Model-driven Design & Synthesis of the SHA-256 Cryptographic Hash Function in ReWire

William L. Harrison and Adam Procter
Department of Computer Science, University of Missouri

Gerard Allwein
US Naval Research Laboratory, Washington, DC

Abstract—**There are many algorithms whose implementations can benefit both from hardware acceleration and formal verification and we would like to develop high assurance implementations as rapidly as possible. Critical computing infrastructure like cryptographic algorithms are prime candidates both for such acceleration and for formal verification. We show how to derive a verifiable, hardware-accelerated implementation of the SHA-256 cryptographic hash in the ReWire functional hardware description language in which the partitioning of the implementation between hardware and software is reflected in the type system itself.**

I. Introduction

Cryptographic algorithms are good candidates for hardware implementation because they typically possess components whose performance benefits from hardware acceleration. But they are also prime candidates for formal verification because of their critical role in maintaining security and integrity in a wide variety of systems. These two concerns are somewhat at odds with one another: how does one relate a reference specification of a cryptographic algorithm—usually defined with mathematics and pseudocode—to an implementation which is, at least in part, written in a language without a formal semantics (e.g., VHDL)? If the implementation of a computation is to be split between hardware and software, how does one specify that partitioning? Furthermore, how does one formally verify the implementation once produced?

This paper explores the model-driven design and synthesis of a hardware accelerator for the SHA-256 cryptographic hash function. The approach is semantics-driven, starting from a reference semantics for SHA-256. The whole derivation takes place within the Haskell functional language [1] and so the hardware accelerator produced by the ReWire compiler is formally verifiable with respect to the reference semantics using equational reasoning in Haskell.

ReWire is a functional hardware description language which is a subset of the Haskell functional language—i.e., every ReWire program is a Haskell program, but not vice versa. Previous work has described the design and implementation of ReWire [2], its use as a target for embedded domain specific languages [3], its support for equational reasoning about reconfigurable hardware [4], and the expressiveness of its programming model [5]. This paper explores ReWire as a vehicle for model-driven design and synthesis of hardware devices.

At the core of the reference semantics is a function, `sha256`, that takes a sequence of 512-bit length blocks and computes their SHA-256 hash. Deriving a hardware accelerator is performed by factoring (or *partitioning* in the lingo of hardware-software codesign) into another function of the same type, `simdev devsha256 . format`. In Section III, this function's meaning is discussed in detail, but the key point is that `devsha256` is a ReWire device (i.e., a Haskell function that belongs to the ReWire subset) that may be compiled directly into a hardware accelerator for SHA-256 using the ReWire compiler. Formally specifying this factoring transformation is straightforward because both `sha256` and `devsha256` are both Haskell functions (in particular, `devsha256` is not in VHDL). Verifying the transformation is straightforward because `devsha256` reuses the code defining `sha256`. Finally, the hardware accelerators for SHA-256 produced in this work exhibit respectable performance with respect to published handcrafted VHDL implementations [6]–[12].

The remainder of this section reviews related work. Section II then presents a reference semantics in the Haskell functional language for SHA-256 based on its published design [13]. This reference semantics, being itself executable, can be tested against known implementations of SHA-256 to gain confidence in its correctness (see Fig. 1 described in Section II-B). Section III presents the first implementation of a hardware accelerator for SHA-256. Section III-D provides a formal specification of the first implementation in terms of the reference semantics. Section IV presents a second implementation of a hardware accelerator for SHA-256. This implementation marshals and unmarshals the 512 bit input and 256 bit output, resp., to accommodate FPGAs with limited I/O pins. Section V presents conclusions and future work.

Related Work

The SHA-2 (Secure Hash Algorithm) is a set of cryptographic hash functions designed by the National Security Agency and SHA-256 is one of these algorithms. These algorithms were defined semi-formally in documents produced by NIST [13].

There have been a host of efforts to use FPGAs to accelerate parts of SHA-256 [6]–[12]. These implement the main loop of SHA-256 in VHDL using various implementation strategies to achieve high performance (see, in particular, Chaves [6] for a comparison). The current approach differs from these in that it is model-driven. Here, hardware accelerators are derived from a Haskell reference semantics for SHA-256 rather than by encoding the algorithm in VHDL. The resulting accelerators, as is shown in Section IV, fall within the low end of the spectrum for performance among this previous work [6]–[12]. In contrast to these, however, the present model-driven approach was both rapid as well as susceptible to formal methods.

