Device-Level Composition in ReWire

A Dissertation presented to
the Faculty of the Graduate School
at the University of Missouri

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

---

by
Ian Graves
Dr. William L. Harrison, Dissertation Supervisor
DEC 2015

The undersigned, appointed by the Dean of the Graduate School, have examined the dissertation entitled:

Device-Level Composition
in ReWire

presented by Ian Graves,
a candidate for the degree of Doctor of Philosophy and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. William L. Harrison

Dr. Michela Becchi

Dr. Sean P. Goggins

Dr. Rohit Chadha

ACKNOWLEDGMENTS

Works such as these can't be accomplished without the tutelage, help, advice, and company of great people. First and foremost, I would like to thank my advisor Dr. William L. Harrison. It has been a privilege to work and learn in a lab that has grown and evolved so much from its inception.

Thanks go to Dr. Adam Procter, Benjamin Schulz, Chris Hathhorn, Dr. Soumya Deepta Sanyal, and my other colleagues in the Center for High Assurance Computing both past and present. Adam Procter in particular has been an incredible source of technical knowledge and creativity on the ReWire project and an all around great guy. Additionally, I would like to thank Dr. Sean P. Goggins for his invaluable advice on career and research matters, Dr. Michela Becchi for her advice and consultation on technical issues related to the work on ReWire.

Graduate work can be trying on a personal level. My case is no exception. I have found that the best way to lean in is to do so with and in good company. I'd like to thank Adam Procter and Ben Schulz for being a great guys to talk to (or share animated GIFs with). I'd like to thank my friends Brian Linquist, Ryan VanMaele, and Mark McLaughlin for their support and friendship through this time as well, despite all of us being so far flung.

Lastly, I'd like to thank my family. My parents Leland and Beverly raised me and gave me the privilege of going to school to study the things that fascinate me and bring me joy. I am forever grateful to them for everything they have done for me. I would also like to thank my sister Emma and my brother Jonah for putting up with me and supporting me as well.

My final thanks goes to my beautiful wife Amanda. Amanda has been my rock through this process and has supported me the entire way while being incredibly graceful and patient in the process. I'm a pretty lucky guy.

TABLE OF CONTENTS

LIST OF LISTINGS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

ABSTRACT

ReWire provides engineers with a tool to specify, verify and implement hardware devices for FPGAs from a high-level Haskell-like language. Previous work has shown ReWire to be a productive source language for developing whole systems in the form of single, monolithic monadic specifications.

To achieve scale, modularity and reusability, some form of modularity principle must be identified and realized within ReWire. The questions we wish to answer are, what are the basic units of a ReWire specification and how may such units be identified, abstracted over and reused to achieve a realistic work flow for device construction in ReWire? This research identifies a modularity principle for ReWire as a suite of language abstractions for breaking apart ReWire specifications into its constituent components called *Connect Logic* and considers its implementation and application.

Adding flexibility to ReWire to support device-level composition would significantly enhance design with ReWire as it would promote reuse of specifications that are complete, tested, and verified and thus reduce redundant work on the part of the designer. These functions allow the developer to manipulate and compose existing device specifications without the need to otherwise modify them. This work integrates a suite of functions into ReWire which provide engineers the ability to incorporate existing specifications into new designs and decompose projects into constituent components using Connect Logic while remaining fully synthesizable to hardware. Connect Logic provides an intuitive way to consider synchronous logic versus combinational logic in ReWire designs. We demonstrate applications of Connect Logic to improve

the performance of complex systems including cryptographic ciphers. We utilize Connect Logic to develop a fully pipelined microprocessor, to implement commonly used high-level concurrency primitives in hardware, and we demonstrate Connect Logic as a substrate for visual programming in ReWire.

Chapter 1

Introduction

This dissertation is an investigation into the composition and modularity of ReWire for hardware design. Reactive Resumption monads have been shown to be verfiable models for designing software [2] and as a basis for the design of verifiable hardware systems [3]. This work introduces connectivity primitives for writing productive and reusable hardware components in the ReWire programming language. We extend the ReWire language with four primitive functions for the composition, manipulation, and introduction of device-level, or Reactive Resumption, devices. We demonstrate that these primitives, called Connect Logic primitives, can be used as tools for optimizing deep combinational circuits (reducing gate delays) with semantics-preserving transformations. We demonstrate foundational software engineering techniques utilizing Connect Logic including modularization, encapsulation and information hiding. In total, this dissertation demonstrates that Connect Logic enables productive design that is modular, composable, and produces efficient circuits in hardware implementations.

1.1 Problems and Questions

This work seeks to address questions pertaining to the composition, modularity and reuse of a functional hardware description laguage (HDL). How can we promote modularity and re-use in a functional Hardware Description Language? Hughes noted in his seminal work [4] that functional programming matters because it promotes modular and composable programs where other programming paradigms do not. In a functional hardware description language, what is our notion of modularity here? What opportunities exist for composition of modular devices in a functional HDL? As a follow on question to promoting modularity, what does modularity and composition looks like in a functional hardware design language? Are good performance and resource characteristics possible with modular and composable design? Can a language like ReWire produce implementations that yield high throughput? What functional design techniques can we incorporate into hardware design? Functional programming has been shown to give programmers productive techniques for software. Can we incorporate any of these techniques into hardware design?

1.2 Hypotheses

This dissertation proceeds with the following hypotheses regarding the problems and questions posed in the previous section. ***First Hypothesis***: we can enable communication between synchronous logic using a series of combinators as an extension to the ReWire programming language called Connect Logic. This gives us complete inter-device communication in ReWire. We can share information between specifications as combinational (pure) functions as well as synchronous (`ReT`) specifications.

Second Hypothesis: Connect Logic enables the composition of synchronous and combination logic give us a higher degree of control over performance characteristic in a functional HDL. This control will enable us to maintain good performance characteristics and resource utilization in hardware implementations. Combinators for device composition promote device reuse as well as other modular-design-enhancing features commonly seen in software engineering. Appropriating these techniques to hardware design will provide us with additional principled approaches to developing hardware. ***Third Hypothesis***: we can compile these combinators in a way that generates device implementations with performance characteristics in line with the state of the art.

1.3 Overview of Literature

Chapter 2 is a discussion of background and related work. We introduce Haskell and monadic programming as well as Reactive Resumptions and related programming techniques in Haskell. We discuss the background work of Haskell-based systems for generating hardware devices as well as related structural and visual idioms for programming (in Haskell and otherwise).

Chapter 3 introduces Connect Logic. We discuss the design and implementation of Connect Logic primitives as well as some non-primitive, but useful transformations consisting of Connect Logic primitives. We also provide Haskell definitions of all functions for reference. Approaches to compiling Connect Logic primitives are discussed in this chapter. The first hypothesis is substantiated partially in this chapter by way of an introduction to Connect Logic.

Chapter 4 is a discussion of modularity principles that apply to ReWire. Modules as they appear in Haskell are not as useful for designing hardware. We consider modularity in the context of hardware design with ReWire. We follow on to this with a discussion of the implementation of ReWire's module system and a discussion of module system support for extra features such as polymorphic functions. The second hypothesis is covered in this discussion of modules and modules systems in ReWire.

Chapter 5, Chapter 6, and Chapter 9 are case studies using ReWire for applications to demonstrate its efficacy in design and implementation efficiency. Chapter 6 describes the design and implementation of high level concurrency primitives in hardware in addition to memory segmentation functionality and redundancy transformations. These case studies serve as a demonstration of the efficacy of ReWire with Connect Logic for integrating functionality by composition of devices and device transformations with complex compositions. Synchronization functionality for concurrent applications is added by transforming a devices to communicate with synchronization primitives. We perform a similar transformation with memory segmentation functionality. For redundancy transformation, we introduce transformations that make arbitrary devices *Triple Modular Redundant* and *Functionally Triple Modular Redundant* and demonstrate pipelining for these devices to minimize overhead to the developer.

In Chapter 9, we demonstrate Connect Logic's application to the construction of a fully pipelined microprocessor architecture, the DLX architecture. Inter-device communciation (the first hypothesis) and composability (second hypothesis) are demonstrated in this chapter. Prior work [3] has utilized ReWire for the implementation of microprocessors. We continue further by using Connect Logic to construct a processor

that is fully pipelined with support for hazard mitigation using Connect Logic. We demonstrate the modularity and encapsulation that Connect Logic brings and how it enables us to design subcomponents of a complex system in isolation. We construct a processor by combining its pipeline phases together with Connect Logic primitives, delivering a synthesizable result that operates with reasonable performance characteristics and resource requirements.

Chapter 5 is a case study in using ReWire as a substrate for a visual programming tool for hardware. We demonstrate a tool and visual editing environment using pre-defined devices as blocks for the user to wire together. We provide a transformation from the visual representation to a synthesizable Connect Logic expression. This chapter demonstrates the additional kinds of tooling that Connect Logic in ReWire can enable and the productivity it provides when coupled with the Haskell programming language for tool design.

Chapter 7 and Chapter 8 are studies in the application of ReWire and Connect Logic. Chatper 7 explores high performance Regular Expression compilation using ReWire as a target. We explore high level reasoning about regular expressions, optimizing high level models, converting the models to ReWire and the performance and resource implications of all of the above in this chapter. Chapter 8 covers an implementation of the Salsa20 stream cipher using ReWire and Connect Logic. This work demonstrates a performant implementation in ReWire using Connect Logic as mentioned by the third hypothesis. Salsa20 is complex cipher that is not feasible to directly implement as a pure function in hardware. Steps need to be taken to reduce the combinational depth of the algorithm and these steps are made feasible in ReWire with Connect Logic. We design two different implementations from the same sub-

component: an iterative version of the cipher and a pipelined version. This illustrates the space/performance tradeoff that is achievable and intuitive using Connect Logic with ReWire.

The dissertation concludes with a summary discussing the work and future works in Chapter 10.

Chapter 2

Background and Related Work

This section establishes the background and related work to the work described in this thesis. The background is framed as it relates to the core contributions of this dissertation. This includes a discussion of Haskell and monadic programming in Haskell, a discussion of the reactive resumption and its monadic representation in Haskell along with similar models of programming. Also discussed are techniques for dataflow-oriented programming both in a visual and textual ways as they related to this work. Lastly, we discuss module systems to support separate compilation in the Haskell programming language.

2.1 Haskell and Monadic Programming

Haskell is a lazy, purely functional, strongly typed, and inferred programming language [5]. The most popular implementation of the language is the Glasgow Haskell Compiler (GHC). The Haskell language itself and GHC are very closely linked with

one another. New additions and extensions to Haskell syntax and the Haskell type system are routinely facilitated through GHC's infrastructure. A number of language extensions have been integrated into Haskell proper over time. This is evident when one compares Haskell 98 to Haskell 2010 [6].

Haskell has seen the integration of many modern programming language innovations over the span of its existence. One of the most importantly used constructions in Haskell is the use of the monad. The monad, first proposed by Moggi as a useful programming feature [7], provides Haskell programmers a way to regain the sequential style of computation seen in imperative programming, while doing it in a way that is type safe and explicit with respect to effects. Monads are incorporated into Haskell from category theory. In Haskell, the Monad is treated as a kind of type class whose methods must adhere to a set of laws equivalent to the monad laws in category theory Monads can be used to model aspects of programs that are considered primitive in many programming languages. Indeed Haskell itself comes without a system for handling exceptions. Programmers are expected to provide a system for handling exceptional cases. Modeling pure, exceptional computations can be accomplished by using a structure as simple as a sum type, or the `Either` data type in Haskell's standard prelude. Additional solutions can be constructed from `Either`. These include Maybe (isomorphic to `Either () a`) or the `Error` or `Exception` monad (sequenced actions in `Either a b` where the appearance of `Left a` raises an exception of type `a`. Popular monads commonly seen "in the wild" include `Maybe`, `Error`, `State`, `Reader`, and `Writer` or simple failure, exceptions, global state, local state, and logging respectively. These monadic components can be useful in isolation but are generally used in concert like one would in a language such as Java. That is, exceptions with

global and local state. To this end, we use monad transformers. Monad transformers are combinations of monads that are themselves monads. They are constructed by way of a simple `lift` morphism [8]. Many commonly used Haskell applications (even GHC) have a monad transformer at their core. The core abstraction of ReWire, the reactive resumption, is also monad transformer [2, 9].

Haskell uses a monad to trap IO actions. IO in Haskell is problematic because Haskell is an otherwise pure, referentially transparent language. The monad abstraction is used to compartmentalize `IO` actions and require the user to explicitly type `IO` and separate it from pure computations [10]. This is in spite of the fact that computations in `IO` can be made to break the monad laws [11]. Pure actions can be lifted into `IO` like any monad, but `IO` actions cannot safely be made "pure" in Haskell. Synthesizable logic in ReWire must occur inside of a reactive resumption monadic computation in the same way that all Haskell programs that do actual work (ex. read or write to a file or port) on a machine occur in an `IO` typed computation. In the case of software, the `IO` interface dictates the boundary between the pure program realm and the rest of the world. In ReWire, the reactive resumption model provides us with an interface to set up how work is done within discrete units of time.

2.2 Reactive Resumptions, Free Monads, and Iteratees

The Reactive Resumption is part of a family of patterns that is commonly seen across the Haskell community. Members of this family of patterns have taken on a number of names over time [12]. Scholz referred to this as a concurrency monad [13], Claessen

refers to it as "a poor man's concurrency monad" [14], and Harrison refers to it as the Reactive Resumpion [2, 9]. The above patterns are more concrete instances of a more general type of monad called the free monad. Free monads in Haskell are a more recent innovation and are based in categorical concepts that bear the same name [15, 16]. All of the aforementioned patterns (less the general-case free monad) can be used to readily model cooperative multitasking.

```
1 --React with Pause wrapping a tuple of output and resumption function
2 data React i o a = Done a | Pause (o, i -> a))
3
4 --Free monad.  Note f is of kind * -> * and is a Functor
5 data Free f a    = Pure a | Free (f (Free f a))
6
7 --A functor to structure our free monad
8 newtype Funktor i o a = Funktor (o, i -> a)
9 instance Functor (Funktor i o) where
10     fmap f (Funktor (o, r)) = Funktor (o, fmap f r)
```

Listing 2.1: Reactive Resumption Monads vs. Free Monads

A common use case of this reactive family of patterns is their use in another design pattern in Haskell called the iteratee [17]. The iteratee pattern and iteratee IO is identified and named in the work of Kiselyov [18]. Iteratees are composable structures (closely resembling that of a reactive resumption) that give programmers a precise way to control evaluation of their programs. This is useful in practice where IO and laziness are concerned. A lack of control over evaluation and IO in Haskell can lead to space leaks [19] (especially thunk leaks and stream leaks) because of Haskell's laziness. Iteratees can alleviate this problem by giving the user a straightforward way to control strictness and avoid leaks. Many different Haskell libraries for data stream

processing including Conduit [20], Pipes [21], and Enumerator [22] are based upon the iteratee design formalized by Kiselyov.

ReWire specifications fall into the family of reactive programming because of their implicit reaction to some input stimulus or an argument accompanying a clock tick. ReWire and other Haskell-based HDL's aren't the first languages to employ a reactive style to model an HDL. SystemC uses the C++ language syntax as a way to model reactive systems and can also be synthesized to VDHL [23]. SystemC leverages the object model of C++ to facilitate reactive system specification. Like ReWire, it can also be synthesized to hardware.

ReWire inherits many ideas from the functional reactive programming (FRP) paradigm [24]. Conal Elliot pioneered the FRP paradigm under the name of functional reactive animation (FRAN) in his initial work [25]. Later work by Hudak has explored FRP and its application to program and device specification as well FRP's relation to Arrows [26]. An important difference between FRP and ReWire is FRP's focus on signals and their timing. Devices in ReWire (typed in React) are synchronously clocked where FRP signals are considered to be continuous.

2.3 Specifying Hardware in Haskell

2.3.1 Lava

Lava is the oldest approach for circuit specification in Haskell [27]. The modern iteration of the Lava is Kansas Lava [28]. Lava leverages the expressiveness of Haskell to specify circuits in succinct and elegant ways, but this isn't without its challenges.

The question of synthesizing a recursive function in a sane and efficient way has resulted in different approaches to the same question. Chalmers Lava relies on the use of monadic constructions in Haskell to accomplish this task while later iterations of the work in Kansas Lava rely on observable sharing, or a form of cyclic graph analysis on Haskell code to identify recursive structures in circuit specification. As a Haskell-hosted domain specific language, Lava can be simulated by using an interpreter written in Haskell. Circuits specified in Haskell using Lava can be simulated and checked for their accuracy using unit testing or other lightweight formal methods.

2.3.2 Clash

Clash (CAES Language for Synchronous Hardware) is a newer solution for specifying hardware in the Haskell programming language [29]. Clash is billed as a Haskell-to-VHDL compiler by way of term rewriting [30]. Like ReWire, Clash places restrictions on the kinds of Haskell programs once can synthesize. Clash's restrictions center on the kinds of algebraic data types, a user can employ when specifying a device using Haskell.

2.3.3 Bluespec

Bluespec is a well-established tool for specifying hardware in a Haskell like language. Arvind [31] describes it as a "relatively a relatively simple DSL (GAAs [Guarded Atomic Actions] and modules) with a fully functioning Haskell-like meta programming layer on top." Bluespec enjoys a successful reputation in industry. The tool appears to be closed source and I personally have not employed it for comparison

purposes to ReWire. From the descriptions that are available, Bluespec appears to be an environment in which developers specify hardware in a synthesizable DSL (of GAAs) and extend the functionality and expressiveness of the tool by using metaprogramming in Haskell. ReWire currently has plans for metaprogramming extensions, but currently does not employ any first-class metaprogramming features.

2.3.4 Forsyde

ForSyDe (Formal System Design) is a methodology for hardware design that starts from high level formal models and maps them to hardware through a series of refinement stages [32]. ForSyDe is both a framework and a shallow-embedded domain specific language in Haskell. Like Lava, ForSyDe designs can be simulated in Haskell. ForSyDe operates on a higher level of abstraction than Lava in that synthesized systems are specified semantically instead of structurally (ala circuit-style devices in Lava). Additionally, both ReWire and ForSyDe specify synchronous devices. ForSyDe emphasizes the use of synchronous devices as targets because it simplifies reasoning about these devices at the higher level [33]. Users of ForSyDe are restricted to specifying only the kinds of devices that can be mapped through a ForSyDe refinement process.

2.3.5 Delite

The Delite DSL compiler framework [34] seeks to address the "three P's" with respect to implementing software on parallel, heterogeneous systems. Delite addresses portability (i.e., retargetability of DSL compilers to a broad range of parallel hardware)

through *language virtualization*.

2.4 Visual or Flow-based Program Specification

There are numerous approaches to programming from an entirely visual standpoint as well as numerous more domain specific approaches or visual representations of specific programming languages [35, 36]. Additionally, some approaches take a reversed approach and use their own syntax to create visual programming techniques in their (non-visual) native languages (Arrows and Profunctors are an example).

2.4.1 Flow-based Programming

Flow-based programming is a visual programming paradigm for specifying applications as asynchronous processes connected visually by directed data flows [37]. The formalization of flow-based programming has seen the creation of numerous visual tools in this vein with new tools continuing to appear on a regular basis.

2.4.2 Arrows and Profunctors in Haskell

There has been a significant amount of work in the Haskell community on methods to generalize composable, functional computations. Arrows are a programming model that generalize functions in Haskell and are an alternative way to structure computations to more traditional mainstream methods like function composition and monadic computation (though certain types of Arrows are equivalent to monads) [38]. Arrows are named for their correspondence to functors in category theory (arrows). The

basic Arrow combinators bear some similarity to the connect logic primitives defined in this work. Indeed, Connect Logic regains similar composability to that of Arrows, specifically for reactive resumptions.

A similar work to Arrows in Haskell is the community's work on structuring computations using profunctors [39]. Profunctors are built on existing Haskell functor types and perform an equivalent function as Arrows. Profunctors are structures in Haskell that are a combination of covariant (i.e. the Functor typeclass) and contravariant functors. A Haskell function is a concrete example of a profunctor its output type can be extended by providing a function from the original output to a new output (covariance) and its input type can be extended by providing a function from a new input type to the original input type (contravariance). The `refold` function in Connect Logic can be used as the basis for treating `React` in ReWire as a profunctor as it can extend both the input and output of a reactive resumption.

2.4.3 Visualizing Functional Programs

Techniques for visualizing programs and visual program manipulation are not new. Functional programming languages make particularly good targets for visualization. The basis of a functional program is an expression of function applications, which is intuitively visualized. Additionally, many functional languages provide a more explicit means to control for side effects, which are inherently hard to visualize, especially when they may not be represented in the source language.

The lambda calculus itself has been a target for visualization. The work of Citrin proposes a series of circle and line arrangements to represent applications in an expression [35]. The work of Massalõgin [36] proposes a similar set of circle arrange-

ments. A sophisticated visual representation representation for the Haskell language was proposed by Reekie [40].

Additional work has taken a more universal approach to defining what a visual language is and what it means. The work of Erwig has seen specifications for semantics of visual languages [41] as well as hybrid visual and textual languages [42]. Other (perhaps more esoteric) work has seen attempts to provide graphical descriptions of existing programming languages including functional languages. Hemann's work in visualizing languages uses unique approaches such as mapping programs onto a Hilbert curve for visualization [43].

2.5 Functional Module Systems

The work detailed in this work centers on a module system for ReWire that supports object code in the form of VHDL. ReWire is a proper subset of Haskell and thus a module system in ReWire syntactically follows the structure and semantics of modules described by the Haskell Report. Work has been done to establish the syntax and semantics of module declarations in the Haskell programming language. The Haskell Report [6] establishes the use and structure of modules for engineers to use. Additional work has been done to establish a formal semantics to the Haskell module system by Diatchki et al. [44] which provides an in-Haskell semantics for reasoning about Haskell modules.

Using Haskell for practical applications inevitably leads one to need an interface with a foreign programming environment. This is also the case where modules are concerned. Can we compile Haskell modules to an object format that interface with

each other? One implementation of a system in this vein is the work by Finnel et al. to interface Haskell with the Microsoft Component Object Model (COM) system [45]. This work, while targeting software instead of hardware, describes a compilation method to extend and compile Haskell so that Haskell programs can be componentized in a main stream object framework. This work addresses the steps taken to address the various impedance mismatches between the source language and the target environment (in this case object-oriented strict imperative vs. purely functional and lazy).

The work of Fortounis and Papaspyrou focuses on supporting separate compilation for a subset of Haskell that includes parametric polymorphism in functions [46, 47]. This work emphasizes support for defunctionalization using a kind of closure conversion. The ReWire compiler includes similar program transformations in its compilation pipeline. This work is relevant in that it discusses separate compilation that allows for defunctionalization across different modules that may have overlapping, but disconnected defunctionalized members. For example functions with the same arity in their higher-order arguments will have different closure types if compiled separately. This work unifies these same, yet differently-named definitions. Separate compilation in ReWire supports parametric polymorphism in compiled (VHDL) modules in a similar way to how this work supports it, but in the C programming language.

2.6 Formal Methods for Hardware Design

There is a long history of formal methods being applied to hardware designs [48]. The general process involves encoding a hardware design in the logic of a theorem

prover by hand[1] and then proving theorems about the encoding. There is an obvious danger that the encoding process—which one might call *semantic archaeology*—will introduce errors as well as a problem of soundness (i.e., how do you know a theorem about the encoding applies to the hardware device itself?).

"Semantic archaeology"—the process of developing a formal specification for an *existing* computing artifact—is the principal reason that formal methods can be so time-consuming and expensive. Sarkar et al. [49] describe the semantic archaeology process in the context of modeling the x86 multiprocessor instruction set architecture: "*The key difficulty was to go from the informal-prose vendor documentation, with its often-tantalising ambiguity, to a fully rigorous definition (mechanised in HOL) that one can be reasonably confident is an accurate reflection of the vendor architectures (Intel 64 and IA-32, and AMD64).*"

Cryptol [50] is a domain-specific language for specifying, verifying and implementing cryptographic algorithms. Given a cryptographic algorithm, one can specify it in Cryptol, run a number of automatic and semi-automatic proof tools over the specification, and ultimately generate C code implementing the algorithm itself. The current open source version of Cryptol (v.2) does not generate hardware implementations, although a previous proprietary version (v.1) did. ReWire, by contrast, is a subset of Haskell compilable to VHDL and is not restricted to cryptographic algorithms. Salsa20 has been specified in Cryptol v.2, but no effort has been made to backport this specification to Cryptol v.1 and synthesize it.

The usual standards for evaluating hardware architectures and design flows are

[1] E.g., Isabelle/HOL (`http://isabelle.in.tum.de`), ACL2 (`http://www.cs.utexas.edu/users/moore/acl2`), and PVS (`http://pvs.csl.sri.com`), are the most commonly used provers for hardware verification.

performance-based metrics (e.g., time and space performance, power usage, etc.). Within the context of mission critical systems, formal analysis and verification are required evaluation modes as well. The Common Criteria for Information Technology Security Evaluation (a.k.a. Common Criteria or CC) is an international standard (ISO/IEC 15408) for computer security certification and the US Federal government mandates following the CC requirements for mission critical systems. The CC sets seven evaluation assurance levels (EAL). The most stringent such level is EAL7, which requires "*extensive formal analysis*" for applications in "*extremely high risk situations and/or where the high value of the assets justifies the higher costs*" ensuing from formal verification [51]. For reconfigurable computing to be applied in the space of mission critical systems, cost effective formal methods techniques must be developed. The current research is a step in this direction.

Previous work demonstrated the construction and verification of a secure many-core system in ReWire [52]. The present work, in contrast, demonstrates the expression of a common hardware design pattern (stall-free pipelining) in ReWire and its verification. The emphasis in the former was on the design and implementation of the ReWire language, while the current work focuses on ReWire as a vehicle for hardware verification.

2.7 Compiling Regular Expression Matchers to Hardware

The conversion of sets of regular expressions into NFAs is a well-known procedure [53]. Sidhu and Prasanna [54] have proposed an efficient FPGA implementation of NFAs.

Their solution is based on the one-hot encoding scheme; the use of an NFA representation avoids the $O(2^n)$ space complexity that is characteristic of DFA (deterministic finite automata) representations, typically adopted in memory-based regular expression matching implementations [55–58]. Subsequent efforts on FPGA [59–62] have refined Sidhu and Prasanna's implementation and achieved gigabit/sec processing throughputs on real-world pattern sets.

Chapter 7 demonstrates that the ReWire compiler works at scale as the generated ReWire programs are on the order of 100K LOC. Great care was taken in the design of ReWire so that it possesses a rigorous denotational semantics to support formal verification while maintaining synthesizability for all of its programs [3]. ReWire is also a virtualized DSL in that it has a separate compiler backend for producing FPGA-based implementations while reusing large parts of its host language's infrastructure—including Haskell's type system, front end, etc. In George, et al., [63], the Delite framework is adapted to the generation of hardware from DSLs, specifically the hardware acceleration of kernels in a heterogeneous setting.

Chapter 3

Connect Logic

The reactive resumption type is demonstrably useful for specifying whole synchronous devices. In practice, however, devices are not usually monolithic specifications. Engineers and developers reuse prior specifications that are applicable to the project at hand. This is the essence of reusability in practice and is a critical component to all modern software design methods. Connect Logic for ReWire introduces new primitives between synchronous devices to further reusability and introduce aspects for information sharing and timing between different devices. This chapter describes the nature of Connect Logic, its main primitive functions, and new functionality in ReWire that can be derived from Connect Logic. A discussion of the implementation of Connect Logic is also provided.

3.1 Motivation

When considering a language or an environment for writing software, one of the first things a programmer considers is how the tool allows for the decomposition of a problem and how easily parts of another previously developed solution can be reapplied to future problems. How easily can we import our old subprograms and subroutines for use in a new program? The same is true for hardware design. Common subcomponents can be factored out of just about any hardware specification. Visual inspection of any hardware device reveals hundreds, if not thousands of embedded subcomponents.

ReWire in its original form supports the composition of non-synchronous logic via pure function composition. Connect Logic seeks to extend the compositionality of ReWire by enabling interactions between combinational/synchronous interfacing and synchronous/synchronous interfacing. It does so in the following ways. First, Connect Logic provides the means to represent parallelized synchronous devices. Second, Connect Logic provides a way to compose synchronous devices with combinational logic. Third and last, through the two prior mentioned extensions, Connect Logic allows for clocked communication between synchronous devices specified in `ReT`. In short, Connect Logic brings communication between components specified in ReWire, and in so doing it also brings reusability.

3.2 Connect Logic Primitives

Connect Logic is comprised of four primitives: `refold`, `parallel`, `iter` and `refoldT`. Additionally there are some important non-primitive functions that follow from these

which include `pipeline`.

3.2.1 Primitive Functions

Primitive functions in Connect Logic are called primitive because of the nature of the types they operate one. ReWire does not provide support for general recursion, higher order functions, or operations that can decompose types in `ReT` (i.e. decompose device specifications). While general recursion is disallowed, ReWire allows for tail recursion in functions where the codomain is in `ReT`. These tail recursive calls in are allowed because correspond to state transitions in a finite state machine and be compiled as such. In order to maintain full synthesizability of all proper ReWire programs, it follows that any extensions to ReWire must preserve this correspondence in addition to corresponding to some synthesizeable structure in hardware. Connect Logic is a set of specific extensions that adheres to both of these principles.

Intuitively, where one pictures synthesizeability of structures in `ReT` by way of a transform-ability to finite state machines, one can picture the synthesizeability of Connect Logic extensions by picturing them as "the wiring" between whole devices on a board. Connect Logic operations work by encapsulating existing specifications in a new, opaque device, that performs wiring "under the hood". In the case of `parallel`, the new device takes the inputs of both interior parallel devices, splits the input and routes it to each corresponding device, then combines the output again for consumption or inspection from the outside. The `refold` function behaves similarly, but instead takes two pure functions to manipulate inputs and outputs of the old device in `ReT` to "route" old input and output types to new ones.

The Parallel Operator

The parallel function is a basic parallel operation for devices in the `ReT` type. Pure functions in ReWire have the benefit of being parallel by their nature. When it comes to whole devices specified in `ReT`, however, we need to be more explicit with an additional caveat. Devices in `ReT` are synchronous logic (or "clocked") unlike pure combinational logic. When merging two devices together using the parallel operator, the devices are now synchronized in lock step with one another because they are now treated as one device on the same clock. This is formalized in the definition of the parallel operator in Listing 3.1.

```
1  (<&>) :: ReT i1 o1 I a -> ReT i2 o2 I a -> ReT (i1,i2) (o1,o2) I a
2  (<&>) (ReT l) (ReT r) =
3   ReT (do
4          l' <- l
5          r' <- r
6          case (l',r') of
7           (Left a,_)                          -> return (Left a)
8           (_,Left a)                          -> return (Left a)
9           (Right (o1,res1),Right (o2,res2)) ->
10           return (Right ((o1,o2),\(i1,i2) -> (res1 i1) <&> (res2 i2))))
```

Listing 3.1: Haskell implementation of the Connect Logic parallel (`<&>`) combinator

The parallel operator (depicted as the ampersand in Listing 3.1) takes two arguments of type `ReT` with different inputs and outputs over the Identity monad and the same termination type `a`. The parallel operator restricts the monad stack to Identity to force devices placed in parallel to be fully encapsulated. Devices can still have internal state, but it cannot be represented in the monad transformer stack here, lest

we imply the existence of unclocked information channels between devices placed in parallel. We make these channels explicit using refolding noted in a later section. The result is a new reactive resumption with input and output types that are tuples of those of the original two devices. The new device is synchronized such that one "tick" corresponds to a complete iteration of all of its internal devices. In other words, the "speed" of the new device is as fast as its slowest sub-component. A visual representation of this intuition is given in Figure 3.1.

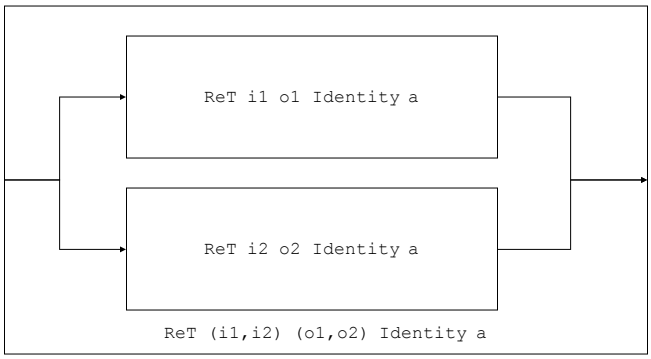


Figure 3.1: An illustration of the parallel operator in Connect Logic.

The `refold` Operator

The `refold` operator provides the developer with a means to alter a given device in `ReT` given combinational logic (pure functions) to affect its inputs and outputs. The `refold` operator can be thought of as a device given two adapters. This is formalized in Listing 3.2.

```
1 refold :: (Monad m) => (o1 -> o2) ->
```

```
2                          (o1 -> i2 -> i1) ->
3                          ReT i1 o1 m a ->
4                          ReT i2 o2 m a
5 refold otpt inpt (ReT r) =
6  ReT (do
7         r' <- r
8         case r' of
9         Left a           -> return (Left a)
10        Right (o1,res1) ->
11         return (Right (otpt o1, \i2 ->
12                 refold otpt inpt (res1 (inpt o1 i2)))))
```

Listing 3.2: Haskell implementation of the Connect Logic `refold` combinator

The first argument to `refold` is a function that converts the device's original output type to a new output type. The second argument adapts the original device's input to accept a new input by way of a function that maps the new input type to the old input type. This conversion of input types observes the original output of the device as part of the mapping. This enables the developer to write adapter functions that allow him or her to observe device state based on device output. This added expressiveness becomes important later on as we use it to implement pipelining with Connect Logic. A visual representation of this mapping is provided in Figure 3.2.

The `refoldT` Operator

We can manipulate devices based on their inputs and outputs with refold and we can pair devices together using the `parallel` operator, but there is a component missing: timing. Reactive Resumptions execute in a step-wise fashion in a manner similar to

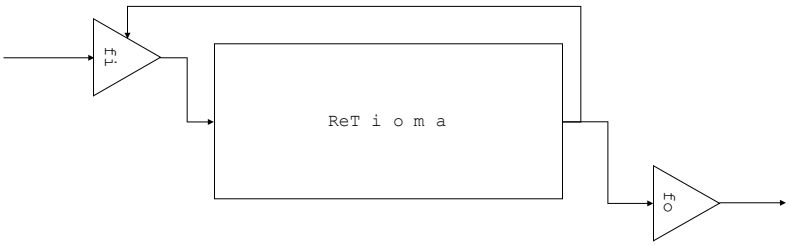


Figure 3.2: An illustration of the `refold` operator's functionality in Connect Logic.

a hardware device reacting to clock pulses. It would be very useful if we had a means to control whether or not a given device steps without needing to alter the device. This is where `refoldT` comes in. The `refoldT` function is a generalization of `refold` developed when working with systems in need of finer grained execution control.

```
1  refoldT :: Monad m => (o1 -> o2) ->
2                        (o1 -> i2 -> Maybe i1) ->
3                        ReT o1 i1 m a ->
4                        ReT o2 i2 m a
5  refoldT fo fi (ReT res) =
6    ReT (do
7            p <- res
8            case p of
9             Left a             -> return (Left a)
10            Right (o,resume) -> return (Right (fo o, dispatch o resume)))
11   where
12     dispatch o1 resume = \ i2 ->
```

```
13        case fi o1 i2 of
14          Nothing -> ReT (return (Right (fo o1, dispatch o1 resume)))
15          Just x  -> refoldT fo fi (resume x)
```

Listing 3.3: Haskell implementation of the Connect Logic `refoldT` combinator

Listing 3.3 shows the type of `refoldT`. The definition refoldT is nearly identical to `refold` with the exception of the result of the input-manipulation function. For values of the input function that are `Nothing`, the internal device is stalled and will be paused. Values in `Just` execute the same way as a normal `refold`. Indeed, `refoldT` is a generalization of sorts of the original `refold` function where `refold fo fi = refoldT fo (\x y -> Just (fi x y))`. We illustrate `refoldT` in Figure 3.3.

