A Programming Model for Reconfigurable Computing Based in Functional Concurrency

William L. Harrison*, Ian Graves*, Adam Procter*, Michela Becchi†, Gerard Allwein‡
*Department of Computer Science, University of Missouri
†Department of Electrical & Computer Engineering, University of Missouri
‡US Naval Research Laboratory, Washington, DC

***Abstract*—FPGA programmability remains a concern with respect to the broad adoption of the technology. One reason for this is simple: FPGA applications are frequently implementations of concurrent algorithms that could be most directly rendered in concurrent languages, but there is little or no first-class support for concurrent applications in conventional hardware description languages. It stands to reason that FPGA programmability would be enhanced in a hardware description language with first-class concurrency. The starting point for this paper is a functional hardware description language with built-in support for concurrency called ReWire. Because it is a concurrent functional language, ReWire supports the elegant expression of common concurrency paradigms; we illustrate this with several case studies.**

I. Introduction

FPGA programmability must improve if they are to gain wider acceptance within computing generally [1]. Andrews [2] argues that a paradigm shift for reconfigurable computing towards what, for lack of a better term, might be called software engineering virtues—abstraction, modularity, program comprehensibility, etc.—is a necessary precondition for wider adoption of reconfigurable technology. Rather than focusing exclusively on performance metrics, the new paradigm must focus as well on dimensions that enable productivity, rapid modifiability, reuse, and scalability. What is required are programming models for reconfigurable computing that embrace the software engineering virtues.

One frequently proposed remedy to the programmability issue is high-level synthesis from functional languages [3], [4], because hardware logic has a functional flavor. More to the point, functional languages support the software engineering virtues through higher-order abstractions and type systems. But many FPGA applications—especially at the system-on-chip level—are implementations of concurrent algorithms, and there is generally little or no first-class support for concurrency in either conventional HDLs or functional languages.

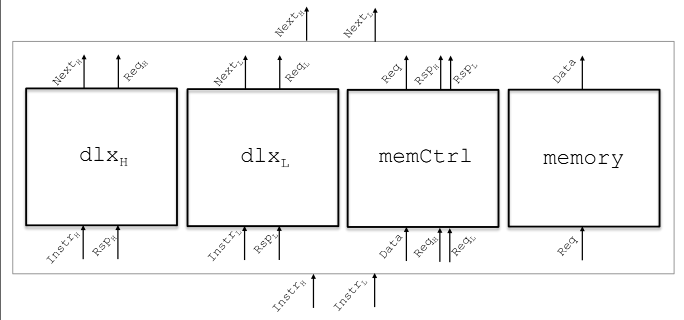
The starting point for this work is a functional language with first-class support for concurrency called ReWire. ReWire, being a concurrent language, offers direct expression of high-level concurrency templates, which, we contend, frees programmers to think at a high level of abstraction, thereby making the programming flow less error prone and effort intensive. High-level concurrency templates in ReWire remove the necessity for programmers to continually reinvent wheels. ReWire goes one better on previous approaches to high-level synthesis from functional languages by proceeding from a functional language for concurrency; this, in turn, supports the direct expression of concurrency templates in ReWire.

The efficiency of the ReWire compiler has been demonstrated in previous work [5]–[7]. The software engineering virtues we ascribe to the ReWire programming model are inherently qualitative in nature: abstraction, modifiability, etc., do not readily admit quantitative measurement. Therefore, we evaluate the software engineering virtues of the ReWire programming model by presentation of a series of use cases that illustrate the ease, elegance, and economy of expression within the ReWire language.

This case studies presented in this article are arranged in increasing order of complexity. First, we show how useful synchronization constructs—mutex (Section IV), triple modular redundancy (Section V), and barriers (Section VI)—are expressed directly in ReWire. These concurrency templates are abstract, modular, and may be modified or extended to other purposes. Fig. 1 presents our most complex example: a dual core system with a secure memory controller (Section VII) expressed in ReWire. It includes two DLX processors [8], high $\mathtt{dlx}_{\mathtt{H}}$ and low $\mathtt{dlx}_{\mathtt{L}}$, a memory controller (`memCtrl`) and a memory (`memory`). The ReWire specification for this system is also pictured below the diagram and will be described in detail in a later section.

The dual core system is "parametric" in the number of cores and in the memory access and scheduling policies, by which it is meant that these concerns may be easily modified or updated. This is supported directly by the ReWire language through its type system, which helps to separate concerns with functional abstraction and static typing. Different concerns (e.g., memory access policies) may be expressed as distinct abstractions and these abstractions are maintained by the type system. ReWire does not merely allow separation of concerns, but supports it directly in its language semantics as well.

