

Fig. 6: ReWire Definitions of Staging Functions.

`DC` on the output port, and returns the next input `i`. `(stage_ x)` is a transition that performs `x` ignoring its computed value, signals `DC` on the output port ignoring the next input, and returns the nil value `()`.

Each application of a staging function creates a distinct Mealy machine transition. Given, for example, two computations `x` and `y` of type `(Storage s ())` for type `s`, define `one` and `two` as:

```
one = stage_ (do x        two = do (stage_ x)
                 y)                (stage_ y)
```

Although `one` and `two` are both of type `Mealy i s o ()` for some types `i` and `o`, they are distinct: `one` ≠ `two`. Intuitively, `one` is a single Mealy transition consuming a single clock cycle, while `two` are two transitions in sequence consuming two clock cycles. Operationally, this gives designers coarse-grained control over the "size" of transitions in the ReWire source and we used this to tune the BLAKE2b staged algorithm.

*Technical Aside:* Although stage functions have the same type as `lift`, they are not monad transformer liftings. The `lift` function in ReWire is an example of monad transformer lifting, meaning intuitively that it redefines a `(Storage s a)` computation as a `(Mealy i s o a)` computation, and also that it must obey the *lifting laws* [14]. Recall that do-notation is syntactic sugar for Haskell's bind operation: `do { v ← x ; f v }` $\triangleq$ `x >>= f`, then the lifting laws are:

$$\begin{aligned}\texttt{lift}\,(\texttt{return}_S\,\texttt{v}) &= \texttt{return}_M\,\texttt{v}\\ \texttt{lift}\,(\texttt{x} >>=_S \texttt{f}) &= (\texttt{lift}\,\texttt{x}) >>=_M (\texttt{lift} \circ \texttt{f})\end{aligned}$$

in which the monadic return and bind operations for `Storage` and `Mealy` are distinguished with subsecripts $S$ and $M$, resp. This means that lifting `x` followed by `f` must be identical to lifting `x` and following it by lifting `f`.

The ReWire formal semantics in Coq, $[\![-]\!]$, is motivated by the Mealy machine in Figure 4b in which, conceptually, the operation of a Mealy machine can be viewed as an infinite stream of "snapshots" of type `(i , s , o)`. Given a ReWire term `e` of type `Mealy i s o a`, an initial snapshot $\texttt{w}_0$, and an infinite stream of inputs `is :: Stream i`, then $([\![\,\texttt{e}\,]\!]\,\texttt{w}_0\ \texttt{is})$ is a sequence of snapshots defining the operation of `e`. (We use ◁ to denote the cons operation for streams and sequences.) The staging theorem characterizing `stage` is:

```
impcsa :: W 8 → W 8 → W 8 → Storage RegFile (W 8, W 8)
impcsa a b c = do
        setReg RA a
        setReg RB b
        setReg RC c
        do a ← readReg RA
           b ← readReg RB
           setReg A_and_B (a .&. b)
        do a ← readReg RA
           c ← readReg RC
           setReg A_and_C (a .&. c)
        do b ← readReg RB
           c ← readReg RC
           setReg B_and_C (b .&. c)
        tmp1 ← do anb ← readReg A_and_B
                  anc ← readReg A_and_C
                  bnc ← readReg B_and_C
                  return (anb .|. anb .|. bnc <<. lit 1)
        tmp2 ← do a ← readReg RA
                  b ← readReg RB
                  c ← readReg RC
                  return ((a ˆ b) ˆ c)
        return (tmp1 , tmp2)

data RegFile = RegFile { ra , rb , rc
                       , a_and_b , a_and_c , b_and_c :: W 8 }
data Reg     = RA | RB | RC | A_and_B | A_and_C | B_and_C

readReg :: Reg → Storage RegFile (W 8)
readReg RA      = do RegFile ra = w ← get
                     return w
        ...
readReg B_and_C = do RegFile b_and_c = w ← get
                     return w

setReg :: Reg → W 8 → Storage RegFile ()
setReg r w = do
    rf ← get
    case r of
         RA      → put (rf  ra = w )
            ...
         B_and_C → put (rf  b_and_c = w )
```

Fig. 7: Reference Algorithm: Carry-Save Adder. This reformulates `purecsa` from Figure 5a in an imperative style in which let bound variables are represented as registers. Some intermediate values are stored in these registers.