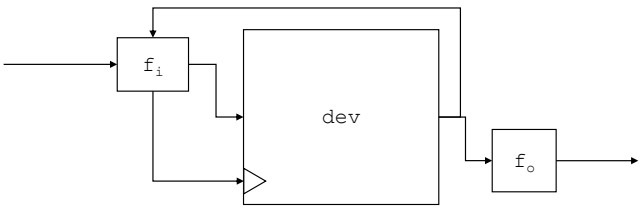


Figure 3.3: Applying `refoldT` to a device `Dev` to manipulate inputs with f_i and outputs with f_o.

The `iter` Operator

The `iter` function is a kind of lifting function for pure functions into the `ReT` monad. The type of iter is demonstrated in Listing 3.4.

```
1 iter :: Monad m => (i -> o) -> o -> ReT i o m a
2 iter f init_o = do
3                   i <- signal init_o
4                   iter' f i
5   where
6     iter' f i = do
7                   i' <- signal (f i)
8                   iter' f i'
```

Listing 3.4: Haskell implementation of the Connect Logic `iter` combinator.

The `iter` function produces a new synchronous device in `ReT` that wraps a pure function. The user is required to provide an output initialization for this device. This combinator gives us a way turn pure functions into synchronous ones.

3.2.2 Non-Primitive Functions

Non-primitive are functions that transform devices using only primitive functions. Put another way, non-primitive functions can be thought of as macros of primitive functions. Here we describe one natural non-primitive function, `pipeline` that follows from primitive Connect Logic functions. Additional non-primitive Connect Logic transformations appear in Chapter 6.

The `pipeline` Operator

The `pipeline` function is a (non-primitive) combination of the parallel and `refold` functions. This function creates a pipeline of devices that perform operations in parallel and feed-forward information for further computation to subsequent devices in the pipeline. This function allows the developer to break large combinational hardware into smaller synchronous devices to increase throughput. The device is defined in Listing 3.5.

```
1 pipeline :: Monad m => ReT i z I a -> ReT z o I a -> ReT i o I a
2 pipeline dev1 dev2 = let combined = dev1 <&> dev2
3                      in refold combined snd pipe
4     where
5         pipe whole_output i_input = (i_input,snd whole_output)
```

Listing 3.5: The Haskell definition of the `pipeline` function in ReWire

The `pipeline` function is formed from the main primitive functions in connect logic. All devices in a pipeline execute in parallel. Connecting inputs and outputs between pipelined devices is accomplished by the `refold` operator and some simple pure functions that select members of a tuple. This action is visualized in Figure 3.4.

3.3 Implementation

Implementation of Connect Logic occurs at multiple levels of the ReWire compilation process. Primitive functions are factored out of early compilation and are used to direct the port mapping of their subexpressions on the back end of the compilation process. Non-primitive functions must be discharged prior to compilation. This can

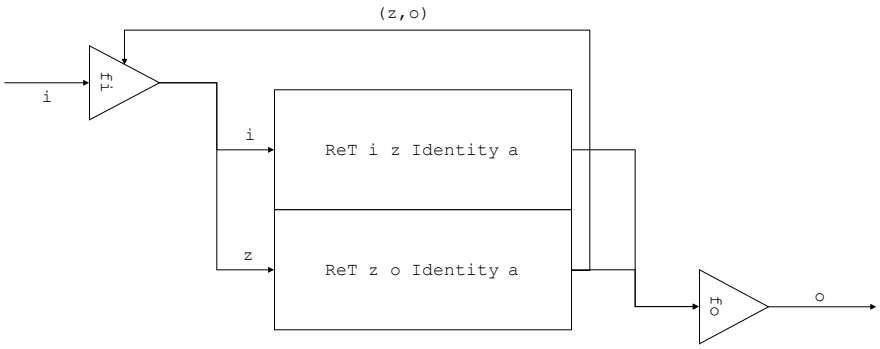


Figure 3.4: An illustration of the `pipeline` operation's functionality in Connect Logic.

be accomplished with inlining, and simplifications.

3.3.1 Structuring Connect Logic Devices

Connect Logic is intended as a system for connecting discrete, seperate components. In the source program, these components are presented as subexpressions of a monolithic, whole ReWire expression. The first step in this process is to identify and separate subcomponents in ReWire expressions. For the purpose of this description, we refer to ReWire component specifications containing no Connect Logic primitives as *trivial* devices and devices containing Connect Logic primitives as *non-trivial* devices. Devices that contain additional Connect Logic will contain device specifications. We call these contained devices *nested devices*. Connect Logic expressions have no limit on expression depth. Pipelining devices is performed in two different phases, one "deep" approach involves nested Connect Logic device specifications and another "shallow" approach is a single, n-length `parallel` expression wrapped by a

single `refold` expression. The shallow approach produces less complex VHDL while the deep approach allows more flexibility to the programmer to design a pipeline without knowledge of its entire structure before-hand. Both of these approaches are discussed in the subsequent section on pipelineing. Tests using pipelining technique employed in the Salsa specification in Chapter 8 created expressions that were both deep and shallow. The synthesized hardware from both approaches yielded hardware with nearly identical performance characteristics.

During the compilation phase, the Connect Logic extension to the ReWire compiler operates by analyzing the expression tree of the `start` (main) function specified by the developer. The Connect Logic extension performs type-directed analysis of the main ReWire device expression. A non-trivial ReWire device contains subexpressions in the type of `ReT`. Trivial device specifications are passed along to the compiler unchanged. This is the same process as a standard non-Connect Logic ReWire compilation procedure. The decomposition process works by constructing a new tree composed of trivial devices, `refold` nodes, and `parallel` nodes. The relevant structures are illustrated in Listing 3.6.

```
1 type Fun = RWCExp
2 data CLTree f a  = Par [CLTree f a]
3                  | ReFold f f (CLTree f a)
4                  | Leaf a
5
6 type CLExp     = CLTree Fun RWCExp
7 type CLNamed   = CLTree Fun String
```

Listing 3.6: A data type for decomposed CL expressions

The `CLTree` type illustrated in Listing 3.6 is a data type parameterized over a

function type `f` and a label type `a`. The type `f` parameterizes the type of the pure function that `refold` uses to manipulate the input and output of a device. The type we use is `RWCExp`, the type of ReWire expressions in the ReWire front end. This is represented in the `CLExp` and `CLNamed` type synonyms. The initial transformation from the ReWire core representation yields a structure in the type `CLExp`. We perform a monadic transformation from `CLExp` that maps to `CLNamed` using the following monadic construction.

```
1 type CLM = WriterT [NCL] (WriterT [NRe] (StateT Int Identity))
2 clExpr  :: RWCExp -> CLExp
3 rnCLExp :: CLExp -> CLM CLNamed
```

Listing 3.7: Types used in the construction of the renamed CL tree

Given a ReWire expression, we can transform it to the equivalent `CLExp` representation, which removes the Connect Logic primitive functions from the expression and structures its subexpressions accordingly in the tree. Given the resulting tree, we remove each node from the tree, starting with the leaves, save it, and replace it with a unique named reference (or node in the type `CLNamed`). First, the trivial (all leaf nodes) are replaced, then the connecting non-trivial nodes (corresponding to Conenct Logic expressions) are replaced. The end result is the root node of the expression, which we then use to drive VHDL code generation.

VHDL code generation is performed on each subexpression (and subdevice) singularly. Given any node from the original `CLExp` tree, we are presented with one of two situations. First, the node we have is a `Leaf` node, and is trivial, or a non-Connect Logic device. We can compile this node as is done in the standard ReWire approach. If the node we are presented with is a `refold` or `parallel` node, then we compile it

in one of the methods described in the next section. The resulting compilation is a series of VHDL entity declarations in a single VHDL file, tied together using VHDL structural mapping features.

3.3.2 Compiling Primtives

The primitive operators in Connect Logic are compiled in one of two different ways. In the case of `parallel`, the operation is given a reference to the two sub components to execute in parallel, and VHDL code is constructed to execute two mapped devices with structural parallelism. In the case of `refold`, a singular reference to a device is given, along with two different pure functions. Both functions are compiled to pure VHDL functions encapsulated in a VHDL entity. These functions are applied to the input and the output of the nested, or encapsulated sub-component given by the device reference. The I/O of the encapsulated device is mapped to internal signals of the `refold` device. The corresponding functions are applied to those signals. The output function transforms the output of the encapsulated device and the input function produces input for the device based on its previous output as well as the input to the `refold` device. A more detailed description of each follows in the subsections.

parallel

A parallel operation is compiled by analyzing its sub-components, compiling them, and then constructing a VHDL specification based on the bit widths of the input and output types of a device in `ReT`. We provide an example of a hand-compiled device composed of a single `parallel` (`parI`) function specified in Listing 3.8. The following example is illustrative of the Connect Logic compilation process for a `parallel` oper-

ator, which produces significantly larger code that includes compiled trivial devices. These are omitted here.

```
1 data Bit = Zero | One
2
3 not :: Bit -> Bit
4 not x = case x of {Zero -> One; One -> Zero}
5
6 not_d :: Bit -> ReT Bit Bit Id ()
7 not_d x = signal (not x)
8
9 start :: ReT (Bit,Bit) (Bit,Bit) Id ()
10 start = (not_d Zero) <&> (not_d Zero)
```

Listing 3.8: An example parallel device

In Listing 3.8 we specify a simple device composed of two devices that `not` their inputs using the `parI` combinator. In an intuitive sense, compiling the `parallel` operation follows from compiling `left Zero` and `right One`. These devices are trivial devices. We can inspect their types from the type checker to determine their type width, which is one bit wide. Thus, the total width for the parallelized device is two bits. We require this information before generating VHDL so we can properly split the input of the combined device to each of its sub-components. The VHDL for the device specified at `start` in Listing 3.8 is illustrated in Listing 3.9.

```
1 entity start is
2   Port ( clk    : in std_logic ;
3          input  : in std_logic_vector (0 to 1);
4          output : out std_logic_vector (0 to 1));
5 end rwcomp4;
```

```
6  architecture behavioral of start is
7    signal dev0in  : std_logic_vector(0 to 0);
8    signal dev0out : std_logic_vector(0 to 0);
9    signal dev1in  : std_logic_vector(0 to 0);
10   signal dev1out : std_logic_vector(0 to 0);
11 begin
12   dev0in  <= input(0 TO 0);
13   dev1in  <= input(1 TO 1);
14
15   dev0 : entity work.not_d(behavioral)
16     port map (clk,dev0input,dev0output)
17
18   dev1 : entity work.not_d(behavioral)
19     port map (clk,dev1input,dev1output)
20
21   output <= dev0out & dev1out;
22 end behavioral;
```

Listing 3.9: VHDL code for the example parallel device.

The code in Listing 3.9 is an abbreviated version of the VHDL-compiled form of the device specified in the `start` function in Listing 3.8. Paralleism specified by Connect Logic is accomplished by port mapping instances of the entity `not_d` (not illustrated). As noted prior, the `input` and `output` for this device is two-bits long. The clock signal, `clk` is passed through to the sub-components, `dev0` and `dev1`. The parallelization of the two sub-components is a simple mapping of inputs and outputs in and imposes none of its own overhead on the performance of the parallelized devices. In other words, a parallelized device is as fast as its slowest sub-component.

refold

Compiling a `refold` function follows a similar process as compiling a `parallel` function. A `refold` operation is, in essence, "wrapping" a device with functions that manipulate its inputs and outputs. Where in the `parallel` operation we place to devices side by side and map their combined inputs and outputs to a wrapping device, in `refold` we wrap one device and map its I/O, but we additionally map the I/O through the pure functions provided as arguments to the `refold` function. An example hand-transformation between a ReWire representation is illustrated in Listing 3.10 and Listing 3.11.

```
1  data Bit = Zero | One
2
3  and :: Bit -> Bit -> Bit
4  and b1 b2 = case (b1,b2) of
5                  (One,One) -> One
6                  _         -> Zero
7
8  not :: Bit -> Bit
9  not x = case x of {Zero -> One; One -> Zero}
10
11 opaque_dev :: ReT Bit Bit Id ()
12 opaque_dev = <..>
13
14 start :: ReT Bit Bit Id ()
15 start = refold not and opaque_dev
```

Listing 3.10: An example refolded device

The example ReWire code in Listing 3.10, illustrates a device consisting of a sin-

gle `refold` that inverts the output of the device `opaque_dev` and uses `and` to map two `Bit` values to one `Bit`, the input type of `opaque_dev`. In this illustration we keep the definition of `opaque_dev` abstract. The compiled form of `start` is given in Listing 3.11.

```
1 entity start is
2   Port ( clk    : in std_logic ;
3          input  : in std_logic_vector (0 to 0);
4          output : out std_logic_vector (0 to 0));
5 end rwcomp4;
6 architecture behavioral of start is
7   signal opaque_dev_in  : std_logic_vector(0 to 0);
8   signal opaque_dev_out : std_logic_vector(0 to 0);
9   function and(arg1 : std_logic_vector, arg2 : std_logic_vector)
     returns std_logic_vector;
10  ...
11  function not(arg1 : std_logic_vector) returns std_logic_vector;
12  ...
13 begin
14  opaque_dev_in  <= and(opaque_dev_out,input);
15  output         <= not(opaque_dev_out);
16
17  dev0 : entity work.opaque_dev(behavioral)
18    port map (clk,input,output)
19 end behavioral;
```

Listing 3.11: VHDL code for the example refold device.

The VHDL generated from `start` in Listing 3.10 is given in the entity in Listing 3.11. In the compiled code, `start` entity wraps the abstract `opaque_dev` device. Input from

the outer device is mapped to the inner device by way of the `opaque_dev_in` signal where the `and` function performs the input mapping specified by the second function argument to `refold`. The output of `opaque_dev` is mapped to the `opaque_dev_out` signal. The output of the whole `start` entity is mapped from the output signal from the interior device, but is first manipulated by the `not` function. We note that the compiled function definitions in the `architecture` of the device are omitted.

3.3.3 Compiling Non-primitives

We note that the non-primitive functions in ReWire are comprised of primitives and as such are not normal synthesizeable functions. We discharge non-primitive functions when they are observed in the parsing of an expression by performing appropriate syntax-level transofrmations of ReWire programs. In the case of `pipeline`, we in-line the definition of pipeline where it appears and perform a beta-reduction.

Chapter 4

Modularity Principles and a Module System

Connect Logic and modular programming are integral concepts in ReWire. We promote modular programming by promoting the reuse of synchronous components and we enable the reuse of synchronous components with Connect Logic. This chapter describes how modular programming in ReWire follows from the implementation of Connect Logic and the module system that is enabled through this addition.

4.1 Modularity with ReWire

ReWire is a subset of Haskell, but it is also a hardware description language (HDL). When designing hardware in ReWire it is important to give consideration to the primary ReWire artifacts, how they translate to artifacts in hardware, and the implications for modularity and reuse in the ReWire language.

4.1.1 Functions

Pure functions in ReWire are where most computation occurs. Work performed by a specification should be placed in a pure function. Pure functions are akin to unclocked combinational logic in a circuit. A pure function of the type $(i -> o)$ can be considered a "black box" that operates on inputs of type i and gives outputs of type o. Functions of a higher arity than one can be considered akin to entities with multiple input ports, but these functions are isomorphic to a unary function by way of uncurrying.

4.1.2 Reactive Resumptions

Reactive Resumptions are the constructions we use to express synchronicity in ReWire. Punctions express what work to be performed, Reactive Resumptions express *when* to perform it. A Reactive Resumption expresses a state machine similar in function to a Moore machine. Designers interface with these structures in ReWire by way of monadic binding ($>>=$ operator) and the non-proper morphism *signal* which allows a device to yield output and resume on the next input after the call to signal. ReWire restricts the developer to these two methods of encoding synchronous logic. Approaches to managing Reactive Resumptions other than the two mentioned are disallowed. Reactive Resumptions are not allowed to be nested or combined in any way other than traditional monadic binding. No inspection of the underlying structure of ReT is allowed (for example in Haskell, the underlying structure is actually an *Either* type).

If there are two devices specified as Reactive Resumptions in ReWire, a programmer can combine them by hand with monadic binding (like do-notation in Haskell)

by sequencing one to take place after the other with some control over how this sequencing occurs (conditionally or otherwise). This is not unlike concatenating two state machines together. We can define transitions into and out of existing device state machines in a number of ways, but the issue remains that we cannot execute two devices *in parallel* without completely refactoring existing specifications. For modular programming and specification, this is not sufficient. If we need inspect the internals of a device in order to integrate it into a design, this severely diminishes the potential for its modularity. In other words, without another way to combine synchronous devices, modularity in ReWire is hampered and a proper module system will not promote code reuse.

4.1.3 Modularity and Composability Follow from Connect Logic

Connect Logic was added to ReWire in order to fully realize modular programming for hardware while maintaining feasible synthesizablility of specifications using Connect Logic primitives. With Connect Logic, we can interface synchronous devices specified in Reactive Resumption form without having to refactor them. We can place devices in parallel, execute them in step with one another, and route input and output values among the devices. With these functions fully supported in the compiler as they are now, we can pave the way to modular programming in ReWire with reuse of commonly used synchronous components as well as combinational functions in a module system.

ReWire gives us two notions of modularity: *synchronous* modularity and *combinational* modularity. The Connect Logic extension to ReWire enables the developer to combine these different types of components. We consider the base form of modu-

	Combinational	Synchronous
Combinational	`left . right`	`refold id (const right) left`
Synchronous	`refold right (flip const) left`	`left ‘pipeline‘ right`

Table 4.1: Composing synchronous and combinational logic in ReWire. Output from the left is fed to the right.

larity to be pure functions (combinational) and Reactive Resumptions (synchronous). Table 4.1 demonstrates how we can use Connect Logic and some traditional Haskell functions (*const* , etc.) to combine synchronous and combinational modules with one another.

This notion of modularity gives hardware developers using ReWire a principled approach to modeling synchronous and combinational systems, separating these concerns, and maximizing reuse through this approach.

4.2 A Module System for ReWire

We equipped the ReWire language and compiler with support for single compilation using the same style of imports and exports seen in Haskell modules. We describe the approach in this section.

4.2.1 The ReWire Module System

The ReWire module system is a single compilation system that functions as a subset of the Haskell namespace and module system. Module imports are handled at compile time by merging modules together by fully qualifying their names.

```
1  module Module1 where
2  data Foo = A | B
3  funA :: Foo -> Foo
4  funA x = ...
5  --A module using Module1
6  module Main where
7  import Module1
8  funB :: Foo -> Foo
9  funB x = .. funA ..
10 --Main module, fully merged and qualified
11 module Main where
12 data Module1.Foo = Module1.A | Module1.B
13 Module1.funA :: Module1.Foo -> Module1.Foo
14 Module1.funA = ...
15 funB :: Module1.Foo -> Module1.Foo
16 funB x = ... Module1.funA ...
```

Listing 4.1: ReWire single compilation transformation example

We provide the same tools as Haskell for qualifying and renaming names. The ReWire module system currently assumes a single working directory structure for imported source files, but a package system similar to the Haskell Cabal package system could be integrated as a future work. Since we restrict ourselves to single compilation, importing modules is equivalent to a source transformation on ReWire files. We illustrate this source transformation in Listing 4.1.

4.2.2 From Separate Compilation and Future Work on Module Systems

With synchronous device reuse fully realized, we added module support to the ReWire compiler. When this project was undertaken, the initial approach considered a fully separate compilation system with support for previously compiled ReWire components. We determined that this approach was impractical and unnecessary. If a separate compilation module system from ReWire to VHDL were to be complete, it would need to have support for separately-compiled polymorphic functions and Reactive Resumptions. ReWire does not support compilation of polymorphic structures. If something is to be compiled to VHDL, it must be monomorphic, or without any quantified types (type variables). That isn't to say that polymorphic functions aren't without their applications in ReWire programming, however. Functions such as *snd* and *fst* are commonly used, and are usually monomorphized by hand by the designer typically by inlining them.

What does this mean for modularity of polymorphic functions in ReWire? We considered VHDL support for polymorphism with VHDL generic entities. These features allow one to specify entities in VHDL that are parameterized over values specifying attributes of the entity. One potential use of generics in entity specifications is for the developer to allow flexibility in the size of the inputs, outputs, and internal storage and wiring of a device. ReWire types are all assumed to be "grounded" or "bitty" types that can be represented and encoded in a bitwise form. Even quantified types (types with type variables), will have an implied "bittyness" to them that isn't fully realized until the types are evaluted to a monomorphic form. We can use this grounded nature of all ReWire types as a potential avenue for compiling polymorphism, but it comes

at a cost. For polymorphic ReWire functions to be represented in terms of VHDL entities with generic arguments, a few requirements need to be met. We must require a way to quantify the size of arguments in polymorphic functions and how this relates to the computation performed by the polymorphic value of the function. We believe that bit size is the only requirement for this to occur. When fully evaluating type level lambdas (or making type variables monomorphic), we must have a way to relay the size of the types to the VHDL entity representing the polymorphic function.

Given that this process works, the VHDL synthesis tools are left to do the work of monomorphizing the device specifications that we opted out of. Generic devices can't be synthesized to hardware. The synthesis tools must discharge all generics with actual values before it can generate an actual design. By adding this feature we will have effectively "kicked the can down the road" to tools that are largely black boxes. Given that the ReWire compiler emphasizes verification in design and implementation and given that this one feature would require a significant increase in compiler complexity for separate compilation where the benefits are somewhat nebulous, we opted to discontinue working towards a separate compilation system and focus instead on a single compilation system while offering an equivalent alternative to support polymorphic functions in a compilation pipeline for ReWire.

We can support polymorphic functions by partially evaluating them with respect to their types, but not their terms. We describe this process as follows. For this example we consider ReWire expressions in their System F form with regards to their types where type variables are bound by a type-level (big) lambda. ReWire functions are valid for compilation if their types contain no big lambdas. We propose a method for monomorphizing polymorphic pure functions in ReWire:

1. Detect all polymorphic functions.

2. In-line (expand) all polymorphic functions.

3. Discharge (purge) all polymorphic function definitions.

4. β-reduce all big-lambdas of the expanded lambda terms from the expanded polymorphic functions

5. Lambda-lift all of the lambda terms from expanded polymorphic functions.

6. Merge all identical lambda-lifted functions to single definitions. Rename replaced definitions accordingly.

The steps enumerated above provide us an analogous replacement to the process that the VHDL tools use in elaboration and synthesis to instantiate generic entities while maximizing sharing of functions. An emphasis is placed on only reducing types and leaving lambda terms as they are without reducing them. This approach allows us to maximize sharing or reduce the amount of redundant work and promote work reuse. The ReWire compiler already contains functionality to perform a number of these functions. We surmise that adding lambda lifting functions and type-reducing functions would be additional labor, but are not novel works and would likely not incur significant work on a compiler engineer.

Chapter 5

Visual Programming in ReWire

This chapter introduces a visual programming environment based largely on Connect Logic as a system for composing devices written in ReWire. We refer to the visual programming programming environment as the ReWire Visual Tool or RVT. Visual Programming in RVT is similar to drawing a block diagram of a hardware system in tool such as Vivado. The user draws wires between connection points on shapes representing synchronous devices and pure logic. We design RVT with two shape types to represent simple devices and functions with constrained inputs and outputs; devices are restricted to one connection point for input and one connection point for output while functions are allowed many input connection points but are constrained to one output connection point. We demonstrate a method to compile a DSL representation of this box diagram to a ReWire implementation.

5.1 Motivation

The motivation for developing the RVT is two-fold. Firstly, we wished to demonstrate the ease of which one could compile a visual representation of a hardware device specification using ReWire. By carefully selecting our programming model and by utilizing ReWire as the back end target for compilation, we are able to recreate a high level system-specification tool usually seen as part of very mature and complex system design suites (such as Vivado by Xilinx). This replication of design demonstrates the advantages that the ReWire back end brings to the design of system tools. Specifically, that a performant, high-level intermediate representation used as a compiler target allows us to build even higher level system tools with a relatively small effort. Secondly we wished to develop the visual tool as a layer on top of a specialized DSL for combining ReWire devices, or Reactive Resumptions, along with pure functions. The development and implementation of the RVT follows the design process used in the implementation of the RexHacc compiler in Chapter 7. The design tool emits a machine specification in its DSL, which is then compiled to ReWire and then to VHDL by way of the ReWire compiler.

5.2 Specification and Features

We begin the development of this tool by specifying its basic features. First, we wish to be able to compose a system of opaque synchronous devices (essentially something specified as a reactive resumption). This is as simple as a box diagram of devices strung together with lines representing wires. A question that arises immediately from this is "how many connection points should we allow for device shapes?" Reactive

resumptions take a single input "argument" and yield a single output "result" when they pause, but those single argument and results, in many cases, are values in some product type that are decomposed by the device. We illustrate an example of this in Listing 5.1.

```
1 type Input  = (Bit, Bit)
2 type Output = (Bit, Bit)
3
4 device :: ReT Input Output Id ()
5 device (left, right) = signal (right, left) >>= device
6
7 start = device (Zero, Zero)
```

Listing 5.1: An Example ReWire device with I/O of product types.

At first glance, an obvious way to allow devices to have many I/O connection points would be to inspect the type of the device and, if it were a tuple type, generate n-many connection points on a rendered device representation that matches the n-tuple of the input or output type. In other words, if an input type were a triple, provide three connection points. The problem with this approach is that it doesn't scale. Haskell does not have one singular product type. Any algebraic data type (ADT) in Haskell with a type constructor that has more than one type argument is a product type. For this reason, we opt to restrict the I/O connection points to a basic device representation to one.

The second feature we wish to establish in our base specification is the ability to manipulate data "in-between" our devices. While at first it seems that one could simply chain devices together to get a desired functionality, but it should be noted that in our model, devices perform their actions synchronously and in parallel with

one another. Intuitively, this means that each device takes an input on one cycle and yields it at the beginning of the next cycle. To chain devices together would require us to institute a pipeline delay where we might not want one. Manipulating data outside of this synchronous, cycle-oriented, or input-oriented model gives us more flexibility to manipulate data without thinking about how it may impact our pipeline with respect to its actual staging. For this we can create a second presentation format that is a pure function box. Given that this visual representation derives its properties from an actual pure function, we allow it many inputs (as arguments) but limit it to one output for the same reasons we limit synchronous devices.

For our base design and implementation of the Visual tool, we assume that all functions and devices available to the user to be included in a visual design are abstract. The user designs a greater implementation by composing specifications from what are abstract (i.e. interals are opaque) functions and devices. An important additional feature will be to allow the user to compose a device or function that they can use later, but can also inspect and manipulate. We defer this functionality to a later iteration of RVT. Lastly, we specify that the composition of devices and pure functions results in a device itself. This allows a user to produce components that can be reused in the environment. To this end we provide an input and output port that devices can connect to. These are given by `Input` and `Output` in Listing 5.2. Counterintuitively, in our model `Input` is an output port and `Output` is an input port.

Given the design specifications laid out in the previous section, we proceed with specifying the data types that we will use to model visual connections in the RVT.

```
1 type NodeRef    =   Text
2 data NodeId     =   Id Text | Input | Output
3 data Node       =   Device  NodeId NodeRef
```

```
4                       | PureFun NodeId NodeRef Int
5 data IAnchor    =   IAnchor NodeId AnchorName
6 data OAnchor    =   OAnchor NodeId
7 type Link       =   (OAnchor,IAnchor)
8 data RVT = RVT {
9                                 nodes :: [Node]
10                               , links :: [Link]
11                              }
```

Listing 5.2: The data types for the first iteration of RVT.

In Listing 5.2 we illustrate the types and data structures used to model RVT. At the top level, a program in `RVT` is comprised of lists of `Node` and `Link`. A `Node` data type models functions and devices in a visual specification. The anchor types (`IAnchor` and `OAnchor` correspond to input anchors and output anchors, or points where a user can connect devices by wires. All types in `Node` contain a `NodeRef` which refers to the name of the device or function as well as a `NodeId` which is a unique identifier for a `Node`. We note that a `NodeId` can appear as an identifier, or the constructors `Input` or `Output`. The `Input` and `Output` constructors are used only as anchor points that connect to the input and output of the whole device being specified. In our model, the user is designing the interior of what becomes a whole synchronous device that takes input and yields output. A correctly specified visual program is a device where all components are fully connected: all inputs have input connections from some output port, and all output ports are connected to some input port. Any number of lines can originate from the input port denoted by `Input` while only one line is allowed to terminate at the output port denoted by `Output`.

5.3 Tool Implementation

The implementation of RVT is split into two components: front end and back end. The front end of RVT is the visual component for user interaction. The back end consists of a small web service with a small compiler that converts front end graph representations to ReWire.

5.3.1 Front End

We implement the front as a web service because of the availability of visual diagramming tools in JavaScript as well as its platform independence for end use. There are visual programming libraries and tools written for Haskell, but they lack the platform independence and maturity of their web-based counterparts. The diagramming tool is based on the JointJS JavaScript library for drawing diagrams with connections. The library includes facilities for serializing and deserializing diagrammatic graph representations we use to JSON formats as well as aesthetic features such as routing libraries for user-drawn connections between components in a graph.

5.3.2 Back End

The back end of RVT is implemented in Haskell. We utilize the Scotty web framework for Haskell as our web service to interface with the RVT front end. The back end serves the tool software to the user and handlers user requests to compile a given visual implementation. Visual implementations are submitted to the server as serialized JSON objects, that are parsed using the Aeson library and transformed to the data types illustrated in Listng 5.2. Once transformed to these types, we conmpile the `RVT`

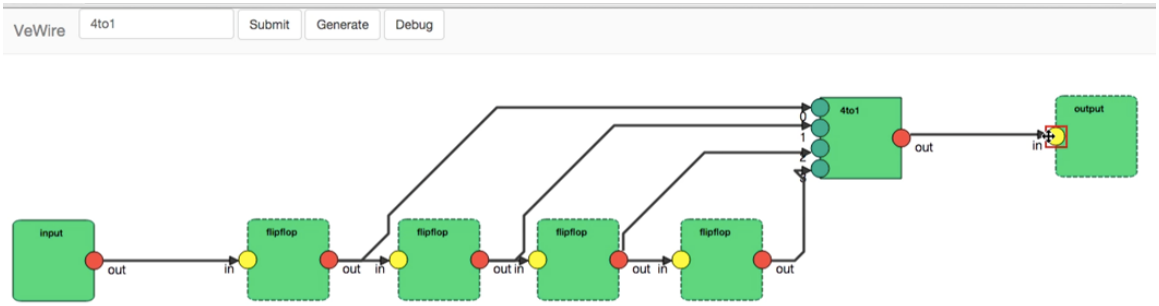


Figure 5.1: Diagramming devices using RVT in a web browser.

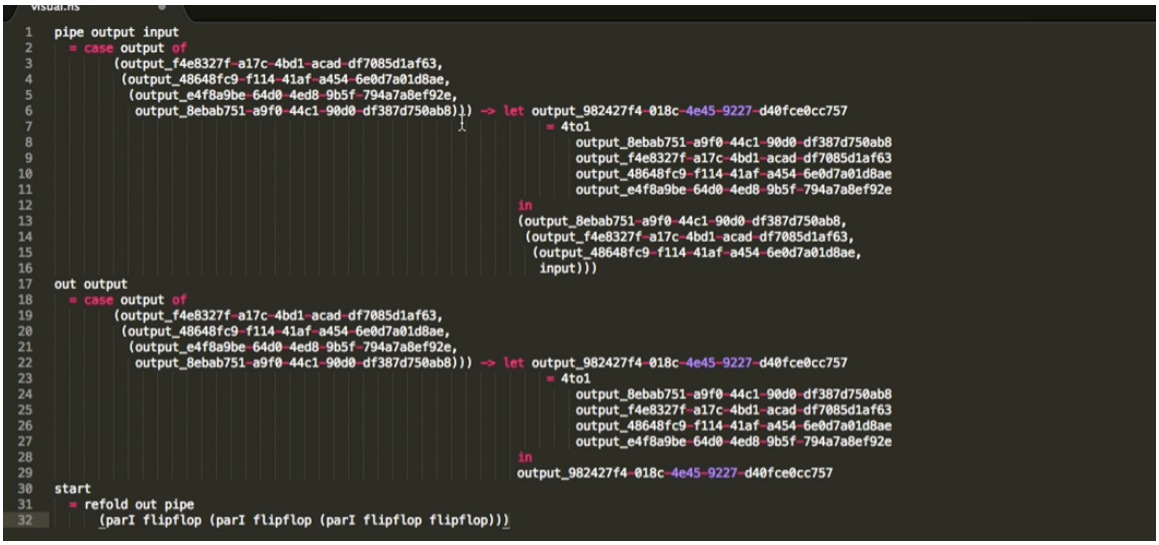
to ReWire code and return the compiled code to the user.

5.4 Using RVT

RVT is utilized as a drawing tool from a web browser. We utilize it as a web service. The tool is hosted by a server and editing is executed on the client. An example diagram from the tool is illustrated in Figure 5.1 where we use the tool to implement a 4-bit deserializer. Functions are represented by squared boxes while synchronous logic components are represented by boxes with rounded edges.

5.5 Code Generation

The service component of the RVT generates code using the graph drawing from the client component to construct a corresponding Connect Logic expression that implements the graph. All synchronous components are placed in parallel execution using the `<&>` operator. Combinational pure functions are applied to arguments in



```
visual.hs
1  pipe output input
2    = case output of
3          (output_f4e8327f-a17c-4bd1-acad-df7085d1af63,
4           (output_48648fc9-f114-41af-a454-6e0d7a01d8ae,
5            (output_e4f8a9be-64d0-4ed8-9b5f-794a7a8ef92e,
6             output_8ebab751-a9f0-44c1-90d0-df387d750ab8))) -> let output_982427f4-018c-4e45-9227-d40fce0cc757
7                                                                   = 4to1
8                                                                       output_8ebab751-a9f0-44c1-90d0-df387d750ab8
9                                                                       output_f4e8327f-a17c-4bd1-acad-df7085d1af63
10                                                                      output_48648fc9-f114-41af-a454-6e0d7a01d8ae
11                                                                      output_e4f8a9be-64d0-4ed8-9b5f-794a7a8ef92e
12                                                              in
13                                                              (output_8ebab751-a9f0-44c1-90d0-df387d750ab8,
14                                                               (output_f4e8327f-a17c-4bd1-acad-df7085d1af63,
15                                                                (output_48648fc9-f114-41af-a454-6e0d7a01d8ae,
16                                                                 input)))
17 out output
18   = case output of
19         (output_f4e8327f-a17c-4bd1-acad-df7085d1af63,
20          (output_48648fc9-f114-41af-a454-6e0d7a01d8ae,
21           (output_e4f8a9be-64d0-4ed8-9b5f-794a7a8ef92e,
22            output_8ebab751-a9f0-44c1-90d0-df387d750ab8))) -> let output_982427f4-018c-4e45-9227-d40fce0cc757
23                                                                  = 4to1
24                                                                      output_8ebab751-a9f0-44c1-90d0-df387d750ab8
25                                                                      output_f4e8327f-a17c-4bd1-acad-df7085d1af63
26                                                                      output_48648fc9-f114-41af-a454-6e0d7a01d8ae
27                                                                      output_e4f8a9be-64d0-4ed8-9b5f-794a7a8ef92e
28                                                              in
29                                                              output_982427f4-018c-4e45-9227-d40fce0cc757
30 start
31   = refold out pipe
32     (parI flipflop (parI flipflop (parI flipflop flipflop)))
```

Figure 5.2: Generated code from an RVT specification.

the same phase as routing data flow between synchronous devices placed in parallel. For this we use the `refold` operator. We construct a routing function by observing the source of each input at each component. Each device input is constructed by a let expression that makes use of the pure functions that may manipulate the input before it arrives at the component. This is an automation of the process that we use to construct the DLX processor in Chapter 9. The tool outputs textual representations of the expression. We illustrate this output code in Figure 5.2.