One might counter saying that modular specifications may be written in VHDL as well and, while true, it is not the case that VHDL and other traditional HDLs support modularity directly. This direct support of the software engineering virtues can alleviate some of the burden of design and refactor from engineers. Static typing, for example, can catch many design errors automatically as such errors can often manifest as type errors, thus supporting rapid modification and refactoring.



```
dlxℓ     :: Dev (Instrℓ,Rspℓ) (Nextℓ,Reqℓ)
memCtrl :: Dev (Data,ReqH,ReqL) (Req,RspH,RspL)
memory  :: Dev Req Data
system  :: Dev (InstrH,InstrL) (NextH,NextL)
system =
  refold
     systemOut
     systemIn
     (dlxH <&> dlxL <&> memCtrl <&> memory)
```

Fig. 1: A Dual Core System realized in ReWire. The ReWire code is described in more detail in Section VII-F. Here, security levels $\ell \in \{\mathtt{H}, \mathtt{L}\}$.

II. The ReWire Programming Model

This section presents a quick overview of Haskell—and, hence, ReWire—syntax necessary to understand this paper. Throughout the paper, we have attempted to explain the meaning of ReWire code as we present it to minimize the burden to the reader. For those requiring more information about ReWire, please consult the references [5], [6].

ReWire is a subset of the Haskell functional programming language [9]—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.

Haskell [9] is a strongly-typed, purely functional language. A Haskell program consists of a number of function and datatype declarations. The type of a function from type a to type b is written, a -> b. The type for a tuple with first and second components a and b, resp., is written (a, b). The fact that a Haskell expression e has type a is written e :: a.

In Haskell/ReWire, we can introduce new datatypes with the data keyword. ReWire has built-in types for words. A 32-bit (128-bit) word belongs to the type W32 (W128), for example. Examples of Haskell data types can be found in Listing 7.

ReWire includes operators for the compositional construction of devices from other devices. ReWire enables two or more existing devices to be composed in parallel and connected together. ReWire supports a compositional style of hardware design akin to structural VHDL. Formulating the design of a hardware device may be accomplished as in previous work [5], or, existing devices may be composed with ReWire operations into bigger devices.

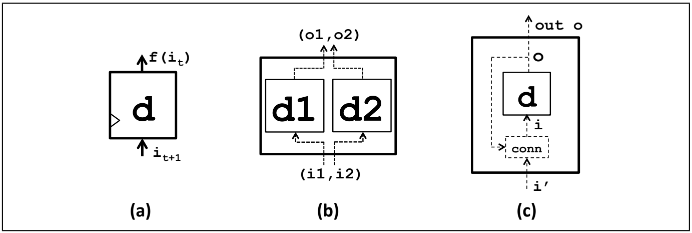
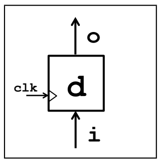


Fig. 2: *Device Constructors*: **(a)** Device d is iter f. **(b)** The device is d1 <&> d2. **(c)** the device is refold out conn d.

There is a type constructor Dev for synchronous devices in ReWire. There are three basic architectural constructors in the ReWire language. The first, iter, constructs a synchronous device from a pure function from inputs to outputs. The second, ⟨&⟩, composes two devices in parallel. The third, refold, is a recursion operator that is used to interconnect devices and/or express feedback loops (i.e., feed back device outputs to inputs).

There is one basic unit of ReWire, devices, for which we introduce the following type: Dev i o for any types i and o. A term of type, Dev i o, represents a clocked computation that, for each clock cycle, takes an input of type i, produces an output of type o, and may possess internal storage. We eschew the formal definition of Dev as it is unnecessary to understanding ReWire and its uses. Device d is clocked, as illustrated in the inset figure. The clock is represented by the underlying structure of Dev i o, rather than as an explicit parameter. A device is created in ReWire by either iterating a function or through composition of existing devices. We introduce operators for constructing devices and composing them into larger, interconnected devices. All ReWire operations are *constructors* for Dev, meaning that they are functions producing Dev i o values for some i and o types.



d :: Dev i o

a) Iteration: The most basic ReWire constructor, iter, iterates a pure function of type i -> o, producing an output corresponding to the input at each clock cycle. The Haskell definition of iter is as follows:

```
iter :: (i -> o) -> o -> Dev i o
iter f o = do i <- signal o
              iter f (f i)
```

Fig. 2**(a)** illustrates the device created with the iter operation. The type declaration above means that iter is a device constructor that takes a function from inputs i to outputs o and an initial output value and constructs a corresponding device. The device (iter f o) will, at the first clock cycle, return output o and, in the next clock cycle after consuming an input i, will produce a new output, (f i). This pattern repeats recursively ad infinitum. The (signal o) operator outputs its argument o and returns the next input. The definition of the (iter f o) constructor above may be read as (1) output o (i.e., `signal o`), (2) receive the next input (i.e., `do i <− signal o`), and then (3) repeat the pattern with new "initial" output (`f i`).