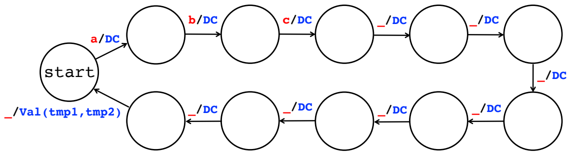
*Theorem 1 (Staging Theorem):* Assuming that `x` has type `Storage s a`, `f` has type `a` → `Mealy i s o b`, For all snapshots `(i , s , o)` and input streams $(\texttt{i}' \triangleleft \texttt{is})$,

$$\begin{aligned}&[\![\texttt{stage x} >>=_M \texttt{f}]\!]\ (\texttt{i},\texttt{s},\texttt{o})\ (\texttt{i}' \triangleleft \texttt{is})\\ &\qquad = (\texttt{i},\texttt{s},\texttt{o}) \triangleleft ([\![\,\texttt{f}\,]\!]\ \ \texttt{a}\ (\texttt{i}',\texttt{s}',\texttt{DC})\ \texttt{is})\\ &\quad \textbf{where}\\ &\qquad (\texttt{a}\,,\,\texttt{s}') = \texttt{runST}\,[\![\,\texttt{x}\,]\!]\,\texttt{s}\end{aligned}$$

Theorem 1 shows how to unwind `stage x` starting from initial snapshot $(\texttt{i},\texttt{s},\texttt{o})$ with input stream $\texttt{i}' \triangleleft \texttt{is}$. Computation `x` is run in initial store `s`, producing value `a` and updated store $\texttt{s}'$. Then, $[\![\,\texttt{f}\,]\!]\,\texttt{a}$ is unwound starting from the next snapshot, $(\texttt{i}',\texttt{s}',\texttt{DC})$, that reflects the next input $\texttt{i}'$, updated store $\texttt{s}'$, and the output signal `DC`. The staging theorems indicate how to unwind the staging functions and, importantly, how the temporal staging maintains the identical computed values.

## IV. First Case Study: Carry-Save Adder

This section describes the first case study in applying temporal staging to the construction of a carry-save adder implementation. The construction follows the steps outlined previously in Figure 3 of Section I. We first present the



(a) Mealy Machine for Pipelined Input CSA Device

```
pcsa :: Mealy (W 8) RegFile (Ans (W 8, W 8)) ()
pcsa = do
  a ← signal DC
  b ← stagei (setReg RA a)
  c ← stagei (setReg RB b)
  stage_ (setReg RC c)
  stage_ (do a ← readReg RA
             b ← readReg RB
             setReg A_and_B (a .&. b) )
  stage_ (do a ← readReg RA
             c ← readReg RC
             setReg A_and_C (a .&. c) )
  stage_ (do b ← readReg RB
             c ← readReg RC
             setReg B_and_C (b .&. c) )
  tmp1 ← stage (do anb ← readReg A_and_B
                   anc ← readReg A_and_C
                   bnc ← readReg B_and_C
                   return (anb .|. anb .|. bnc <<. lit 1) )
  tmp2 ← stage (do a ← readReg RA
                   b ← readReg RB
                   c ← readReg RC
                   return ((a ˆ b) ˆ c) )
  signal (Val (tmp1 , tmp2))
  pcsa

start :: Mealy (W 8) RegFile (Ans (W 8, W 8)) ()
start = pcsa
```

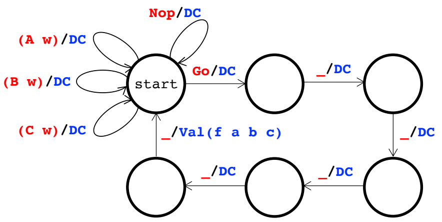
(b) ReWire for Pipelined Carry-Save Adder

Fig. 8: Carry-Save Adder with Pipelined Input

reference algorithm, which is based on `purecsa` (Figure 5a) in which its inputs and let-bound variables are treated as registers. We then construct two example adders using temporal staging, one with pipelined input and the other with asynchronous input. The full correctness verification of the second example can be found in the codebase [13].