RVT is a proof-of-concept tool. The code it generates isn't typechecked, but is subject to type checking before synthesis. Early test examples were synthesized to hardware and produced good implementations. Further iterations of the tool could see more integration with the compiler as well as better support for saved designs and visual testing and debugging. We leave these features for future work on a more robust and integrated tool.

Chapter 6

Concurrent Devices in ReWire and Connect Logic

In this chapter we illustrate the applicability of Connect Logic to a variety of use-cases involving inter-device communication. The applications in this chapter demonstrate the usefulness of Connect Logic in ReWire as a method for implementing concurrent systems in hardware in addition to parallel systems. Parallel programming in ReWire is accomplished by utilizing combinational pure functions. Execution of parallel functions in ReWire is deterministic in nature. Concurrent programming in ReWire is not necessarily deterministic. We can execute synchronous logic in `ReT` concurrently where threads have internal state and may complete work at different points in time. Concurrent programming employs a number of useful idioms to manage shared resources between concurrently-running devices. We illustrate some of these idioms in this chapter in addition to a redundancy transformation on synchronous logic in ReWire.

6.1 Barrier Synchronization

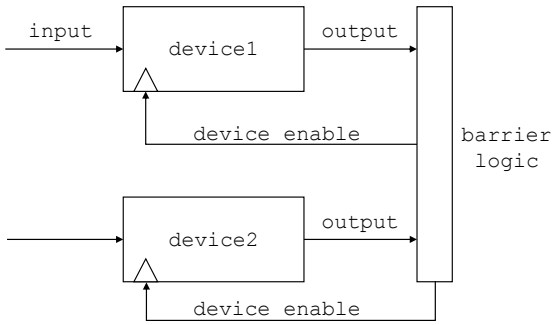


Figure 6.1: Hardware threads in ReWire in a system with barrier synchronization.

In software, one of the most commonly seen concurrent synchronization idioms is the barrier. Barriers act as a synchronization point between concurrent threads [64]. 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. This concept is illustrated in Figure 6.1. Barriers are employed in the execution of parallel loops seen in OpenMP [65] and Pthreads (POSIX threads) library [66]. The principle of barrier-halted execution also applies to devices in hardware. Different components in a hardware device may need to synchronize

before continuing execution just like their threaded software counterparts.

Connect Logic 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.

```
data Stall a = Stall | Continue a
data Busy  a = Busy  | Complete a
refoldT :: Monad m => (o1 -> o2) ->
                      (o1 -> i2 -> Stall i1) ->
                      ReT i1 o1 m a ->
                      ReT i2 o2 m a
makeStaller :: Monad m => ReT i o m a -> ReT (Stall i) o m a
makeStaller dev = refoldT (\o -> o)
                          (\_ -> \mi -> mi) dev
```

Listing 6.1: A transformation to make any device in ReT a stalling device using the `refoldT` primitive. Here we use the isomorphic types `Stall` and `Busy` in place of a `Maybe` type.

Before constructing a barrier transformation, we need a method by which to make an arbitrary device a *stalling device*. Listing 6.1 defines the function `makeStaller` 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 `makeStaller` 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`.

```
barrier :: ReT i1 (Busy o1) I a       ->
           ReT i2 (Busy o2) I a       ->
           ReT (i1,i2) (Busy (o1,o2)) I a
```

```
4  barrier d1 d2 = let dp = (makeStaller d1) <&> (makeStaller d2)
5                  in refold out inp dp
6
7  inp :: (Busy o1, Busy o2) -> (i1,i2) -> (Stall i1,Stall i2)
8  inp o (i1,i2) = case o of
9                    --If neither device has produced output,
10                   --keep allowing input to both
11                   (Busy,Busy)              -> (Continue i1,Continue
       i2)
12                   --If the left device has produced, stall it
13                   (Complete l, Busy)       -> (Stall, Continue i2)
14                   --If the right device has produced, stall them
15                   (Busy, Complete r)       -> (Continue i1,Stall)
16                   --If both devices have produced,
17                   --let them both continue
18                   (Complete l, Complete r) -> (Continue i1,Continue
       i2)
19
20 out :: (Busy o1, Busy o2) -> Busy (o1, o2)
21 out o = case o of
22              (Busy,_) -> Busy
23              (_,Busy) -> Busy
24              (Complete a,Complete b) -> Complete (a,b)
```

Listing 6.2: Creating a barrier in ReWire for devices typed in `ReT`

In Listing 6.2 we define the barrier-constructing function that acts on two different devices to produce a single device. The `barrier` function defined on lines 1-5 takes two devices that yield output in the type of `Busy o`. It 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)` where 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 Connect Logic 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.

The barrier device demonstrates a critical application of `refoldT`: managed execution of ReWire components. While this example exhibits a basic usage of the Connect Logic combinator, it enables a very commonly used concurrency idiom found in software.

6.2 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. These effects have been observed to be common in high altitude or outer space environments where system failures can lead to disastrous results [67]. 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, was initially conceived by

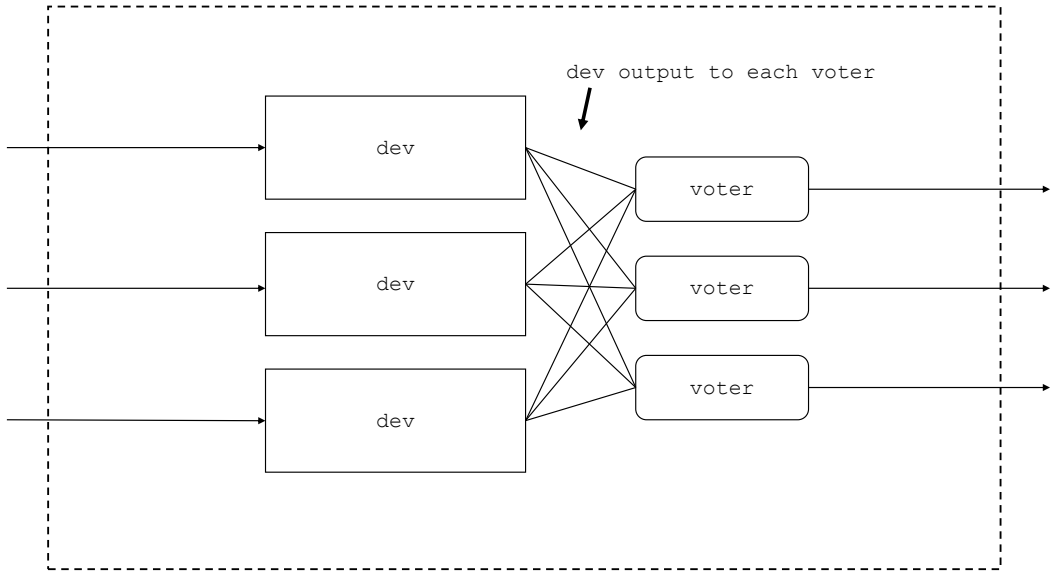


Figure 6.2: A Functional Triple Modular-redundant system constructed using the `ftmr` transformation.

von Neumann [68] and formalized by Lyons [1]. In TMR regimes, the core logic replicated three times and executes over the same input, yielding the same output in ideal conditions. The output of each redundant component is given to a voting process that determines the correct output by the mode of its input. In the event of an SEU corrupting one of the redundant components, the other two components will yield correct output to carry forward to the voting process and the correct result prevails.

We can develop transformations on devices in ReWire using Connect Logic to enable TMR in a transparent and scalable way. We demonstrate a simple `tmr` transformation in Listing 6.3. In this example we rely on the `Eq` typeclass to provide us with a definition of Eq for polymorphic types, but these could be easily instantiated in ReWire to monomorphic forms. Connect Logic makes this transformation on devices straightforward. We construct a voting-transformation that uses a majority-wins fil-

tering approach on the output of three different devices. The `tmr` transformation replicates a given device three times using the `voter` transformation.

```
vote :: Eq a => ((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 :: Eq o => ReT i o I a ->
                 ReT i o I a ->
                 ReT i o I a ->
                 ReT i o I a
voter d1 d2 d3 = refold vote fan ((d1 <&> d2) <&> d3)

tmr :: Eq o => ReT i o I a -> ReT i o I a
tmr dev = voter dev dev dev
```

Listing 6.3: Simple Triple Modular Redundancy with Connect Logic

There exist more sophisticated (and better) redundancy regimes. The simple TMR regime described by Listing 6.3 has an obvious flaw: the voting logic is not redundant. If an SEU were to affect the voting logic, then the correct results of the components would be for naught!

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

```
4                   v3 = vote a
5                in ((v1,v2),v3)
6
7  ftmr :: Eq o => ReT i o I a -> ReT ((i,i),i) ((o,o),o) I a
8  ftmr dev = refold
9                   voteRed
10                  (\_ i -> i)
11                  ((dev <&> dev) <&> dev)
12
13 pipeline_ftmr :: (Eq z, Eq o) => ReT i z I a ->
14                                    ReT z o I a ->
15                                    ReT ((i,i),i) ((o,o),o) I a
16 pipeline_ftmr left right = refold
17                                  snd
18                                  (\(left_out,right_out) inp ->
19                                        (inp,left_out)
20                                  )
21                                  ((ftmr left) <&> (ftmr right))
```

Listing 6.4: Functional TMR [1] with redundant voting logic in Connect Logic.

In Listing 6.4 we modify our previous transformation from using the `voter` transformation to one based on the notion of *Functional Triple Modular Redundancy* (FTMR). This transformation on a given device `dev` is illustrated in Figure 6.2. In the `ftmr` transformation we transform a device by replicating it three times and routing unique inputs to each device. We compute the output by way of three different redundant voting components (pure logic, so this is indicated by the application of the pure function `vote`) in `voteRed`. Unlike our previous definition, the `ftmr` transformation changes the input and output types. This is a necessity because if we were to use logic

to "merge" the outputs, this logic create a single point of failure. We also require inputs to be redundant as well as outputs.

The usefulness of this choice is exhibited in the pipelining function in `pipeline_ftmr`. This is an alternative to the simple `pipeline` idiom discussed in Chapter 3 which sequences devices together by their inputs and outputs that transforms both devices into an FTMR form. Similar to `pipeline` a designer could take any number of components to sequence and transform them into having redundant sequentiality. The result being a single device taking redundant inputs and yielding redundant outputs. A designer could then choose how to "merge" the outputs and "fan out" inputs in a way consistent with their redundancy regime. Additionally as an alternative, we can apply `ftmr` to any number of devices and then combine them by using the canonical `pipeline` transformation.

6.3 Mutual Exclusion

Mutual exclusion locks are a useful primitive for guaranteeing that only one thread of execution has access to a critical section. A design using a mutex component with two concurrent devices is illustrated in Figure 6.3. We implement a non-blocking mutex for two hardware threads as a device with three states: `unlocked`, `left-locked`, and `right-locked` with each state corresponding to which argument, if any, has the mutex lock. In this implementation, the mutex encapsulates the critical section, which is a value (`val`) of type `a`.

```
1 data Req a = ReqLock | Release  | Write a  | NullReq
2 data Rsp = LockGrant | Ack | NullResp
3
```

Figure 6.3: Constructing a system that utilizes a mutex for protecting a value of type a. The mutex is a distinct device in this design and communicates with concurrent devices with synchronous connections via Connect Logic primitives.

```
4  unlocked :: (Req val,Req val) ->
5              val                ->
6              ReT (Req val, Req val) (val,(Rsp,Rsp)) I ()
7  unlocked reqs val = case reqs of
8                (ReqLock,_) -> do
9                   i <- signal (val,(LockGrant,NullResp))
10                  leftLocked i val
11               (_,ReqLock) -> do
12                  i <- signal (val,(NullResp,LockGrant))
13                  rightLocked i val
14               _           -> do
15                  i <- signal (val,(NullResp,NullResp))
16                  unlocked i val
```

```
17
18 leftLocked :: (Req val,Req val) ->
19                 val                 ->
20                 ReT (Req val, Req val) (val,(Rsp,Rsp)) I ()
21 leftLocked reqs val = case reqs of
22                   (Write v,_) -> do
23                     i <- signal (v,(Ack,NullResp))
24                     leftLocked i v
25                   (Release,_) -> do
26                     i <- signal (val,(Ack,NullResp))
27                     unlocked i val
28                   (ReqLock,_) -> do
29                     i <- signal (val,(LockGrant,NullResp))
30                     leftLocked i val
31                   _           -> do
32                     i <- signal (val,(LockGrant,NullResp))
33                     leftLocked i val
34
35 rightLocked :: (Req val,Req val) ->
36                 val                 ->
37                 ReT (Req val, Req val) (val,(Rsp,Rsp)) I ()
38 rightLocked reqs val = case reqs of
39                     (_,Write v) -> do
40                       i <- signal (v,(NullResp,Ack))
41                       rightLocked i v
42                     (_,Release) -> do
43                       i <- signal (val,(NullResp,Ack))
44                       unlocked i val
45                     (_,ReqLock) -> do
```

```
46                      i <- signal (val,(NullResp,LockGrant))
47                      rightLocked i val
48                  _           -> do
49                      i <- signal (val,(NullResp,LockGrant))
50                      rightLocked i val
```

Listing 6.5: A left-argument-biased mutex specification for two ReWire devices.

In Listing 6.6 we utilize the mutex previously defined in Listing 6.5. We place devices that make use of the mutex lock to execute in parallel with one another and we use `refold` to connect the devices together for requests and responses for the mutex lock.

The mutex device is *external* to the devices requesting use of the lock. Requests will be delayed by a clock cycle. At time t_0 if a device requests a lock, the mutex will process this request at time t_1. The response from the mutex will be received by the requesting device at the clock cycle t_2. Thus, the device needs to do work in the intermediate cycle t_1 or otherwise stall while a response is calculated by the mutex. This design choice modularizes our mutex, but comes at the cost of a lost cycle for devices waiting for a response. In certain situations this trade off could be acceptable, but there are alternative approaches that sacrifice modularity to regain the cost of the clock cycle. We demonstrate this approach in our semaphore implementation.

```
1 devLeft :: ReT Rsp (Req Int) I ()
2 devLeft = do
3             rsp <- signal ReqLock
4             case rsp of
5               LockGrant -> do
6                                rsp <- signal (Write 1)
7                                signal Release
8                                signal NullReq
```

```
9                                   signal NullReq
10                                  devLeft
11                  _           -> do
12                                  devLeft
13
14 devRight :: ReT Rsp (Req Int) I ()
15 devRight = do
16              rsp <- signal ReqLock
17              case rsp of
18                 LockGrant -> do
19                                  rsp <- signal (Write 2)
20                                  signal Release
21                                  signal NullReq
22                                  devRight
23                 _          -> do
24                                  devRight
25
26 mutex :: ReT (Req Int, Req Int) (Int,(Rsp,Rsp)) I ()
27 mutex = unlocked (NullReq,NullReq) 0
28
29 device :: ReT () Int I ()
30 device = refold (\(_,(_,(i,_))) -> i)
31                 (\(lreq, (rreq, (oval, (lrsp,rrsp)))) () ->
32                  (lrsp,(rrsp, (lreq,rreq))))
33                 (devLeft <&> (devRight <&> mutex))
```

Listing 6.6: Utilizing the semaphore as a device in a closed system

6.4 Semaphore Constructions

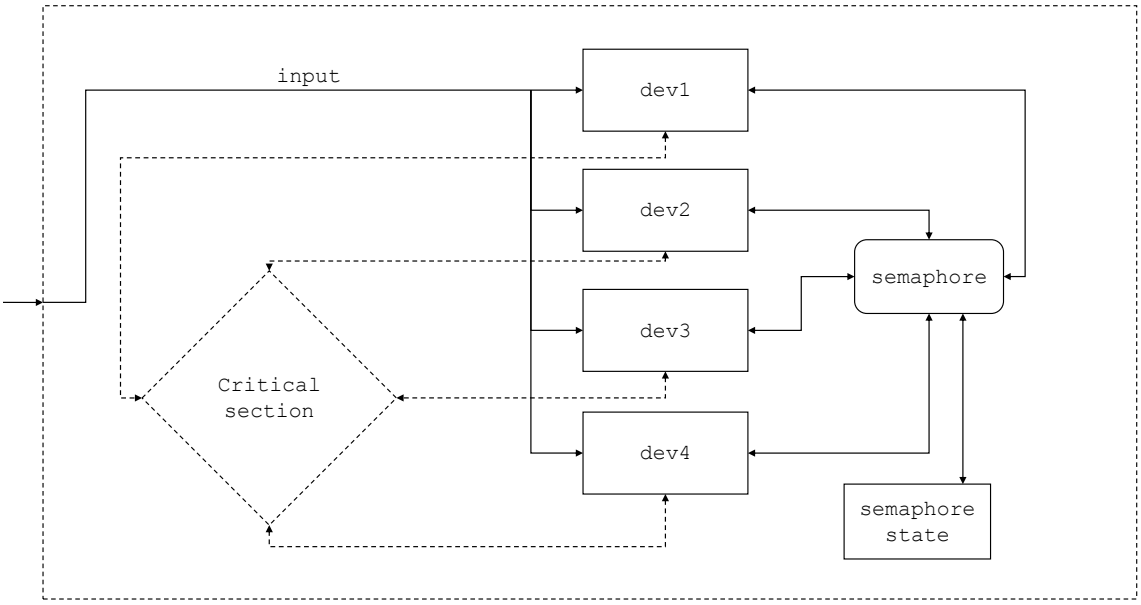


Figure 6.4: A semaphore construction in ReWire. Four concurrent devices request access to a critical section with a semaphore in pure logic. The semaphore maintains its own state in a separate device. The critical section is kept abstract in this design and is illustrated with dashed borders and connections.

Semaphores are more general mutexes. We can use semaphores to regulate resource sharing that allows more than one active process into a critical section. We implement a non-blocking semaphore in a K by N fashion where we have K concurrent processes requesting access to N different resources. In the subsequent code listings, we demonstrate the implementation of a 4 by 2 semaphore. The first implementation treats the semaphore as a separate device with a longer latency for requesting access to the critical section. The revised version of the semaphore, illustrated in Figure 6.4 combines semaphore logic with devices to eliminate request latency. This implementation is a non-blocking semaphore. Devices requesting a lock get an imme-

diate response, but may not successfully receive a lock and must act accordingly in response. We note that blocking implementations are also possible utilizing `refoldT` in a manner similar to the barrier implementation seen earlier in this chapter.

```
1  --Two slots to work with
2  data Count = Z | One | Two
3
4  --Connection Priority
5  data Priority = C0 | C1 | C2 | C3
6
7  --Protocol
8  data Req = NRq | P | V
9  data Rsp = NRp | Ack
10
11 inc :: Count -> Count
12 inc c = case c of
13              Z -> One
14              One -> Two
15              Two -> Two
16
17 dec :: Count -> Count
18 dec c = case c of
19              Z -> Z
20              One -> Z
21              Two -> One
22
23 adv :: Priority -> Priority
24 adv p = case p of
25              C0 -> C1
26              C1 -> C2
```

```
27              C2 -> C3
28              C3 -> C0
```

Listing 6.7: Types for a 2-semaphore device implementation.

We define our data types and helper functions in Listing 6.7. We define a `Count` data type for tracking how many slots are available in the critical section. The `Priority` data type specifies names for each concurrent process which we use for managing who has the highest priority when requesting the an available slot. The `Req` and `Rsp` data types are our request and response protocols. The functions we define are for managing the counter of available spots and iterating which process has the highest request priority in a round-robin fashion.

```
1  --Rotations for Round-Robin priority
2  rotate :: Priority -> (a,a,a,a) -> (a,a,a,a)
3  rotate p (l0,l1,l2,l3) = case p of
4                                C0 -> (l0,l1,l2,l3)
5                                C1 -> (l3,l0,l1,l2)
6                                C2 -> (l2,l3,l0,l1)
7                                C3 -> (l1,l2,l3,l0)
8
9  rotate' :: Priority -> (a,a,a,a) -> (a,a,a,a)
10 rotate' p (l0,l1,l2,l3) = case p of
11                               C0 -> (l0,l1,l2,l3)
12                               C1 -> (l1,l2,l3,l0)
13                               C2 -> (l2,l3,l0,l1)
14                               C3 -> (l3,l0,l1,l2)
15
16 lock :: Count -> (Req,Req,Req,Req) -> (Count,(Rsp,Rsp,Rsp,Rsp))
17 lock count reqs = case count of
```

```
18                          Two -> case reqs of
19                                      --Two Requests
20                                      (P,P,_,_) -> (Z,(Ack,Ack,NRp,NRp))
21                                      (P,_,P,_) -> (Z,(Ack,NRp,Ack,NRp))
22                                      (P,_,_,P) -> (Z,(Ack,NRp,NRp,Ack))
23                                      (_,P,P,_) -> (Z,(NRp,Ack,Ack,NRp))
24                                      (_,P,_,P) -> (Z,(NRp,Ack,NRp,Ack))
25                                      (_,_,P,P) -> (Z,(NRp,NRp,Ack,Ack))
26                                      --One Request
27                                      (P,_,_,_) -> (One,(Ack,NRp,NRp,NRp))
28                                      (_,P,_,_) -> (One,(NRp,Ack,NRp,NRp))
29                                      (_,_,P,_) -> (One,(NRp,NRp,Ack,NRp))
30                                      (_,_,_,P) -> (One,(NRp,NRp,NRp,Ack))
31                                      --No Requests
32                                      _         -> (count,(NRp,NRp,NRp,NRp))
33                          One -> case reqs of
34                                      (P,_,_,_) -> (Z,(Ack,NRp,NRp,NRp))
35                                      (_,P,_,_) -> (Z,(NRp,Ack,NRp,NRp))
36                                      (_,_,P,_) -> (Z,(NRp,NRp,Ack,NRp))
37                                      (_,_,_,P) -> (Z,(NRp,NRp,NRp,Ack))
38                                      --No Requests
39                                      _         -> (count,(NRp,NRp,NRp,NRp))
40                          Z    -> (Z,(NRp,NRp,NRp,NRp))
41
42  yield :: Count -> (Req,Req,Req,Req) -> Count
43  yield count reqs = let c1 = inc count
44                     in case reqs of
45                          --Two Requests
46                          (V,V,_,_) -> Two
```

```
47                                  (V,_,V,_) -> Two
48                                  (V,_,_,V) -> Two
49                                  (_,V,V,_) -> Two
50                                  (_,V,_,V) -> Two
51                                  (_,_,V,V) -> Two
52                                 --One Request
53                                  (V,_,_,_) -> c1
54                                  (_,V,_,_) -> c1
55                                  (_,_,V,_) -> c1
56                                  (_,_,_,V) -> c1
57                                 --No Requests
58                                  _         -> count
```

Listing 6.8: Pure functions for managing semaphore state and incoming requests

In Listing 6.8 we define pure functions for managing the locking and rotating priority of processes for requesting semaphore locks. We note that the `rotate` functions are inverse to one another. The lock function manages incoming lock requests from processes giving the leftmost argument the highest priority. We use the rotation functions to alternate which process sits in the left most positions of the 4-tuple and thus has the highest priority. The `yield` function adjusts the count based on how many devices yield their lock on the semaphore.

```
1 sem :: (Req,Req,Req,Req) -> Count -> Priority -> ReT (Req,Req,Req,Req)
      (Rsp,Rsp,Rsp,Rsp) I ()
2 sem reqs count pri = let reqsp  = rotate pri reqs --Re-arrange for
      priority
3                          countp = yield count reqs
4                      in case lock countp reqsp of
5                             (final_count,rsps) -> do
```

```
6                                                    inp <- signal
    (rotate' pri rsps)
7                                                    sem inp final_count
    (adv pri) --
```

Listing 6.9: The first semaphore device implementation. A stand-alone semaphore device.

Like the mutex before, we implement a discrete semaphore device. Each cycle the count is updated first, then the requests are rotated by priority and responses are computed by the `lock` function. The results are signaled and the next cycle resumes with an advanced priority and the next set of incoming requests. Similarly to the mutex implementation, using this device to manage other devices incurs a clock cycle delay from request to response.

```
1  dev0, dev1, dev2, dev3 :: ReT Rsp Req I ()
2
3  countDev :: ReT Count Count I ()
4  priorityDev :: ReT Priority Priority I ()
5
6
7  devs :: ReT (Rsp,Rsp,Rsp,Rsp) (Req,Req,Req,Req) I ()
8  devs = refold (\(d0,(d1,(d2,d3))) -> (d0,d1,d2,d3))
9                (\_ -> \(d0,d1,d2,d3) -> (d0,(d1,(d2,d3))))
10               (dev0 <&> (dev1 <&> (dev2 <&> dev3)))
11
12 system :: ReT () () I ()
13 system = refold (const ())
14                 (\out -> \_ -> case out of
15                    ((count,priority),reqs) ->
```

```
16                      let reqsp  = rotate priority reqs
17                          countp =  yield count reqs
18                       in case lock countp reqsp of
19                            (final_count,rsps) ->
20                             ((final_count,()),rotate' priority rsps)
21
22
23                  )
24                  ((countDev <&> priorityDev) <&> devs)
```

Listing 6.10: The second semaphore device implementation. A semaphore integrated in primarily pure logic refolded with its constituent devices.

As an alternative to discrete devices with cycle delays for communication, we introduce another implementation of the semaphore that integrates the functionality with its four processes using `refold` and parallel functions. In this example we introduce two additional devices `countDev` and `priorityDev` that save a value for a clock cycle. Where we saved values as function parameters in the previous example, we feed them forward into these respective devices here using Connect Logic. The result is a `refold` over the parallel devices that looks very similar to the definition of the `sem` function seen in Listing 6.9. The result is an integrated 4-device semaphore without the one clock cycle delay between request and response.

6.5 Segmentation

We utilize techniques exhibited in the previous examples to implement a policy-enforcing memory controller. The concept of this controller is illustrated in Figure 6.5.

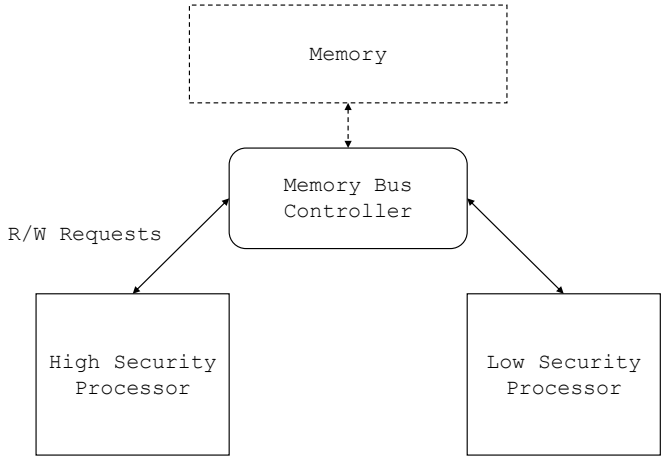


Figure 6.5: Constructing a segmented memory controller from a high and low security processor. Memory requests made by processors are screened by a smart bus controller.

In this scenario, we have two processors making concurrent requests to a single memory module with a single input bus. The processors exist in two different domains: a high domain that can read all of memory while only writing to its own region of memory, and a low domain that can only read and write in its own region.

We begin with a rough sketch of the requirements of such a segmentation device. First, requests made simultaneously by both processors need to be handled so that one request prevails, but no processor is starved. Second, the memory module must "proxy" valid requests to the memory unit itself. Responses from the memory unit return on some subsequent cycle. We assume that memory read responses return on the immediate subsequent cycle when generally in many settings they may return in one or more cycles. The is generally not the case in real world situations, but our

specification can be easily modified to support any type of read/write latency from a memory unit. We begin by defining types and basic devices in Listing 6.11.

6.5.1 Data Types

```
1  module Segmenter where
2
3  import Types
4  import Data.Word
5
6  type Address = Word32
7  type Data    = Word8
8
9  data MemAcc  = NoReq    | Read Address    | Write Address Data
10 data MemRsp  = NoRsp    | Success | Retry | ReadResult Data
11
12 data RspMask = NoRes
13                | Written    --Notifies memory written
14                | Busy       --Device busy, reattempt
15                | ReadRes    --Result of Read
16
17 data Priority = C0 | C1
18
19 adv :: Priority -> Priority
20 adv p = case p of
21           C0 -> C1
22           C1 -> C0
```

Listing 6.11: Types and helper functions for a memory segmenter

In this implementation we assume a 32-bit byte-addressable memory unit. Memory accesses are represented by `MemAcc` encoding whether or not there is a read (`Read`), a write (`Write`), or no memory access request (`NoReq`) is taking place. Responses to processors are defined as type constructors for `MemRsp` on Line 10. These indicate no action (`NoRsp`), write successes (`Success`), a signal to retry if the bus is busy (`Retry`), and the result of a read operation (`ReadResult`). Memory requests and responses occur on different clock cycles. This example makes the simplifying assumption that the result of the memory access is available on the clock cycle immediately following the clock cycle of the request. The `RspMask` type is an internal encoding for how to handle the response from the memory module. The "response mask" is fed to the response master device from the request master device (both detailed later). The response master device considers the response masks and routes the memory response in `MemRsp` to both requesting devices. We note the presence of a `Priority` type for managing which requesting device has top priority in the event of simultaneous requests. The priority is rotated after every conflict is resolved so the losing device will be first priority in a subsequent conflict.

6.5.2 Security Policy Functions

```
1 policyH :: MemAcc -> (MemAcc, RspMask)
2 policyH req = case req of
3                   --No Request
4                   r@(NoReq)            -> (r, NoRes)
5                   --A read request reads from any address
6                   r@(Read _)           -> (r, ReadRes)
7                   --A Write Request
8                   r@(Write addr _) -> if addr >= 0x7FFFFFFF
```

```
9                                          --Valid address range
10                                         then (r, Written)
11                                         --Illegal address range
12                                         else (NoReq,NoRes)
13
14 policyL :: MemAcc -> (MemAcc,RspMask)
15 policyL req = case req of
16                   --No Request
17                   r@(NoReq)             -> (r,NoRes)
18                   --Read Request
19                   r@(Read addr)         -> if addr < 0x7FFFFFFF
20                                            --Valid read range
21                                            then (r,ReadRes)
22                                            --invalid read
23                                            else (NoReq,NoRes)
24                   --Write Request
25                   r@(Write addr _) -> if addr < 0x7FFFFFFF
26                                            --Valid write
27                                            then (r,Written)
28                                            --Invalid write
29                                            else (NoReq,NoRes)
```

Listing 6.12: Policy functions for a memory bus master

In Listing 6.12 we define memory access policies as two functions. One policy is for the high security (`policyH`) domain 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 pro-

cessor 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.

6.5.3 The Request Master

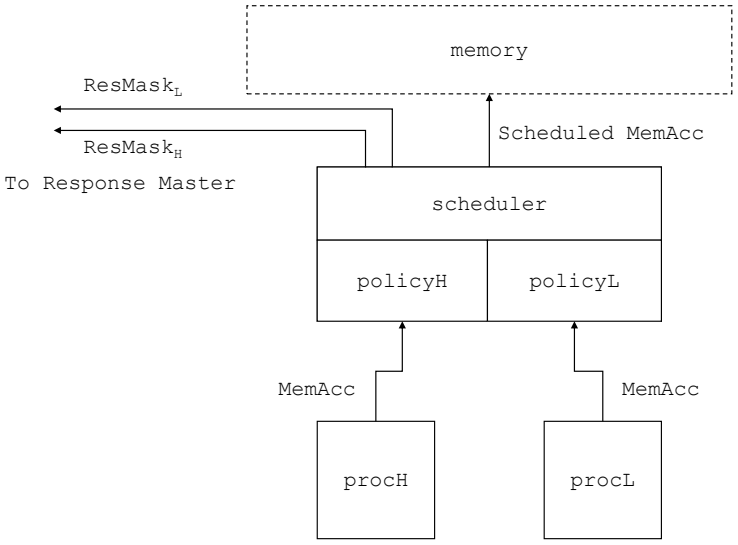


Figure 6.6: The request master component. Up to two requests are received in a cycle, are processed by the policy functions and then scheduled. A single memory request is sent to the memory module while response masks are sent to the response master component.

The request master component is one of two subcomponents that comprise the memory segmenting device. A block diagram of this device is illustrated in Figure 6.6. This component handles inbound memory requests from two processors. The requsts are checked by the policy functions and in the event of two valid requests, a winning function is selected by the scheduler. Response masks are sent to the response master subcomponent which is detailed later.

```
reqMaster_ :: Priority
              -> (MemAcc,MemAcc)
              -> ReT (MemAcc,MemAcc)
                     (MemAcc,(RspMask,RspMask))
                     I ()
reqMaster_ p reqs =
  case reqs of
    (NoReq,NoReq)     -> do
                         i <- signal (NoReq,(NoRes,NoRes))
                         reqMaster_ p i
    (req,NoReq)       -> do
                         let (acc,rsp) = policyH req
                         i <- signal (acc,(rsp,NoRes))
                         reqMaster_ p i
    (NoReq,req)       -> do
                         let (acc,rsp) = policyL req
                         i <- signal (acc,(NoRes,rsp))
                         reqMaster_ p i
    (high,low)        -> case p of
                         C0 -> do
                                 let (acc,rsp) = policyH high
                                 i <- signal (acc,(rsp,Busy))
                                 reqMaster_ (adv p) i
                         C1 -> do
                                 let (acc,rsp) = policyL low
                                 i <- signal (acc,(Busy,rsp))
                                 reqMaster_ (adv p) i
--Here we initialize the request master
reqMaster :: ReT (MemAcc,MemAcc) (MemAcc,(RspMask,RspMask)) I ()
```

```
30 reqMaster = reqMaster_ C0 (NoReq,NoReq)
```

Listing 6.13: Definitions for the request master function. The transition function is given by `reqMaster_` and the initialized device is given by `reqMaster`.

The code for the request master device is listed in Listing 6.13. The first three branches of the case statement (lines 3-13) handle cases where a single request or no request is made and the scheduling mechanism is not invoked. The final case handles a 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.