Listings 1 and 8 use another iteration primitive, `iterS`, which behaves similarly to `iter`.

b) Parallelism: Parallelism is expressed with the device constructor, ⟨&⟩, that composes two existing devices, `d1` and `d2`, into a single device, `d1` ⟨&⟩ `d2`, in which both devices operate in parallel and in isolation from one another. N.b., we are assuming, here and elsewhere, that both arguments `d1` and `d2` are non-terminating. The type declaration of ⟨&⟩ is:

```
⟨&⟩ :: Dev i₁ o₁ ->
          Dev i₂ o₂ ->
             Dev (i₁,i₂) (o₁,o₂)
```

Fig. 2**(b)** presents a pictorial version of `d1` ⟨&⟩ `d2`. The type signature of ⟨&⟩ means that the input and output types of constructed device `d1` ⟨&⟩ `d2` are pairs of the inputs and outputs of `d1` and `d2`, resp. Both subdevices `d1` and `d2` are isolated from one another in `d1` ⟨&⟩ `d2`—i.e., there is no intercommunication or shared state between them. Such interaction may be added explicitly using the `refold` operator described below. The parallelism operator may be generalized to arbitrary numbers of devices (i.e., beyond two), but, for lack of space, we only present the simplest case.

c) Interdevice Communication & Feedback: Making interconnections between devices occurs using another device level operator, `refold`. The `refold` operator can be used to connect sub-devices within its third argument and to hide internal connections as well. The use of `refold` is illustrated in Fig. 2**(c)**. Given a device `d` :: `Dev` i_1 o_1, and two pure functions, `out` :: o_1 `->` o_2 and `conn` :: (o_1 `->` i_2 `->` i_1), `refold out conn d` is a new device with the following behavior. Given an external input i' and current value output `o` by internal device `d`, the new input to `d` is `conn o` i' and the new external output is `out o`. The type of `refold` is:

```
refold :: (o₁ -> o₂)         ->
          (o₁ -> i₂ -> i₁)  ->
          Dev i₁ o₁          ->
          Dev i₂ o₂
```

Listing 4 uses a function, `refoldT` that behaves similarly to `refold`.

III. Related Work

One approach to the programmability issues with FPGAs is to provide languages and tools for high-level synthesis. The most popular strategy adapts C-like languages: LegUp [10], Vivado HLS [11], FPM [12], Streams-C [13] and ROCCC [14]. Such tools confront the challenges of extracting coarse-grain parallelism from C-like languages [15].

High-level synthesis (HLS) based on languages and libraries for parallelism [16], [17] are common. Applying functional languages to HLS is also quite common [3], [4], [18] because aspects of hardware's notion of computation (notably combinational circuitry) are inherently functional in nature. The ReWire methodology combines aspects of both of these camps, differing in that ReWire is a concurrent functional language. ReWire exhibits all of the benefits of a functional approach (e.g., expressiveness and concision) while allowing the specification of fine-grain concurrent algorithms and mechanisms. ReWire was designed as a formal methods tool [6] and the authors are motivated by the prospect of verifying security properties of hardware artifacts.

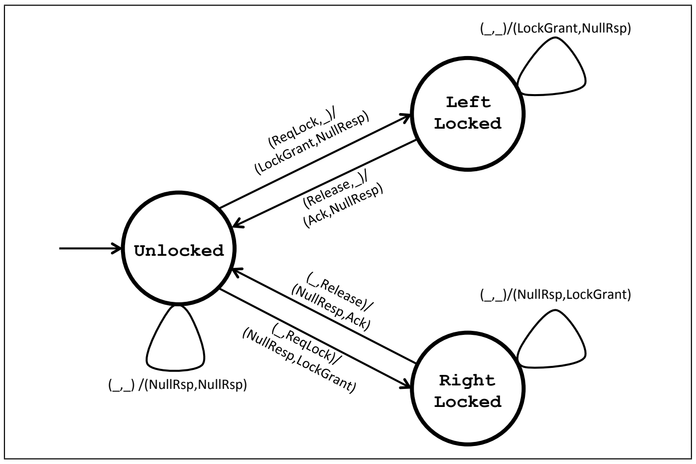

(_,_)/(LockGrant,NullRsp)
Left Locked
(ReqLock,_)/ (LockGrant,NullResp)
(Release,_)/ (Ack,NullResp)
Unlocked
(_,Release)/ (NullResp,Ack)
(_,_)/(NullRsp,LockGrant)
(_,ReqLock)/ (NullResp,LockGrant)
(_,_) /(NullRsp,NullRsp)
Right Locked

Fig. 3: Mealy Machine for Mutex. The "_" stands for a wildcard pattern.

Nikhil [4] argues that conventional software languages are unsuitable for hardware design because of their adherence to a von Neumann programming model. Hardware parallelism is "massive, fine-grain, heterogeneous and reactive" and, as such, is fundamentally inharmonious with conventional languages like C, C++, etc. Nikhil then argues that Bluespec, a functional language based on term-rewriting, does fit well with hardware's notion of computation. ReWire's language design is organized by a *reactive resumption monad* [5], a mathematical structure that models hardware parallelism which may be directly represented in Haskell.

Huffmire et al. [19] present a domain-specific language for memory access policies, based in regular expressions, and a compilation flow implementing it. Their DSL thus encapsulates a class of memory controllers with configurable security policies. How memory access policies are represented here is different in that policies are encapsulated as functions of type `Policy` expressed directly in ReWire. Given previous work on regular expression compilation in ReWire [7], this DSL approach could be easily adapted to ReWire.