*a) Reference Algorithm for Carry-Save Adder:* The imperative reference algorithm is presented in Figure 7. The function `impcsa` reflects the structure of the `purecsa` function (Figure 5a) although `impcsa` is typed in `Storage RegFile`. The inputs (`a`, `b`, and `c`) are stored in corresponding registers (resp., `RA`, `RB`, and `RC`) that are included in `RegFile` as are registers for some of the intermediate values let-bound in `purecsa`. The register file for both CSA designs is defined in Figure 7. There are six registers included in `RegFile` with corresponding names defined in data type `Reg`. There are operations for reading and writing individual registers—resp., `readReg` and `writeReg`. These operations are defined in terms of `Storage` primitives `put` and `get`.

*b) Carry-Save Adder with Pipelined Input:* The first carry-save adder is defined in Figure 8 with its control behavior expressed conventionally by the Mealy machine at the top of the figure. During the first three clock cycles, it accepts inputs `a`, `b`, and `c` and stores them in corresponding registers. For the next seven cycles, it computes and stores intermediate values while ignoring inputs. In the eleventh cycle, the computed carry-save addition of the inputs is signaled to the output port. N.b., `pcsa` performs the same computations as the reference algorithm `impcsa`, but the computation is time-sliced by the staging functions. The `pcsa` is a tail-recursive function, looping *ad infinitum* just as the Mealy machine in the top of the figure. ReWire designs must include a defined `start` to be compiled.



(a) Mealy Machine for Asynchronous Input CSA Device

```
data In w  = A w | B w | C w | Nop | Go
acsa :: In (W 8) →
          Mealy (In (W 8)) RegFile (Ans (W 8, W 8)) ()
acsa Nop   = do i ← signal DC
                acsa i
acsa (A a) = do i ← stagei (setReg RA a)
                acsa i
acsa (B b) = do i ← stagei (setReg RB b)
                acsa i
acsa (C c) = do i ← stagei (setReg RC c)
                acsa i
acsa Go    = do
    stagei (do a ← readReg RA
               b ← readReg RB
               setReg A_and_B (a .&. b) )
    stagei (do a ← readReg RA
               c ← readReg RC
               setReg A_and_C (a .&. c) )
    stagei (do b ← readReg RB
               c ← readReg RC
               setReg B_and_C (b .&. c) )
    tmp1 ← stage (do
              anb ← readReg A_and_B
              anc ← readReg A_and_C
              bnc ← readReg B_and_C
              return (anb .|. anb .|. bnc <<. lit 1) )
    tmp2 ← stage (do a ← readReg RA
                     b ← readReg RB
                     c ← readReg RC
                     return ((a ˆ b) ˆ c) )
    i ← signal (Val (tmp1 , tmp2))
    acsa i
start :: Mealy (In (W 8)) RegFile (Ans (W 8, W 8)) ()
start = acsa Nop
```

(b) ReWire for Asynchronous Input CSA Device

Fig. 9: Designs with Asynchronous Inputs.

*c) Carry-Save Adder with Asynchronous Input:* A second carry-save adder is presented in Figure 9. This design accepts inputs asynchronously (and potentially out of order), saving them to registers. The carry-save addition computation is then triggered by the `Go` input. The control-flow for this design is expressed by the Mealy machine in Figure 9a, and the full ReWire design is in Figure 9b.