One approach to creating a verified cryptographic implementation is to use a proprietary toolchain like Cryptol [14]. Cryptol is a DSL embedded in Haskell designed specifically for cryptographic algorithms. There is, apparently, a Cryptol implementation for a SHA-256 hardware accelerator, but it is not freely available and so there is no basis for comparison with the current work. ReWire is a general-purpose hardware description language not dedicated to any specific class of algorithms or systems.

Appel [15] uses the formally verified CompCert C compiler and related tools [16] to verify a C implementation of the SHA-256 algorithm. Appel verifies the entire SHA-256 implementation, including the pre-processing formatting phase that is left out of the hardware accelerators in Sections III and IV and the previous work [6]–[12]. The pre-processing phase of SHA-256 involves potentially unbounded data—i.e., the input stream—that cannot readily be accommodated in hardware.

The approach we take to model-driven design is reminiscent of what is known as "Bird-Wadler" style program derivation in the functional programming community (so-named after an influential textbook [17]). A Bird-Wadler derivation starts from a reference specification for an algorithm in a functional language and, through a series of semantics-preserving program transformations, produces a more efficient implementation. The correctness of the implementation can then specified equationally and verified in terms of the reference semantics. The essence of the approach here is to identify the portion of the reference specification (in this case of SHA-256) that is amenable to hardware acceleration and to reformulate that portion in terms of the built-in monadic constructions of the ReWire language.

Hardware-software codesign [18] develops the hardware and software components for systems in tandem. The current approach derives a hardware component from a reference semantics describing an algorithm as a whole, and, therefore, the approach is a partial implementation technique because the software component is not derived. Therefore, the current approach cannot be considered hardware-software codesign as it stands. The hardware-software partitioning of the reference semantics is, however, reflected in type structure of the SHA-256 implementations produced, and so it is an intriguing open question whether the current approach could be extended to support codesign.

II. A Reference Semantics for the SHA-256 Cryptographic Hash Function

This section presents a high-level overview of the SHA-256 algorithm as it is rendered in the Haskell functional programming language. The reference semantics is little more than a straightforward transliteration of the pseudocode from [13] into Haskell. Haskell notation is explained as it is used throughout this paper, although some previous experience with functional programming would be helpful.

```
ghci> refsha256 msg1
  Oct 3128432319 2399260650 1094795486 1571693091
      2953011619 2518121116 3021012833 4060091821
ghci> hashed1
  Oct 3128432319 2399260650 1094795486 1571693091
      2953011619 2518121116 3021012833 4060091821
ghci> refsha256 msg2
  Oct 613247585 3523623096 3854575251 205414457
      2738676825 1694441831 4142722516 433784513
ghci> hashed2
  Oct 613247585 3523623096 3854575251 205414457
      2738676825 1694441831 4142722516 433784513
ghci> refsha256 msg3
  Oct 3452399196 2568289170 2174863330 2228698727
      4051737160 2761367566 74267084 3339791568
ghci> hashed3
  Oct 3452399196 2568289170 2174863330 2228698727
      4051737160 2761367566 74267084 3339791568
```

Fig. 1: Transcript showing test of reference semantics against values reported in [13].

The cryptographic hash algorithm SHA-256 is defined semi-formally using pseudocode in an online document from NIST [13]. The NIST document [13] has since been superseded by newer editions of the document. We refer to this edition because it contains example hash computations against which we can check our reference semantics.

A. Top Level

The input to the SHA-256 algorithm is assumed to be a sequence of bytes and its output is a *digest* of eight 32-bit words. Respectively, these are represented in Haskell as the types `[Char]` (i.e., a list of ASCII characters) and `Oct Word32`. A Haskell function implementing this algorithm, `refsha256`, is presented in Listing 1; its type is `[Char] -> Oct Word32`. The function `refsha256` is the composition of functions `pad`, `sha256`, and `runM` which are discussed in more detail below.

Listing 1 SHA-256 in Haskell

```
refsha256 :: [Char] -> Oct Word32
refsha256 = runM . sha256 . pad

pad    :: [Char] -> [Hex Word32]
sha256 :: [Hex Word32] -> M (Oct Word32)
runM   :: M a -> a

data Oct a = Oct a a a a a a a a
data Hex a = Hex a a a a a a a a a a a a a a a a
```

The SHA-256 algorithm first pads the input sequence so that the result has length as a multiple of 512 bits. The padded input is then parsed into 512 bit blocks, $M^{(1)}, \dots, M^{(N)}$. Each of these blocks, $M^{(i)}$, is further parsed into sixteen 32-bit words, $M_0^{(i)}, \dots, M_{15}^{(i)}$ and represented in Haskell with the type `Hex Word32`—thus, the type of the `pad` function. The code for `pad` (as well as for `runM`) is found in the code repository [19]; their definitions would not be illuminating here.