IV. State Machine Constructors: Mutex

We implement a non-blocking mutex for two hardware threads as a device with three states: `Unlocked`, `LeftLocked`, and `RightLocked`. The states are represented by the type `State` below in Listing 1. On any clock cycle, the left and right threads may request the lock, release the lock, or make no request; these requests are defined as the `Req` type in Listing 1. The mutex will provide two responses on each clock cycle, one for the left and right threads, resp. The mutex may respond by granting the lock, acknowledging

Listing 1 Transition function corresponding to Fig. 3

```
data State = Unlocked | LeftLocked | RightLocked
data Req   = ReqLock | Release  | NullReq
data Rsp   = LockGrant | Ack | NullRsp

delta :: State -> (Req,Req) -> (State,(Rsp,Rsp))
delta Unlocked (ReqLock,_)
            = (LeftLocked, (LockGrant,NullRsp))
delta Unlocked (_,ReqLock)
            = (RightLocked, (NullRsp,LockGrant))
delta Unlocked (_,_)
            = (Unlocked, (NullRsp,NullRsp))
delta LeftLocked (Release,_)
            = (Unlocked, (Ack,NullRsp))
delta LeftLocked (_,_)
            = (LeftLocked, (LockGrant,NullRsp))
delta RightLocked (_,Release)
            = (Unlocked, (NullRsp,Ack))
delta RightLocked (_,_)
            = (RightLocked, (NullRsp,LockGrant))

mutex :: Dev (Req, Req) (Rsp, Rsp)
mutex = iterS delta (Unlocked,(NullRsp,NullRsp))
```

a release, or with no response; these responses are encoded by the `Rsp` type in Listing 1.

The Mealy machine for the mutex is presented in Fig. 3. The transition function for this machine appears as the `delta` function in Listing 1. Note that this mutex is left-biased because of the top-to-bottom order in which pattern-matching proceeds in ReWire/Haskell. For example, in `UnLocked`, if both left and right request the lock in the same cycle (i.e., `(ReqLock,ReqLock)`), the first clause (line 2 above) will match always because it occurs above the second clause (line 4 above). This biasing is not expressed directly in Fig. 3. The ReWire code for this device is presented in its entirety in Listing 1.

V. Devices in Parallel: Triple Modular Redundancy

Electronic components can suffer from *single event upsets* (SEUs) or non-destructive, "soft errors" that change the state of a circuit to an erroneous one. One approach to mitigating the risk of error propagation in mission-critical devices is through redundancy. Redundancy regimes have been studied extensively for decades and one of the most common regimes, the Triple Modular Redundancy (TMR) regime [20]. In TMR regimes, the core logic is replicated typically three times and executes over the same input, yielding the same output in ideal conditions. The output of each redundant device is then given to voting logic. In the event of an SEU corrupting one of the redundant devices, the other two devices will yield correct output to carry forward to the voting process and the correct result prevails.

Devices constructors in ReWire express TMR in a transparent, straightforward and scalable way (Listing 2). The voting constructor uses majority-wins vote on the output of three different devices. The `tmr` constructor replicates a given device three times using the `voter` constructor.

Listing 2 Simple TMR with ReWire

```
vote :: ((a,a),a) -> a
vote ((a1,a2),a3) | a1 == a2  = a1
                  | a1 == a3  = a1
                  | a2 == a3  = a2
                  | otherwise = a1

fan :: a -> i -> ((i,i),i)
fan _ i = ((i,i),i)

voter :: Dev i o -> Dev i o -> Dev i o -> Dev i o
voter d1 d2 d3 = refold vote fan (d1 <&> d2 <&> d3)

tmr :: Dev i o -> Dev i o
tmr dev = voter dev dev dev
```

Reuse and Extensibility: The simple TMR regime described by Listing 2 has a flaw: the voting logic is not redundant. If an SEU were to affect the voting logic, then the correct results of the devices would be for naught!

Listing 3 modifies the previous construction to replicate the voting logic. The `ftmr` constructor replicates a device three times and routes unique inputs to each device. The output is computed by three different redundant voting devices (pure logic, so this is indicated by the application of the pure function `vote`) in `voteRed`. The `ftmr` device constructor changes the input and output types because logic to "merge" the outputs would create a single point of failure.

VI. Device Synchronization: Barriers

A common synchronization construct in hardware and software is the barrier. Barriers act as a synchronization point between concurrent threads. Given a collection of threads synchronized to a single barrier, a single thread must pause execution once it reaches the barrier until all other threads synchronized to the barrier reach the barrier. Once all of the threads in the barrier have reached the barrier, all threads are subsequently un-paused and concurrent execution can resume in the same manner as before: running until they reach the barrier again, pausing, and continuing yet again.

ReWire can express barriers using `refoldT`. With `refoldT`, we can develop barriers that can pause concurrently running hardware devices until all devices have reached the barrier. The barrier device demonstrates a critical applica-

Listing 3 Functional TMR [20] with redundant voting logic.

```
voteRed :: ((a,a),a) -> ((a,a),a)
voteRed a = let v1 = vote a
                v2 = vote a
                v3 = vote a
            in ((v1,v2),v3)

ftmr :: Dev i o -> Dev ((i,i),i) ((o,o),o)
ftmr dev = refold
              voteRed
              (\ _ i -> i)
              (dev <&> dev <&> dev)
```

Listing 4 The `makeStall` device constructor

```
data Stall a = Stall | Continue a

makeStall :: Dev i o -> Dev (Stall i) o
makeStall dev = refoldT
                  (\ o -> o)
                  (\ _ -> \ mi -> mi)
                  dev
```

tion of `refoldT`—managed execution of ReWire devices—and can be easily generalized to arbitrary nubers of devices.