6.5.4 The Response Master

The response master is the second half of the memory segmenting component. The block diagram for this subcomponent is illustrated in Figure 6.7. When a memory request is made to the memory module, we send response masks to the response component 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.

```
1 rspMaster_ :: (Data,(RspMask,RspMask))
2               -> ReT (Data,(RspMask,RspMask))
3                      (MemRsp,MemRsp) I ()
4 rspMaster_ (dta,(hmask,lmask)) =
```

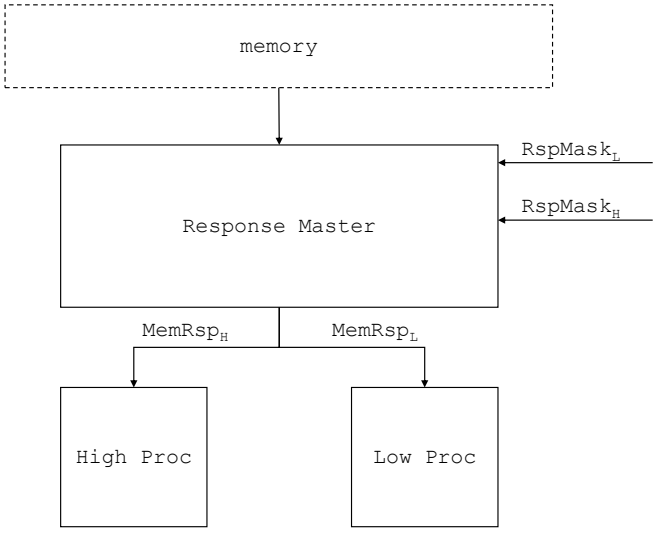


Figure 6.7: The response master component. The response master component computes the responses to send to both processors based on the output from the memory module unit and the requests masks in a given cycle.

```
5     case (hmask, lmask) of
6       (NoRes, NoRes) -> do
7                         r <- signal (NoRsp, NoRsp)
8                         rspMaster_ r
9       (ReadRes, NoRes) -> do
10                        r <- signal (ReadResult dta, NoRsp)
11                        rspMaster_ r
12      (ReadRes, Busy) -> do
13                        r <- signal (ReadResult dta, Retry)
14                        rspMaster_ r
15      (NoRes, ReadRes) -> do
16                        r <- signal (NoRsp, ReadResult dta)
17                        rspMaster_ r
```

```
18        (Busy,ReadRes) -> do
19                          r <- signal (Retry,ReadResult dta)
20                          rspMaster_ r
21        (Written,NoRes) -> do
22                          r <- signal (Success,NoRsp)
23                          rspMaster_ r
24        (Written,Busy) -> do
25                          r <- signal (Success,Retry)
26                          rspMaster_ r
27        (NoRes,Written) -> do
28                          r <- signal (NoRsp,Success)
29                          rspMaster_ r
30        (Busy,Written) -> do
31                          r <- signal (Retry,Success)
32                          rspMaster_ r
33        --All pathological cases result
34        --in no responses to both processors.
35        pathological   -> do
36                          r <- signal (NoRsp,NoRsp)
37                          rspMaster_ r
38
39 rspMaster :: ReT (Data,(RspMask,RspMask))
40                  (MemRsp,MemRsp) I ()
41 rspMaster = rspMaster_ (0,(NoRes,NoRes))
```

Listing 6.14: Definitions for the response master. The transition function is given by `rspMaster_` and the initialized device is given by `rspMaster`.

The code for the response master device is given in Listing 6.14. The device operates on an input tuple that includes data (typed `Data`) from the memory module and a pair

of response masks (typed `(RspMask,RspMask)`) given by the input type of the device on Line 2. The device outputs a pair of `MemRsp` responses to be fed to requesting processors. The case statement beginning on Line 5 scrutinizes a pair of response masks and acts on all valid pairs of them. A valid pair 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. All valid pairs are explicit branches in this case statement and all non-listed pairs are considered pathological cases. If a pathological case occurs then no response is sent to either processor.

6.5.5 Composing the Bus Master

The bus master of the memory segmenting component is composed from the request and response master subcomponents using the parallel and `refold` combinators. We illustrate this in the code given in Lising 6.15.

```
1  outputSelect :: ((MemAcc, (RspMask, RspMask)), (MemRsp, MemRsp))
2                  -> (MemAcc,(MemRsp,MemRsp))
3  outputSelect ((memacc,_),rsp) = (memacc,rsp)
4
5  inputSelect :: ((MemAcc, (RspMask, RspMask)), (MemRsp, MemRsp))
6                 -> (Data,(MemAcc,MemAcc))
7                 -> ((MemAcc, MemAcc), (Data, (RspMask, RspMask)))
8  inputSelect ((_,masks),_) (dta, accs) = (accs,(dta,masks))
9
10 busMaster :: ReT (Data,(MemAcc,MemAcc)) (MemAcc,(MemRsp,MemRsp)) I ()
11 busMaster = refold
12               outputSelect
```

```
13                inputSelect
14                (reqMaster <&> rspMaster)
```

Listing 6.15: The bus master is composed from the request and response master. We use routing logic in the functions `outputSelect` and `inputSelect` in a `refold` over the paralleized `reqMaster` and `reqMaster` devices.

The bus master is the top level definition of the memory segmenting device. It is given by `busMaster` in Listing 6.15. 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 `busMaster` definition on Line 10 encapsulate the interconnections between the two subcomponents. That is, the response masks are kept internal and are not available for external interfacing in this definition. The bus master takes input in the form of `Data` from an external memory module with a pair of memory access requests. It yields a single memory access request (`MemAcc`) to an external memory module as well as request responses to the processors. The high level processor is represented by the leftmost memory access and response request while the low is on the right.

6.5.6 Using the Segmenter with Processors

The bus master is a stand-alone component in ReWire that we can use by interfacing it with two processor devices and a memory module. We illustrate a use case with a modified ReWire DLX implementation in Listing 6.16.

```
1 memory :: ReT MemAcc MemRsp I ()
2 proc :: ReT (Instr, MemRsp) (NextInst, MemAcc) I ()
```

```
3
4  type SystemOut = ((NextInst, MemAcc),
5                    ((NextInst, MemAcc),
6                     ((MemAcc, (MemRsp, MemRsp)), Data)))
7
8  type SystemIn = ((Instr, MemRsp),
9                   ((Instr, MemRsp),
10                   ((Data, (MemAcc, MemAcc)), MemAcc)))
11
12 systemOut :: SystemOut
13             -> (NextInst, NextInst)
14 systemOut ((nextInstH,_),((nextInstL,_),_)) = (nextInstH,nextInstL)
15
16
17 systemIn  :: SystemOut
18             -> (Instr,Instr)
19             -> SystemIn
20 systemIn ((nextInstH,memAccH),((nextInstL,memAccL),
21            ((memAccM, (memRspH,memRspL)), dta))) (instH,instL) =
22            ((instH,memRspH),((instL,memRspL),
23             ((dta, (memAccH,memAccL)),memAccM)))
24
25 system :: ReT (Instr,Instr) (NextInst,NextInst) I ()
26 system = refold
27            systemOut
28            systemIn
29            (parI proc (parI proc (parI busMaster memory)))
```

Listing 6.16: Using the bus master to interface two processors to a memory module unit in ReWire.

In Listing 6.16 we define a system composed of two identical processors `proc`, a `busMaster`, and a memory module `memory`. The memory module has an input type `MemAcc` and an output type `Data` to be compatible with the bus master. The memory module takes a request on cycle n and returns the data response if the request is a read on cycle $n+1$. The memory module makes no impositions or restrictions on requests. This is handled by the segmentation bus master. The processors are a modified version of the ReWire DLX implementation from Chapter 9 that is compatible with this bus master. The specification is modular in a way such that making this change or other similar changes with regards to memory units is trivial. The types `SystemIn` and `SystemOut` on Lines 4-10 are the "raw" input types of the device that is the result composing all the subcomponents together in parallel as is done on Line 29.

The functions `systemIn` and `systemOut` are the routing functions used in our refold of the parallelized components. 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 combined system of processors, bus, and memory is `(NextInst,NextInst)` or the addresses of the next instructions to be fetched. These fetched instructions are the input of the system (`(Instr,Instr)`) listed in the type on Line 25. 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 26. We define the composed and refold device as `system` on Line 25. 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 instruc-

tions are provided as inputs (`(Instr,Instr)`) and two addresses are yielded for the next instruction fetch (`(NextInst,NextInst)`).

Chapter 7

Case Study: Regular Expression Compilation

The following chapter is from an accepted paper (Applied Reconfigurable Computing 2015) on regular expression compilation in ReWire. This paper demonstrates the use of ReWire to compile domain specific languages to regular expression pattern matchers, primarily for the use in network packet inspection, and compares the results to the state of the art [55]. Additionally, we demonstrate a novel and effective method for domain specific language-driven device specification. In the synthesis experiments described in this paper, ReWire is able to generate VHDL specifications that match or exceed the state of the art.

7.1 Abstract

Although FPGAs have the potential to bring software-like flexibility and agility to the hardware world, designing for FPGAs remains a difficult task divorced from standard software engineering norms. A better programming flow would go far towards realizing the potential of widely deployed, programmable hardware. We propose a general methodology based on domain specific languages embedded in the functional language Haskell to bridge the gap between high level abstractions that support programmer productivity and the need for high performance in FPGA circuit implementations. We illustrate this methodology with a framework for regular expression to hardware compilers, written in Haskell, that supports high programmer productivity while producing circuits whose performance matches and, indeed, exceeds that of a state of the art, hand-optimized VHDL-based tool. For example, after applying a novel optimization pass, throughput increased an average of 28.3% over the state of the art tool for one set of benchmarks. All code discussed in the paper is available online [69].

7.2 Introduction

FPGAs are notably difficult to program and this has motivated research into high-level synthesis (HLS) from high level programming languages and, in particular, from domain-specific languages [63]. This language-based approach is attractive because of its potential to make hardware engineering more like software engineering with its support for modularity, reuse, and abstraction, and thereby create a wider group of developers for programmable hardware. This paper describes a methodology for deriving performant hardware implementations directly from high-level functional

embedded domain-specific languages (EDSL).

This work makes the following contributions. We present ReWire [70], a subset of the Haskell functional language as a compiler target for compiling domain-specific languages to FPGAs. We show that ReWire can be effectively used as a compiler target because it supports the compilation of large input programs (over 100K LOC) and can generate competitively fast hardware implementations versus state of the art, domain-specific tools.

These contributions comprise a methodology supporting the "three P's" [34] for programming reconfigurable hardware: productivity, performance and portability. DSLs address the first two P's directly because domain specialization supports programmer productivity and, furthermore, allows aggressive optimization of domain-specific idioms. Portability is achieved by using ReWire, a retargetable language for specifying hardware devices.

New language constructs raise issues with respect to performance. Is there a performance price to be paid and, if so, is the increased expressiveness worth it? Does the increased expressiveness enable better performance and programmer productivity? In light of these questions, we evaluate our methodology via two case studies. The case studies presented here consider a purely functional framework for REHC construction, called RexHacc (for "Regular EXpression HArdware compiler-compiler"). RexHacc is an EDSL-structured compiler-compiler, implemented in Haskell, for Perl-compatible regular expressions (PCRE) similar to those seen in popular intrusion detection systems (e.g., Snort [71]).

Overview of Methodology.

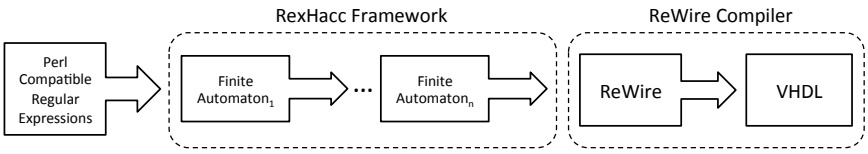


Figure 7.2: Combining the ease of use of traditional EDSLs with the power and run-time performance of a virtualized language.

Figure 7.1: FP Methodology for HLS

The methodology factors the problem of HLS into a series of translations between EDSLs. An EDSL is a domain-specific language that is defined as a collection of constructs within an existing high level language. The methodology is illustrated in the inset figure. A problem domain can be realized as a DSL embedded in Haskell. DSL cross-compilers targeting ReWire enable synthesis onto an FPGA via the ReWire compiler. Sec. 7.3 presents a more in-depth discussion of our methodology.

The case studies involve regular expression to hardware compilation (see Fig. 7.2) in which we generate artifacts that perform as well as and often better than state of the art approaches. The case studies reported here consider the problem domain of regular expression to hardware compilers (REHC) [54]. Following Fig. 7.1, we developed a reusable and modular framework for REHC called *RexHacc* and demonstrated that circuits produced with it meet or exceed the performance of state-of-the-art REHC.

The RexHacc Framework. We performed an experiment in which we compared RexHacc to the performance of the state-of-the-art REHC of Becchi and Crowley [60] (henceforth `reg2vhdl`) against its own benchmarks. The goal is to demonstrate both the productivity gain and high performance achievable via our method-

ology in the construction and testing of compilers generated by RexHacc. The presentation here is deliberately high-level. We suppress the definitions of functions and data types; the code is online [69].

The entry point for RexHacc is the function `rexhacc` with Haskell type:

```
rexhacc :: (NFA a -> NFA a) -> RegEx a -> ReWire
```

The declaration form "`::`" is pronounced "has type". The function `rexhacc` takes two inputs, an optimization function (of type `NFA a -> NFA a`) as well as a regular expression (of type `RegEx a`). The type `NFA a` (resp., `RegEx a`) represents nondeterministic finite automata (resp., regular expressions) over an alphabet of type `a`. A regular expression compiler is generated with RexHacc by applying the top-level `rexhacc` function to an optimization pass, `opt`:

```
                compiler :: RegEx a -> ReWire
(‡)             compiler = rexhacc opt
                      where opt = (o1 . ··· . on)
```

Each $\mathtt{o}_i$ is an optimization pass of functional type `NFA a -> NFA a`, all of which are composed using Haskell's function composition operator (i.e., the infix "`.`") into a single pass. This composition corresponds to the middle box in Fig. 7.2 and each $\mathtt{o}_i$ is a phase inside that box. The generated `compiler` takes a regular expression over an alphabet of type `a` and converts it into an `NFA a`, which is then fed to the optimization pass `opt`. The optimization pass produces an `NFA a` from which ReWire code is generated. The ReWire output from this compiler can either be translated into VHDL by the ReWire compiler or executed as software in any standard Haskell environment.

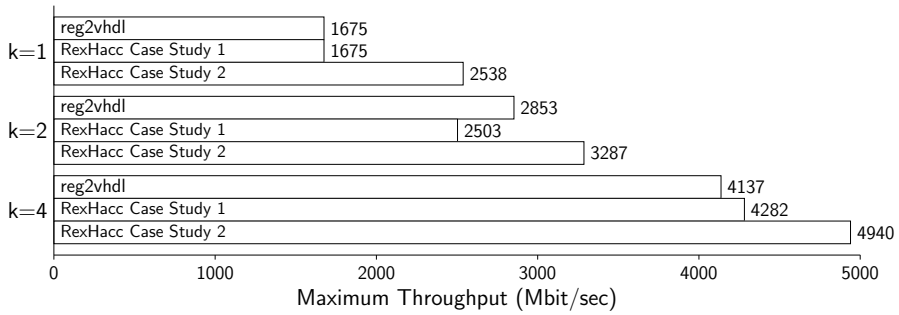


Figure 7.3: Maximum throughput for the `tcp25` benchmark, comparing `reg2vhdl` and the RexHacc case study compilers (Secs. 7.4 and 7.5). Parameter `k` indicates stride length (Sec. 7.4). Case study 2 shows an average of 28.3% throughput increase over `reg2vhdl`.

Summary of Case Study Results. Secs. 7.4 and 7.5 each describe the definition of an REHC in the RexHacc framework. Each case study was tested against `reg2vhdl` using existing test suites [60] with respect to standard metrics for circuit size, clock speed and throughput (see Fig. 7.3). The first case study (Sec. 7.4) implements the same optimization passes as `reg2vhdl`, and it was clear that this compiler generally matched or exceeded the performance of the hand-optimized compiler `reg2vhdl` with a tiny increase in circuit size. It was observed that one of the benchmarks (`tcp25`) seemed to be particularly challenging for both the first case study compiler and `reg2vhdl` with respect to throughput. This observation motivated the second case study (Sec. 7.5), which improves on the first with an (apparently novel) optimization pass that results in better performance than `reg2vhdl` on the `tcp25` benchmark.

7.3 A Methodology for Synthesis from Functional EDSLs

Synthesis from pure functional languages (e.g., Haskell, `www.haskell.org`) is appealing because combinational hardware is functional in nature, functional languages have powerful features supporting programmer productivity (e.g., modularity, expressive data types, static type inference, etc.), and the absence of side effects (e.g., destructive update) simplifies synthesis. But general purpose functional languages also contain a number of features that cannot be represented in hardware (e.g., general recursion and garbage collection) and this makes HLS directly from existing functional languages more challenging.

ReWire [70] is a proper *sublanguage* of Haskell—i.e., any ReWire program is a Haskell program, but not all Haskell programs are ReWire programs. ReWire programs, in contrast with general purpose functional languages like Haskell, are always synthesizable to hardware. ReWire restricts Haskell by disallowing the use of higher-order functions and general recursion at runtime (though techniques like partial evaluation may enable their use at compile time). RexHacc uses the ReWire hardware compiler as a back-end for producing VHDL implementations.

Front End.

The RexHacc compilation process begins with a collection of regular expressions written in Perl-compatible regular expression (PCRE) syntax. We use the parser combinator library Parsec in Haskell to parse the regular expressions in the source file. The regular expression is converted to the `NFA` type via a textbook translation of regular expressions to NFAs [53]. The resulting `NFA` is passed to the optimization portion of

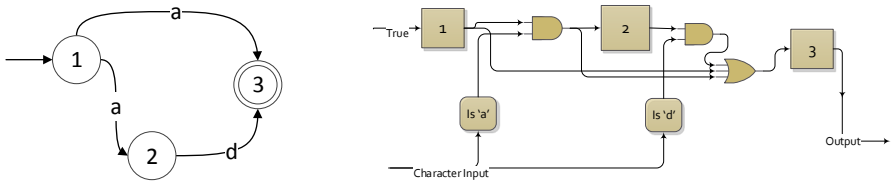


Figure 7.4: An NFA and its corresponding Sidhu and Prasanna-style implementation.

the compilation chain.

Simulating Circuits in Haskell.

Because ReWire is a sublanguage of Haskell, we can execute ReWire code as software in any Haskell environment with a test harness for executing reactive resumptions. The implementation of `rexhacc`was tested and debugged using a test harness in Haskell which is included in the code base [69].

7.4 Case Study 1: Matching State of the Art

We undertake the construction of a tool equivalent in functionality to the state of the art [60] (`reg2vhdl`) and to examine the feasibility of duplicating this functionality with our approach. The purpose of this case study is to demonstrate the *ease* with which such a tool can be constructed. The optimizations were chosen to match those of Becchi and Crowley [60] and include *head zipping*, *striding*, *alphabet compression*, and *epsilon elimination*. These results indicate that the `rexhacc`-based compiler compares favorably to and often surpasses `reg2vhdl` where throughput is concerned, and area utilization is similarly competitive. Each optimization phase was implemented in a few dozen lines of Haskell code; this is a rough indication that the

amount of programmer effort required is small.

- *Head zipping.* Head zipping is a transformation that merges outbound transitions from a state that have the same transition labels. Nodes with more than one inbound transition are not head zipped because this would result in a non-equivalent NFA. Head zipping is performed by merging the destination nodes of the matching transitions into one node that includes all of the outbound transitions from the merged nodes.

- *Striding.* Striding is an optimization pass that doubles the number of characters an NFA matches at each transition. Striding traverses the graph's edges and looking two transitions ahead from each state, converts two-transition sequences to a single transition consuming two characters.

- *Alphabet compression.* Alphabet compression is a technique that increases sharing of logic by exploiting the identical treatment of different characters by an NFA. If two characters always result in the same transitions between all states, then these characters are compressed into one character class.

- *Epsilon elimination.* Eliminating ϵ-transitions reduces the complexity and size of NFAs and simplifies code generation. NFAs with ϵ-transitions allow state transitions without consuming input. States connected to an NFA solely by ϵ-transitions can be eliminated. Eliminating unnecessary states reduces the number of flip flops required to implement the NFA on an FPGA. A textbook ϵ-elimination algorithm is used [53].

Experiments and Evaluation.

To test the performance of RexHacc, we selected three benchmark sets of regular expressions from the literature [55, 60]. `Snort24` is a set of 24 regular expressions drawn from the Snort network intrusion detection system [71]. `Tcp25` is a set of 79 regular expressions designed to match malicious SMTP traffic, also drawn from the Snort NIDS. `Bro217` is a set of 217 regular expressions drawn from the Bro NIDS [72]. Matchers for each of these benchmarks were generated using `reg2vhdl`, as well as RexHacc. Each benchmark was tested at stride lengths $k = 1$, $k = 2$, and $k = 4$, producing circuits that consume input streams at one, two, and four bytes per clock cycle. The resulting VHDL was then synthesized using Xilinx's XST synthesis tool for the Xilinx Spartan-3E X3CS500E FPGA, speed grade -4. The synthesis tools are optimized for speed. The frequencies that we list are synthesis estimates.

Fig. 7.5 compares the resulting circuits in terms of three performance metrics: (a) logic slice utilization, (b) LUT utilization, and (c) maximum throughput as measured in megabits per second. (Flip flop utilization was extremely close between the two tools and thus is not shown.) RexHacc compares favorably with `reg2vhdl` on virtually all fronts.

Throughput.

RexHacc matches or exceeds `reg2vhdl`'s total throughput for all but one of the nine benchmarks. In the best case (benchmark `bro217`, $k = 1$) throughput is around 60% higher. In the worst case (benchmark `tcp25`, $k = 2$) throughput is around 13% lower. Both tools, in all cases, are capable of processing input at a rate of more than 1 Gbit/sec. In the best case, RexHacc is capable of handling input rates up to 7.5

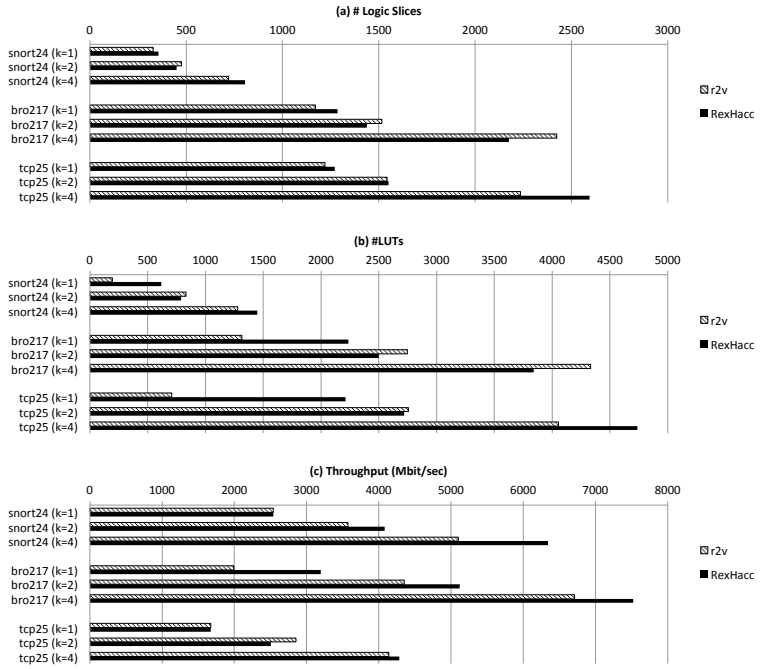


Figure 7.5: Performance comparisons of RexHacc to `reg2vhdl` tool (here, "r2v").

Gbit/sec on a Xilinx Spartan-3E FPGA at a relatively low clock rate. Tests on a Xilinx 7-series platform (not presented here, but available online [69]) indicate that throughputs of up to 25 Gbit/sec are achievable with a more modern FPGA.

Logic utilization.

With the exception of the single-strided ($k = 1$) benchmarks, LUT utilization for RexHacc-generated circuits ranged from 88% to 116% of their `reg2vhdl` counterparts. In the specific case where $k = 1$, RexHacc tends to produce circuits with higher LUT counts (up to 219% higher), suggesting that the combinational next-state logic produced by the RexHacc code generator is more complicated for these circuits. For all benchmarks, flip flop utilization for RexHacc was close to, but slightly higher

than, the results generated by `reg2vhdl`. This is not surprising since each state in the NFA is represented by a single flip flop, and both tools tend to generate similar numbers of NFA states. RexHacc, however, pays a small penalty here, because it generates output signals synchronously, storing them in flip flops, while `reg2vhdl` does not. Please note, however, that the choice of synchronous outputs rather than asynchronous ones is optional in the most recent version of ReWire.

The results exhibited here suggest that the case study compiler is competitive with the state of the art. The extra flexibility of the modular, purely functional design does not come at a prohibitive cost in terms of circuit size, and indeed brings substantial benefits with respect to throughput.

7.5 Case Study 2: Surpassing State of the Art

In this case study, we demonstrate the *agility* of the RexHacc approach by identifying an opportunity for an optimization, and rapidly implementing that optimization as a compiler phase in RexHacc. The modular nature of RexHacc made it easy both to identify a key performance bottleneck, and to implement a new optimization pass to address it.

Identifying the bottleneck.

While conducting the experiments of Sec. 7.5, we noticed that one of the benchmarks, `tcp25`, stood out for its relatively low maximum throughput when processed by RexHacc as well as by `reg2vhdl`. While striding enabled our compiler to produce circuits with maximum throughput in excess of 6 Gbit/sec for `snort24` and

`bro217`, maximum throughput for `tcp25` just barely exceeded 4 Gbit/sec. The throughput advantage over `reg2vhdl` observed for `snort24` and `bro217` was essentially nonexistent for `tcp25`.

To explore the reasons for this, we instrumented our compiler pipeline by using the Haskell Functional Graph Library's built-in support for generating graph visualizations via GraphViz (www.graphviz.org). We observed that the `tcp25` NFA exhibited a structural feature that was not present in the `snort24` and `bro217` NFAs. Specifically, the `tcp25` NFA contained one state that had a large number of inbound transitions. A simplified example of this problem is exhibited in Fig. 7.6 (left), where state 9 has eight inbound transitions. A large number of inbound transitions emerges when the source regular expression contains a long chain of choice operators. This pattern is not uncommon in packet inspection rulesets (e.g., consider a long chain of alternative filenames followed by the common suffix ".exe").

In the circuit implementation the inbound transitions translate to a large fan-in of signals that must be ORed together to determine whether to activate that state. As the size of this fan-in grows large, the combinational logic involved begins to dominate the critical path of the circuit. The result is a sharp reduction in maximum operating clock frequency, and therefore throughput. This suggested an opportunity for optimization: namely, to transform the NFA in such a way as to reduce the number of inbound transitions to heavily-loaded states.

State Splitting Optimization.

To address the performance bottleneck, we extended the compiler of Sec. 7.4 with an optimization called *state splitting*. Suppose we have in our NFA a state s with

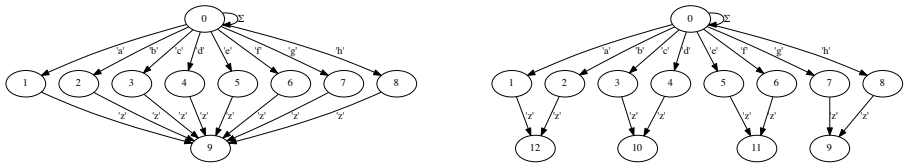


Figure 7.6: NFA for $(a|b|c|d|e|f|g|h)z$, before state splitting (left) and after (right).

inbound transitions $e_1, \cdots, e_n$, and assume without loss of generality that s has no self-loops. Observe that we can produce an *equivalent* NFA by "splitting" s in two: that is, introducing a new state (call it s'), and reassigning half of the inbound transitions (say, $e_1, \cdots, e_{\lceil n/2 \rceil}$) to s' instead of s. State splitting works by applying this transformation to each node whose indegree exceeds a certain fixed threshold t. Fig. 7.6 (right) illustrates the results of applying state splitting to the NFA for $t = 2$. N.b., the maximum indegree has been reduced from 8 to 2 in this example.

The reader may note that this optimization may have the effect of *increasing* the number of inbound transitions for successor states of split nodes. This is generally not a problem for two reasons: first, as long as state splitting succeeds in reducing the *maximum* indegree, it is likely to pay off even if some states see their number of inbound transitions increased. Second, state splitting may be iterated; if the splitting of state s_1 results in state s_2 exceeding the split threshold, s_2 itself may be split.

The full code for the state-splitting optimization, consisting of 17 lines of code, is given as the `splitStates` function in the code base [69]. We can insert the state-splitting into the optimization pipeline simply by adding an extra phase to the `rexhacc` call; this is an instance of (‡) from Sec. 7.2:

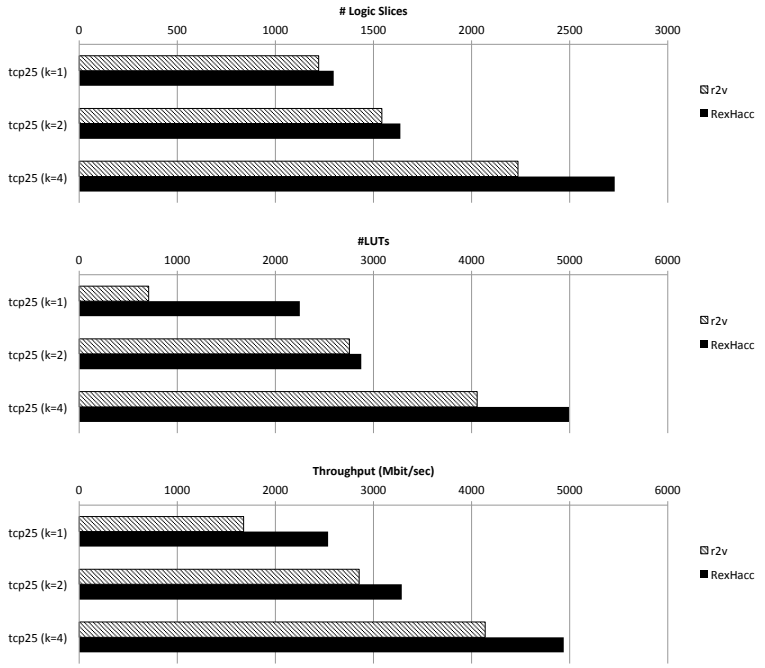


Figure 7.7: Comparisons of RexHacc with state splitting enabled to `reg2vhdl` (here, "r2v") tool.

7.6 Conclusions and Future Work

This research is a substantial case study utilizing the ReWire compiler at scale. ReWire is a subset of Haskell limited in expressive power to ensure the synthesizability of every ReWire program. There is a potential drawback to such restrictions: it excludes many powerful functional programming idioms. In spite of this potential drawback, we demonstrate that ReWire maintains sufficient expressiveness to support the design and implementation of high level DSLs for specifying fast hardware accelerators. Future work aims to improve the resource usage of ReWire-generated devices by optimizing ReWire's code generation stages.

The methodology leverages the intrinsic power of Haskell and functional program-

ming. RexHacc is modular and customizable in the sense that optimization passes can be easily added and removed. Because the ordering of passes is exposed as function composition in Haskell, experimentation with optimization ordering is enabled. A RexHacc-generated compiler can be instrumented in a straightforward manner as we did with GraphViz and take advantage of existing external Haskell tools.

The flexibility of the RexHacc framework derives from the cross-compilation to ReWire and the ability of ReWire to generate VHDL synthesizable to efficient circuits. The methodology we have introduced lowers the barrier to entry for reconfigurable computing for functional programmers. At the same time, it provides an opportunity for hardware designers to leverage the power of the functional paradigm to improve productivity. The choice of a purely functional language does not come at a performance cost: our benchmarking demonstrates that we match or exceed the performance of a state-of-the-art hand-tuned compiler for a number of real-world tests.

The two research directions we are pursuing have to do with increasing the expressiveness of the type system to support metaprogramming and hardware security. The current methodology is based on metaprogramming (i.e., ReWire/Haskell programs are generated by Haskell programs) and there are type systems for staged programming (e.g., MetaML [73]) that we believe will improve programmer productivity further while automatically enforcing type safety. We developed a type system for enforcing fault isolation on ReWire [74] and we are currently extending to information flow security.

7.7 Acknowledgments

The authors would like to thank Jason Agron of Intel Corporation and David Andrews of the University of Arkansas for their helpful feedback.

Chapter 8

Case Study: Implementing the Salsa20 Cipher

This chapter describes the design and implementation of the Salsa20 stream cipher algorithm using ReWire and Connect Logic. It emphasizes equational reasoning to prove correct the pipelining transformation described in Chapter 3. We implement an iterative and pipelined form of Salsa20 here and show that the implementation is performant with good resource characteristics.

8.1 Abstract

There is a semantic gap between the hardware definition languages used to design and implement hardware and the languages and logics used to formally specify and verify them. Bridging this gap—i.e., constructing formal models from existing hardware artifacts—can be costly, time-consuming, and error prone—and yet utterly necessary

if formal verification is to proceed. This work demonstrates that this gap can be collapsed by starting in a pure functional language that is also a hardware description language, and that equational style verifications may be performed directly on the source text of a hardware design, thereby significantly lowering the verification cost for reconfigurable designs. When combined with an efficient compiler, this methodology achieves both good performance and low cost verification.

8.2 Introduction

Reconfigurable computing emphasizes a "mix and match" approach to system construction, frequently involving specially tailored "one off" components. Formal methods can provide high confidence that systems obey critical properties (e.g., safety and security), but, by reputation, they can also involve a substantial investment of time and effort. Formal methods may, therefore, seem somewhat antithetical to reconfigurable computing. Can it make economic sense to invest the resources for formal methods on potentially "one off" reconfigurable systems?

The proposed methodology aims to make hardware verification cost effective for reconfigurable designs via a functional programming language that also serves as a hardware description language. The principal hypothesis of this research is that following this methodology can significantly reduce the effort of verifying hardware designs, thereby making formal verification cost effective for reconfigurable computing. The functional language—ReWire [52]—plays a dual rôle for both hardware description and formal specification. We support this hypothesis with a demonstration of the approach in which the stream cipher Salsa20 [75] is implemented efficiently in

ReWire and verified using equational reasoning on the implementation source code.

In the functional programming community, equational reasoning about programs frequently goes by the moniker "Bird-Wadler style" (so named for the influential textbook [76]). Functional programmers reason about source programs in an equational style, by replacing equals for equals, making simplifications, induction and coinduction, etc. Equational reasoning is commonly used to justify, among other things, source-to-source transformations and program correctness. This is precisely what we use Bird-Wadler reasoning for in this paper, although, in ReWire, programs *are* hardware descriptions.

This research demonstrates that formal methods and reconfigurable systems are not antithetical to one another at all. **The contributions of this paper are as follows. (1)** We describe a methodology for developing high assurance, reconfigurable systems leveraging pure functional languages and equational reasoning. A standard practice in functional programming—Bird-Wadler reasoning—is repurposed to hardware design with this methodology. **(2)** We introduce an extension to ReWire called *Connect Logic*, which consists of domain specific language abstractions for hardware devices that support a mixture of functional and structural design styles. **(3)** Encapsulation of a pipelining structuring technique in Connect Logic is exhibited along with **(4)** several performant implementations of the Salsa20 stream cipher based on it.

Reconfigurable Salsa20 without ReWire

Consider the following experiment. A hardware designer decides to implement the Salsa20 cipher in hardware. There are a number of good reasons to do so, not the

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

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

Theorem (Fib)**.** *For all* $\mathtt{n} \geq 0$*,* $fib(n) = fst\ (fib2(n))$*.*

Figure 8.1: Bird-Wadler Program Development

least of which is that reconfigurable hardware can increase the possible throughput compared to a software implementation. The hardware designer uses a tried and true hardware definition language (HDL) like VHDL or Verilog. The implementation path is straightforward—she implements Bernstein's defining equations [75] in terms of the HDL and performs her usual development process involving synthesis, simulation, and testing.

This first implementation is one step removed from Bernstein's high-level specification, and, furthermore, is expressed in a language without a formal semantics. So, how does she prove that the first implementation is correct? It becomes clear to the hardware engineer that the first implementation does not suffice: even implemented in the most optimized fashion, it contains too many gates for most FPGAs. So, the hardware engineer produces a second implementation structured in an explicitly pipelined form resulting in a circuit that fits on her FPGA.