The function `sha256` implements the main loop of the SHA-256 algorithm. It takes the sequence of blocks formatted by `pad` (i.e., a list of type `Hex Word32`) as input and produces a computation of a digest (i.e., a computation of type `M (Oct Word32)`) as output. Here, `M` is a *monad*, which is a construction used for representing side-effects in the pure functional language Haskell. We define `M` and its use below in Section II-E.

B. Testing Against the NIST Standard

The NIST document defining SHA-256 [13] provides three example hash computations against which one may test implementations of the algorithm. Figure 1 summarizes the tests applied to `refsha256`. The tests apply `refsha256` to messages `msg1`, `msg2`, and `msg3`; `msg1` is just `"abc"` while the other two are too long to print easily here.

In Figure 1, a test harness is loaded into the GHC (Glasgow Haskell Compiler) interpreter. First the `refsha256` is applied, producing the hash digest calculated by the reference semantics. Then, the published hash digests from [13], Appendix B, are displayed—these are `hashed1`, `hashed2`, and `hashed3`, respectively—and note that they are identical to the corresponding output of `refsha256`. By default, these digests are displayed in decimal format rather than the hexadecimal format in which they appear in [13].

C. Logical Operators

SHA-256 defines a number of operations on 32-bit words called "logical operators" in Sections 3.2 and 4.1.2 of [13]. Each of these functions transliterates simply into Haskell using built-in functions and data from the Haskell libraries, `Data.Bit` and `Data.Word`. Within Listing 2, `.&.`, `complement`, and `xor` are, resp., the bitwise and, complement and exclusive or operations. The `rotateR` and `shiftR` functions are the rotate right and shift right operations.

Listing 2 Logical Operators

```
ch :: Word32 -> Word32 -> Word32 -> Word32
ch x y z = (x .&. y) `xor` (complement x .&. z)

maj :: Word32 -> Word32 -> Word32 -> Word32
maj x y z = (x .&. y) `xor` (x .&. z) `xor` (y .&. z)

bigsigma0 :: Word32 -> Word32
bigsigma0 x = (rotateR x 2) `xor` (rotateR x 13)
                            `xor` (rotateR x 22)

bigsigma1 :: Word32 -> Word32
bigsigma1 x = (rotateR x 6) `xor` (rotateR x 11)
                            `xor` (rotateR x 25)

sigma0 :: Word32 -> Word32
sigma0 x = (rotateR x 7) `xor` (rotateR x 18)
                         `xor` (shiftR x 3)

sigma1 :: Word32 -> Word32
sigma1 x = (rotateR x 17) `xor` (rotateR x 19)
                          `xor` (shiftR x 10)
```

D. Constants and Counters

The inner loop of the SHA-256 algorithm is, in effect, a for-loop counting from 0 to 63. To allow expression of this in Haskell, we define a counter type `Ctr` in Listing 3. It would be possible in the reference semantics to use the built-in `Int` type as a loop variable, but, anticipating the shift to hardware implementation, a discrete, finite type like `Ctr` makes more sense. The SHA-256 algorithm also makes use of 64 constants (these are defined in Section 4.2.2 of [13]). We index these constants via the `seed` function defined in Listing 3.

Listing 3 Counter types and Constants

```
-- Counter type
data Ctr = C0  | C1  | C2  | C3  | C4  | C5  | C6  | C7  |
           C8  | C9  | C10 | C11 | C12 | C13 | C14 | C15 |
           C16 | C17 | C18 | C19 | C20 | C21 | C22 | C23 |
           C24 | C25 | C26 | C27 | C28 | C29 | C30 | C31 |
           C32 | C33 | C34 | C35 | C36 | C37 | C38 | C39 |
           C40 | C41 | C42 | C43 | C44 | C45 | C46 | C47 |
           C48 | C49 | C50 | C51 | C52 | C53 | C54 | C55 |
           C56 | C57 | C58 | C59 | C60 | C61 | C62 | C63

-- Increment counter          -- Constants
incCtr :: Ctr -> Ctr          seed :: Ctr -> Word32
incCtr C0  = C1               seed C0  = 0x428a2f98
           ⋮                            ⋮
incCtr C62 = C63              seed C62 = 0xbef9a3f7
incCtr C63 = C0               seed C63 = 0xc67178f2
```

E. Memory Layout of SHA-256

SHA-256 is an imperative algorithm, meaning that it involves assignments to registers, loops, etc. To program in imperative style in Haskell, one normally uses state monads. This section will provide a quick overview of the fundamentals of monadic programming in Haskell for the sake of being as self-contained as possible. For readers requiring more information, please consult the references [20].