Before defining the barrier constructor, we need a method by which to make an arbitrary device a *stalling device*. Listing 4 defines the function `makeStall` which transforms a device in this way using the `refoldT` primitive. The `refoldT` function stalls a device if the input of this function is `Stall`. The `makeStall` function exposes this functionality by extending the input type `i` (from a device `ReT i o m a`) to be `Stall i`. Systems using a transformed device then have a method to pause it, which is to supply `Stall` instead of `Continue a`.

The `barrier` constructor defined on lines 1-5 of Listing 5 takes two devices that yield output in the type of `Busy o` and combines them into a single device that accepts a pair of inputs, one for each device, and yields output in the type `Busy (o1,o2)`. The types `o1` and `o2` correspond to the outputs of the internal devices. The barrier device yields output when both devices have produced a value. The device parameters to `barrier` are transformed to stalling devices by applying `makeStaller` to them and then placing them in parallel with the ReWire parallel combinator. The synchronization of the devices is managed by the `inp` input-processing function defined on lines 7-18. Once a device has produced output in the form of `Complete x`, we feed that device `Stall` to pause further execution from that device until the other device has also produced output. Once both devices have produced output, both are allowed to proceed executing once again. In the same cycle devices are allowed to proceed, the barrier yields both of their outputs.

Listing 5 Barrier device constructor

```
data Busy a  = Busy | Complete a

barrier :: Dev i1 (Busy o1)       ->
           Dev i2 (Busy o2)       ->
           Dev (i1,i2) (Busy (o1,o2))
barrier d1 d2 = refold
                  out
                  inp
                  (makeStall d1 <&> makeStall d2)
  where
    inp (Busy,Busy) (i1,i2)
                       = (Continue i1,Continue i2)
    inp (Complete l,Busy) (i1,i2)
                       = (Stall, Continue i2)
    inp (Busy,Complete r) (i1,i2)
                       = (Continue i1,Stall)
    inp (Complete l,Complete r) (i1,i2)
                       = (Continue i1,Continue i2)
    out (Busy,_)                 = Busy
    out (_,Busy)                 = Busy
    out (Complete a,Complete b) = Complete (a,b)
```

Listing 6 Memory protection policies as functional abstraction

```
--Policy type:
type Policy = Req -> (Req,Mask)

-- Generic "all access" policy
policy :: Policy
policy NoReq        = (NoReq,NoRes)
policy (Read a)     = (Read a,ReadRes)
policy (Write a v) = (Write a v, Written)

policyH :: Policy
policyH NoReq          = (NoReq,NoRes)
policyH (Read a)       = (Read a,ReadRes)
policyH (Write a v) | a >= 0x7FFFFFFF
                       = (Write a v, Written)
                     | otherwise
                       = (NoReq,NoRes)
policyL :: Policy
policyL NoReq       = (NoReq,NoRes)
policyL (Read a) | a < 0x7FFFFFFF
                    = (Read a,ReadRes)
                  | otherwise
                    = (NoReq,NoRes)
policyL (Write a v) | addr < 0x7FFFFFFF
                    = (Write a v,Written)
                     | otherwise
                    = (NoReq,NoRes)
```

VII. System Integration for SoCs: Memory Protection via Functional Abstraction

This section describes the construction of the policy-parameterized memory controller (i.e., `memCtrl` in Fig 1). We describe the types underlying it and then discourse further on representing memory access policies as functional abstractions. The memory controller is itself composed of two subdevices, the request and response masters, and these are described in Sections VII-C and VII-D, resp. Section VII-E describes the composition of the request and response masters into the memory controller. The construction of the dual core system using `memCtrl` is presented in Section VII-F.

Each Harvard-architecture processor receives a instruction word ($\mathtt{Instr}_\ell$) and a signal from the `memCtrl` of type $\mathtt{Rsp}_\ell$ with the result of a previous access request. A processor produces the address of the next instruction to load as well as a memory access request each cycle ($\mathtt{Next}_\ell$ and $\mathtt{Req}_\ell$, resp.).

The memory controller arbitrates processor requests to avoid starvation and, importantly, to enforce a memory access policy. The memory controller is parameterized over memory access policies. A *memory access policy* is a function of type:

```
type Policy = Req -> (Req,Mask)
```

A policy `p`, applied to a request `q`, results in a pair $(\mathtt{q}', \mathtt{m})$, where $\mathtt{q}'$ indicates the request that will ultimately be processed by `memory`. If `q` is an invalid request (e.g., an access out of bounds), then the resulting pair will be $(\mathtt{NoReq}, \mathtt{NoRes})$, indicating that `q` is treated as a non-request that produces no result. The `Mask` type indicates the sort of result returned.