Is she all done? Not if formal proof is required that the second implementation is correct. The second implementation is two steps removed from Bernstein's high level specification and it is written in a language without a formal semantics. To verify its correctness, where does she even start? She could attempt to verify the

implementation by encoding it in the logic of a theorem prover, but, observe that this involves yet another translation—and one which is not straightforward. With this approach, how can we be sure that her logical specification faithfully relates Bernstein's high-level specification to a VHDL implementation?

Bird-Wadler Provably Correct Development

To illustrate the formal methodology we advocate for reconfigurable computing, consider first this classic example (p.131, [76]) of Bird-Wadler style equational reasoning in Fig. 8.1. On the left is the usual recursive definition of the Fibonacci function. It serves as a *reference specification* defining the meaning of the Fibonacci function, but it has terrible $O(2^n)$ performance. The other version of the Fibonacci function on the right is in an optimized, "accumulator-passing style" form with $O(n)$ performance.

The hallmark of Bird-Wadler development is that there is a reference specification (e.g., *fib*) and one or more transformations from it (e.g., into *fib*2) that give rise to an equational verification (e.g., the Fib theorem in Fig. 8.1). This verification justifies using the optimized version (i.e., replacing $fib(n)$ with $fst(fib2(n))$).

Provably Correct Development of Salsa20 with ReWire

It is precisely the Bird-Wadler style of development that ReWire enables for reconfigurable computing. Fig. 8.4 presents the hash function from the Salsa20 stream cipher [75] represented in a Haskell-like syntax. We discuss this figure in some detail as well as explain the requisite Haskell syntax in subsequent sections. It suffices to say that Fig. 8.4 contains a functional program defining the Salsa20 hash function that also serves as the high-level reference specification in the Bird-Wadler development

presented in our case study. To render it into a synthesizable form, we add some Connect Logic annotations to produce the ReWire code in Fig. 8.5. The ReWire compiler can now synthesize a circuit for Salsa20. This new implementation can now be measured in two ways: against standard performance metrics as in Table 8.1 or by verifying that it produces the same answers as the reference specification (Theorem 1). The first ReWire implementation is now rewritten using pipelining constructs also written in Connect Logic (the ten and twenty stage pipelines in Figures 8.6 and 8.7, resp.). The correctness of the pipelining transformation is given in Theorem 2.

Section 8.3 introduces Connect Logic and the pipelining structuring technique applied and verified in Sections 8.4 and 8.5, resp. Section 8.6 summarizes and concludes. Most of the subject matter in this paper relates to provably correct development of reconfigurable hardware rather than on more traditional areas of reconfigurable computing. The targeted audience for the paper is, however, the reconfigurable computing community and so considerable effort has been made to make the paper as self-contained as possible.

8.3 Connect Logic in ReWire

Connect Logic has operations for composing and connecting smaller devices to create larger ones. Sec. 8.3.2 below introduces Connect Logic at a high level; for reasons of space, a semantic treatment of Connect Logic is left for future work. We then illustrate the use of Connect Logic via the design of a pipelining transformation for ReWire. Section 8.3.1 gives background information on pure functional languages and equational verification.

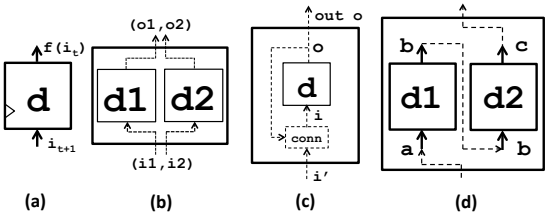


Figure 8.2: Device Constructors

8.3.1 Pure Functional Languages & Equational Verification

Primer on Haskell/ReWire Syntax

For the sake of being as self-contained as possible, this section presents a quick overview of Haskell—and, hence, ReWire—syntax necessary to understand this paper.

Haskell [5] 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 $e :: a$. Haskell has a built-in list type constructor: $[a]$ is the type of all lists of elements of type a. Because of Haskell's lazy evaluation strategy, lists can have an infinite number of elements—such lists are also called *streams*.

Below are a number of function declarations. The simplest function is the identity function, which takes its argument and simply returns it.

In Haskell/ReWire, we can introduce new datatypes with the *data* keyword. In the following declarations, *Quad* and *Hex* are *type constructors* that, given any type a, construct new types, *Quad a* and *Hex a*, resp. To construct a value of a datatype, apply a *data constructor*; the data constructors below are Q and H. For example, a value Q `1 2 3 4` is of type *Quad Int*; we write this type declaration as

Q `1 2 3 4` :: *Quad Int*. A *Bit* is either *High* or *Low*.

```
data Quad a = Q a a a a
data Hex  a = H a a a a a a a a a a a a a a a a
data Bit    = High | Low
```

ReWire has built-in types for words. A 32-bit (128-bit) word belongs to the type $W32$ ($W128$). For example, a value of type ($Quad\ W32$) has the form ($Q\ w1\ w2\ w3\ w4$), which is nothing more than four 32-bit words.

Purity and Equational Verification

Haskell (and, hence, ReWire) is a pure language, which is a critical foundation for equational reasoning. Purity means that the type of a Haskell program faithfully represents its value and behavior. If a Haskell function has type *Int* -> *Int*, then the function takes an *Int* as input and produces an *Int* as output. Furthermore, we can conclude that the function possesses no side effects whatsoever because, in Haskell, side effects are reflected accurately in the types. The expression `(print "Hello World")`, for instance, prints out *Hello World* to the prompt and, therefore, `(print "Hello World") :: IO ()`—it produces the value nil, (), which is tagged in its type with *IO*, meaning it performs input/output in some form.

To prove an equation, $e = e'$, one starts from e and "replaces equals for equals" until e' is reached. In symbols, this proof is $e = e_1 = e_2 = \cdots = e_n = e'$ in which each step is justified by a known equation x=y—as in "replace x in e_i by y to obtain e_{i+1}". Purity supports this style of reasoning because, being all Haskell expressions are side effect free, they cannot interact unpredictably with the expressions in which they are substituted.

8.3.2 Extending ReWire with Connect Logic

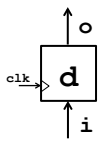
This section presents the ReWire operators for the compositional construction of devices from other devices. We refer to these particular operators as "Connect Logic". Connect Logic enables two or more existing devices to be composed in parallel and connected together. Connect Logic supports a compositional style of hardware design akin to structural VHDL. Formulating the design of a hardware device may be accomplished as in previous work [52] (i.e., without Connect Logic), or, existing devices may be composed with Connect Logic operations into bigger devices.

There is a type constructor *Dev* for synchronous devices in ReWire. There are three basic architectural constructors that Connect Logic adds to the ReWire language. The first, `iter`, constructs a synchronous device from a pure function from inputs to outputs. The second, $\langle\&\rangle$, 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).

Types for Devices

There is one basic unit of Connect Logic, 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 Connect Logic 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 Connect Logic 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 Connect Logic 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*

Iteration

The most basic Connect Logic 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. 8.2**(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*).

Parallelism

Parallelism is expressed with the device constructor, $\langle\&\rangle$, that composes two existing devices, $d1$ and $d2$, into a single device, $d1 \langle\&\rangle d2$, 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 $\langle\&\rangle$ is:

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

We omit its Haskell definition as doing so would require an unnecessary excursion into Haskell's syntax and semantics. Fig. 8.2**(b)** presents a pictorial version of $d1 \langle\&\rangle d2$. The type signature of $\langle\&\rangle$ means that the input and output types of constructed device $d1 \langle\&\rangle d2$ are pairs of the inputs and outputs of $d1$ and $d2$, resp. Both subdevices $d1$ and $d2$ are isolated from one another in $d1 \langle\&\rangle d2$—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.

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. 8.2**(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 be-

havior. 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
```

Defining a Pipeline

The form of pipeline we consider is a simple one, namely stall-free pipelines, in which the output from a stage flows directly into the input of the next stage. It is possible to define more complex pipelines (e.g., instruction pipelines that stall, etc.) with Connect Logic, but we leave that subject for a follow-on publication.

Stall-free pipelines—henceforth simply "pipelines"—have the flavor of functional composition, and the architectural combinators of ReWire allow the formalization of this intuition. For functions, $\mathtt{f_j}$, of appropriate type, the composition, $f_n \circ \cdots \circ f_1$, resembles a pipeline. Of course, this ignores the timing aspect of a pipeline. In ReWire, we can express this pipeline, along with its timing, as the following:

$$\mathit{iter}\ f_1\ o_1\ \rightsquigarrow \cdots \rightsquigarrow\ \mathit{iter}\ f_n\ o_n$$

where f_j :: a_j -> a_{j+1} are pure functions from input of type a_j to output of type a_{j+1} and each o_j :: a_{j+1} is the initial output value produced by pipeline stage *iter* f_j o_j. The $\rightsquigarrow$ combinator chains each stage together, connecting the output of the j^{th} stage to the input of the $j{+}1^{th}$ stage. The combinators for pipelining, etc., are defined below.

$$
\begin{array}{ll}
R1 & \left[\begin{array}{ll}
1 & \left[\begin{array}{ll} x[4] \;\oplus\!=(x[0] \boxplus x[12]) \lll 7 & x[9] \oplus\!=(x[5] \boxplus x[1]) \lll 7 \\ x[14] \oplus\!=(x[10] \boxplus x[6]) \lll 7 & x[3] \oplus\!=(x[15] \boxplus x[11]) \lll 7 \end{array}\right] \\
2 & \left[\begin{array}{ll} x[8] \oplus\!=(x[4] \boxplus x[0]) \lll 9 & x[13] \oplus\!=(x[9] \boxplus x[5]) \lll 9 \\ x[2] \oplus\!=(x[14] \boxplus x[10]) \lll 9 & x[7] \;\oplus\!=(x[3] \boxplus x[15]) \lll 9 \end{array}\right] \\
3 & \left[\begin{array}{ll} x[12] \oplus\!=(x[8] \boxplus x[4]) \lll 13 & x[1] \;\oplus\!=(x[13] \boxplus x[9]) \lll 13 \\ x[6] \;\oplus\!=(x[2] \boxplus x[14]) \lll 13 & x[11] \oplus\!=(x[7] \boxplus x[3]) \lll 13 \end{array}\right] \\
4 & \left[\begin{array}{ll} x[0] \;\oplus\!=(x[12] \boxplus x[8]) \lll 18 & x[5] \;\oplus\!=(x[1] \boxplus x[13]) \lll 18 \\ x[10] \oplus\!=(x[6] \boxplus x[2]) \lll 18 & x[15] \oplus\!=(x[11] \boxplus x[7]) \lll 18 \end{array}\right]
\end{array}\right] \\
R2 & \left[\begin{array}{ll}
5 & \left[\begin{array}{ll} x[1] \;\oplus\!=(x[0] \boxplus x[3]) \lll 7 & x[6] \;\oplus\!=(x[5] \boxplus x[4]) \lll 7 \\ x[11] \oplus\!=(x[10] \boxplus x[9]) \lll 7 & x[12] \oplus\!=(x[15] \boxplus x[14]) \lll 7 \end{array}\right] \\
6 & \left[\begin{array}{ll} x[2] \oplus\!=(x[1] \boxplus x[0]) \lll 9 & x[7] \;\oplus\!=(x[6] \boxplus x[5]) \lll 9 \\ x[8] \oplus\!=(x[11] \boxplus x[10]) \lll 9 & x[13] \oplus\!=(x[12] \boxplus x[15]) \lll 9 \end{array}\right] \\
7 & \left[\begin{array}{ll} x[3] \oplus\!=(x[2] \boxplus x[1]) \lll 13 & x[4] \;\oplus\!=(x[7] \boxplus x[6]) \lll 13 \\ x[9] \oplus\!=(x[8] \boxplus x[11]) \lll 13 & x[14] \oplus\!=(x[13] \boxplus x[12]) \lll 13 \end{array}\right] \\
8 & \left[\begin{array}{ll} x[0] \;\oplus\!=(x[3] \boxplus x[2]) \lll 18 & x[5] \;\oplus\!=(x[4] \boxplus x[7]) \lll 18 \\ x[10] \oplus\!=(x[9] \boxplus x[8]) \lll 18 & x[15] \oplus\!=(x[14] \boxplus x[13]) \lll 18 \end{array}\right]
\end{array}\right]
\end{array}
$$

Figure 8.3: Salsa20 Hashing Algorithm [77]. Operation $\oplus$ is bitwise exclusive OR and $\boxplus$ is addition modulo 2^{32}, and $\lll$ is left rotate. Each set of four assignments numbered 1–8 is a *quarter round*, and each *round*, $R1$ and $R2$, consists of four quarter rounds each. The algorithm consists of repeating each *double round* $(R1; R2)$ ten times in succession. Argument x is a 16 element array of 32 bit words.

Note that $\rightsquigarrow$ is not syntactic sugar for function composition. For example, while it is true that $id \circ f = f$, it is also the case that $iter\ id\ o_1 \rightsquigarrow iter\ f\ o_2 \neq iter\ f\ o_2$. The LHS of this inequality is a two stage pipeline while the RHS is a one stage pipeline. The outputs both pipelines produce will be related, of course.

Given two devices, $d1$ and $d2$, the ReWire code for connecting them in pipelined sequence is below. This construction is illustrated in Fig. 8.2**(d)**. The two devices are first placed unconnected in parallel (i.e., $d1$ `<&>` $d2 :: Dev\ (a, b)\ (b, c)$) and, in this context, both devices operate in isolation. The combined device consumes a single input of type (a, b) and produces a single output of type (b, c). The output type for $(d1 \rightsquigarrow d2)$ is c; i.e., the second component of the output tuple of $d1$ `<&>` $d2$. The external input (of type a) to $(d1 \rightsquigarrow d2)$ is passed to the subdevice $d1$ and the output of $d1$ to the input of $d2$; thus the routing function *pipe* is as defined below:

$$
\begin{aligned}
&\mathit{salsa20} \;::\; \mathit{W128} \texttt{ -> } \mathit{Hex}\ \mathit{W32}\\
&\mathit{salsa20}\ \mathit{nonce} \;=\; \mathit{hash}\ (\mathit{initialize}\ \mathit{key}_0\ \mathit{key}_1\ \mathit{nonce})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{hash} \;::\; \mathit{Hex}\ \mathit{W32} \texttt{ -> } \mathit{Hex}\ \mathit{W32}\\
&\mathit{hash}\ x \;=\; x \;+\; \underbrace{\texttt{doubleround}(\cdots(\texttt{doubleround}}_{10}(x))\cdots)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{doubleround} \;::\; \mathit{Hex}\ \mathit{W32} \texttt{ -> } \mathit{Hex}\ \mathit{W32}\\
&\mathit{doubleround}\ x \;=\; \mathit{rowround}\ (\mathit{columnround}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{quarterround} \;::\; \mathit{Quad}\ \mathit{W32} \texttt{ -> } \mathit{Quad}\ \mathit{W32}\\
&\mathit{quarterround}\ (y_0, y_1, y_2, y_3) \;=\; (z_0, z_1, z_2, z_3)\\
&\quad \textbf{where}\\
&\qquad z_1 = y_1 \oplus (y_0 + y_3) \lll 7\\
&\qquad z_2 = y_2 \oplus (z_1 + y_0) \lll 9\\
&\qquad z_3 = y_3 \oplus (z_2 + z_1) \lll 13\\
&\qquad z_0 = y_0 \oplus (z_3 + z_2) \lll 18
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{rowround} \;::\; \mathit{Hex}\ \mathit{W32} \texttt{ -> } \mathit{Hex}\ \mathit{W32}\\
&\mathit{rowround}\ (y_0, \ldots, y_{15}) \;=\; (z_0, \ldots, z_{15})\\
&\quad \textbf{where}\\
&\qquad (z_0, z_1, z_2, z_3) \;\;\;\;\;\;\;\; = \mathit{quarterround}\ (y_0, y_1, y_2, y_3)\\
&\qquad (z_5, z_6, z_7, z_4) \;\;\;\;\;\;\;\; = \mathit{quarterround}\ (y_5, y_6, y_7, y_4)\\
&\qquad (z_{10}, z_{11}, z_8, z_9) \;\;\;\; = \mathit{quarterround}\ (y_{10}, y_{11}, y_8, y_9)\\
&\qquad (z_{15}, z_{12}, z_{13}, z_{14}) = \mathit{quarterround}\ (y_{15}, y_{12}, y_{13}, y_{14})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{columnround} \;::\; \mathit{Hex}\ \mathit{W32} \texttt{ -> } \mathit{Hex}\ \mathit{W32}\\
&\mathit{columnround}\ (x_0, \ldots, x_{15}) \;=\; (y_0, \ldots, y_{15})\\
&\quad \textbf{where}\\
&\qquad (y_0, y_4, y_8, y_{12}) \;\;\;\; = \mathit{quarterround}\ (x_0, x_4, x_8, x_{12})\\
&\qquad (y_5, y_9, y_{13}, y_1) \;\;\;\; = \mathit{quarterround}\ (x_5, x_9, x_{13}, x_1)\\
&\qquad (y_{10}, y_{14}, y_2, y_6) = \mathit{quarterround}\ (x_{10}, x_{14}, x_2, x_6)\\
&\qquad (y_{15}, y_3, y_7, y_{11}) = \mathit{quarterround}\ (x_{15}, x_3, x_7, x_{11})
\end{aligned}
$$

Figure 8.4: Reference Specification of Salsa20 Hash Function [75], which plays the rôle of reference specification in our case study. Operation $\oplus$ is bitwise exclusive OR, $+$ is addition modulo 2^{32}, and $\lll$ is left rotate.

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

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

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

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

out      :: (Hex W32, Hex W32) -> Hex W32
out ((x0, ..., x15), (y0, ..., y15)) = (x0+y0, ..., x15+y15)

conn :: (Hex W32, Hex W32) ->
        (Bit, W128) -> (Hex W32, Hex W32)
conn (o1, o2) (Low, nonce)    = (o1, o2)
conn (o1, o2) (High, nonce)) = (x, x)
  where
    x = initialize key0 key1 nonce
```

Figure 8.5: Iterative Salsa20 Device in ReWire.

```
(⇝) :: Dev a b -> Dev b c -> Dev a c

d1 ⇝ d2 = refold snd pipe (d1 ⟨&⟩ d2)

  where

    pipe (b, c) a = (a, b)
```

Compiling Connect Logic

A pure function `f` in ReWire will be compiled into a combinational circuit of fixed depth that, in turn, determines a fixed delay. If $\texttt{f} = \texttt{f}_n \circ \cdots \circ \texttt{f}_1$, then its depth is additive as is its delay. Composition of pure functions exposes an opportunity for a pipelining optimization to reduce the average propagation delay of the entire circuit.

The two operators $\langle\&\rangle$ and *refold* are treated as primitives in the ReWire compilation process. These operations correspond directly to structural features in generated VHDL. The $\langle\&\rangle$ operation is compiled to a single VHDL entity that handles the combined IO of two ReWire devices, and port maps it accordingly. The *refold* operator is a single entity with included functions to manipulate the IO of a device in the manner prescribed by the type of the *refold* function.

8.4 Provably Correct Development of Salsa20 Devices in ReWire and Connect Logic

Salsa20 is a stream cipher developed by Bernstein [77] and is part of the ECRYPT ESTREAM [78] portfolio of cryptographic ciphers. Salsa20 was originally intended for software implementation, but can also be synthesized on an FPGA with careful consideration given to space and mapping constraints. Fig. 8.3 presents the Salsa20 hashing algorithm, which is the heart of the Salsa20 algorithm itself and where the bulk of its computation occurs. The inputs to the algorithm include a 16-element array of 32 bit words, called x in the figure.

8.4.1 Salsa20 Reference Specification

Fig. 8.4 contains the reference specification for Salsa20. This specification simply recasts Bernstein's functional specification [75] using Haskell syntax. The function *hash* formulates the original specification from Fig. 8.3 and the function *salsa*20 is the entry point for the whole algorithm. There are certain details which we have left out of this code for the sake of brevity and comprehensibility; these include routines to

change endianness, to reformat words as sequences of bytes, and similar such routines. The function *initialize* sets up the initial input; its definition is omitted as well.

8.4.2 Salsa20 Iterative Implementation

Fig. 8.5 contains the additional ReWire code to create an iterative version of Salsa20. Two devices are created, *dblrd* and *passthru*, using the *iter* constructor in Connect Logic. A diagrammatic view of the circuit produced is found in Fig. 8.8(a). Synthesis estimates of resource usage and FMax for $sls20dev$ are in Table 8.1.

There is one functional unit performing the *doubleround* operation. This operates ten cycles to produce on answer. When the inputs to the device $sls20dev$ are $[(High, n), (Low, n_0), \ldots, (Low, n_9), \ldots]$, then, on the cycle with input (Low, n_9), the output will be $salsa20\ n$. The *High* bit signifies that the device should start hashing n. The (Low, n') input signifies that n' should be ignored and that the iteration should continue.

8.4.3 Pipelining Salsa20

The numbers for the iterative device are reasonable, but the structure of the cipher algorithm would indicate that there is room for improvement. There is an apparent performance gap with this approach: nine cycles of the device do not yield useful output. Pipelining our base components together gives us a way to keep our performance characteristics with respect to clock speed roughly the same while enabling our device to be productive on every clock cycle. We do so by placing ten different *passthru* $\langle\&\rangle$ *dblrd* devices in sequence, connecting their inputs and outputs together

$$
\begin{array}{l}
pipe10 \;::\; Dev\ W128\ (Hex\ W32) \\
pipe10 \;=\; refold\ out\ inpt\ tenstage \\
\quad \textbf{where} \\
\qquad tenstage = \underbrace{\texttt{stage} \rightsquigarrow \cdots \rightsquigarrow \texttt{stage}}_{10} \\
\qquad stage \quad\; = passthru\ \langle\&\rangle\ dblrd
\end{array}
$$

Figure 8.6: Ten Stage Pipeline

to obtain *pipe*10 in Fig. 8.6.

A twenty stage pipeline may be created by increasing the granularity of each stage. Now, instead of staging each *doubleround* as before, each component *columnround* and *rowround* is staged (see Fig. 8.7).

8.5 Evaluating Provably Correct Salsa20 Devices

This section evaluates the devices created in the previous section according to two modes: performance and verification. The devices synthesized by the ReWire compiler exhibit performance comparable to a previously published, hand optimized design [79] We sketch the verification of general theorem which characterizes the correctness of the pipelining transformation applied in Section 8.4.3.

In this section, we sketch the verification of the pipelining transformation defined in Section 8.4. There is a function of the following type that serves to run a device on stream of inputs: *feed* :: $[i]$ -> *Dev i o* -> $[o]$. For a stream of inputs *is* :: $[i]$ and a device d :: *Dev i o*, *feed is d* is the stream of outputs created by running the device d on *is*. N.b., *feed* preserves the order of outputs with respect to inputs; i.e., if i is the n^{th} input in *is*, then the $(n+1)^{st}$ item in *feed is d* was produced by d on i. We

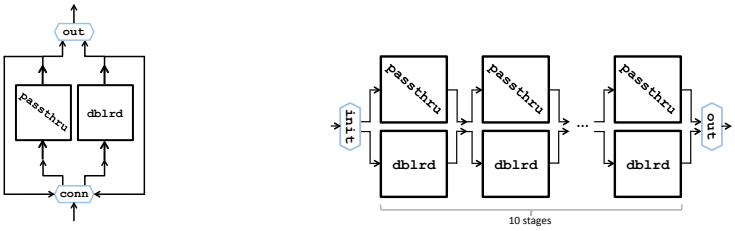
$$
\begin{aligned}
\mathit{crstage} &= \mathit{passthru}\ \langle\&\rangle\ \mathit{crdev} \\
&\quad \textbf{where} \\
&\qquad \mathit{crdev} = \mathit{iter}\ \mathit{columnround}\ (\mathit{columnround}\ \mathit{zeros}) \\
\\
\mathit{rrstage} &= \mathit{passthru}\ \langle\&\rangle\ \mathit{rrdev} \\
&\quad \textbf{where} \\
&\qquad \mathit{rrdev} = \mathit{iter}\ \mathit{rowround}\ (\mathit{rowround}\ \mathit{zeros}) \\
\\
\mathit{pipe}20 &= \begin{pmatrix} \texttt{crstage} \rightsquigarrow \texttt{rrstage} \rightsquigarrow \\ \vdots \\ \texttt{crstage} \rightsquigarrow \texttt{rrstage} \rightsquigarrow \\ \texttt{crstage} \rightsquigarrow \texttt{rrstage} \end{pmatrix} \qquad (10)
\end{aligned}
$$

Figure 8.7: Twenty Stage Pipeline.

omit the definition of *feed*.

8.5.1 Performance

We evaluated the performance of the VHDL generated from our high level specifications by synthesizing it using Xilinx ISE targeting a Kintex 7 FPGA (xc7k160t-3fbg676). The synthesis results detailed in Table 8.1 show an increase in throughput



(a) Iterative Salsa20

(b) Ten Stage Pipelined Salsa20 Device

Figure 8.8: Diagrammatic views of circuits produced by ReWire in Figs. 8.5 and 8.6.

and resource utilization as we pipeline that is in line with intuitive expectations. The 10-stage pipeline and the iterative implementation are the same design core replicated tenfold. We observe a nearly tenfold increase of flip-flop usage and a notable increase in LUT usage (likely impacted by optimizations in the synthesis tools). In the 20-stage pipeline, we divide our basic unit into separate `rowRound` and `columnRound` pipeline stages. This introduces some addtional LUT usage, but doubles flip-flop (slice) usage because the number of stages in the pipeline are doubled. The maximum frequency of the 20-stage pipeline increases by approximately 1.7 times which indicates a doubling effect from doubling the pipeline with a moderate amount of overhead. These numbers demonstrate that our approach is competitive with similar work in the area of synthesizing Salsa20 [79] on modern FPGAs.

8.5.2 Testing the Iterative Salsa20 Device Automatically

We used the QuickCheck tool [80] to test the putative correctness of the relationship between the reference specification *salsa*20 and the iterative ReWire definition *sls*20*dev* (from Figs. 8.4 and 8.5, resp.). Below is a *Bool*-valued function, *test*, that takes a $W128$ nonce n as input and computes an equation. Note that the value of input stream *is* is of the form $[(High, n), (Low, undefined), (Low, undefined), \cdots]$ where *undefined* is a special "don't care" constant built-in to Haskell.

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

```
is = (High,n) : repeat (Low,undefined)
```

QuickCheck can generate random inputs to *test* and, if *test* returns *True* for each input, then QuickCheck remarks that the tests were passed; below is a transcript of running QuickCheck on this correctness condition for $sls20dev$:

```
GHCi, version 7.10.1: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Salsa20          ( Salsa20.hs, interpreted )
Ok, modules loaded: Salsa20.
*Salsa20> quickCheck test
+++ OK, passed 100 tests.
*Salsa20>
```

The correctness condition is neatly summed up in the following theorem (stated without proof):

Theorem 1 (Correctness of Iterative Salsa20)**.** *For all nonces* $n, n_0, \ldots, n_9 :: W128$*, assume input stream is has the form* $[(High, n), (Low, n_0), \cdots, (Low, n_9), \ldots]$*. Then, the following equation holds:* $salsa20\ n = nth\ 10\ (feed\ is\ sls20dev)$*.*

	LUTs	Slices	Fmax (MHz)	T (Gbit/s)
Iterative	3459	651	99.4	5.1
10 Stage	22840	6019	97.5	49.9
20 Stage	25519	12309	167.4	85.7

Table 8.1: Resource usage, Fmax, and throughput (T) of the Salsa20 algorithm as implemented and compiled in ReWire.

8.5.3 Verification of Pipelining

Lemmas

This section states the Lemmas used in proving the correctness of pipelining (Theorem 2 below). Each lemma is left unproven, although we describe the intuitive meaning of each.

Lemma 1 says that the pipelining operator is associative. The associativity of $\rightsquigarrow$ allows for "parentheses to be dropped"; i.e., $(f \rightsquigarrow g \rightsquigarrow h)$ can stand for either the right- or left-hand sides of the equation in the lemma.

Lemma 1 (Associativity)**.** *The $\rightsquigarrow$ operation is associative.*

$$f \rightsquigarrow (g \rightsquigarrow h) = (f \rightsquigarrow g) \rightsquigarrow h$$

Lemma 2 relates stages in a pipeline of devices created with `iter`. The LHS below performs f and g in succession. The RHS performs f and g in the first stage and the identity function in the second stage. N.b., the RHS is *not* identical to $iter\ (g \circ f)\ (g\ o_2)$ because the former has two stages while the latter has one.

Lemma 2. *Let g :: b -> c, f :: a -> b, o_1 :: c, and o_2 :: b. Then, we have:*

$$\texttt{iter f } o_2 \rightsquigarrow \texttt{iter g } o_1$$
$$= iter\ (g \circ f)\ (g\ o_2) \rightsquigarrow iter\ id\ o_1$$

Lemma 3 relates *feed l* with $\rightsquigarrow$ in terms of infinite streams. It gives a condition under which the pipeline may be reduced by one stage.

Lemma 3. *Let l be an infinite stream and φ :: Dev i o, then: feed l $(\varphi \rightsquigarrow iter\ id\ o) = o$: feed l φ*

Lemma 4 characterizes the interaction of *feed* and *iter* in terms of a stream recording the outputs of device argument to *feed*. The first is just the initial output of one stage pipeline device (*iter f o*) and the rest are simply *f mapped* onto *l*.

Lemma 4. *Let* $l :: [i]$ *be an infinite stream and* $f :: i \to o$. *Then, feed* l $(iter\ f\ o) = o : map\ f\ l$.

Correctness Theorem

The following theorem says that *feed*ing an n-stage pipeline a stream of inputs is the same as mapping a composite function across those inputs, as long as the first n outputs are ignored.

Theorem 2 (Correctness of Pipelining)**.** *Assuming that* $f = f_1 \circ \cdots \circ f_n$ *and that* l *is an infinite stream, then:*

$$\texttt{map f l} = drop\ n\ (feed\ l\ (iter\ f_n\ o_n \rightsquigarrow \cdots \rightsquigarrow iter\ f_1\ o_1))$$

□

First, define: $F_0 = id$ and $F_{i+1} = F_i \circ f_{i+1}$. Observe that, by Lemmas 1 and 2 ($\texttt{n}-1$ times),

$$\begin{aligned}
&\texttt{iter f}_\texttt{n}\ \texttt{o}_\texttt{n} \rightsquigarrow \cdots \rightsquigarrow \texttt{iter f}_1\ \texttt{o}_1 \\
&\quad = \texttt{iter F}_\texttt{n}\ (\texttt{F}_{\texttt{n}-1}\ \texttt{o}_\texttt{n}) \\
&\qquad \rightsquigarrow \texttt{iter id}\ (\texttt{F}_{\texttt{n}-2}\ \texttt{o}_{\texttt{n}-1}) \\
&\qquad \rightsquigarrow \cdots \\
&\qquad \rightsquigarrow \texttt{iter id}\ (\texttt{F}_0\ \texttt{o}_1) \\
&\{f = F_n, F_0 = \texttt{id}\}
\end{aligned}$$

$$
\begin{aligned}
= \quad & \texttt{iter}\ \texttt{f}\ (\texttt{F}_{\texttt{n}-1}\ \texttt{o}_{\texttt{n}}) \\
& \quad \rightsquigarrow \texttt{iter}\ \texttt{id}\ (\texttt{F}_{\texttt{n}-2}\ \texttt{o}_{\texttt{n}-1}) \qquad (\ddagger) \\
& \quad \rightsquigarrow \ldots \\
& \quad \rightsquigarrow \texttt{iter}\ \texttt{id}\ \texttt{o}_1
\end{aligned}
$$

Working from the RHS of the theorem statement:

$$drop\ n\ (feed\ l\ (iter\ f_n\ o_n\ \rightsquigarrow \cdots \rightsquigarrow\ iter\ f_1\ o_1))$$

{By (‡)}

$$= drop\ n\ (feed\ l\ \left(\begin{array}{l} \texttt{iter f}\ (\texttt{F}_{\texttt{n}-1}\ \texttt{o}_{\texttt{n}}) \\ \quad \rightsquigarrow\ \texttt{iter id}\ (\texttt{F}_{\texttt{n}-2}\ \texttt{o}_{\texttt{n}-1}) \\ \quad \rightsquigarrow\ \cdots \\ \quad \rightsquigarrow\ \texttt{iter id o}_1 \end{array}\right))$$

{ Lemma 3, $\texttt{n}-1$ times }

$$= drop\ n\ (o_1 : \cdots : F_{n-2}\ o_{n-1} : feed\ l\ (iter\ f\ (F_{n-1}\ o_n)))$$

{ (†), Section 8.3.1 }

$$= drop\ 1\ (feed\ l\ (iter\ f\ (F_{n-1}\ o_n)))$$

{ Lemma 4 }

$$= drop\ 1\ (F_{n-1}\ o_n\ :\ map\ f\ l)$$

{ Defn. `drop` }

$$= map\ f\ l$$

8.6 Summary and Conclusions

This paper considered the provably correct development of several reconfigurable designs and implementations of the Salsa20 stream cipher. The vehicle for this development is the ReWire language. ReWire is a sublanguage of the pure, functional language Haskell, and, as such, possesses a rigorous semantics that supports formal verification. Functional languages are generally quite expressive, and, consequently, the Salsa20 specifications in ReWire were quickly produced, concise and comprehensi-

ble, and elegant. Connect Logic—a previously unpublished part of ReWire—supports a structural style of development in a functional HDL. Connect Logic was key to rapidly prototyping Salsa20 in ReWire, especially in the introduction of pipelining optimizations to the specifications.

It is commonplace for hardware engineers to "think in diagrams". Any circuit or device specification will include a diagram depicting the high-level structure of the device. This diagram domain abstraction is used as an informal guide for comprehending the design. But how do we express such structural notions in a functional language-based HDL like ReWire? To this end, we introduced an extension to ReWire called Connect Logic, that encapsulates the diagrammatic style directly in the syntax of ReWire. This paper defines Connect Logic and illustrates its use with a case study of the construction of an efficient, pipelined hardware design and implementation of the Salsa20 stream cipher. Furthermore, and more to the point, we verify the correctness of this device through equational reasoning on the ReWire source text.

New language abstractions are not typically cost free. There is usually some trade-off with respect to performance and language implementers attempt to minimize such overheads. Furthermore, new abstractions tend to be more useful in some situations than in others. The Salsa20 cipher was chosen as a test for ReWire to evaluate (1) how well cryptographic algorithms might be expressed in ReWire and (2) what performance trade-off, if any, might arise with respect to carefully hand-optimized implementations? The performance of the synthesized ReWire devices (as shown in Table 8.1) was quite good and, although there are not any published numbers on hand-optimized implementations of Salsa20 that afford direct comparison with our results, the achieved performance was in line with the only relevant publication in the

area [79]. Question (1) concerns what is, admittedly, more of an aesthetic issue than a measurable quantity. Still, it is safe to say that the Salsa20 specifications in ReWire would be readily comprehensible to those with experience in functional programming.

More importantly, a clear advantage of the ReWire methodology is that the artifacts we produced were verified in the manner of ordinary functional programs directly on the text of the design. This is a point worth emphasizing: verification of ReWire programs takes place *on the program itself*. Because VHDL has no mathematical semantics, artifacts produced in VHDL (or in Verilog for that matter) would require an additional step in which the formal specification of the device would encoded by hand in the logic of a theorem prover [48]. This hand-encoding is fraught with the potential for error as well as being quite time-consuming.