There are four types of storage in the algorithm:

1) Intermediate Digest: temporary storage for a digest value; of type `Oct Word32`.
2) Current Block: the current, formatted block; of type `Hex Word32`.
3) Digest: the value of the digest; of type `Oct Word32`.
4) Loop Counter: the current value of the loop counter register of type `Ctr`.

To create storable registers for each of these types, we create a monad `M` using the state monad transformer `StateT` in Listing 4. Notice that, for each type of register above, there is a corresponding application of the state monad transformer. This stack of `StateT` applications applies to the `Identity` monad, which has no registers at all.

Listing 4 Defining the Monad `M`

```
type M =
  StateT (Oct Word32)                    -- Intermediate Digest
    (StateT (Hex Word32)                 -- Current Block
       (StateT (Oct Word32)              -- Digest
          (StateT Ctr Identity)))        -- Loop Counter
```

For each register type, there are corresponding read and write operations in `M` for manipulating the register. These operations are prepended with "`get`" and "`put`", respectively, and are defined in Listing 5. Paying attention to the types of the operation is the best way to understand them. For example, to read the current digest value, one uses `getDigest :: M (Oct Word32)`. This type signifies that `getDigest` is an `M` computation that returns an `Oct Word32` value. To update the current value of the digest register, one uses `putDigest :: Oct Word32 -> M()`. For a given `d :: Oct Word32`, `(putDigest d) :: M()` is a computation that sets the current value of the digest register to `d`.

Listing 5 Register Operations in `M`

```
getIntDig :: M (Oct Word32)
getIntDig = get
putIntDig :: Oct Word32 -> M ()
putIntDig = put

getBlock :: M (Hex Word32)
getBlock = lift get
putBlock :: Hex Word32 -> M ()
putBlock = lift . put

getDigest :: M (Oct Word32)
getDigest = lift (lift get)
putDigest :: Oct Word32 -> M ()
putDigest = lift . lift . put

getCtr :: M Ctr
getCtr = lift (lift (lift get))
putCtr :: Ctr -> M ()
putCtr = lift . lift . lift . put
```

To chain together the operations from Listing 5, we use Haskell's `do` notation. In Listing 6, first the current digest is read with `getDigest`, then the current intermediate digest is read with `getIntDig`, and finally they are added together component-wise and stored back in the digest register using `putDigest`. This `intermediate` operation is part of the Haskell reference semantics for SHA-256.

Listing 6 The `intermediate` function

```
intermediate :: M ()
intermediate =
   do
      Oct h1 h2 h3 h4 h5 h6 h7 h8 <- getDigest
      Oct a b c d e f g h         <- getIntDig
      putDigest (Oct (a+h1) (b+h2) (c+h3) (d+h4)
                     (e+h5) (f+h6) (g+h7) (h+h8))
```

There is one more `M` operation to describe: `return :: a -> M a`. This polymorphic operation takes a value `v` of some type `a` and produces an `M` computation that simply returns `v`. The `return` operation produces no side effects and is, in some sense, a "do nothing" operation. One can reason about Haskell programs equationally and, for example, the following equation holds:

```
getDigest = do d <- getDigest
               return d
```

F. The Main Function

Listing 7 displays the Haskell code for the main loop of the SHA-256 algorithm, which is represented by the recursive list function `sha256`. The `sha256` function first initializes the digest and then passes the parsed input on to the `mainloop` function. The `mainloop` function is a straightforward rendering of the pseudo-code defining the main loop of SHA-256 (see Section 6.2.2 of [13]). The code for the message scheduling and compression routines can be found in the code repository [19].