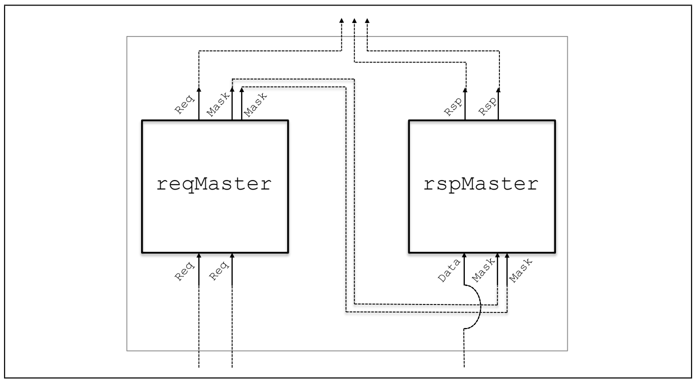


Fig. 4: Layout Diagram of the Memory Controller. The ReWire code for this controller is in Listings 8-10.

There is a subdevice of `memCtrl`—the "request master" `reqMaster_` in Listing 8—which takes the access policies as functional parameters:

```
reqMaster_
    :: Policy ->
       Policy ->
       Dev (Req,Req) (Req,(Mask,Mask))
```

Up to two requests are received in a cycle, are processed by the policy functions and then scheduled. A single memory request is sent to the memory module while response `Masks` are sent to the response master device. Note that we assume the memory will always respond to accesses in a single clock cycle; this is consistent with the behavior of block RAM resources on Xilinx FPGAs. The design could easily be adapted to memories that have different timing behavior, but we focus on this case for simplicity of presentation. The `reqMaster_` takes two access policies and creates a device enforcing those policies—this is described in detail below in Section VII-C. Parameterization supports rapid reconfiguration of memory access policies and it does so as a consequence of the choice of a *functional* HDL. To update the memory access policy, one need only change the `Policy` functions accordingly and recompile with the ReWire compiler.

The entirety of the ReWire code for `memCtrl` is presented in this section, except in several small cases where noted. The ReWire compiler and all of the ReWire code for the entire dual core system is available upon request.

A. Types

The basic types underlying this design are presented in Listing 7. ReWire contains built-in word types and, for this case study, instructions (`Instr`) and addresses (`Address`) are 32 bit words and data (`Data`) are 8 bit words. The type `Req` encapsulates the requests that can be made of the bus-master by the DLX processors: either no request (`NoReq`) or a read or write. Responses by the bus-master are: either no response (`NoRsp`); `Success` for successful writes; `Retry` to indicate memory was busy; or `ReadResult d` returning the successful result `d` of a read request.

Listing 7 Types for the Dual Core System

```
type Instr   = Word32
type Address = Word32
type Data    = Word8

data Req  = NoReq | Read Address
          | Write Address Data
data Rsp  = NoRsp | Success | Retry
          | ReadResult Data

data Mask = NoRes
          | Written --Memory written
          | Busy    --Device busy/reattempt
          | ReadRes --Result of Read

data Priority = C0 | C1

adv :: Priority -> Priority
adv p = case p of
          C0 -> C1
          C1 -> C0
```

B. Policies as Functional Abstractions

In Listing 6 we define memory access policies as two functions. One policy is for the high security domain (`policyH`) and the other is for the low security domain (`policyL`). The high domain policy function restricts writes to the upper half the addressable memory bank while the low policy restricts reads and writes to the lower half. The policies are defined as transformations on memory accesses given by processor devices. A function yields a tuple of a memory access crossed with a response `Mask` to be fed to the response master. If the request is not allowed by the policy, it is treated as no request and silently fails.

C. Request Master

The request master device is one of two subdevices that comprise the memory segmenting device. A block diagram of this device is illustrated in Fig. 4 (l.h.s.). This device handles inbound memory requests from two processors. The requests are checked by the policy functions and in the event of two valid requests, a winning request is selected by the scheduler. Response `Masks` are sent to the response master subdevice which is detailed later.

The code for the request master device is listed in Listing 8. The first three clauses of `f` handle cases where a single request or no request is made and the scheduling mechanism is not invoked. The final two clauses handle contention for the bus. If the priority is `C0`, the high processor gets the bus, otherwise the low processor wins. The priority is then advanced so the loser will win in the next contention. In this specification, if a winning processor makes an illegal request during a contention, it will win the contention, but no read or mutation will occur on the memory module.

D. Response Master

The response master is the second half of the memory segmenting device. The block diagram for this subdevice is illustrated in Fig. 4 (r.h.s.). When a memory request is made to the memory module, we send response Masks to the response device to indicate how the result from the memory unit should be handled in the next clock cycle. Every clock cycle, the response master reads the Masks and the data from the memory unit (if necessary) and signals responses to each processor accordingly.