This style of asynchronous input is common in digital design, and it is the input style used for the BLAKE2b design described in the next section. The correctness verification in Coq for this design can be found in the codebase [13].

```
data Reg  =
    V0 | V1 |  V2 | ··· | V15 -- working vectors v[0..15]
  | M0 | M1 |  M2 | ··· | M15 -- message buffer  m[0..15]
  | H0 | H1 |  H2 | ··· |  H7 -- hash state      h[0..7]

data RegFile = RegFile
       { v0 , v1 ,  v2 , ··· , v15
       , m0 , m1 ,  m2 , ··· , m15
       , h0 , h1 ,  h2 , ··· ,  h7  :: W 64 }
```

Fig. 10: `RegFile` for BLAKE2b Reference & Staged Specifications. The `readReg` and `writeReg` operations are defined analogously to Figure 7.

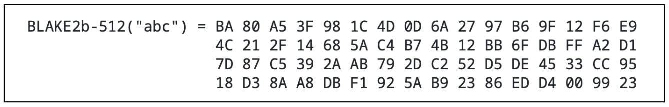
## V. Second Case Study: BLAKE2b

This section describes the second case study applying temporal staging to the construction of a BLAKE2b hardware implementation. As before, the construction followed the same steps outlined in Figure 3 of Section I. Because BLAKE2b is a much larger specification than the carry-save adder examples, however, we can only present an overview of the construction here. The full reference algorithm, staged algorithm, and correctness verification for BLAKE2b can be found in the codebase [13]. Importantly, while the BLAKE2b construction is larger than the previous carry-save adder examples, it does follow along precisely the same lines as before—i.e., temporal staging worked at this larger scale.

Figures 2 and 12a present examples of functions from which the BLAKE2b is composed. An executable reference algorithm for BLAKE2b is created by transliterating these pseudocodes and others into ReWire. Figures 12a and 12c provide a representative example of this transliteration.

Figure 10 presents the definition of the `RegFile` for the reference algorithm. BLAKE2b uses three vectors of 64-bit words. These are the 16 *working* and *message buffer* vectors, `v` and `m`, and the *hash state* vector containing eight words. These are represented by the `Reg` data type whose contents are represented by the `RegFile` data type. `(Storage RegFile)` maintains the current register file and has operations to read and write each register, `readReg` and `setReg`, that are defined analogously to the previous section. An assignment operation `<==` is defined that corresponds directly to `:=` in Figure 12a.

The BLAKE2b reference algorithm is also just a Haskell program, written in imperative style using `Storage`, and it is useful to take advantage of this fact to test the reference algorithm itself. Figure 11 presents an example of this conformance testing. The documentation for BLAKE2b [8] contains sample inputs with the correct outputs so that designers can perform "sanity checks" on their implementations. Figure 11a presents one of these examples taken directly from the BLAKE2b documentation. The file `Blake2b-reference.hs` from the figure includes the reference algorithm and it is loaded in the figure into GHCi, the Glasgow Haskell interpreter. There is a wrapper function `_BLAKE2b_512` that runs the reference on the appropriate inputs and formats the resulting hash value. The authors of the BLAKE2 documentation also provided a C program, and we ran extensive tests that conformed to outputs of this program.



```
BLAKE2b-512("abc") = BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
                     4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
                     7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
                     18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

(a) BLAKE2b document contains examples to check implementation conformance. Screenshot from RFC7693 [8], Appendix A.