Listing 7 The Main Function

```
sha256 :: [Hex Word32] -> M (Oct Word32)
sha256 hws = do
                putDigest initialSHA256State
                mainloop hws
                getDigest

mainloop :: [Hex Word32] -> M ()
mainloop []            = return ()
mainloop (hw32 : hw32s) = do
                             hi_1 <- getDigest
                             putIntDig hi_1
                             putBlock hw32
                             putCtr C0
                             innerloop
                             mainloop hw32s

innerloop :: M ()
innerloop = do
               ctr <- getCtr
               s <- sched
               compress (seed ctr) s
               putCtr (incCtr ctr)
               case ctr of
                    C63 -> intermediate
                    _   -> innerloop

-- SHA-256 message scheduling and compression routines
sched    :: M Word32
compress :: Word32 -> Word32 -> M ()
```

III. Partitioning the Reference Semantics into Hardware and Software

The reference semantics as a whole will not translate directly to hardware: it makes extensive use of list types in Haskell (i.e., list types) that are of potentially unbounded size. Hardware circuits, which require a finite storage footprint, cannot accommodate such types generally. However, large parts of the reference semantics may be re-used in a ReWire specification of a circuit that computes the `sha256` function from Listing 7, albeit with a different calling convention to eliminate lists.

A. Background: ReWire

This section provides a brief introduction to ReWire and its use. For a more complete introduction to ReWire and an explanation of its design and semantics, please refer to the references [2].

ReWire is a subset of the Haskell functional programming language [1]—i.e., ReWire programs are Haskell programs, but not necessarily *vice versa*. All ReWire programs can be compiled to synthesizable VHDL with the ReWire compiler. The principal difference between Haskell and ReWire is that recursion in ReWire is restricted to tail recursion so that every ReWire program requires only a finite, bounded memory footprint. Unbounded recursion requires an unbounded stack or heap for compilation and such dynamic control structures are anathema to hardware's fixed storage. One may view ReWire as a language for creating clocked, synchronous devices.

Generally speaking, at each clock "tick", a synchronous device consumes an input signal and produces an output signal. Let us call the types of these signals, `Inp` and `Out`, resp., and we will give particular definitions of these types below. There may also be, during each clock cycle, changes to the device's internal storage. In ReWire, the internal storage of a device is encapsulated by a state monad (e.g., `M`). The type for synchronous devices with input/output signals, `Inp` and `Out`, with internal storage encapsulated by `M` is written:

```
type Device = ReacT Inp Out M ()
```

We eschew the formal definition of the type constructor `ReacT` as it is unnecessary to understanding ReWire and its uses (see [2] for a full explanation). Like `M`, `ReacT Inp Out M` is also a monad, and, consequently, ReWire devices may be specified using `do`-notation. Operations on `M` must be "lifted" to be used within a `Device`. For instance, to use `getDigest` within a `Device`, one must apply the `lift` to it first (see Listing 8 for examples), where the polymorphic function, `lift :: M a -> ReacT Inp Out M a`, converts `M` operations into `ReacT Inp Out M` operations.

A `Device` specification may also use the following operation, `signal`, which encapsulates an interactive style of I/O:

```
signal :: Out -> ReacT Inp Out M Inp
```

It is best to explain `signal` with the following example. Given an output `o :: Out` and a function, `f :: Inp -> Device`, the following code firsts outputs `o`, then waits to receive input `i`, and finally passes `i` onto `f`:

```
do i <- signal o
   f i
```

B. Hardware Accelerator for SHA-256

Listing 8 presents the input and output signals for the hardware accelerator device; these are the `Inp` and `Out` types, resp. The `Init` input signal tells the accelerator to initialize the current block and begin a new hash computation. The `Load` signal instructs the accelerator to load new input into the current block register. The `DigestQ` signal is used to return the computed hash digest. The `DigestR` output signal returns the computed hash digest and the `Nix` output signal is the default output of the accelerator.

Listing 8 presents the definition of the SHA-256 accelerator. The most important thing to note is that `devsha256` reuses most of the code directly from the reference semantics `sha256`. First, `devsha256` outputs the `Nix` signal, receives an input `i` and passes it to `dev`. The function `dev` handles each `Inp` signal as described in the previous paragraph.