Chapter 9

Case Study: Implementing a Pipelined DLX Processor

9.1 Introduction

In this chapter we detail the design and implementation of the DLX microprocessor using ReWire with Connect Logic. Prior work has demonstrated ReWire's viability as a tool to design and implement processors [3]. In this case, we choose a pipelined architecture to illustrate Connect Logic's usefulness in building pipelined devices in the manner of those seen in processor architectures. This chapter describes the implementation from the high level and how we arrive at the resulting device illustrated in Listing 9.1.

```
1 dlx :: ReT (Data, Instruction) (Address, Maybe Data, PC) I ()
2 dlx = refold output_select wiring
```

```
3           (fetch <&> decode <&> execute <&> memory <&> writeback)
```

Listing 9.1: The type of the DLX processor device

9.2 The DLX Processor

The DLX processor architecture is a micro-architecture designed by Hennessy and Patterson for didactic purposes in their seminal book on micro-architecture design [81]. The DLX architecture is similar in form to the MIPS processor family. DLX is intended to be implemented as a pipelined processor with five processor phases: fetch, decode, execute (ALU), memory access, and register write back. We design the stages as separate ReWire components in near isolation from one another to demonstrate the modularity that Connect Logic brings to VLSI design in ReWire. The DLX specification, while less common than more popular processor architectures (MIPS, POWER, x86, etc.) has a large enough community to provide us with mature tools to program our processor. We use these tools to demonstrate that our specification has correct behavior in testing in Haskell. From there, we move to converting to ReWire and synthesizing to a Xilinx FPGA to report resource utilization and performance characteristics.

9.3 Constructing the Processor

The specification of a physical process follows primarily from its instruction set. The features that we need to support in the instruction set translate to physical features in our final product. We do this by building each processor phase individually, giving

consideration to the instructions we support. Finally, we combine these isolated components into DLX using Connect Logic.

9.3.1 Instructions and Architecture

The DLX processor is a reduced instruction set computer (RISC) consisting of thirty-two, 32-bit registers. These registers include two special-purpose registers. The first register `R0` is a zero constant value and is read only. The last register `R31` is intended for use as a return address register for procedure calls [82] and similar routines. In this work, we treat DLX as a big-endian processor in the spirit of Sailer et al. [82]. The DLX architecture includes a number of logic, arithmetic, control-flow, and data-flow-affecting instructions. For brevity, this implementation covers a representative subset of the DLX architecture. Supported instructions are listed in Table 9.1, Table 9.2, and Table 9.3. Specifically it excludes all floating-point operations, but it also only includes a subset of the remaining instructions as well.

DLX instructions are classified into three different types. The first type, the R-type instruction, is the type of instructions that utilize register-to-register functionality in the ALU stage of the processor. These instructions include a function opcode and three register opcodes: two source registers and one destination register. Tables 9.1-9.3 list the instructions supported by our DLX implementation with their opcode values in addition to their semantics. Source and destination register variables are denoted by rd and rs_x respectively. Immediate values provided by a programmer in an encoded instruction are denoted by *immediate*. The *extend* function is a sign-extension operation. The *MEM* keyword serves as a C-like array interface to system memory in our semantics.

The selection of R-Type instructions supported by our DLX implementation are listed in Table 9.1. I-type instructions are similar to their R-type cousins, except they contain one fewer source register and reserve 16-bits for an immediate value for a programmer to determine in advance. I-type functions contain immediate-argument variations of R-type instructions as well as jumping, branching, and memory access instructions. The final group of instructions, the J-Type instructions, are a smaller group of unconditional jumping instructions. These are listed in Table 9.3. This format consists of an opcode and an address for a jump target. This is the smallest set of instruction types for the DLX ISA.

Instruction	**Opcode**	**Semantics**
add	0x20	$rd \leftarrow (rs_1) + (rs_2)$
and	0x24	$rd \leftarrow (rs_1)\ \&\ (rs_2)$
or	0x25	$rd \leftarrow (rs_1) \parallel (rs_2)$
seq	0x28	$rd \leftarrow (rs_1) = (rs_2)\ ?\ (0^{31} \parallel 1) : (0^{32})$
sle	0x2C	$rd \leftarrow (rs_1) \leq (rs_2)\ ?\ (0^{31} \parallel 1) : (0^{32})$
sll	0x04	$rd \leftarrow (rs_1)[(rs_2\ \%\ 8) : 31] \parallel 0^{rs_2\%8}$
slt	0x2A	$rd \leftarrow (rs_1) < (rs_2)\ ?\ (0^{31} \parallel 1) : (0^{32})$
sne	0x29	$rd \leftarrow (rs_1) \neq (rs_2)\ ?\ (0^{31} \parallel 1) : (0^{32})$
sra	0x07	$rd \leftarrow (rs_1[0])^{rs_2\%8} \parallel rs_1[0 : (31 - (rs_2\ \%\ 8))]$
srl	0x06	$rd \leftarrow 0^{rs_2} \parallel rs_1[0 : (31 - (rs_2\ \%\ 8))]$
sub	0x22	$rd \leftarrow (rs_1) - (rs_2)$
xor	0x26	$rd \leftarrow (rs_1) \oplus (rs_2)$

Table 9.1: DLX R-Type instructions encoding and semantics.

9.3.2 Fetch

The fetch phase of the execution pipeline of the DLX is the first phase. This phase of the pipeline facilitates requests with the program counter for memory reads from

Instruction	Opcode	Semantics
addi	0x08	$rd \leftarrow (rs_1) + immediate$
andi	0x0C	$rd \leftarrow (rs_1)\ \&\ immediate$
beqz	0x04	$PC \leftarrow (rs_1 = 0\ ?\ extend(immediate) + 4 : 4)$
bnez	0x05	$PC \leftarrow (rs_1 \neq 0\ ?\ extend(immediate) + 4 : 4)$
jalr	0x13	$R31 \leftarrow (PC + 8); PC \leftarrow rs_1$
jr	0x12	$PC \leftarrow rs_1$
lhi	0x0F	$rd \leftarrow (immediate \parallel 0^{16})$
lw	0x23	$rd \leftarrow MEM[rs_1 + extend(immediate)]$
ori	0x0D	$rd \leftarrow rs_1 \parallel immediate$
seqi	0x18	$rd \leftarrow rs_1 = extend(immediate)\ ?\ 1 : 0$
slei	0x1C	$rd \leftarrow rs_1 \leq extend(immediate)\ ?\ 1 : 0$
slli	0x14	$rd \leftarrow (rs_1)[(immediate\ \%\ 8) : 31] \parallel 0^{immediate\%8}$
slti	0x1A	$rd \leftarrow (rs_1) < immediate\ ?\ (0^{31} \parallel 1) : (0^{32})$
snei	0x19	$rd \leftarrow (rs_1) \neq immediate\ ?\ (0^{31} \parallel 1) : (0^{32})$
srai	0x17	$rd \leftarrow (rs_1[0])^{immediate\%8} \parallel rs_1[0 : (31 - (immediate\ \%\ 8))]$
srli	0x16	$rd \leftarrow 0^{immediate} \parallel rs_1[0 : (31 - (immediate\ \%\ 8))]$
subi	0x0A	$rd \leftarrow (rs_1) - immediate$
sw	0x2B	$MEM[rs_1 + extend(immediate)] \leftarrow rd$
xori	0x0E	$rd \leftarrow (rs_1) \oplus immediate$

Table 9.2: DLX I-Type instructions encoding and semantics.

Instruction	Opcode	Semantics
j	0x02	$PC \leftarrow PC + extend(value)$
jal	0x03	$R31 \leftarrow PC + 4; PC \leftarrow rs_1$

Table 9.3: DLX J-Type instructions encoding and semantics.

program memory. The fetch stage receives an instruction word from the program memory each cycle. In our design we assume a one-cycle read time from program memory. Every cycle the fetch phase can receive a new instruction to feed forward. For example, given clock cycle at time t_n, the instruction received by the fetch component corresponds to the instruction at the program counter corresponding to the program counter at $t_{(n-1)}$ (or PC_{n-1}). The types for the fetch phase are illustrated in Listing 9.2. The fetch component feeds forward the instruction from memory to the decode phase in addition to an updated program counter (`NextInst`) and the current program counter to the decode phase (`PC`). We note that the fetch component accepts an additional value of type `NewAdd` as input. This value represents new values to assign to the program counter in the event of a jump or branch processed further down the instruction pipeline. We omit the definition of this device in this section (the code for the implementation is in the Appendix beginning on Page 187), but note that if the value of type `NewAdd` is `Nothing` then the program counter is incremented. If it is not `Nothing` the program counter is set to the argument of the `Just`. Lastly we note that the first member of the 3-tuple input argument to the fetch device is a stall bit. All components up to and including the execute stage include a stall to support stalling messages from the memory access component. The fetch component requires two IO connections that route outside of the processor. The fetched instruction comes from a program memory bank outside of the processor and the address of the next instruction is to be fed to a program memory module for a read for the subsequent instruction fetch.

```
1 type Instr   = Vector32 Bit
2 type NewAdd  = Maybe (Vector32 Bit)
3 type PC      = Vector32 Bit
```

```
4 type NextInst = Vector32 Bit
5 type Stall    = Bit
6 type FetchI = (Stall,Instr,NewAdd)
7 type FetchO = (NextInst,Instr,PC)
8
9 fetch :: ReT FetchI FetchO I ()
```

Listing 9.2: Types for the fetch component given in Haskell

9.3.3 Decode

The decode phase of the DLX pipeline is responsible for decoding instructions as they appear in 32-bit binary form to a form that subsequent processor stages can interpret and act on. The decode phase is responsible for loading source register values from the register file. In our DLX implementation, the values of the register file are stored and managed by the writeback phase, with read-only values being fed backwards to the decode stage. The decoding component takes a stall bit, the un-decoded instruction as a `Vector32 Bit`, values comprising the register file (`RegFile`) and a register value (`RegVal`) indicating the address of the instruction (the value of the program counter when the instruction was fetched). The decoder outputs a decoded version of the instruction along with register names in Haskell types. We note that this transformation incurs a minimal cost. The DLX instruction encoding is such that microcode isn't required to represent parsed instructions because there are as many registers as can be represented by a 5-bit register encoding, as well as instructions represented by a 6-bit ocpode encoding. When expressing these in as algebraic data types in ReWire, the compiler re-encodes the constructorsin a form represented by

just as many bits so long as we choose to keep the same number of instruction opcodes and registers (or less). Register values require an additional 64-bits of routing to be feed forward to the ALU from the decoder. The output is expressed as a 4-tuple of the opcode, the destination register, and the first and second source registers paired with their values retrieved from the register file, respectively.

```
1 type RegFile = Vector32 (Vector32 Bit)
2 type RegVal  = Vector32 Bit
3 data Opcode  = ADD | .. | NOP --All supported opcodes
4 data Reg     = R0 | .. | R32
5 type DecodeI = (Stall,Vector 32 Bit, RegFile, RegVal)
6 type DecodeO = (Opcode,Reg,(Reg,RegVal),(Reg,RegVal))
7
8 decode :: ReT DecodeI DecodeO I ()
```

Listing 9.3: Types for the decode component given in Haskell

9.3.4 Execute

The execute component, or the ALU, is where the bulk of the computation occurs in the DLX processing pipeline. The most combinationally expensive computations such as arithmetic operations occur here and the purpose of this phase is to isolate those operations so they are at most 1-operation deep. Other phases include some fixed addition (fetch increments the program counter) that is combinationally separate from this phase, but pipelined as to balance out the delay to different phases and keep this separate work executing in parallel.

```
1 type Flush = Bit
2 type ExecuteI = (Stall,Opcode,Flush,Reg,RegVal,RegVal)
```

```
3 type ExecuteO = Maybe (Opcode, RegDest, RegVal, RegVal)
4 execute :: ReT ExecuteI ExecuteO I ()
```

Listing 9.4: Types for the execution (ALU) phase given in Haskell.

On line 2 of Listing 9.4 the input type of `execute` is specified by the first argument of `ReT` as a 6-Tuple. The execute phase component takes a stall bit, opcode, flush bit, destination register for writebacks, and two source register values. The opcode determines which functionality we execute in the ALU. The register values are used according to the semantics of the instruction corresponding to the opcode. Two values are supplied, but are not always used and are sometimes meaningless (zero) values. Likewise, destination registers are not required by all instructions and are ignored by some instructions. The flush bit indicates whether or not a branch in the subsequent memory phase has been evaluated to be taken. In this case, the execution phase accepts the number of instructions remaining in the pipeline equivalent to the number of delay slots in our implementation and ignores the remaining ones until receiving the branch target instruction.

9.3.5 Memory

The memory access phase of the DLX pipeline is responsible for interfacing with the system memory external to the processor. In this case we define the memory as a RAM with an output 32-bit data bus, an input 32-bit data bus, and an input 32-bit address bus with an additional signalling bit for specifying a read or a write.

```
1 type Data    = Vector32 Bit
2 type Address = Vector32 Bit
3 type MemI = (Data,Maybe (Opcode,Reg,RegVal,RegVal))
```

```
4 type MemO = (Maybe Data, Address, Stall, Flush,
5                Maybe (Reg, RegVal), Maybe PC)
6 mem :: ReT MemI MemO I ()
```

Listing 9.5: Types for the memory access phase given in Haskell.

The input type to the memory access phase is represented in Listing 9.5 on line 3 by `MemI` consisting of a tuple of the output data bus from a memory module as well as a potential instruction to process. The memory unit acts upon memory access instructions, but is also responsible for calculating jumps and branches. Instructions not within these two groups are fed forward to the writeback phase of the pipeline in the same way as NOP instructions.

The output from the device is specified on lines 4-5 of Listing 9.5 is a 6-tuple consisting of a number of signals and data outputs going to memory and every other component in the pipeline. The first argument is a value of type `Maybe Data`. This argument encapsulates reading and writing. If it is `Nothing` this signifies a read operation, as no data is sent to the attached memory unit. If the argument is a `Just`, its argument is considered the data to be written to memory. Encoding a `Maybe` is equivalent to using a separate read/write signalling bit because ReWire only needs one bit to encode the two different constructors of a `Maybe` type which are appended to the encoding of the type variable (`Data` here) of `Maybe`. The second argument to the output tuple is the address to be read or written to. The `Stall` and `Flush` bits are signals to the previous devices to stall or drop current work. A high stall bit indicates that work should be held. A high flush bit indicates a branch is to be taken and all work aside from the delay slot should be ignored. This is performed by the other devices inserting yielding the work equivalent to a NOP instruction when

receiving a signal to flush. The fifth argument is a register and register value pair. When this value is populated (not `Nothing`), the writeback stage will save this value to the indicated register in the register file. The final argument is the value to set the program counter to in the event of a jump or branch. If the memory access unit encounters a jump or a branch that is to be taken, this value is populated with the branch or jump target.

The memory component of the pipeline assumes that a read instruction will require two clock cycles to complete. This is one cycle to signal the memory unit and one cycle to receive the result. We choose this form because it is both reasonable and amenable to expression using reactive resumptions in ReWire with `signal` i.e. three signals, two with stalled (NOP) outputs and the third with the output containing the read result from memory. The delay for a write instruction is one cycle. This is less than a read because the processor doesn't need to wait on a result from the memory to continue processing. The implementation of our processor is not dependent upon the admittedly high requirements we place on our memory modules with regards to speed. If we were to change the number of cycles required to wait on loads and stores, the only changes we would need to make to our processor would be in the number of cycles to signal stalls.

9.3.6 Writeback

The writeback phase of the DLX pipeline is responsible for managing updates to the register file. In our implementation, the writeback receives a value that is populated by a register-value pair as input. This pair indicates what value to assign to the specified register as an update (if the value is not `Nothing`). We note again that the

register R0 is fixed to a zero value and cannot be altered.

```
1 type WriteBackI = Maybe (Reg, RegVal)
2 type WriteBackO = RegFile
3 writeback :: ReT WriteBackI WriteBackO I ()
```

Listing 9.6: Types for the writeback phase given in Haskell.

9.4 Composing the DLX Processor

Our DLX processor implementation is composed of the subcomponents described in the previous section. We construct the whole processor by using the Connect Logic primitives `refold` and `parI`. We begin by specifying the processor's type in Listing 9.7.

```
1 dlx :: ReT (Data, Instruction) (Address, Maybe Data, PC) I ()
```

Listing 9.7: The type of the DLX processor device

The `dlx` device is our top level device that we ultimately want to synthesize. For this reason, we keep the types of the inputs, outputs, and failure (here we use unit, failure is not used) monomorphic even though failure could remain polymorphic. The inputs of our DLX processor are a pair consisting of `Data` values from memory read operations and `Instruction` values from program memory for instruction fetches. The output of the processor is a triple consisting of an `Address` for memory operations, `Maybe Data` signalling a potential read or write with a value, and a program counter value for reading the next fetched instruction from program memory. This type outlines the goal of what remains of our work. We need to compose our subcomponents in a way that is true to the behavior of the DLX specification as well as following the type

laid out at the top level. The following subsections illustrate the necessary steps to reach this end.

9.4.1 Parallelizing and Connecting Devices

The first step is to construct an intermediate device that consists of all components placed in parallel. We do this by using the parallel operator and we illustrate this step in Listng 9.8.

```
1  type InterI = (
2                  (Stall,Instr,NewAdd),   --Fetch
3                  (Stall,Instr,RegFile,RegVal), --Decode
4                  (Stall,Opcode,Flush,RegDest,RegVal,RegVal), --Exec
5                  (Data, Maybe (Opcode,Reg,RegVal,RegVal)), --Mem
6                  Maybe (Reg,RegVal) --Writeback
7                )
8  type InterO = (
9                  (NextInst,Instr,PC), --Fetch
10                 (Opcode,Reg,(Reg,RegVal),(Reg,RegVal)), --Decode
11                 (Maybe (Opcode, RegDest, RegVal, RegVal)), --Execute
12                 (Maybe Data,Address,Stall,
13                  Flush,Maybe (Reg,RegVal), Maybe PC), --Memory
14                 RegFile --Writeback
15               )
16 dlx_inter :: ReT InterI InterO I ()
17 dlx_inter = fetch   <&> decode
18             execute <&> memory <&> writeback
```

Listing 9.8: Constructing the intermediate ReWire device.

On lines 1-13 of Listing 9.8 show the input types of the combined devices using the parallel operator. We illustrate the types as flat tuples, which are isomorphic to the nested structure that would be produced by subsequent applications of the parallel combinator. The combined `dlx_inter` device is akin to an un-wired device that has output and input ports open, but unconnected. We connect these ports in the next steps by defining connecting functions for each device based on the combined device type and the types of the top level DLX processor defined earlier.

```
1  type DLXI = (Data, Instruction)
2  type DLXO = (Address, Maybe Data, PC) I ()
3
4  connFetch   :: InterO -> DLXI -> (Instr,NewAdd)
5  connDecode  :: InterO -> DLXI -> (Instr, RegFile, RegVal)
6  connExecute :: InterO -> DLXI -> (Opcode, Flush,RegDest,RegVal,RegVal)
7  connMem     :: InterO -> DLXI ->
8                 (Data, Maybe (Opcode,Reg,RegVal,RegVal))
9  connWB      :: InterO -> DLXI -> RegFile
10
11 dlxOut      :: InterO -> DLXO
12
13 dlx :: ReT DLXI DLXO I ()
14 dlx = refold
15         dlxOut
16         (\interO -> \dlxI -> let f = connFetch interO dlxI
17                                  d = connDecode interO dlxI
18                                  e = connExecute interO dlxI
19                                  m = connMem interO dlxI
20                                  w = connWB interO dlxI
21                              in (f,d,e,m,w))
```

```
22          dlx_inter
```

Listing 9.9: Connective functions for each pipelining phase of the DLX processor.

We re-introduce the input and output types of the top level DLX device as `DLXI` and `DLXO` on lines 1 and 2 of Listing 9.9. These types, along with the intermediate `dlx_inter` output types, appear in the typing of the five different connecting functions on lines 4-9. The definitions of these functions are omitted. Trivial connection functions construct their output by selecting the corresponding members of the input provided to them. No additional computation is performed by a trivial connection function and they are akin to "wiring". We will discuss particular non-trivial functions in a later section and note here that the non-trivial functions are different from trivial ones because they facilitate register value forwarding to account for data flow hazards in the pipeline. The `dlxOut` function on line 11 selects the values from the `InterO` type to comprise the output for the whole processor. These are the address for the memory address bus, the read/write `Maybe Data` type and the program counter for reading the next instruction from program memory.

The top level device representing the whole processor is the `dlx` definition that appears on lines 13-22. We `refold` over the `dlx_inter` device by providing the output selection function `dlxOut` as the output modification function. An anonymous function constructs the input for `dlx_inter` by using each connection function to first construct the input for each individual stage in the `let` bindings on lines 16-20. The input is the tuple constructed on line 21.

9.4.2 Considering and Mitigating Pipelining Hazards

Pipelined architecutres do not come without drawbacks. Aside from the latency introduced by multiple pipeline stages, the more critical issue that arises from pipelining a processor comes in the form of hazards. We describe the applicable hazards to our implementation as they appear in the Hennesy and Patterson text [81] and discuss our remedies to these hazards in our implementation as well as how they relate to the connection functions described in Listing 9.8.

Data Hazards

Data flow hazards arise when an earlier stage of the pipeline depends on a value produced at a later stage, but that value hasn't yet reached the register file via the writeback (final) phase of the pipeline. The hazard creates a window of subsequent instructions in the pipeline that require a "forward glance" to the results of their prior instructions at different phases. This is called *register forwarding* [81].

```
1  fwd2 :: Maybe (Opcode, RegDest, RegVal, RegVal) ->
2          Maybe (Reg, RegVal) -> (Reg, RegVal) -> RegVal
3  fwd2 aluo memo otro = case aluo of
4              Nothing            -> fwd1 memo otro
5              Just (_,frd,frv,_) -> case otro of
6                                      (rd,rv) -> case frd == rd of
7                                                    H -> frv
8                                                    L -> fwd1 memo otro
9
10 fwd1 :: Maybe (Reg, RegVal) -> (Reg,RegVal) -> RegVal
11 fwd1 memo otro = case memo of
12                        Nothing -> (snd otro)
```

```
13                          Just (frd,frv) -> case otro of
14                                              (rd,rv) -> case frd == rd of
15                                                           H -> frv
16                                                           L -> rv
17 connExecute  :: InterO -> DLXI ->
       (Stall,Opcode,Flush,RegDest,RegVal,RegVal)
18 connExecute (_,(dcOp,dcDreg,regA,regB),
19                aluO,_,_,stall,flush,mbWbReg,_,_) _ =
20                                                    (stall,dcOp,flush,dcDreg,
21                                                     fwd2 aluO mbWbReg regA,
22                                                     fwd2 aluO mbWbReg regB)
```

Listing 9.10: Functions for forwarding register values.

Listing 9.10 illustrates our approach for forwarding written register values not yet written to the register file back to the `execute` phase of the processor pipeline. We utilize two functions `fwd2` and `fwd1` to facilitate forwarding. Values for assignment are computed at the execute or memory phases of the pipeline by either computation from the ALU or reads from memory. The freshest (or most recently written) place to find a potential value of a register is in the output of the execute phase. The second freshest place is the output of the memory access phase. Lowest priority of register freshness is given to the register file, which is read at the decode phase.

Our forwarding functions work by checking whether or not either of the register-value pairs emitted by the decoder overlap with a register assignment emitted by the `execute` phase. If this is the case, we replace the incoming register value from the decoder with the value emitted by `execute` phase. This is performed in `fwd2`. If there isn't an overlap with the output of the execution phase, we check for an overlap with the output of the memory access phase in `fwd1`. If neither functions discover a

match, the value from the decode phase is fed through. Note that after the forwarding test, source register information is no longer required and none of the subsequent components accept it as input. This saves us resources and reduces the complexity of post-decode phases as well as reduces resource utilization post-synthesis.

Control Flow Hazards

Control flow hazards occur in pipelined architectures when branching occurs, but the pipeline is saturated with instructions that occur after the branching instruction and shouldn't be executed. Handling a control flow hazard is in essence making sure that these instructions do not result in an effect on the machine. In our processor implementation, we manage this by directing the `execute` phase to replace these invalid instructions in the pipeline with something equivalent to a NOP. This is, in essence a small state machine that states that when the `Flush` bit (output from the `memory` phase) is high, the `execute` phase will drop instructions that are invalid given a branch.

9.4.3 Delay Slot Implementation

We don't discard *all* extra instructions in the pipeline, however. One instruction after a branch in DLX is considered a valid instruction known as the *delay slot* instruction. Listing 9.11 demonstrates a section of DLX assembly code that shows an instruction i_0 occupying the delay slot on line 3.

```
1 START:
2     BEQZ r0, TARGET ;branchIns
3     i0
```

```
4       i1
5       i2
6       invalidFetch
7       ...
8   TARGET:
9       validFetch0
10      validFetch1
11      validFetch2
12      ...
```

Listing 9.11: DLX assembly code illustrating the the appearance of a delay slot instruction on line 3.

A timing diagram in Figure 9.1 illustrates the execution of the code in Listing 9.11 in our the DLX pipeline. The lines represented are the clock `CLK`, the various input lines to each execution phase of the processor, and a selected flushing bit signal `FLUSH`. The state of the processor can be interpreted by viewing the timing diagram in a column-wise fashion. We abstract the particular values of each input in this diagram. The inputs of each execution phase represent an instruction as it moves along the pipeline. The diagram begins at the first upward clock cycle with the branch instruction reaching the input of the `memory` execution phase as its input. In the subsequent clock cycle, it has been determined that the branch is to be taken, so the `FLUSH` signal is raised to high, notifying the execution phase that the pipeline currently contains invalid instructions and to "drop" the next three instructions it receives on the pipeline. The execution phase of the processor inserts `NOP` instructions in the pipeline for subsequent pipeline phases. This ensures that no effects will occur from the invalid instructions currently in the pipeline. This process is referred to by

Hennesy and Patterson as "bubbling" [81].

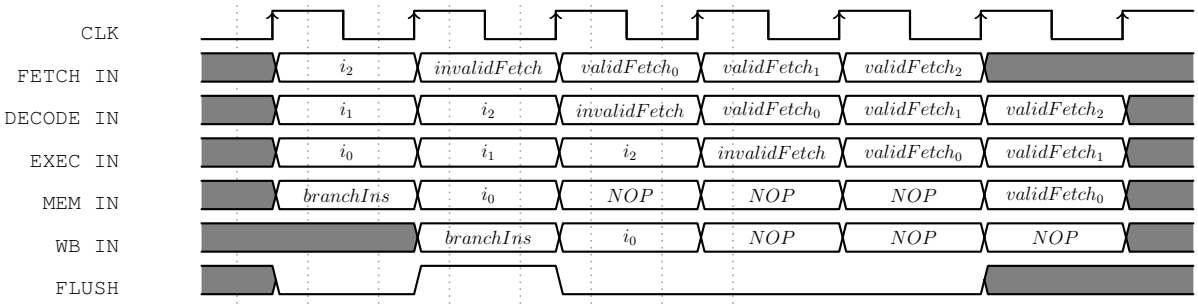


Figure 9.1: Timing diagram illustrating branching and delay slots. Pipeline state can be read column-wise.

Instruction i_0, however, has already proceeded past the `execute` device before bubbling begins. If we did not have a delay slot, the `memory` phase would have to be extended to ignore the instruction in this circumstance. In the case of a single delay slot instruction for pipelined processors, it appears that allowing for a single delay slot actually decreases pipeline complexity! Indeed this prevents us from having to alter the design of our memory access phase in our implementation, and only focusing on adding extra states to the execute phase.

```
1 exec_step :: (Stall,Opcode,RegDest,RegVal,RegVal) ->
2               ReT (Stall,Opcode,RegDest,RegVal,RegVal)
3                   (Maybe (Opcode, RegDest, RegVal, RegVal)) I ()
4 exec_step inp = case inp of
5                     (H,_,_,_,_,_) -> do
6                                        signal Nothing
7                                        signal Nothing
8                                        vinp <- signal Nothing
9                                        exec_step vinp
```

```
10                    ...non-flushing cases ...
```

Listing 9.12: Haskell code for flushing the pipeline in the execution phase of the ReWire DLX processor implementation.

Adding flushing or bubbling functionality to our `execute` phase is a fairly straightforward process. Listing 9.12 shows the code of our execution stepping function that checks whether or not the flush bit is high. If the bit is high, then we `signal Nothing` three times using the input provided by the third signal to continue evaluation. We note that adding this type of stalling functionality adds three "dead states" to state machine that comprises the `exec` device. This comes at a cost of increased flip flops (potentially) and circuit complexity to facilitate these additional state transitions. We localize this additional complexity to one device instead of three other stages to simplify the code and types for the other two devices.

9.4.4 Stalling Functionality

Certain DLX instructions take longer than one cycle to execute at a single stage. In our implementation, these instructions are memory access instructions at the `memory` device. Read instructions take two cycles to complete, for example. To ameliorate the *structural hazard* of resource contention on the memory device, we must stall the pipeline so subsequent instructions cannot access the device while it is still in use by a previous instruction. We accomplish this by adding stall support in each processor phase prior to the `memory` phase by adding output memoization to each device's stepping function.

```
1 fetch_step    :: FetchO -> FetchI -> ReT FetchI FetchO I ()
2 fetch_step o i = case i of
```

```
3                     (H,_,_) -> do
4                                 i' <- signal o
5                                 fetch_step o i'
6                     ..non-stalling code..
7
8  decode_step  :: DecodeO -> DecodeI -> ReT DecodeI DecodeO I ()
9  decode_step o i = case i of
10                    (H,_,_,_) -> do
11                                 i' <- signal o
12                                 decode_step o i'
13                    ..non-stalling code..
14
15 execute_step :: ExecuteO -> ExecuteI -> ReT ExecuteO ExecuteI I ()
16 execute_step o i = case i of
17                    (H,_,_,_,_,_) -> do
18                                 i' <- signal o
19                                 execute_step o i'
20                    ..non-stalling code..
```

Listing 9.13: Stepping functions with output memoization for stalling.

We refer to stepping functions as functions that allow us to accept input and transition (or step) to another state in a state machine. This is similar to the kind of transitions seen in Moore machines. Stepping functions for Reactive Resumptions of the type `ReT i o m a` have the type `i -> ReT i o m a`. For stalling devices, we require an extra argument to the stepping function to remember the previous cycle's output when evaluating the current cycle's input for a stall. If a stalling device receives a signal to stall, the previous cycle's output is re-used and saved for the next cycle (using it again as the first argument to our memoizing stepping function) where we evaluate

the next cycle's input for a stall. The stalling function of each stalling portion of our DLX implementation appears in Listing 9.13. In each definition, there is a case for acting on a stall signal. We omit non-stalling cases. In each case, we `signal` with the previous cycle's output (saved as the first argument) and save the output again for use in the next cycle by supplying it as an argument a tail recursive call to the stepping function.

9.5 Testing

The ReWire language is a subset of the Haskell programming language and as such device specifications written in ReWire can be tested in Haskell. We validate our DLX implementation by testing it in a Haskell-hosted runtime.

9.5.1 A Haskell Test Bench

Testing ReWire devices in Haskell is akin to setting up a test bench in a popular VLSI testing suite, except that Haskell provides us with a more powerful and more expressive substrate to work with. The process of establishing a testing suite for a ReWire device begins with the type of the device "under test".

```
1 import qualified Data.ByteString as BS
2 import Data.Array.IO
3
4 --Treating I as IO instead of Identity for tests
5 type I = IO
6
7 --A reactive run function
```

```
runReacT :: Monad m => ReacT input output m a ->
                       (output -> m input)    -> m a
runReacT = ...

--Top level DLX device
dlx  ::  ReT (Data,Instruction)  (Address,Maybe Data,PC)  I  ()
dlx = ...

--A Memory stepping function
memory :: (Address,Maybe Data) -> IOUArray Int32 Word8 ->
          ReT (Address,Data) (Vector32 Bit) I ()
memory = ...

mem4096 :: ReT (Address,Maybe Data) (Vector32 Bit) IO ()
mem4096 = do
              arr <- lift (newArray (0,4096) 0)
              inp <- signal 0
              memory inp arr

loadMem :: FilePath -> ReT (Address,Maybe Data) (Vector32 Bit) IO ()
loadMem f = do
                 bs <- lift (BS.readFile f)
                 let bs' = BS.unpack bs
                 arr <- lift (newListArray (0::Int32,fromIntegral
    (length bs' - 1)) bs')
                 inp <- signal 0
                 memory inp arr

program :: FilePath -> ReT () RegFile IO ()
```

```
36 program f = do
37                 let prog = refold
38                                 id
39                                 (\_ i -> (i,Nothing))
40                                 (ramFromFile f)
41                 let paired = parI mem65535 (parI (prog) (dlx_testreg))
42                 refold devout devinp paired
43   where
44       devout (ramout,(progout,(nextinst,rwdata,addr,rf))) =
45          (nextinst,addr,rwdata,rf)
46       devinp (ramout,(progout,(nextinst,rwdata,addr,rf))) () =
47          ((addr,rwdata),(nextinst, (progout,ramout)))
48
49 test :: FilePath -> IO ()
50 test f = runReacT (program f) (\(regfile -> do
51                             ..analyze and report processor state here..
52                                 )
```

Listing 9.14: The top level DLX device type for testing

Listing 9.14 illustrates a test bench for the ReWire DLX implementation. To make testing with files and terminal I/O easier, we begin by swapping out the underlying Identity monad (`I`) in the ReWire DLX stack for the IO monad. Haskell utilizes the IO monad to represent side-effecting IO operations which include file operations and other standard operations like printing and reading from the command line. Our implementation makes no use of its inner monad stack because it consists of only Identity. Making this alteration for test doesn't affect any of our components. The inner monad is given by `I` which is treated as a type synonym for `IO` on line 5. The other components require for testing are memory banks for system and for program

memory. We provide the type of our run function for Reactive Resumptions on line 8. This function allows us to operate on outputs from the device during test and represent them for observation in the `IO` monad. On line 17 we provide the type of the stepping function for a memory device (definition omitted) that is readable and writable. This device is a reactive wrapper over an unboxed Haskell array to emulate a byte-addressable memory bank. The device accepts an `Address` to read or write and a `Maybe Data` specifying data to write, if any. We utilize this memory simulator for both system memory and program memory for DLX, but we only allow reads from program memory. Lines 21-25 are a small memory device for testing. The `loadMem` function on lines 27-33 produces a memory device that is populated with values from a file. This enables us to create program memory for assembled DLX programs using third-party assemblers. The `program` function on lines 35-42 creates a sealed system around our processor that accepts the `Unit` type as input and yields the state of the register value as output. We yield the register file as output here to easily inspect the values of the registers in test. The input of type `Unit` is akin to a hardware device receiving a clock pulse signalling it to step forward. The top level testing function takes a program file path as an argument and simulates a processor run and takes a user-provided inspection function that produces the next input in `IO` to test execute a device. This function is executed for every cycle of the device. In the case of this test bench, we inspect the register file at every clock cycle. More elaborate tests could select additional values for inspection including specific registers, memory locations, signals between processor stages, and any other value yielded from a reactive device contained in the processor. We stay with the top level signals in this example, but with some tweaks to the underlying implementation, a programmer can expose any

signal for top level evaluation in test.

Using Haskell as a substrate for testing devices is immensely helpful, especially when validating a device using third party tools. In this case, we utilized an assembler found online to test the behavior of the processor to ensure it complied with expected norms by observing register values when doing normal arithmetic, loading and storing to memory, and executing branches.

9.6 Synthesizing the Design

Our DLX implementation was synthesized to an FPGA by converting the specification to VHDL. The process is straightforward and we note the stages involved in synthesizing the device here.