```
$ ghci Blake2b-reference.hs
GHCi, version 9.2.5: https://www.haskell.org/ghc/
[1 of 1] Compiling (Blake2b-reference.hs, interpreted )
ghci> _BLAKE2b_512 "abc"

      BA 80 A5 3F 98 1C 4D 0D 6A 27 97 B6 9F 12 F6 E9
      4C 21 2F 14 68 5A C4 B7 4B 12 BB 6F DB FF A2 D1
      7D 87 C5 39 2A AB 79 2D C2 52 D5 DE 45 33 CC 95
      18 D3 8A A8 DB F1 92 5A B9 23 86 ED D4 00 99 23
```

(b) Reference conformance checked with `ghci` Haskell interpreter.

Fig. 11: Reference Algorithms & Conformance Checking.

## VI. Related Work

Cryptol [15] is a domain-specific language for specifying cryptographic algorithms. The purpose of Cryptol is to verify properties of what we have termed "reference" algorithms and the purpose of the present work (and previous work [7]) with ReWire is to produce high assurance hardware implementations of cryptographic algorithms. As such, ReWire and Cryptol are neatly orthogonal: one could easily imagine, for example, building a ReWire backend for Cryptol.

Earlier work by the second author [16] presented a case study of rapid hardware design of the SHA-256 cryptographic function. This work demonstrated how a reasonable design for SHA-256 can be formulated quickly in ReWire. At the time of that writing, the ReWire language and compiler were still at an early stage of development. Formal verification was not part of that effort.

Staging transformations for software languages have been explored for many years; a manifestly incomplete list of such work includes Scherlis [5] and Taha [17]. The idea behind staging is to use program transformations or annotations to distinguish the static and dynamic parts of a (software) program. Temporal staging obtained its name from this body of research, although it applies within a hardware context via the FHLS language ReWire. Temporal staging time-slices computations into discrete transitions rather than between traditional software notion of static and dynamic.

High-level synthesis languages have generally been conceived as a means of bringing traditional software virtues to hardware design—principally the desired software virtues where abstraction and modularity. FHLS can also act as a vector to transfer other ideas from functional programming and languages research into hardware design as well including software-like formal methods. ReWire—an FHLS language subset of Haskell—is a monadic language as well, and temporal staging really works along monadic lines. The staging functions defined in this article are similar (but importantly distinct as noted) to the lifting functions associated with monad transformers [14]. There are other FHLS languages [18]–[21]

```
FUNCTION F( h[0..7], m[0..15], t, f )
|  // Initialize local
|  //   work vector v[0..15]
|        ...
|  v[12] := v[12] ^ (t mod 2**w)
|  v[13] := v[13] ^ (t >> w)
|  IF f = TRUE THEN
|  |   v[14] := v[14] ^ 0xFF..FF
|  END IF.
|
|  // Cryptographic mixing
|        ...
|
|  FOR i = 0 TO 7 DO
|  |   h[i] := h[i] ^ v[i] ^ v[i + 8]
|  END FOR.
|
|  RETURN h[0..7]
|
END FUNCTION.
```

(a) BLAKE2b Compression Function

```
type M = Storage RegFile

_F :: W 128 → Bit → M ()
_F t f = do
     init_local_work_vector
     V12 <== V12 ^ lowword t
     V13 <== V13 ^ highword t
     if f then
         V14 <== V13 ^ 0xF...F
       else
         return ()
     cryptographic_mixing
     xor_two_halves

(<==)    :: Reg → M (W 64) → M ()
w <== e = do v ← e
             setReg w v
```

(b) BLAKE2b Reference Algorithm.

```
type Dev = Mealy W64x4 RegFile (Ans W64x8)