The code for the response master device is given in Listing 9. The response master device computes the responses to send to both processors based on the output from the memory module unit and the requests Masks in a given cycle. The device operates on an input tuple that includes data (typed Data) from the memory module and a pair of response Masks (typed (Mask,Mask)) given by the input type of the device. The device outputs a pair of Rsp responses to be fed to requesting processors. The function f scrutinizes a pair of response Masks and acts on all valid pairs of them. Valid pairs of Masks are ones such that there is one "acting" Mask (i.e. a read or a write) paired with a non-request (NoRes) or "busy" Mask (Busy) to imply that the device lost a contention and should retry. The first nine clauses of f are valid pairs. The last clause (i.e., "f _ = ...") handles pathological cases for which no response is sent to either processor.

E. Composing the Memory Controller

The memory controller, memCtrl, is the top level definition of the memory protection device. It is given in Listing 10. We compose it by placing reqMaster and rspMaster in parallel with the <&> combinator and refolding over the combined device with routing logic with the functions inputSelect and outputSelect. We note that the input and output types of the memCtrl device definition on lines 1 and 2 encapsulate the interconnections between the two subdevices. That is, the response Masks are kept internal to memCtrl and are not available for external interfacing in this definition. The memory controller takes input in the form of (Data,(Req,Req)). The Data component arrives from an external memory module and is paired with two memory access requests. It produces a single memory access request to an external memory module as well as responses to the processors; i.e., it produces a tuple of type (Req,(Rsp,Rsp)).

Listing 8 The request master device

```
reqMaster = reqMaster_ policyH policyL
reqMaster_ :: Policy ->
              Policy ->
              Dev (Req,Req) (Req,(Mask,Mask))
reqMaster_ polH polL
            = iterS f (C0,(NoReq,(NoRes,NoRes)))
  where
    f p (NoReq,NoReq) = (p,(NoReq,(NoRes,NoRes)))
    f p (req,NoReq)   = (p,(acc,(rsp,NoRes)))
        where (acc,rsp) = polH req
    f p (NoReq,req)   = (p,(acc,(NoRes,rsp)))
        where (acc,rsp) = polL req
    f C0 (high,low)   = (C1,(acc,(rsp,Busy)))
        where (acc,rsp) = polH high
    f C1 (high,low)   = (C0,(acc,(Busy,rsp)))
        where (acc,rsp) = polL low
```

Listing 9 The response master device

```
rspMaster :: Dev (Data,(Mask,Mask)) (Rsp,Rsp)
rspMaster = iter f (NoRsp,NoRsp)
  where
    f :: (Data,(Mask,Mask)) -> (Rsp,Rsp)
    f (dta,(NoRes,NoRes))    = (NoRsp,NoRsp)
    f (dta,(ReadRes,NoRes)) = (ReadResult dta,NoRsp)
    f (dta,(ReadRes,Busy))  = (ReadResult dta,Retry)
    f (dta,(NoRes,ReadRes)) = (NoRsp,ReadResult dta)
    f (dta,(Busy,ReadRes))  = (Retry,ReadResult dta)
    f (dta,(Written,NoRes)) = (Success,NoRsp)
    f (dta,(Written,Busy))  = (Success,Retry)
    f (dta,(NoRes,Written)) = (NoRsp,Success)
    f (dta,(Busy,Written))  = (Retry,Success)
    f _                     = (NoRsp,NoRsp)
```

Listing 10 Memory Controller in ReWire

```
memCtrl :: Dev (Data,(Req,Req))
               (Req,(Rsp,Rsp))
memCtrl = refold
             outputSelect
             inputSelect
             (reqMaster <&> rspMaster)

outputSelect ::
  ((Req, (Mask, Mask)), (Rsp, Rsp)) ->
  (Req,(Rsp,Rsp))
outputSelect ((req,_),rsp2) = (req,rsp2)

inputSelect ::
  ((Req, (Mask, Mask)), (Rsp, Rsp)) ->
  (Data,(Req,Req))                  ->
  ((Req, Req), (Data, (Mask, Mask)))
inputSelect ((_,masks),_) (dta, accs)
                      = (accs,(dta,masks))
```

The high security processor is represented by the leftmost memory access response while the low security processor is on the right.

F. Using the Memory Controller with Processors

The memory controller is a stand-alone device written in ReWire that can interface with two processor devices and a memory module. We illustrate a use case of memCtrl with an implementation of DLX [8] in ReWire in Fig. 1. In Fig. 1, we define a system composed of two identical processors proc, the memCtrl device, and a memory module memory. This system is composed from these subdevices using the parallel combinator <&> and refold. N.b., we have not included the definitions of the systemOut and systemIn function arguments to refold as they are routine and not illuminating for this presentation.

The memory module has an input type Req and an output type Data; it takes a request on cycle n and returns the ReadResult response on cycle $n+1$ if the request at cycle n was a Read. The memory module makes no constraints or restrictions on requests. These are arbitrated by the memory controller. The specification is modular in such a way that making changes with regards to memory units is trivial.

The functions systemIn and systemOut are the routing functions used in the composition of the parallelized devices.