9.6.1 Proper Compilable ReWire

ReWire that is synthesizable to hardware needs to follow the conventions established by Procter [3]. That is, all pure functions are prohibited from being recursively defined. Functions in `ReT` (called stepping functions in this chapter) are allowed to be tail recursive, but only if the tail calls are *guarded*. Guarded tail calls are tail calls following a call to `signal` in a monadic expression. Anecdotally, these restrictions, while perhaps unusual to adhere to at first, become entirely natural when writing device specifications with consideration to clocked behavior. The core ReWire language is a subset of the whole Haskell syntax. At the time of this writing, we eschew some of the more sugar-y aspects of Haskell to simplify parsing and compilation (whitespace rule, irrefutable patterns in let-bindings, Haskell-style do-notation, etc.).

It is feasible to write Haskell in this impoverished style, however. Doing so can speed the process to testing on hardware, but we acknowledge this comes at a cost to some expressive freedom at the source level.

All DLX code was written in the ReWire-style when being implemented and tested in Haskell. Migrating the code to ReWire requires a few minor tweaks, but is a fairly painless process. We took care to write the device specifications so they adhered to ReWire's guardedness and recursion requirements so these parts of the code would not need to be rewritten during the code migration process.

Connect Logic allows us a kernel of operators that act on devices typed in `ReT`. It is allowable to write non-primitive functions comprised of these calls, but before compilation, these functions must be inlined. Functions typed in `ReT` that are not called from a tail-position in a larger expression typed in `ReT` must be inlined. The ReWire compiler provides an interactive REPL phase where the user can invoke commands to inline and reduce (beta-reduce) expressions in ReWire. Functions that must be discharged through this process can be thought of as user-directed macros. Currently, this process is done by hand and automated approaches to discharging these macros is left to future work on the ReWire compiler.

9.6.2 Back end Primitives

There are many functions that, while feasible to do so in ReWire, make sense to defer to synthesis tools to generate. Such features generally include arithmetic and logical functionality. ReWire includes a `vhdl` keyword similar to Haskell's `foreign` keyword that one can use to declare the existence of VHDL functions on the back and treat them as first-class ReWire functions. FPGA synthesis tools carry significant

insight for optimizing functions (such as addition) for the targetable devices. We opt to utilize standard VHDL functions over "home-grown" ReWire implementations where possible and practical. The majority of the functionality of the arithmetic logic unit in our DLX implementation relies on VHDL standard functions. We use ReWire to faciliate how these functions are applied, but the functions themselves we treat as black boxes when composing our specifications.

9.6.3 Synthesis Results

ReWire compiled to VHDL was synthesized using the Xilinx ISE development environment. We targeted a Kintex-7 family **xc7k160t-3fbg676** chip. The results of the synthesis are given in Table 9.4.

Emphasis	**Effort**	**LUT Usage**	**Flip Flops**	**Maximum Frequency (f_{max})**
Speed	Normal	26928	7371	106.200MHz
Speed	High	26928	7371	106.200MHz
Area	Normal	26928	7367	105.076MHz
Area	High	26928	7367	105.076MHz

Table 9.4: FPGA synthesis results for our DLX implementation.

Synthesis runs were given to enumerate the combinations of high and normal effort versus speed and area emphasis. As shown in Table 9.4, the results are essentially identical in all cases. There is a slight difference between speed and area emphasis, but it is negligible. The LUT usage amounted to about 26% of the chip area and the flip flop usage utilized about 3% of the chip's available flip flops. A large amount of LUT usage is likely due to the space requirements of our arithmetic logic unit in the `execute` phase of the processor. Implementing the full DLX ISA would result in

a significant increase in area usage. ReWire currently does not utilize chip-specific features that could result in a reduction of space requirements (this includes features such as DSP slices). Work to enhance the ReWire compiler in this area will be FPGA family specific and perhaps chip specific. We leave these enhancements to future work.

9.7 Conclusion

In this chapter we demonstrate various ReWire Connect Logic features and their application to building a pipelined processor implementing the DLX ISA. Previous work by Procter [3] illustrates ReWire's applicability to processor design with a Xilinx Picoblaze implementation in ReWire. This work goes a step further by using the Connect Logic primitives to implement a fully pipelined processor architecture with support for pipeline hazard avoidance. We demonstrate that ameliorating hazards is a straightforward process that is natural when using the Connect Logic primitives to compose devices. Furthermore, this process enables developers to incorporate popular software engineering techniques such as encapsulation and data hiding in natural ways when implementing and testing DLX and its subcomponents.

Chapter 10

Summary and Future Works

10.1 Summary of Results

This dissertation delivers three results. First, we selected and implemented primitive functions in ReWire to support composition and manipulation of device-level constructs. Second, we established a principle of modularity in ReWire utilizing our first set of primitives. Lastly, we demonstrate novel design techniques utilizing ReWire and Connect Logic to implement sophisticated and efficient designs in hardware.

10.1.1 Connect Logic Primitives

In this dissertation, we introduced four primitive Connect Logic functions, `refold`, `<&>`, `iter`, and `refoldT`. These functions were implemented in Haskell and support has been added to the ReWire compiler to support the compilation of these functions as primitives. Prior to Connect Logic, functions taking Resumptions had

been disallowed. In this work, we soften this restriction with primitives that allow for significant control over structuring devices and modifying them by input and output values as well as providing a lifting for pure functions to synchronous devices and a method for controlling device stalling.

10.1.2 Modularity and Modules

Modularity in ReWire follows from Connect Logic. With Connect Logic in place, we establish a notion of modularity centered on synchronous devices or Resumptions. This idea of modularity is similar to that seen in HDLs such as VHDL or Verilog. Connect Logic allows us to compose preexisting devices with different devices, pure functions, or elevate pure functions to create complex systems that we can synthesize to hardware. Connect Logic reinforces some common best practices usually seen in high level programming languages such as encapsulation and data hiding by giving designers fine grained control over device placement and input and output routing: he or she can introduce and hide ports when composing devices for maximum flexibility and better work flow.

10.1.3 Novel Designs with ReWire and Connect Logic

This work introduces a number of novel design techniques for hardware using ReWire with Connect Logic as a medium. We use ReWire as a target language for implementing efficient pattern matching for regular expression files while giving careful consideration to resource and performance implications by applying transformations at the intermediate level. We utilize Connect Logic with ReWire to construct a pro-

cessor from pipeline stages, and implement a stalling pipeline from those stages with Connect Logic. We provide a number of implementations for important hardware and concurrency features including concurrency idioms like mutexes and semaphores as well as transformations on devices to create redundant hardware at minimal overhead to the designer. Theses techniques exhibit encapsulation and data hiding: integrating concurrency into groups of devices requires very little knowledge about the devices themselves other than they support a communication protocol with the synchronization logic. Redundancy transformations require no introspection into the devices they replicate and thus impose very little cost to the design process.

In summary, the applications demonstrated in this work demonstrate that although hardware composition with Connect Logic is a novel process, we are able to reincorporate more classical software engineering design concepts using Connect Logic.

10.2 Future Works

10.2.1 Structural Metaprogramming With Connect Logic

In this dissertation we introduce a non-primitive function called `pipeline` which is, in essence, a macro or metaprogrammatic function that we must discharge before compilation because it isn't a primitive function. In fact, the Connect Logic implementation requires that Connect Logic primitives as they are named can be the only devices that act upon Reactive Resumptions in ReWire. This limits our ability to reuse useful idioms when combining devices together. While it's fairly easy to rewrite

or inline instances of a `pipeline` function, there may exist more complicated structural idioms that we would wish to reuse. As such, it would be exceptionally useful if we could devise a way to enable functions that manipulate devices, but only do so in ways that rely on Connect Logic primitives.

Simple approaches to this problem involve simply inlining any function that makes calls to Connect Logic primitives, but this may not be enough. Work needs to be done to establish the formalisms of Connect Logic and how they extend ReWire so that we can easily express these structural macros without the need for hand-inlining them at compile time. This work could likely include syntactic analysis of Connect Logic expressions that is type-driven.

10.2.2 Network-on-Chip Paradigms

The Network-on-Chip (NoC) is a hardware design paradigm [83, 84] that emphasizes component-level reuse and reduced complexity of inter-component wiring (where in some cases component connectivity witnessed by networks on chip is infeasible without networking). ReWire provides a significant offering with regards to high assurance and ease of programming for desingers targeting HDLs. With Connect Logic, we can proceed a step further by focusing on combinators for modeling a variety of network topologies, switching, and communication between devices. We can build on previous work with equational reasoning with ReWire to provide a variety of assurances to inter-device networking on-chip. This has implications in NoC security properties, performance characteristics, and network quality-of-service (QoS) guarantees.

10.2.3 Type Level Naturals and Vectors

Circuits can be designed with great flexibility when the designer is able to abstract the size of types in designs. Extending ReWire with vector types that are parameterized by their size would be immensely useful for implementing operations that are flexible with regards to their input size (logical functions, addition, etc.). It is possible that we could implement this with an approach rooted in partial evaluation. Partial evaluation for constructing synthesizable logic for varying-length structures is particularly challenging. The result of partial evaluation needs to "construct" a functional specification that can be converted to hardware, or the compiler itself needs to be instrumented to manage how to go about this process automatically if this process is computationally possible or feasible.

10.2.4 Program Transformations for Power Consumption and Circuit Depth

Circuit power consumption and heat profile are critical factors to consider when evaluating a design. Currently, ReWire offers very little for evaluating high level specifications and how they may behave with regards to power requirements and heat generation. A method for evaluating potential power usage based on expression structure could go far to strengthening ReWire's viability as an HDL. Additionally, evaluating the depth of a behavioral ReWire specification could lead us to a method for automating circuit depth reduction by automated pipelining. This work illustrates the equivalence between pipelined and unpipelined expressions (pull equivalence). The final piece of the pipeline puzzle in a compiler would be a method for identifying points to pipline and exploit them with optimizations. Even if this process is based solely on

heuristics we believe that the performance implications are present to warrant more work in this area.

BIBLIOGRAPHY

[1] Robert E Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[2] W. L. Harrison and A. Procter. Cheap (but functional) threads. *Submitted to Journal of Functional Programming*, 2005.

[3] Adam Procter. *Semantics-Driven Design and Implementation of High-Assurance Hardware*. PhD thesis, University of Missouri, 2014.

[4] John Hughes. Why functional programming matters. *The computer journal*, 32(2):98–107, 1989.

[5] Simon L. Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[6] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[7] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[8] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM.

[9] William L. Harrison. The essence of multitasking, 2006.

[10] SP Jones. Tackling the awkward squad: monadic input/output, concurrency, execptions and foreign-language calls. *Lecture Notes for a tutorial given at Marktoberdorf Summer School*, 2002.

[11] Russel O'Connor. Io is not a monad.

[12] S. Marlow. The haxl project at facebook, 2014.

[13] Enno Scholz. *A concurrency monad based on constructor primitives, or, being first-class is not enough*. Freie Univ., Fachbereich Mathematik, 1995.

[14] Koen Claessen. A poor man's concurrency monad. *Journal of Functional Programming*, 9(03):313–323, 1999.

[15] Janis Voigtländer. *Asymptotic Improvement of Computations over Free Monads*, volume 5133 of *Lecture Notes in Computer Science*, book section 20, pages 388–403. Springer Berlin Heidelberg, 2008.

[16] E. Kmett. Free monads for less (part 1 of 3): Codensity, 2011.

[17] John W Lato. Iteratee: Teaching an old fold new tricks. *The Monad. Reader*, 16:19–35, 2010.

[18] Oleg Kiselyov. *Iteratees*, pages 166–181. Springer, 2012.

[19] E.Z. Yang. Space leak zoo, 2011.

[20] M. Snoyman. Conduit library on hackage.

[21] G. Gonzalez. Pipes library on hackage.

[22] J. Millikin. Enumerator library on hackage.

[23] Stan Liao, Steve Tjiang, and Rajesh Gupta. An efficient implementation of reactivity for modeling hardware in the scenic design environment. In *Proceedings of the 34th annual Design Automation Conference*, pages 70–75. ACM.

[24] Edward Amsden. A survey of functional reactive programming. *Unpublished*, 2011.

[25] Conal Elliott and Paul Hudak. Functional reactive animation. *Proceedings of the second ACM SIGPLAN international conference on Functional programming - ICFP '97*, pages 263–273, 1997.

[26] P. Hudak. Modular domain specific languages and tools. *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 134–142, 1998.

[27] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. *ACM SIGPLAN Notices*, 1998.

[28] Andy Gill, ygill@ittc.ku.edu, andygill@ittc.ku.edu, Tristan Bull, tbull@ittc.ku.edu, Garrin Kimmell, kimmell@ittc.ku.edu, Erik Perrins, esp@ittc.ku.edu, Ed Komp, komp@ittc.ku.edu, Brett Werling, and bwer-

ling@ittc.ku.edu. Introducing kansas lava. *Implementation and Application of Functional Languages*, pages 18–35, 2011.

[29] R. Wester, C. Baaij, and Kuper. A two step hardware design method using clash. 2012.

[30] Christiaan Baaij, c.p.r.baaij@utwente.nl, Jan Kuper, and j.kuper@utwente.nl. Using rewriting to synthesize functional languages to digital circuits. *Trends in Functional Programming*, pages 17–33, 2014.

[31] Arvind. Bluespec and haskell, 2013.

[32] A. Acosta. Hardware synthesis in forsyde. *Sweden: KTH/ICT/ETS*, 2007.

[33] Ingo Sander, er, and Axel Jantsch. Modelling adaptive systems in forsyde. *Electronic Notes in Theoretical Computer Science*, 200(2):39–54, 2008.

[34] HyoukJoong Lee, Kevin Brown, Arvind Sujeeth, Hassan Chafi, Tiark Rompf, Martin Odersky, and Kunle Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, September 2011.

[35] W. Citrin, R. Hall, and B. Zorn. Programming with visual expressions. In *Visual Languages, Proceedings., 11th IEEE International Symposium on*, pages 294–301.

[36] Viktor Massalõgin. *Visual lambda calculus*. Thesis, 2008.

[37] J Paul Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[38] John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1-3):67–111, 2000.

[39] Dan Piponi. Profunctors in haskell, 2011.

[40] H John Reekie. Visual haskell: a first attempt. Report, Citeseer, 1994.

[41] Martin Erwig. Abstract syntax and semantics of visual languages. *Journal of Visual Languages and Computing*, 9(5):461–483, 1998.

[42] M. Erwig and B. Meyer. Heterogeneous visual languages-integrating visual and textual programming. *Proceedings of Symposium on Visual Languages*, pages 318–325, 1995.

[43] Jason Hemann and Eric Holk. Visualizing the turing tarpit. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling and design*, pages 71–76. ACM.

[44] Iavor S Diatchki, Mark P Jones, and Thomas Hallgren. A formal specification of the haskell 98 module system. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 17–28. ACM, 2002.

[45] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *ACM SIGPLAN Notices*, volume 34, pages 114–125. ACM, 1999.

[46] Georgios Fourtounis and Nikolaos S Papaspyrou. Supporting separate compilation in a defunctionalizing compiler. In *SLATE*, pages 39–49, 2013.

[47] Georgios Fourtounis, Nikolaos S Papaspyrou, and Panagiotis Theofilopoulos. Modular polymorphic defunctionalization. *Computer Science and Information Systems*, (00):30–30, 2014.

[48] T. Melham. *Higher Order Logic and Hardware Verification*, volume 31 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1993.

[49] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-cc multiprocessor machine code. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 379–391, 2009.

[50] Levent Erkok, Dylan McNamee, Joe Kiniry, Iavor Diatchki, and John Launchbury. *Programming Cryptol*. Galois Inc., 2014.

[51] NIAP-CCEVS. Common criteria for information technology security evaluation part 3: Security assurance components. Technical Report CCMB-2012-09-003, National Information Assurance Partnership, September 2012. `https://www.niap-ccevs.org/`.

[52] Adam Procter, William L. Harrison, Ian Graves, Michela Becchi, and Gerard 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.

[53] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.

[54] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using FPGAs. In *Proc. of the the 9th Annual IEEE Symp. on Field-Programmable Custom Computing Machines*, pages 227–238, 2001.

[55] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proc. of the 2007 ACM/IEEE Symp. on Architecture for Networking and Communications Sys.*, pages 145–154.

[56] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *Proc. of the 2006 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 339–350, 2006.

[57] Benjamin C. Brodie, David E. Taylor, and Ron K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *2006 ISCA*, pages 191–202.

[58] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proc. of the 2007 ACM CoNEXT Conf.*, pages 1–12.

[59] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *Proc. of the 2007 ACM/IEEE Symp. on Architecture for Networking and Communications Sys.*, pages 127–136.

[60] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proc. of the 4th ACM/IEEE Symp. on Architectures for Networking and Communications Systems*, pages 50–59. ACM, 2008.

[61] Ioannis Sourdis, João Bispo, João M. Cardoso, and Stamatis Vassiliadis. Regular expression matching in reconfigurable hardware. *J. Signal Process. Syst.*, 51(1):99–121, April 2008.

[62] Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. Compact architecture for high-throughput regular expression matching on fpga. In *Proc. of the 2008 ACM/IEEE Symp. on Architectures for Networking and Communications Sys.*, pages 30–39.

[63] Nithin George, Hyoukjoong Lee, David Novo, Tiark Rompf, Kevin Brown, Arvind Sujeeth, Martin Odersky, Kunle Olukotun, and Paolo Ienne. Hardware system synthesis from domain-specific languages. In *Proc. of 24th Int. Conf. on Field Prog. Logic and App. (FPL '14).*

[64] Gregory R Andrews. *Concurrent programming: principles and practice.*

[65] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[66] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* " O'Reilly Media, Inc.", 1996.

[67] James F Ziegler and WA Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

[68] John Von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. *Automata studies*, 34:43–98, 1956.

[69] ARC 15 Code Base. `http://goo.gl/efJ6SO`.

[70] A. Procter, W.L. Harrison, I. Graves, M. Becchi, and G. Allwein. Semantics-directed machine architecture in ReWire. In *2013 Int. Conf. on Field Programmable Technology (FPT '13)*, pages 446–449.

[71] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proc. of the 13th USENIX Conf. on System Administration*, LISA '99, pages 229–238, 1999.

[72] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proc. of the 1998 Conf. on USENIX Security Symp.*, pages 3–3.

[73] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1?2):211 – 242, 2000.

[74] William L. Harrison, Adam Procter, and Gerard Allwein. The confinement problem in the presence of faults. In *Proc. 14th ICFEM*, pages 182–197, 2012.

[75] Daniel J. Bernstein. Salsa20 specification, 2005. `http://cr.yp.to/snuffle/spec.pdf`.

[76] Richard Bird and Phillip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[77] Daniel J. Bernstein. New stream cipher designs. chapter The Salsa20 Family of Stream Ciphers, pages 84–97. 2008.

[78] Daniel J. Bernstein. The eSTREAM project - eSTREAM phase 3 - Salsa20 (portfolio profile 1), 2005. Retrieved November 11, 2014.

[79] Jaroslaw Sugier. Low-cost hardware implementations of salsa20 stream cipher in programmable devices. *Journal of Polish Safety and Reliability Association Summer Safety and Reliability Seminars*, 4(1), 2013.

[80] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4):53–64, May 2011.

[81] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 1990.

[82] Patty M. Sailer, Philip M. Sailer, and David R. Kaeli. *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1996.

[83] Luca Benini and Giovanni De Micheli. Networks on chip: a new paradigm for systems on chip design. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 418–419. IEEE, 2002.

[84] Axel Jantsch, Hannu Tenhunen, et al. *Networks on chip*, volume 396. Springer, 2003.

Appendix A

Connect Logic Implementation in Haskell

A.1 Parallel Combinator

This is the Parallel combinator.

```
1 (<&>) :: ReT i1 o1 I a -> ReT i2 o2 I a -> ReT (i1,i2) (o1,o2) I a
2 (<&>) (ReT l) (ReT r) =
3  ReT (do
4         l' <- l
5         r' <- r
6         case (l',r') of
7          (Left a,_)                          -> return (Left a)
8          (_,Left a)                          -> return (Left a)
9          (Right (o1,res1),Right (o2,res2)) ->
10          return (Right ((o1,o2),\(i1,i2) -> (res1 i1) <&> (res2 i2))))
```

Listing A.1: Haskell implementation of the Connect Logic parallel (<&>) combinator

A.2 Refold Combinator

This is the refold combinator.

```
1 refold :: (Monad m) => (o1 -> o2) -> (o1 -> i2 -> i1) -> ReT i1 o1 m a
    -> ReT i2 o2 m a
2 refold otpt inpt (ReT r) =
3  ReT (do
4         r' <- r
5         case r' of
6         Left a           -> return (Left a)
7         Right (o1,res1) ->
8          return (Right (otpt o1, \i2 ->
9                  refold otpt inpt (res1 (inpt o1 i2)))))
```

Listing A.2: Haskell implementation of the Connect Logic `refold` combinator

A.3 RefoldT Combinator

This is the refoldT combinator.

```
1 refoldT :: Monad m => (o1 -> o2) -> (o1 -> i2 -> Maybe i1) -> ReT o1 i1
    m a -> ReT o2 i2 m a
2 refoldT fo fi (ReT res) =
3   ReT (do
4           p <- res
5           case p of
6            Left a           -> return (Left a)
7            Right (o,resume) -> return (Right (fo o, dispatch o resume)))
8   where
```

```
9     dispatch o1 resume = \ i2 ->
10      case fi o1 i2 of
11        Nothing -> ReT (return (Right (fo o1, dispatch o1 resume)))
12        Just x  -> refoldT fo fi (resume x)
```

Listing A.3: Haskell implementation of the Connect Logic `refoldT` combinator

A.4 Iter Combinator

This is the iter combinator.

```
1 iter :: Monad m => (i -> o) -> o -> ReT i o m a
2 iter f init_o = do
3                   i <- signal init_o
4                   iter' f i
5   where
6     iter' f i = do
7                   i' <- signal (f i)
8                   iter' f i'
```

Listing A.4: Haskell implementation of the Connect Logic `iter` combinator.

Appendix B

DLX Component Implementation

B.1 Types for DLX

```
1  module Redux.Types (
2    Bit(..),Vector5(..),Vector6(..),Vector16(..),
3    Vector26(..),Vector32(..),Reg(..),
4    Opcode(..),Maybe(..),RegVal,regEq,zeroReg, oneReg, I, ReT,
5    StT,parI,refold,refoldT,signal,extrude,get,put,lift,return
6  ) where
7
8  import Control.Monad.Resumption.Reactive
9  import Control.Monad.Resumption.Connectors
10 import Data.Functor.Identity
11 import Control.Monad.State hiding (lift)
12 import Control.Monad.Morph
13
14
```

```
15 type ReT = ReacT
16 --Over IO for testing in Haskell
17 type I = IO
18 type StT s m = StateT s m
19 type RegVal = Vector32 Bit
20
21 parI :: ReT i o I a -> ReT j p I a -> ReT (i,j) (o,p) I a
22 parI = (<||>)
23
24 data Bit = L | H deriving Show
25
26 data Vector5 a = Vector5 a a a a a
27 data Vector6 a = Vector6 a a a a a a
28 data Vector16 a = Vector16 a a a a a a a a
29                             a a a a a a a a
30
31 data Vector26 a = Vector26 a a a a a a a a
32                             a a a a a a a a
33                             a a a a a a a a
34                             a a
35
36 data Vector32 a = Vector32 a a a a a a a a
37                             a a a a a a a a
38                             a a a a a a a a
39                             a a a a a a a a deriving Show
40
41 data Reg = R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7
42            | R8 | R9 | R10 | R11 | R12 | R13 | R14
43            | R15 | R16 | R17 | R18 | R19 | R20 | R21
```

```
44                 | R22 | R23 | R24 | R25 | R26 | R27 | R28
45                 | R29 | R30 | R31 | R32
46
47 regEq :: Reg -> Reg -> Bit
48 regEq =
49   \r1 -> \r2 -> case (r1,r2) of
50                          (R1,R1) -> H
51                          (R2,R2) -> H
52                          (R3,R3) -> H
53                          (R4,R4) -> H
54                          (R5,R5) -> H
55                          (R6,R6) -> H
56                          (R7,R7) -> H
57                          (R8,R8) -> H
58                          (R9,R9) -> H
59                          (R10,R10) -> H
60                          (R11,R11) -> H
61                          (R12,R12) -> H
62                          (R13,R13) -> H
63                          (R14,R14) -> H
64                          (R15,R15) -> H
65                          (R16,R16) -> H
66                          (R17,R17) -> H
67                          (R18,R18) -> H
68                          (R19,R19) -> H
69                          (R20,R20) -> H
70                          (R21,R21) -> H
71                          (R22,R22) -> H
72                          (R23,R23) -> H
```

```
73                          (R24,R24) -> H
74                          (R25,R25) -> H
75                          (R26,R26) -> H
76                          (R27,R27) -> H
77                          (R28,R28) -> H
78                          (R29,R29) -> H
79                          (R30,R30) -> H
80                          (R31,R31) -> H
81                          (R32,R32) -> H
82                          _         -> L
83
84 data Opcode = ADD | ADDI | AND | ANDI | BEQZ | BNEZ
85              | J | JAL | JALR | JR | LHI | LW | OR
86              | ORI | SEQ | SEQI | SLE | SLEI | SLL
87              | SLLI | SLT | SLTI | SNE | SNEI | SRA
88              | SRAI | SRL | SRLI | SUB | SUBI | SW
89              | XOR | XORI | NOP deriving Show
90
91 extrude :: (Monad m) => ReT i o (StT s m) a ->
92           s ->
93           ReT i o m (a,s)
94 extrude (ReacT m) s =
95     let a = flip evalStateT s $ do
96                                   res <- m
97                                   s   <- get
98                                   case res of
99                                    Left a      -> return $ Left (a,s)
100                                   Right (o,r) -> return $ Right (o,\i
    -> extrude (r i) s)
```

```
101       in ReacT a
102
103 zeroReg :: Vector32 Bit
104 zeroReg = Vector32 L L L L L L L L L L L L L L L L L L L L L L L L L L L L
      L L L L L L
105
106 oneReg :: Vector32 Bit
107 oneReg = Vector32 L L L L L L L L L L L L L L L L L L L L L L L L L L L L
      L L L L H
```

Listing B.1: Types defined for DLX processor implementation

B.2 DLX Fetch Stage

```
1  module Redux.Fetch where
2
3  import Redux.Types
4  --Stand in function for primitive VHDL adding function
5  import Redux.Test.ArithLogic (add_)
6
7  type Instr    = Vector32 Bit
8  type NewAdd   = Maybe (Vector32 Bit)
9  type PC       = Vector32 Bit
10 type NextInst = Vector32 Bit
11 type FetchI   = (Instr,NewAdd)
12 type FetchO   = (NextInst,Instr,PC)
13
14
15 fourReg :: Vector32 Bit
16 fourReg =
```

```
17   Vector32 L L L L L L L L
18            L L L L L L L L
19            L L L L L L L L
20            L L L L L H L L
21
22
23 fetch_ :: Monad m => PC -> FetchI -> ReT FetchI FetchO m ()
24 fetch_ pc inp = case inp of
25              (_,(Just addr)) -> do
26                                 inp <- signal (addr,zeroReg,zeroReg)
27                                 fetch_ addr inp
28              (inst,Nothing) -> do
29                                let pc4 = add_ pc fourReg
30                                inp <- signal (pc4, inst, pc)
31                                fetch_ pc4 inp
32
33 fetch :: Monad m => ReT FetchI FetchO m ()
34 fetch = fetch_ zeroReg (zeroReg,Just zeroReg)
```

Listing B.2: Haskell implementation of DLX Fetch Stage

B.3 DLX Decode Stage

```
1 module Redux.Decode where
2
3 import Redux.Types
4 import Redux.Writeback
5 import Redux.Test.ArithLogic
6 import Debug.Trace
7
```

```
8  data IJ = I | Jm
9  extend :: Vector16 Bit -> Vector32 Bit
10 extend vec =
11  case vec of
12    Vector16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 ->
13     Vector32 b15 b15 b15 b15 b15 b15 b15 b15
14              b15 b15 b15 b15 b15 b15 b15 b15
15              b15 b14 b13 b12 b11 b10 b9  b8
16              b7  b6  b5  b4  b3  b2  b1  b0
17
18 lextend :: Vector16 Bit -> Vector32 Bit
19 lextend vec =
20  case vec of
21   Vector16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 ->
22    Vector32  L L L L L L L L L L L L L L L L
23              b15 b14 b13 b12 b11 b10 b9 b8
24              b7  b6  b5  b4  b3  b2  b1 b0
25
26
27 highHalf :: Vector16 Bit -> Vector32 Bit
28 highHalf vec = case vec of
29     Vector16 b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0 ->
30     Vector32  b15 b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4 b3 b2 b1 b0
31               L L L L L L L L L L L L L L L L
32
33
34 decodeReg :: (Bit,Bit,Bit,Bit,Bit) -> Reg
35 decodeReg inp = case inp of
36             (L,L,L,L,L) -> R0
```

```
(L,L,L,L,H) -> R1
(L,L,L,H,L) -> R2
(L,L,L,H,H) -> R3
(L,L,H,L,L) -> R4
(L,L,H,L,H) -> R5
(L,L,H,H,L) -> R6
(L,L,H,H,H) -> R7
(L,H,L,L,L) -> R8
(L,H,L,L,H) -> R9
(L,H,L,H,L) -> R10
(L,H,L,H,H) -> R11
(L,H,H,L,L) -> R12
(L,H,H,L,H) -> R13
(L,H,H,H,L) -> R14
(L,H,H,H,H) -> R15
(H,L,L,L,L) -> R16
(H,L,L,L,H) -> R17
(H,L,L,H,L) -> R18
(H,L,L,H,H) -> R19
(H,L,H,L,L) -> R20
(H,L,H,L,H) -> R21
(H,L,H,H,L) -> R22
(H,L,H,H,H) -> R23
(H,H,L,L,L) -> R24
(H,H,L,L,H) -> R25
(H,H,L,H,L) -> R26
(H,H,L,H,H) -> R27
(H,H,H,L,L) -> R28
(H,H,H,L,H) -> R29
```

```
66                (H,H,H,H,L) -> R30
67                (H,H,H,H,H) -> R31
68
69 decodeIJ :: (Bit,Bit,Bit,Bit,Bit,Bit) -> (Opcode,IJ)
70 decodeIJ inp = case inp of
71                (L,L,L,L,L,L) -> (NOP,I)
72                (L,L,L,L,L,H) -> (NOP,I)
73                (L,L,L,L,H,L) -> (J,Jm)
74                (L,L,L,L,H,H) -> (JAL,Jm)
75                (L,L,L,H,L,L) -> (BEQZ,I)
76                (L,L,L,H,L,H) -> (BNEZ,I)
77                (L,L,L,H,H,L) -> (NOP,I)
78                (L,L,L,H,H,H) -> (NOP,I)
79                (L,L,H,L,L,L) -> (ADDI,I)
80                (L,L,H,L,L,H) -> (NOP,I)
81                (L,L,H,L,H,L) -> (SUBI,I)
82                (L,L,H,L,H,H) -> (NOP,I)
83                (L,L,H,H,L,L) -> (ANDI,I)
84                (L,L,H,H,L,H) -> (ORI,I)
85                (L,L,H,H,H,L) -> (XORI,I)
86                (L,L,H,H,H,H) -> (LHI,I)
87                (L,H,L,L,L,L) -> (NOP,I)
88                (L,H,L,L,L,H) -> (NOP,I)
89                (L,H,L,L,H,L) -> (JR,I)
90                (L,H,L,L,H,H) -> (JALR,I)
91                (L,H,L,H,L,L) -> (SLLI,I)
92                (L,H,L,H,L,H) -> (NOP,I)
93                (L,H,L,H,H,L) -> (SRLI,I)
94                (L,H,L,H,H,H) -> (SRAI,I)
```

```
95    (L,H,H,L,L,L) -> (SEQI, I)
96    (L,H,H,L,L,H) -> (SNEI, I)
97    (L,H,H,L,H,L) -> (SLTI, I)
98    (L,H,H,L,H,H) -> (NOP, I)
99    (L,H,H,H,L,L) -> (SLEI, I)
100   (L,H,H,H,L,H) -> (NOP, I)
101   (L,H,H,H,H,L) -> (NOP, I)
102   (L,H,H,H,H,H) -> (NOP, I)
103   (H,L,L,L,L,L) -> (NOP, I)
104   (H,L,L,L,L,H) -> (NOP, I)
105   (H,L,L,L,H,L) -> (NOP, I)
106   (H,L,L,L,H,H) -> (LW, I)
107   (H,L,L,H,L,H) -> (NOP, I)
108   (H,L,L,H,H,L) -> (NOP, I)
109   (H,L,L,H,H,H) -> (NOP, I)
110   (H,L,H,L,L,H) -> (NOP, I)
111   (H,L,H,L,H,L) -> (NOP, I)
112   (H,L,H,L,H,H) -> (SW, I)
113   (H,L,H,H,L,L) -> (NOP, I)
114   (H,L,H,H,L,H) -> (NOP, I)
115   (H,L,H,H,H,L) -> (NOP, I)
116   (H,L,H,H,H,H) -> (NOP, I)
117   (H,H,L,L,L,L) -> (NOP, I)
118   (H,H,L,L,L,H) -> (NOP, I)
119   (H,H,L,L,H,L) -> (NOP, I)
120   (H,H,L,L,H,H) -> (NOP, I)
121   (H,H,L,H,L,L) -> (NOP, I)
122   (H,H,L,H,L,H) -> (NOP, I)
123   (H,H,L,H,H,L) -> (NOP, I)
```

```
124                 (H,H,L,H,H,H) -> (NOP, I)
125                 (H,H,H,L,L,L) -> (NOP, I)
126                 (H,H,H,L,L,H) -> (NOP, I)
127                 (H,H,H,L,H,L) -> (NOP, I)
128                 (H,H,H,L,H,H) -> (NOP, I)
129                 (H,H,H,H,L,L) -> (NOP, I)
130                 (H,H,H,H,L,H) -> (NOP, I)
131                 (H,H,H,H,H,L) -> (NOP, I)
132                 (H,H,H,H,H,H) -> (NOP, I)
133
134 decodeR :: (Bit,Bit,Bit,Bit,Bit,Bit) -> Opcode
135 decodeR inp = case inp of
136                 (L,L,L,L,L,L) -> NOP
137                 (L,L,L,L,L,H) -> NOP
138                 (L,L,L,L,H,L) -> NOP
139                 (L,L,L,L,H,H) -> NOP
140                 (L,L,L,H,L,L) -> SLL
141                 (L,L,L,H,L,H) -> NOP
142                 (L,L,L,H,H,L) -> SRL
143                 (L,L,L,H,H,H) -> SRA
144                 (L,L,H,L,L,L) -> NOP
145                 (L,L,H,L,L,H) -> NOP
146                 (L,L,H,L,H,L) -> NOP
147                 (L,L,H,L,H,H) -> NOP
148                 (L,L,H,H,L,L) -> NOP
149                 (L,L,H,H,L,H) -> NOP
150                 (L,L,H,H,H,L) -> NOP
151                 (L,L,H,H,H,H) -> NOP
152                 (L,H,L,L,L,L) -> NOP
```

```
153                (L,H,L,L,L,H) -> NOP
154                (L,H,L,L,H,L) -> NOP
155                (L,H,L,L,H,H) -> NOP
156                (L,H,L,H,L,L) -> NOP
157                (L,H,L,H,L,H) -> NOP
158                (L,H,L,H,H,L) -> NOP
159                (L,H,L,H,H,H) -> NOP
160                (L,H,H,L,L,L) -> NOP
161                (L,H,H,L,L,H) -> NOP
162                (L,H,H,L,H,L) -> NOP
163                (L,H,H,L,H,H) -> NOP
164                (L,H,H,H,L,L) -> NOP
165                (L,H,H,H,L,H) -> NOP
166                (L,H,H,H,H,L) -> NOP
167                (L,H,H,H,H,H) -> NOP
168                (H,L,L,L,L,L) -> ADD
169                (H,L,L,L,L,H) -> NOP
170                (H,L,L,L,H,L) -> SUB
171                (H,L,L,L,H,H) -> NOP
172                (H,L,L,H,L,L) -> AND
173                (H,L,L,H,L,H) -> OR
174                (H,L,L,H,H,L) -> XOR
175                (H,L,L,H,H,H) -> NOP
176                (H,L,H,L,L,L) -> SEQ
177                (H,L,H,L,L,H) -> SNE
178                (H,L,H,L,H,L) -> SLT
179                (H,L,H,L,H,H) -> NOP
180                (H,L,H,H,L,L) -> SLE
181                (H,L,H,H,L,H) -> NOP
```

```
182               (H,L,H,H,H,L) -> NOP
183               (H,L,H,H,H,H) -> NOP
184               (H,H,L,L,L,L) -> NOP
185               (H,H,L,L,L,H) -> NOP
186               (H,H,L,L,H,L) -> NOP
187               (H,H,L,L,H,H) -> NOP
188               (H,H,L,H,L,L) -> NOP
189               (H,H,L,H,L,H) -> NOP
190               (H,H,L,H,H,L) -> NOP
191               (H,H,L,H,H,H) -> NOP
192               (H,H,H,L,L,L) -> NOP
193               (H,H,H,L,L,H) -> NOP
194               (H,H,H,L,H,L) -> NOP
195               (H,H,H,L,H,H) -> NOP
196               (H,H,H,H,L,L) -> NOP
197               (H,H,H,H,L,H) -> NOP
198               (H,H,H,H,H,L) -> NOP
199               (H,H,H,H,H,H) -> NOP
200
201 rtype :: Vector32 Bit -> RegFile -> RegVal
202         -> (Opcode,Reg,(Reg,RegVal),(Reg,RegVal))
203 rtype vec rf pc = case vec of
204             Vector32 _   _   _   _   _
205                      _   b25 b24 b23 b22
206                      b21 b20 b19 b18 b17
207                      b16 b15 b14 b13 b12
208                      b11 b10 b9  b8  b7
209                      b6  b5  b4  b3  b2 b1 b0 ->
210               let rs1 = decodeReg (b25,b24,b23,b22,b2)
```

```
211                     rs2 = decodeReg (b20,b19,b18,b17,b16)
212                 in (decodeR    (b5,b4,b3,b2,b1,b0),
213                     decodeReg (b15,b14,b13,b12,b11),
214                     (rs1, getReg rs1 rf),
215                     (rs2, getReg rs2 rf)
216                    )
217
218 ijtype :: Vector32 Bit -> RegFile -> RegVal ->
      (Opcode,Reg,(Reg,RegVal),(Reg,RegVal))
219 ijtype vec rf pc = case vec of
220             Vector32 b31 b30 b29 b28 b27
221                      b26 b25 b24 b23 b22
222                      b21 b20 b19 b18 b17
223                      b16 b15 b14 b13 b12
224                      b11 b10 b9  b8  b7
225                      b6  b5  b4  b3  b2 b1 b0 ->
226               let exvalue  = Vector32 b25 b25 b25 b25 b25 b25 b25
227                                       b24 b23 b22 b21 b20 b19 b18
228                                       b17 b16 b15 b14 b13 b12 b11
229                                       b10 b9 b8 b7 b6 b5 b4 b3 b2
230                                       b1 b0
231                   imm      = Vector16 b15 b14 b13 b12 b11 b10 b9
232                                       b8 b7 b6 b5 b4 b3 b2 b1 b0
233                   eximm    = extend imm
234                   leximm   = lextend imm
235             in case decodeIJ (b31,b30,b29,b28,b27,b26) of
236                 (opc,I) -> let rs1  = decodeReg (b25,b24,b23,b22,b21)
237                                rs1v = getReg rs1 rf
238                                rd   = decodeReg (b20,b19,b18,b17,b16)
```

```
239                 in case opc of
240                     ADDI -> (opc,rd,(rs1,rs1v),(R0,eximm))
241                     ANDI -> (opc,rd,(rs1,rs1v),(R0,leximm))
242                     BEQZ -> (opc,R0,(rs1,rs1v),(R0,add_ pc eximm))
243                     BNEZ -> (opc,R0,(rs1,rs1v),(R0,add_ pc eximm))
244                     JALR -> (opc,rd,(R0,pc),(rs1,rs1v))
245                     JR   -> (opc,R0,(rs1,rs1v),(R0,zeroReg))
246                     LHI  -> (opc,rd,(R0,highHalf imm),(R0,zeroReg))
247                     LW   -> (opc,rd,(rs1,rs1v),(R0,leximm))
248                     ORI  -> (opc,rd,(rs1,rs1v),(R0,leximm))
249                     SEQI -> (opc,rd,(rs1,rs1v),(R0,eximm))
250                     SLEI -> (opc,rd,(rs1,rs1v),(R0,eximm))
251                     SLLI -> (opc,rd,(rs1,rs1v),(R0,leximm))
252                     SLTI -> (opc,rd,(rs1,rs1v),(R0,eximm))
253                     SNEI -> (opc,rd,(rs1,rs1v),(R0,eximm))
254                     SRAI -> (opc,rd,(rs1,rs1v),(R0,eximm))
255                     SRLI -> (opc,rd,(rs1,rs1v),(R0,leximm))
256                     SUBI -> (opc,rd,(rs1,rs1v),(R0,eximm))
257                     SW   -> (opc,rd,(rs1,add_ rs1v eximm),
258                              (rd,getReg rd rf))
259                     XORI -> (opc,rd,(rs1,rs1v),(R0,leximm))
260                     _    -> (NOP,R0,(R0,zeroReg),(R0,zeroReg))
261             (opc,Jm) -> case opc of
262                             J   -> (J,R0,
263                                     (R0,exvalue),(R0,pc))
264                             JAL -> (JAL,R31,(R0,pc),
265                                     (R0,exvalue))
266                             _   -> (NOP,R0,
267                                     (R0,zeroReg),(R0,zeroReg))
```

```
268
269 decodeInst :: Vector32 Bit -> RegFile
270                -> RegVal     ->
271                (Opcode,Reg,(Reg,RegVal),(Reg,RegVal))
272 decodeInst inst rf pc =
273     case inst of
274       Vector32 L L L L L L
275                b25 b24 b23 b22
276                b21 b20 b19 b18 b17
277                b16 b15 b14 b13 b12
278                b11 b10 b9  b8  b7
279                b6  b5  b4  b3  b2 b1 b0 ->
280                 rtype inst rf pc
281       _ -> ijtype inst rf pc
282
283 decoder_ :: Monad m => (Vector32 Bit, RegFile, RegVal)
284                         -> ReT (Vector32 Bit, RegFile, RegVal)
285                                (Opcode,Reg,(Reg,RegVal),(Reg,RegVal))
286                                m ()
287 decoder_ inp = case inp of
288             (i,rf,pc) -> do
289                          i <- signal (decodeInst i rf pc)
290                          decoder_ i
291
292 decode :: Monad m => ReT (Vector32 Bit, RegFile, RegVal)
293                          (Opcode,Reg,(Reg,RegVal),(Reg,RegVal))
294                          m ()
295 decode = decoder_ (zeroReg,zeroFile,zeroReg)
```