C. Top Level for `devsha256`

Listing 9 presents the top-level functions for testing the `devsha256` accelerator in Haskell. Below, we discuss the calling convention for `devsha256`, which is encapsulated as the `format` function. The `simdev` function takes a `Device`

Listing 8 ReWire Code for a SHA-256 Hardware Accelerator

```
data Inp = Init (Hex Word32) | Load (Hex Word32) | DigestQ
data Out = DigestR (Oct Word32) | Nix

devsha256 :: Device
devsha256 = do
               i <- signal Nix
               dev i

dev :: Inp -> Device
dev (Init hw32) = do
                     lift (do
                              putDigest initialSHA256State
                              hi_1 <- getDigest
                              putIntDig hi_1
                              putBlock hw32
                              putCtr C0)
                     signal Nix
                     innerloop
dev (Load hw32) = do
                     lift (do
                              hi_1 <- getDigest
                              putIntDig hi_1
                              putBlock hw32
                              putCtr C0)
                     signal Nix
                     innerloop
dev DigestQ     = do
                     h_n <- lift getDigest
                     i <- signal (DigestR h_n)
                     dev i

innerloop :: Device
innerloop   = do
                ctr <- lift (do
                                c <- getCtr
                                s <- sched
                                compress (seed c) s
                                putCtr (incCtr c)
                                return c)
                i <- signal Nix
                case ctr of
                     C63 -> do lift intermediate
                               dev i
                     _   -> innerloop
```

and a list of `Inp`s and computes the final `Out` signal produced by the device on those inputs. The code for both of the functions is found in the repository [19].

Importantly, the hardware accelerator may be tested in Haskell: using `godev256` produces the same results as the `refsha256` did in Fig. 1.

Listing 9 Partitioned SHA-256.

```
godev256 :: [Char] -> Out
godev256 = runM . simdev devsha256 . format . pad
format    :: [Hex Word32] -> [Inp]
simdev    :: Device -> [Inp] -> M Out
```

To call the main function from the reference semantics, one simply applies the function `sha256` to the list of input blocks. That is, **sha256** $[M^{(1)}, \ldots, M^{(N)}]$ computes a digest. The device version, `devsha256`, requires that this input be formatted differently, and this is accomplished with the function, `format :: [Hex Word32] -> [Inp]`. The action of the `format` function is described below:

$$\begin{aligned}\texttt{format}[M^{(1)},\ldots,M^{(N)}] = \\ [\texttt{Init}\ M^{(1)}, \underbrace{\texttt{DigestQ},\ldots,\texttt{DigestQ}}_{\times 64}, \\ \texttt{Load}\ M^{(2)}, \underbrace{\texttt{DigestQ},\ldots,\texttt{DigestQ}}_{\times 64}, \\ \vdots \\ \texttt{Load}\ M^{(N)}, \underbrace{\texttt{DigestQ},\ldots,\texttt{DigestQ}}_{\times 64}, \\ \texttt{DigestQ}]\end{aligned}$$

This calling convention is analogous to those found in the references [6]–[12]. Note that the sixty four `DigestQ` signals are used only as padding to wait for the inner loop within the device to finish. The final `DigestQ` will result in the completed hash value being returned.

D. Formal Specification

The formal specification of the hardware accelerator defined in Listing 8 is formulated as the equation:

For all finite `str :: [Char],`
`DigestR (refsha256 str) = godev256 str`

This equation requires that the hash digest returned by the reference semantics (i.e., `refsha256 str`), when wrapped by a `DigestR` constructor, is precisely the same thing returned by the `devsha256` accelerator.

Note that the above equation may be demonstrated by ordinary equational reasoning in Haskell (e.g., induction on lists, etc.) of the type made popular by Bird and Wadler [17], although we leave it for future work to do so. What makes this simple form of formal specification possible is that both the reference semantics and the hardware accelerator given above are defined in Haskell. Traditional hardware verification [21] would have to encode the semantics and implementation in the logic of a theorem prover with the latter encoding being particularly challenging because the implementation would be in a hardware description language without a formal semantics (e.g., VHDL or Verilog).

IV. Refactoring for Implementation

Recall that the inputs to the `devsha256` are 512 bits wide—i.e., a `Hex Word32` value requires 512 bits ($= 16 \times 32$) to represent. Similarly, the `Oct Word32` outputs of `devsha256` require 256 ($= 8 \times 32$) bits to represent. Consequently, we would need an FPGA with at least 768 (= 512+256) I/O pins in order to synthesize devsha256 directly; this is an unrealistically large number. To bring the pin count down, we can refactor the devsha256 specification to marshal and unmarshal the input and output 64 bits at a time.