They operate on the raw input and output of the combined devices. The function `systemOut` selects the outputs from the combined devices that are meant for external interfacing. The output type of the whole dual core system is $(\mathtt{Next_H}, \mathtt{Next_L})$; these are the addresses of the next instructions to be fetched. These fetched instructions are the input of the system $(\mathtt{Instr_H}, \mathtt{Instr_L})$ listed in the type of the `system` device on the fourth line of code in Fig. 1. The function `systemIn` routes the external input and internal outputs between the combined devices. The memory acccesses and responses are routed between the processors, bus, and memory unit, which is encapsulated by the `refold` on line 6 of Fig. 1.

This definition encapsulates the memory module used for reading and writing, but leaves an interface for two separate program memory modules for the high and low-level processors. At a given cycle two fetched instructions are provided as inputs $(\mathtt{Instr_H}, \mathtt{Instr_L})$ and two addresses are yielded for the next instruction fetch $(\mathtt{Next_H}, \mathtt{Next_L})$.

VIII. Conclusions and Future Work

We demonstrate that a concurrent functional programming paradigm, when properly designed, can encapsulate a wide variety of popular concurrency templates useful at the hardware level. Each of these templates exhibits the "software engineering virtues" including modularity, abstraction, and program comprehensibility. Our approach has all the advantages of the usual functional language methodology with respect to FPGA programming and more: concurrency templates are extensible and reusable and produced in a language environment that uses strong, static typing to check designs and detect errors early.

Software is soft—i.e., modifiable, extensible, and reusable. Pure functional languages like Haskell and ReWire, in particular, support softness via functional abstraction: design concerns are encapsulated as individual functions and whole designs are manifested through functional composition. Hardware is hard—i.e., fixed, difficult to modify, and designed with tools that do not support abstraction or reasoning. Hardness makes it difficult, if not impossible, to inoculate hardware against emerging threats or correct errors via patching. The crux of Andrew's argument [2] is that a paradigm shift for reconfigurable hardware towards softness is necessary for the broader adoption of reconfigurable technology.

The hypothesis of this work is that the software engineering virtues are well supported for hardware by the concurrent functional programming model of ReWire. Maintaining a reconfigurable hardware system over time presumes the ability to quickly modify, refactor, and reimplement designs. The case study supports this hypothesis because, for example, the dual core system and its subdevices are easily modified and refactored. Furthermore, this refactorability is supported directly by the ReWire language itself with functional abstraction and static typing. ReWire provides language constructs for circuit design that abstract away from implementation issues with respect to timing, synchronization, and communication.

The authors are currently formalizing ReWire with the Coq proof assistant [21] as both a means of automating formal verification of ReWire security specifications and of verifying the ReWire compiler. Hardware engineers frequently view designs in graphical terms. We are planning to build a graphical front end to ReWire to aid hardware engineers and encourage adoption of the ReWire tools.

References

[1] D. Bacon, R. Rabbah, and S. Shukla, "Fpga programming for the masses," *Queue*, vol. 11, no. 2, pp. 40:40–40:52, Feb. 2013.

[2] D. Andrews, "Will the future success of reconfigurable computing require a paradigm shift in our research community's thinking?" Keynote, ARC, 2015, http://hthreads.csce.uark.edu/mediawiki/images/d/d8/Arc-presentation.pdf.

[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *DAC*, 2012, pp. 1216–1225.

[4] R. S. Nikhil, "Abstraction in hardware system design," *Queue*, vol. 9, no. 8, pp. 40–54, Aug. 2011.

[5] 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.

[6] I. Graves, W. L. Harrison, A. Procter, and G. Allwein, "Provably correct development of reconfigurable hardware designs via equational reasoning," in *Proceedings of ICFPT*, 2015.

[7] I. Graves, A. Procter, W. 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.

[8] P. M. Sailer, P. M. Sailer, and D. R. Kaeli, *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann Publishers Inc., 1996.

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

[10] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 1–27, Sep. 2013.

[11] "Vivado high-level synthesis," http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[12] C. Wang, X. Li, J. Zhang, P. Chen, X. Feng, and X. Zhou, "Fpm: A flexible programming model for mpsoc on fpga," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 477–484.

[13] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented fpga computing in the streams-c high level language," in *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, 2000, pp. 49–56.

[14] J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 127–134.

[15] S. A. Edwards, "The challenges of synthesizing hardware from C-like languages," *IEEE Design and Test of Computers*, vol. 23, no. 5, pp. 375–386, 2006.

[16] D. Greaves and S. Singh, "Kiwi: Synthesis of fpga circuits from parallel programs," in *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*. IEEE Computer Society, April 2008.

[17] J. Choi, S. Brown, and J. Anderson, "From software threads to parallel hardware in high-level synthesis for fpgas," in *Field-Programmable Technology (FPT), 2013 International Conference on*, Dec 2013, pp. 270–277.

[18] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp, "Types and type families for hardware simulation and synthesis," in *Trends in Functional Programming*, ser. LNCS, 2011, vol. 6546, pp. 118–133.

[19] T. Huffmire, T. Sherwood, R. Kastner, and T. Levin, "Enforcing memory policy specifications in reconfigurable hardware," *Computers & Security*, vol. 27, no. 56, pp. 197 – 215, 2008.

[20] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

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