_F :: W 128 → Bit → Dev ()
_F t f = do
    stage (do init_local_work_vector
              V12 <== V12 ^ lowword t
              V13 <== V13 ^ highword t
              if f then
                  V14 <== V13 ^ 0xF...F
                else
                  return () )
    stage cryptographic_mixing
    stage xor_two_halves
```

(c) BLAKE2b Staged Algorithm.

Fig. 12: CbyC BLAKE2b Case Study. (a) BLAKE2 Compression Function F (page 8, [8]) abbreviated with ellipses. (b) Reference algorithm for F is composed from functions like these. (c) After temporal staging of the reference algorithm.

that may be able to incorporate temporal staging in some form.

## VII. Summary, Conclusions and Future Work

Temporal staging follows a traditional approach to CbyC software development [1], [3] in which a given reference algorithm is transformed into an implementation via "semantics-preserving" program transformations. Reframing these ideas within a hardware context is, to the authors' best knowledge, completely novel. Temporal staging transformations are simply ordinary function applications of the stage functions (Figure 6), and these stage functions are themselves just ReWire functions. The stage functions slice the reference algorithm temporally, partitioning the reference algorithm into single Mealy machine transitions that preserve the calculated values as indicated in Section III. Because the reference algorithm, the temporal staging transformations, and the resulting implementation are all expressed in ReWire, formal verification can proceed immediately via a previously published formalized ReWire semantics in Coq [7]—i.e., temporal staging can "shrink the conceptual divide" between reference algorithm and hardware implementation.

This article has focused on the temporal staging program transformation while providing some details about correctness verification. The BLAKE2b implementation we produced had sufficient performance to be included in an ASIC for FHE [6] that is, as of this writing, awaiting production. As of this writing, we are applying temporal staging to other cryptographic algorithms of equal or greater complexity to BLAKE2b.

## References

[1] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT*, vol. 8, no. 3, p. 174–186, sep 1968.

[2] N. Wirth, "Program development by stepwise refinement," *Commun. ACM*, vol. 14, no. 4, p. 221–227, apr 1971.

[3] R. M. Burstall and J. Darlington, "A transformation system for developing recursive programs," *J. ACM*, vol. 24, no. 1, p. 44–67, jan 1977.

[4] D. Gries, *The Science of Programming*, 1st ed. Berlin, Heidelberg: Springer-Verlag, 1981.

[5] U. Jørring and W. Scherlis, "Compilers and staging transformations," in *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, 1986, pp. 86–96.

[6] S. K. Moore, "Chips to compute with encrypted data are coming: Fully homomorphic encryption could make data unhackable," *IEEE Spectr.*, vol. 61, no. 01, p. 38–40, January 2024.

[7] W. L. Harrison, I. Blumenfeld, E. Bond, C. Hathhorn, P. Li, M. Torrence, and J. Ziegler, "Formalized high level synthesis with applications to cryptographic hardware," in *NASA Formal Methods Symposium (NFM23)*, 2023.

[8] M.-J. O. Saarinen and J.-P. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)," RFC 7693, Nov. 2015. [Online]. Available: https://www.rfc-editor.org/info/rfc7693

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

[10] T. Hansen and D. E. E. 3rd, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234, May 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6234

[11] G. H. Mealy, "A method for synthesizing sequential circuits," *The Bell System Technical Journal*, vol. 34, no. 5, pp. 1045–1079, Sep. 1955.

[12] R. Katz and G. Borriello, *Contemporary Logic Design*, 2005.

[13] Y. Forman and W. Harrison, "Correct-by-Construction BLAKE2 Codebase," Available from RSP24 Codebase, Jul. 2024.

[14] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995, p. 333–343.

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

[16] W. L. Harrison, A. Procter, and G. Allwein, "Model-driven design & synthesis of the sha-256 cryptographic hash function in rewire," in *Proceedings of the 27th International Symposium on Rapid System Prototyping (RSP)*, 2016, pp. 1–7.

[17] W. Taha, *A Gentle Introduction to Multi-stage Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 30–50.

[18] R. S. Nikhil and J. Clarke, "Designing hardware systems and accelerators with open-source bh (bluespec haskell)," https://github.com/rsnikhil/ICFP2020_Bluespec_Tutorial.

[19] C. Baaij and J. Kuper, "Using rewriting to synthesize functional languages to digital circuits," in *Trends in Fun. Prog.*, ser. LNCS, vol. 8322, 2014, pp. 17–33.

[20] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of bluespec: A core language for rule-based hardware design," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020, 2020, p. 243–257.

[21] R. Jordão, F. Bahrami, R. Chen, and I. Sander, "A multi-view and programming language agnostic framework for model-driven engineering," in *2022 Forum on Spec. & Design Languages (FDL)*, 2022, pp. 1–8.