Listing 10 displays the new definitions of the `Inp` and `Out` data types to accomplish the marshaling and unmarshaling of data on and off the targeted board. Each load signal (i.e., either `Init` or `Load0` through `Load7`) contains two 32 bit words, so that words are loaded two at a time onto the targeted board. Each output digest signal (i.e., `DigestR`) off loads two 32 bit

Listing 10 I/O Signals for Marshaled SHA-256 Device

```
data Inp = Init Word32 Word32  | Load0 Word32 Word32 |
           Load1 Word32 Word32 | Load2 Word32 Word32 |
           Load3 Word32 Word32 | Load4 Word32 Word32 |
           Load5 Word32 Word32 | Load6 Word32 Word32 |
           Load7 Word32 Word32 |
           DigestQ0 | DigestQ1 | DigestQ2 | DigestQ3
data Out = DigestR Word32 Word32 | Nix
```

words per cycle, so that it takes four digest request signals (i.e., `DigestQ0` through `DigestQ3`) to output one digest. The calling convention for the new marshaling accelerator device changes as well, and it is defined similarly to the `format` function from the previous section (and can be found in the code repository [19]). The code for the refactored device is found in Listing 11.

Listing 11 Marshaling Device

```
devsha256' :: Device
devsha256' = do i <- signal Nix
                dev i

dev :: Inp -> Device
dev (Init w1 w2) = do
                       lift (do
                               putDigest initialSHA256State
                               hw <- getBlock
                               putBlock (load0 w1 w2 hw))
                       i <- signal Nix
                       dev i
dev (Load0 w1 w2) = do
                        lift (do
                                hw <- getBlock
                                putBlock (load0 w1 w2 hw))
                        i <- signal Nix
                        dev i
                    ⋮
dev (Load7 w1 w2) = do
                        lift (do
                                putCtr C0
                                hi_1 <- getDigest
                                putIntDig hi_1
                                hw <- getBlock
                                putBlock (load7 w1 w2 hw))
                        signal Nix
                        innerloop
dev DigestQ0      = do
                        h_n <- lift getDigest
                        i <- signal (digest0 h_n)
                        dev i
                    ⋮
dev DigestQ3      = do
                        h_n <- lift getDigest
                        i <- signal (digest3 h_n)
                        dev i
```

The `devsha256'` accelerator has been tested in Haskell to show that it computes the same hashes as the reference semantics. Furthermore, it could be formally specified as well along the same lines as in Section III-D, although we leave this for future work. Using the ReWire compiler, we can produce synthesizable VHDL and synthesize to a FPGA target—here we target the Xilinx Spartan-3E XC3S500E-4FG320 featured on the Xilinx Spartan-3E Starter Kit.

The table below presents the synthesis numbers for `devsha256'` for the Xilinx ISE toolset targeting the Spartan-

3E. The maximum clock rate was 60 MHz.

Slices	Flip-Flops	LUTs	IOBs
1424 (30%)	1106 (11%)	2716 (29%)	134 (57%)

There are 76 cycles per output, and so the total throughput for `devsha256'` is calculated as:

$$(60MHz \times 512bits)/76cycles = 404Mbps$$

This throughput for the `devsha256'` accelerator is on the low end of the performance spectrum of the hand-crafted VHDL implementations reported in Chaves et al. [6]—higher than Sklavos and Koufopavlou [11] (376 *Mbps*) and lower than Chaves et al. [6] (1370 *Mbps*). The aforementioned prior works are not targeting the same FPGA, and so the above is something of a sort of "apples-to-oranges" comparison. The performance numbers should be taken merely as indicative that performance is reasonable for a rapidly developed prototype.

V. Conclusions & Future Work

This paper explores the application of ideas and techniques from functional languages to the model-driven design and synthesis of hardware artifacts and demonstrates the methodology with a significant case study from cryptographic algorithms. Starting from a reference semantics for SHA-256, several formally verifiable implementations with acceptable performance are derived. Because this model-driven design takes place in the Haskell language, formal specification is straightforward.

One aspect of this derivation is its speed—most of the time in this experiment was spent in comprehending the semi-formal specification of SHA-256 [13]. Given the reference semantics, deriving both accelerator implementations took on the order of one hour's time. Admittedly, this evidence is anecdotal and, furthermore, the derivation was performed by the authors who have considerable experience with Haskell and ReWire. Still, functional programmers will recognize the basic process illustrated in Sections III and IV as mainly (in the jargon of monadic programming [20]) lifting a function through a monad transformer. Most functional programmers could fairly quickly understand and apply this methodology.

The methodology has things in common with hardware-software codesign, in particular that a given application is partitioned into separate hardware and software components. This partitioning is reflected in the types in that the hardware component must have, what we called in Section III, the type `Device`. As it stands, the hardware and software components are completely disconnected, therefore, to use the hardware accelerators, one would have to write a new software wrapper to call the synthesized accelerator. An intriguing next step for this research explores the automatic generation of integration code to connect the Haskell "software partition" to the device generated from the "hardware partition." This would provide a uniform, functional language for seamlessly constructing heterogeneous applications.

The authors are currently formalizing ReWire with the Coq proof assistant [22] as both a means of automating formal verification of ReWire security and correctness specifications and of verifying the ReWire compiler itself. In conjunction with the aforementioned functional language for heterogeneous applications, formalized ReWire would provide a basis for formally verifying heterogeneous applications.

References

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

[2] 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.

[3] I. Graves, A. Procter, W. L. Harrison, M. Becchi, and G. Allwein, "Hardware synthesis from functional embedded domain-specific languages: A case study in regular expression compilation," in *Applied Reconfigurable Computing*, ser. LNCS, 2015, vol. 9040, pp. 41–52.

[4] I. Graves, A. M. Procter, W. L. Harrison, and G. Allwein, "Provably correct development of reconfigurable hardware designs via equational reasoning," in *International Conference on Field-Programmable Technology (FPT15).* IEEE, 2015, pp. 160–171.

[5] W. L. Harrison, I. Graves, A. M. Procter, M. Becchi, and G. Allwein, "A programming model for reconfigurable computing based in functional concurrency (to appear)," in *11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC16)*, 2016.

[6] R. Chaves, G. Kuzmanov, L. Sousa, and S. Vassiliadis, "Improving SHA-2 hardware implementations," in *8th International Workshop on Cryptographic Hardware and Embedded Systems*, 2006, pp. 298–310.

[7] I. Algredo-Badillo, C. Feregrino-Uribe, R. Cumplido, and M. Morales-Sandoval, "FPGA-based implementation alternatives for the inner loop of the secure hash algorithm SHA-256," *Microprocessors and Microsystems*, vol. 37, no. 6-7, pp. 750–757, 2013.

[8] K. K. Ting, S. C. L. Yuen, K. H. Lee, and P. H. W. Leong, "An FPGA based SHA-256 processor," in *12th International Conference on Field-Programmable Logic and Applications (FPL)*, 2002, pp. 577–585.

[9] R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane, "Optimisation of the SHA-2 family of hash functions on FPGAs," in *IEEE Symposium on Emerging VLSI Technologies and Architectures (ISVLSI)*, March 2006, pp. 317–322.

[10] R. Garcia, I. Algredo-Badillo, M. Morales-Sandoval, C. Feregrino-Uribe, and R. Cumplido, "A compact fpga-based processor for the secure hash algorithm sha-256," *Computers and Electrical Engineering*, vol. 40, no. 1, pp. 194–202, 2014.

[11] N. Sklavos and O. Koufopavlou, "Implementation of the sha-2 hash family standard using fpgas," *J. Supercomput.*, vol. 31, no. 3, pp. 227–248, Mar. 2005.

[12] F. Kahri, H. Mestiri, B. Bouallegue, and M. Machhout, "Efficient FPGA hardware implementation of secure hash function SHA-256/Blake-256," in *12th International Multi-Conference on Systems, Signals Devices (SSD)*, March 2015, pp. 1–5.

[13] "Secure Hash Standard (SHS)," Federal Information Processing Standards Publication FIPS Pub 180-2, Aug. 2002. [Online]. Available: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[14] L. Erkök, M. Carlsson, and A. Wick, "Hardware/software co-verification of cryptographic algorithms using cryptol," in *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, Nov 2009, pp. 188–191.

[15] A. W. Appel, "Verification of a cryptographic primitive: Sha-256," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 2, pp. 7:1–7:31, Apr. 2015.

[16] "The compcert project." [Online]. Available: http://compcert.inria.fr

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

[18] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, pp. 1411–1430, May 2012.

[19] "Code repository for rps16." [Online]. Available: https://goo.gl/xuGkja

[20] S. Liang, P. Hudak, and M. Jones, "Monad transformers and modular interpreters," in *22nd ACM Symposium on Principles of programming Languages*, 1995, pp. 333–343.

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

[22] "The Coq Proof Assistant," https://coq.inria.fr.