Listing B.3: Haskell implementation of DLX Decode Stage

B.4 DLX Execute Phase

This is the DLX execute phase.

```
1  module Redux.ALU where
2
3  import Redux.Types
4  import Redux.Test.ArithLogic
5
6  type RegDest = Reg
7  type Flush   = Bit
8
9  type ALUI = (Opcode, Flush,RegDest,RegVal,RegVal)
10 type ALUO = (Maybe (Opcode, RegDest, RegVal, RegVal))
11
12 mod8 :: RegVal -> (Bit,Bit,Bit)
13 mod8 rv =case rv of
14             (Vector32 _ _ _ _ _ _ _ _
15                       _ _ _ _ _ _ _ _
16                       _ _ _ _ _ _ _ _
17                       _ _ _ _ _ b2 b1 b0) -> (b2,b1,b0)
18
19 proc :: (Opcode, Flush, RegDest, RegVal, RegVal) ->
20         (Maybe (Opcode, RegDest, RegVal, RegVal))
21 proc inp = case inp of
22             (op,_,rdest,ra,rb) -> case op of
23             ADD  -> Just (op,rdest,add_ ra rb, zeroReg)
24             ADDI -> Just (op,rdest,add_ ra rb, zeroReg)
25             AND  -> Just (op,rdest,and_ ra rb, zeroReg)
26             ANDI -> Just (op,rdest,and_ ra rb, zeroReg)
```

```
27              BEQZ -> case eq_ ra zeroReg of
28                        H -> Just (op,R0,rb,oneReg)
29                        L -> Just (op,R0,zeroReg,zeroReg)
30              BNEZ -> case not_ (eq_ ra zeroReg) of
31                        H -> Just (op,R0,rb,oneReg)
32                        L -> Just (op,R0,zeroReg,zeroReg)
33              J    -> Just (op,R0,add_ ra rb,zeroReg)
34              JAL -> Just (op,R31,add_ ra (lit (4 :: BWord)), add_ ra rb)
35              JALR -> Just (op,R31,add_ ra (lit (4 :: BWord)), rb)
36              JR   -> Just (op,R0,ra,zeroReg)
37              LHI  -> Just (op,rdest,ra,zeroReg)
38              LW   -> Just (op,rdest,add_ ra rb,zeroReg)
39              OR   -> Just (op,rdest,or_ ra rb,zeroReg)
40              ORI  -> Just (op,rdest,or_ ra rb,zeroReg)
41              SEQ  -> case eq_ ra rb of
42                        H -> Just (op,rdest,oneReg,zeroReg)
43                        L -> Just (op,rdest,zeroReg,zeroReg)
44              SEQI  -> case eq_ ra rb of
45                         H -> Just (op,rdest,oneReg,zeroReg)
46                         L -> Just (op,rdest,zeroReg,zeroReg)
47              SLE  -> case lte_ ra rb of
48                        H -> Just (op,rdest,oneReg,zeroReg)
49                        L -> Just (op,rdest,zeroReg,zeroReg)
50              SLEI  -> case lte_ ra rb of
51                         H -> Just (op,rdest,oneReg,zeroReg)
52                         L -> Just (op,rdest,zeroReg,zeroReg)
53              SLL   -> case ra of
54                Vector32 b31 b30 b29 b28 b27 b26 b25 b24
55                         b23 b22 b21 b20 b19 b18 b17 b16
```

```
56                      b15 b14 b13 b12 b11 b10 b9  b8
57                      b7  b6  b5  b4  b3  b2  b1  b0 ->
58    case mod8 rb of
59        (L,L,L) -> Just (op,rdest,ra,zeroReg)
60        (L,L,H) -> let vec =  Vector32 b30 b29 b28 b27 b26 b25 b24
61                              b23 b22 b21 b20 b19 b18 b17 b16  b15
62                              b14 b13 b12 b11 b10 b9 b8 b7 b6 b5 b4
63                              b3 b2 b1 b0 L
64                   in Just (op,rdest,vec,zeroReg)
65        (L,H,L) -> let vec =  Vector32 b29 b28 b27 b26 b25 b24
66                              b23 b22 b21 b20 b19 b18 b17 b16
67                              b15 b14 b13 b12 b11 b10 b9  b8
68                              b7  b6  b5  b4  b3  b2  b1  b0 L L
69                   in Just (op,rdest,vec,zeroReg)
70        (L,H,H) -> let vec =  Vector32 b28 b27 b26 b25 b24
71                              b23 b22 b21 b20 b19 b18 b17 b16
72                              b15 b14 b13 b12 b11 b10 b9  b8
73                              b7  b6  b5  b4  b3  b2  b1  b0 L L L
74                   in Just (op,rdest,vec,zeroReg)
75        (H,L,L) -> let vec =  Vector32 b27 b26 b25 b24
76                              b23 b22 b21 b20 b19 b18 b17 b16
77                              b15 b14 b13 b12 b11 b10 b9  b8
78                              b7  b6  b5  b4  b3  b2  b1  b0 L L L L
79                    in Just (op,rdest,vec,zeroReg)
80        (H,L,H) -> let vec =  Vector32 b26 b25 b24
81                              b23 b22 b21 b20 b19 b18 b17 b16
82                              b15 b14 b13 b12 b11 b10 b9  b8
83                              b7  b6  b5  b4  b3  b2  b1  b0 L L L L L
84                    in Just (op,rdest,vec,zeroReg)
```

```
85          (H,H,L) -> let vec =  Vector32 b25 b24
86                                b23 b22 b21 b20 b19 b18 b17 b16
87                                b15 b14 b13 b12 b11 b10 b9  b8
88                                b7  b6  b5  b4  b3  b2  b1  b0 L L L L L L
89                      in Just (op,rdest,vec,zeroReg)
90          (H,H,H) -> let vec =  Vector32 b24 b23 b22 b21
91                                b20 b19 b18 b17 b16
92                                b15 b14 b13 b12 b11 b10 b9  b8
93                                b7  b6  b5  b4  b3  b2  b1  b0
94                                L   L   L   L   L   L   L
95                      in Just (op,rdest,vec,zeroReg)
96              SLLI    -> case ra of
97                           Vector32 b31 b30 b29 b28 b27 b26 b25 b24
98                                    b23 b22 b21 b20 b19 b18 b17 b16
99                                    b15 b14 b13 b12 b11 b10 b9  b8
100                                   b7  b6  b5  b4  b3  b2  b1  b0 ->
101             case mod8 rb of
102               (L,L,L) -> Just (op,rdest,ra,zeroReg)
103               (L,L,H) -> let vec =
104                            Vector32 b30 b29 b28 b27 b26 b25 b24
105                                     b23 b22 b21 b20 b19 b18 b17 b16
106                                     b15 b14 b13 b12 b11 b10 b9  b8
107                                     b7  b6  b5  b4  b3  b2  b1  b0 L
108                           in Just (op,rdest,vec,zeroReg)
109               (L,H,L) -> let vec = Vector32 b29 b28 b27 b26 b25 b24
110                                     b23 b22 b21 b20 b19 b18 b17 b16
111                                     b15 b14 b13 b12 b11 b10 b9  b8
112                                     b7  b6  b5  b4  b3  b2  b1  b0 L L
113                           in Just (op,rdest,vec,zeroReg)
```

```
114                 (L,H,H) -> let vec =
115                                Vector32 b28 b27 b26 b25 b24
116                                    b23 b22 b21 b20 b19 b18 b17 b16
117                                    b15 b14 b13 b12 b11 b10 b9  b8
118                                    b7  b6  b5  b4  b3  b2  b1  b0 L L L
119                                    in Just (op,rdest,vec,zeroReg)
120                 (H,L,L) -> let vec =
121                                Vector32 b27 b26 b25 b24
122                                b23 b22 b21 b20 b19 b18 b17 b16
123                                b15 b14 b13 b12 b11 b10 b9  b8
124                                b7  b6  b5  b4  b3  b2  b1  b0 L L L L
125                                in Just (op,rdest,vec,zeroReg)
126                 (H,L,H) -> let vec =
127                                Vector32 b26 b25 b24
128                                b23 b22 b21 b20 b19 b18 b17 b16
129                                b15 b14 b13 b12 b11 b10 b9  b8
130                                b7  b6  b5  b4  b3  b2  b1  b0 L L L L L
131                                in Just (op,rdest,vec,zeroReg)
132                 (H,H,L) -> let vec =
133                                Vector32 b25 b24
134                                b23 b22 b21 b20 b19 b18 b17 b16
135                                b15 b14 b13 b12 b11 b10 b9  b8
136                                b7  b6  b5  b4  b3  b2  b1  b0
137                                L   L   L   L   L   L
138                               in Just (op,rdest,vec,zeroReg)
139                 (H,H,H) -> let vec =
140                                Vector32 b24
141                                b23 b22 b21 b20 b19 b18 b17 b16
142                                b15 b14 b13 b12 b11 b10 b9  b8
```

```
143                         b7   b6   b5   b4   b3   b2   b1   b0
144                         L    L    L    L    L    L    L
145                       in Just (op,rdest,vec,zeroReg)
146           SLT -> case lt_ ra rb of
147                     H -> Just (op,rdest,oneReg,zeroReg)
148                     L -> Just (op,rdest,zeroReg,zeroReg)
149           SLTI -> case lt_ ra rb of
150                     H -> Just (op,rdest,oneReg,zeroReg)
151                     L -> Just (op,rdest,zeroReg,zeroReg)
152           SNE -> case not_ (eq_ ra rb) of
153                     H -> Just (op,rdest,oneReg,zeroReg)
154                     L -> Just (op,rdest,zeroReg,zeroReg)
155           SNEI -> case not_ (eq_ ra rb) of
156                     H -> Just (op,rdest,oneReg,zeroReg)
157                     L -> Just (op,rdest,zeroReg,zeroReg)
158           SRA -> case ra of
159                      Vector32 b31 b30 b29 b28 b27 b26 b25 b24
160                               b23 b22 b21 b20 b19 b18 b17 b16
161                               b15 b14 b13 b12 b11 b10 b9  b8
162                               b7  b6  b5  b4  b3  b2  b1  b0 ->
163         case mod8 rb of
164           (L,L,L) -> Just (op,rdest,ra,zeroReg)
165           (L,L,H) -> let vec =
166                            Vector32 b31 b31 b30 b29 b28 b27 b26 b25 b24
167                                     b23 b22 b21 b20 b19 b18 b17 b16
168                                     b15 b14 b13 b12 b11 b10 b9  b8
169                                     b7  b6  b5  b4  b3  b2  b1
170                          in Just (op,rdest,vec,zeroReg)
171           (L,H,L) -> let vec =
```

```
172                     Vector32 b31 b31 b31 b30
173                              b29 b28 b27 b26 b25 b24
174                              b23 b22 b21 b20 b19 b18 b17 b16
175                              b15 b14 b13 b12 b11 b10 b9  b8
176                              b7  b6  b5  b4  b3  b2
177                        in Just (op,rdest,vec,zeroReg)
178             (L,H,H) -> let vec =
179                     Vector32 b31 b31 b31 b31 b30
180                              b29 b28 b27 b26 b25 b24
181                              b23 b22 b21 b20 b19 b18 b17 b16
182                              b15 b14 b13 b12 b11 b10 b9  b8
183                              b7  b6  b5  b4  b3
184                        in Just (op,rdest,vec,zeroReg)
185             (H,L,L) -> let vec =
186                     Vector32 b31 b31 b31 b31 b31 b30
187                              b29 b28 b27 b26 b25 b24
188                              b23 b22 b21 b20 b19 b18 b17 b16
189                              b15 b14 b13 b12 b11 b10 b9  b8
190                              b7  b6  b5  b4
191                        in Just (op,rdest,vec,zeroReg)
192             (H,L,H) -> let vec =
193                     Vector32 b31 b31 b31 b31 b31 b31
194                              b30 b29 b28 b27 b26 b25 b24
195                              b23 b22 b21 b20 b19 b18 b17 b16
196                              b15 b14 b13 b12 b11 b10 b9  b8
197                              b7  b6  b5
198                        in Just (op,rdest,vec,zeroReg)
199             (H,H,L) -> let vec =
200                     Vector32 b31 b31 b31 b31 b31
```

```
201                                 b31 b31 b30 b29 b28 b27 b26 b25 b24
202                                 b23 b22 b21 b20 b19 b18 b17 b16
203                                 b15 b14 b13 b12 b11 b10 b9  b8
204                                 b7  b6
205                             in Just (op,rdest,vec,zeroReg)
206           (H,H,H) -> let vec =
207                           Vector32 b31 b31 b31 b31 b31 b31
208                                    b31 b31 b30 b29 b28 b27 b26 b25 b24
209                                    b23 b22 b21 b20 b19 b18 b17 b16
210                                    b15 b14 b13 b12 b11 b10 b9  b8
211                                    b7
212                             in Just (op,rdest,vec,zeroReg)
213           SRAI -> case ra of
214                           Vector32 b31 b30 b29 b28 b27 b26 b25 b24
215                                    b23 b22 b21 b20 b19 b18 b17 b16
216                                    b15 b14 b13 b12 b11 b10 b9  b8
217                                    b7  b6  b5  b4  b3  b2  b1  b0 ->
218         case mod8 rb of
219           (L,L,L) -> Just (op,rdest,ra,zeroReg)
220           (L,L,H) ->
221             let vec = Vector32 b31 b31 b30 b29
222                                b28 b27 b26 b25 b24
223                                b23 b22 b21 b20 b19 b18 b17 b16
224                                b15 b14 b13 b12 b11 b10 b9  b8
225                                b7  b6  b5  b4  b3  b2  b1
226                           in Just (op,rdest,vec,zeroReg)
227           (L,H,L) -> let vec = Vector32 b31 b31 b31 b30 b29
228                                b28 b27 b26 b25 b24
229                                b23 b22 b21 b20 b19 b18 b17 b16
```

```
230                                 b15 b14 b13 b12 b11 b10 b9  b8
231                                 b7  b6  b5  b4  b3  b2
232                             in Just (op,rdest,vec,zeroReg)
233         (L,H,H) -> let vec = Vector32 b31 b31 b31 b31 b30
234                                 b29 b28 b27 b26 b25 b24
235                                 b23 b22 b21 b20 b19 b18 b17 b16
236                                 b15 b14 b13 b12 b11 b10 b9  b8
237                                 b7  b6  b5  b4  b3
238                             in Just (op,rdest,vec,zeroReg)
239         (H,L,L) -> let vec = Vector32 b31 b31 b31 b31 b31
240                                 b30 b29 b28 b27 b26 b25 b24
241                                 b23 b22 b21 b20 b19 b18 b17 b16
242                                 b15 b14 b13 b12 b11 b10 b9  b8
243                                 b7  b6  b5  b4
244                              in Just (op,rdest,vec,zeroReg)
245         (H,L,H) -> let vec = Vector32 b31 b31 b31 b31 b31
246                                 b31 b30 b29 b28 b27 b26 b25 b24
247                                 b23 b22 b21 b20 b19 b18 b17 b16
248                                 b15 b14 b13 b12 b11 b10 b9  b8
249                                 b7  b6  b5
250                               in Just (op,rdest,vec,zeroReg)
251         (H,H,L) -> let vec = Vector32 b31 b31 b31 b31 b31 b31
252                                 b31 b30 b29 b28 b27 b26 b25 b24
253                                 b23 b22 b21 b20 b19 b18 b17 b16
254                                 b15 b14 b13 b12 b11 b10 b9  b8
255                                 b7  b6
256                             in Just (op,rdest,vec,zeroReg)
257         (H,H,H) -> let vec = Vector32 b31 b31 b31 b31 b31 b31
258                                 b31 b31 b30 b29 b28 b27 b26 b25 b24
```

```
259                                 b23 b22 b21 b20 b19 b18 b17 b16
260                                 b15 b14 b13 b12 b11 b10 b9  b8
261                                 b7
262                             in Just (op,rdest,vec,zeroReg)
263             SRL -> case ra of
264                         Vector32 b31 b30 b29 b28 b27 b26 b25 b24
265                                 b23 b22 b21 b20 b19 b18 b17 b16
266                                 b15 b14 b13 b12 b11 b10 b9  b8
267                                 b7  b6  b5  b4  b3  b2  b1  b0 ->
268         case mod8 rb of
269             (L,L,L) -> Just (op,rdest,ra,zeroReg)
270             (L,L,H) -> let vec = Vector32 L b31 b30 b29 b28
271                                 b27 b26 b25 b24
272                                 b23 b22 b21 b20 b19 b18 b17 b16
273                                 b15 b14 b13 b12 b11 b10 b9  b8
274                                 b7  b6  b5  b4  b3  b2  b1
275                            in Just (op,rdest,vec,zeroReg)
276             (L,H,L) -> let vec = Vector32 L L b31 b30 b29
277                                 b28 b27 b26 b25 b24
278                                 b23 b22 b21 b20 b19 b18 b17 b16
279                                 b15 b14 b13 b12 b11 b10 b9  b8
280                                 b7  b6  b5  b4  b3  b2
281                            in Just (op,rdest,vec,zeroReg)
282             (L,H,H) -> let vec = Vector32 L L L b31 b30 b29
283                                 b28 b27 b26 b25 b24
284                                 b23 b22 b21 b20 b19 b18 b17 b16
285                                 b15 b14 b13 b12 b11 b10 b9  b8
286                                 b7  b6  b5  b4  b3
287                            in Just (op,rdest,vec,zeroReg)
```

```
288             (H,L,L) -> let vec = Vector32 L L L L b31 b30 b29
289                                  b28 b27 b26 b25 b24
290                                  b23 b22 b21 b20 b19 b18 b17 b16
291                                  b15 b14 b13 b12 b11 b10 b9  b8
292                                  b7  b6  b5  b4
293                           in Just (op,rdest,vec,zeroReg)
294             (H,L,H) -> let vec = Vector32 L L L L L b31 b30 b29
295                                  b28 b27 b26 b25 b24
296                                  b23 b22 b21 b20 b19 b18 b17 b16
297                                  b15 b14 b13 b12 b11 b10 b9  b8
298                                  b7  b6  b5
299                           in Just (op,rdest,vec,zeroReg)
300             (H,H,L) -> let vec = Vector32 L L L L L L b31
301                                  b30 b29 b28 b27 b26 b25 b24
302                                  b23 b22 b21 b20 b19 b18 b17 b16
303                                  b15 b14 b13 b12 b11 b10 b9  b8
304                                  b7  b6
305                        in Just (op,rdest,vec,zeroReg)
306             (H,H,H) -> let vec = Vector32 L L L L L L L b31
307                                  b30 b29 b28 b27 b26 b25 b24
308                                  b23 b22 b21 b20 b19 b18 b17 b16
309                                  b15 b14 b13 b12 b11 b10 b9  b8
310                                  b7
311                           in Just (op,rdest,vec,zeroReg)
312             SRLI -> case ra of
313                          Vector32 b31 b30 b29 b28 b27 b26 b25 b24
314                                   b23 b22 b21 b20 b19 b18 b17 b16
315                                   b15 b14 b13 b12 b11 b10 b9  b8
316                                   b7  b6  b5  b4  b3  b2  b1  b0 ->
```

```
317             case mod8 rb of
318                     (L,L,L) -> Just (op,rdest,ra,zeroReg)
319                     (L,L,H) -> let vec = Vector32 L b31 b30 b29
320                                           b28 b27 b26 b25 b24
321                                           b23 b22 b21 b20 b19 b18 b17 b16
322                                           b15 b14 b13 b12 b11 b10 b9  b8
323                                           b7  b6  b5  b4  b3  b2  b1
324                                    in Just (op,rdest,vec,zeroReg)
325                     (L,H,L) -> let vec = Vector32 L L b31 b30
326                                           b29 b28 b27 b26 b25 b24
327                                           b23 b22 b21 b20 b19 b18 b17 b16
328                                           b15 b14 b13 b12 b11 b10 b9  b8
329                                           b7  b6  b5  b4  b3  b2
330                                    in Just (op,rdest,vec,zeroReg)
331                     (L,H,H) -> let vec = Vector32 L L L
332                                           b31 b30 b29 b28 b27 b26 b25 b24
333                                           b23 b22 b21 b20 b19 b18 b17 b16
334                                           b15 b14 b13 b12 b11 b10 b9  b8
335                                           b7  b6  b5  b4  b3
336                                    in Just (op,rdest,vec,zeroReg)
337                     (H,L,L) -> let vec = Vector32 L L L L
338                                           b31 b30 b29 b28 b27 b26 b25 b24
339                                           b23 b22 b21 b20 b19 b18 b17 b16
340                                           b15 b14 b13 b12 b11 b10 b9  b8
341                                           b7  b6  b5  b4
342                                    in Just (op,rdest,vec,zeroReg)
343                     (H,L,H) -> let vec = Vector32 L L L L L
344                                           b31 b30 b29 b28 b27 b26 b25 b24
345                                           b23 b22 b21 b20 b19 b18 b17 b16
```

```
346                                         b15 b14 b13 b12 b11 b10 b9  b8
347                                         b7  b6  b5
348                                     in Just (op,rdest,vec,zeroReg)
349                     (H,H,L) -> let vec = Vector32 L L L L L L
350                                         b31 b30 b29 b28 b27 b26 b25 b24
351                                         b23 b22 b21 b20 b19 b18 b17 b16
352                                         b15 b14 b13 b12 b11 b10 b9  b8
353                                         b7  b6
354                                   in Just (op,rdest,vec,zeroReg)
355                     (H,H,H) -> let vec = Vector32 L L L L L L L
356                                         b31 b30 b29 b28 b27 b26 b25 b24
357                                         b23 b22 b21 b20 b19 b18 b17 b16
    b15
358                                         b14 b13 b12 b11 b10 b9  b8 b7
359                                     in Just (op,rdest,vec,zeroReg)
360               SUB -> Just (op,rdest,sub_ ra rb, zeroReg)
361               SUBI -> Just (op,rdest,sub_ ra rb, zeroReg)
362               SW  -> Just (op,rdest,ra, rb)
363               XOR -> Just (op,rdest,xor_ ra rb, zeroReg)
364               XORI -> Just (op,rdest,xor_ ra rb, zeroReg)
365               _    -> Nothing
366
367 procp :: Monad m => ALUI -> ReT ALUI ALUO m ()
368 procp inp = do
369             case inp of
370               (_,H,_,_,_) -> do
371                                     signal Nothing
372                                     signal Nothing
373                                     signal Nothing
```

```
374                                     inp' <- signal Nothing
375                                     procp inp'
376                 _ -> do
377                         inp' <- signal (proc inp)
378                         procp inp'
379
380 alu :: Monad m => ReT ALUI ALUO m ()
381 alu = procp (NOP, L , R0, zeroReg, zeroReg)
```

Listing B.4: The DLX execute phase implemented in Haskell

B.5 DLX Memory Access Phase

```
1  module Redux.Memory where
2
3  import Redux.Types
4
5
6  type Stall      = Bit
7  type Flush      = Stall
8  type Data       = RegVal
9  type Address    = Data
10 type PC         = RegVal
11
12 type MemI = (
13              --Data bus coming from Memory unit
14              Data,
15              --Output from the ALU.  Opcode, Dest Reg Name, Register A,
       Register B values
```

```
16                Maybe (Opcode,Reg,RegVal,RegVal)
17               )
18
19 type MemO = (
20                --Data to be written to memory, if Nothing, we are reading
21                Maybe Data,
22                --Address to be read/written
23                Address,
24                --Shall we stall the pipeline?
25                Stall,
26                --Shall we flush the pipeline (in the event of a branch)?
27                Flush,
28                --Destination register and value to be written back to it
29                Maybe (Reg,RegVal),
30                --New value of the PC in the event of a branch
31                Maybe PC
32               )
33
34 lastBit :: Vector32 Bit -> Bit
35 lastBit vect = case vect of
36                   Vector32 _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
37                            _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
38                            _ _ _ _ _ _ _ b -> b
39
40 feed :: Monad m => (Reg,RegVal) -> ReT MemI MemO m MemI
41 feed fwd = signal (Nothing,zeroReg,L,L,Just fwd,Nothing)
42
43 nop :: Monad m => ReT MemI MemO m MemI
44 nop =  signal (Nothing,zeroReg,L,L,Nothing,Nothing)
```

```
45
46 readStall :: Monad m => Address -> ReT MemI MemO m MemI
47 readStall addr = do
48                 _ <- signal (Nothing, addr, H, L, Nothing,Nothing)
49                 signal (Nothing,addr,H,L,Nothing,Nothing)
50
51 writeStall :: Monad m => Data -> Address -> ReT MemI MemO m MemI
52 writeStall dta addr = signal (Just dta, addr, L, L, Nothing,Nothing)
53
54
55 branch :: Monad m => Maybe (Reg,RegVal) -> RegVal -> ReT MemI MemO m
       MemI
56 branch mb val = signal (Nothing, zeroReg, L, H, mb , Just val)
57
58 memProc :: Monad m => MemI -> ReT MemI MemO m ()
59 memProc inp = case inp of
60               (dta,Nothing) -> do
61                                     i <- signal (Nothing, zeroReg, L, L,
       Nothing,Nothing)
62                                     memProc i
63               --RA is also the value to be written for all others
64               (dta,Just (opcode, reg, ra,rb)) ->
65               case opcode of
66                           LW -> do
67                                     i <- readStall ra
68                                     case i of
69                                        (dta, _) ->
70                                         do
71                                          --Stall one last time
```

```
72                                  inp <-
73                                   signal
74                                   (Nothing, zeroReg, H,
75                                   L, Just (reg,dta),Nothing)
76                                  memProc inp
77                          SW -> do
78                                  i <- writeStall rb ra
79                                  memProc i
80                          J  -> do
81                                  inp <- branch Nothing ra
82                                  memProc inp
83                          JAL -> do
84                                  --RA is PC+4
85                                  --RB is PC += extend(value)
86                                  inp <- branch (Just (R31,ra)) rb
87                                  memProc inp
88                          JALR -> do
89                                    --RA is PC+4
90                                    --RB is PC = Rs1
91                                    inp <- branch (Just (R31,ra)) rb
92                                    memProc inp
93                          JR -> do
94                                    --RA is Rs1 (value)
95                                    inp <- branch Nothing ra
96                                    memProc inp
97                          BEQZ -> do
98                                    case lastBit rb of
99                                      H -> do
100                                             _   <- branch Nothing ra
```

```
101                                                 inp <- signal (Nothing,
       zeroReg, L, L, Nothing, Nothing)
102                                                 memProc inp
103                                           L -> do
104                                                  inp <- nop
105                                                  memProc inp
106                               BNEZ -> do
107                                         case lastBit rb of
108                                           H -> do
109                                                  _   <- branch Nothing ra
110                                                  inp <- signal (Nothing,
       zeroReg, L, L, Nothing, Nothing)
111                                                  memProc inp
112                                           L -> do
113                                                  inp <- nop
114                                                  memProc inp
115                               NOP -> do
116                                         inp <- nop
117                                         memProc inp
118                               _   -> do
119                                         --Feed RA and reg forward
120                                         inp <- feed (reg,ra)
121                                         memProc inp
122
123 mem :: Monad m => ReT MemI MemO m ()
124 mem = memProc (zeroReg, Nothing)
```

Listing B.5: Haskell implementation of the DLX Memory Access processor phase.

B.6 DLX Writeback Phase

```
1  module Redux.Writeback where
2
3  import Prelude (Monad)
4
5  import Redux.Types
6  import Redux.Instructions
7  import Control.Monad.Resumption.Reactive
8
9  type RegFile = Vector32 (RegVal)
10
11 zeroFile :: RegFile
12 zeroFile =
13   Vector32 zeroReg zeroReg zeroReg zeroReg
14            zeroReg zeroReg zeroReg zeroReg
15            zeroReg zeroReg zeroReg zeroReg
16            zeroReg zeroReg zeroReg zeroReg
17            zeroReg zeroReg zeroReg zeroReg
18            zeroReg zeroReg zeroReg zeroReg
19            zeroReg zeroReg zeroReg zeroReg
20            zeroReg zeroReg zeroReg zeroReg
21
22 --Reg Muxer
23 getReg :: Reg -> RegFile -> RegVal
24 getReg reg regfile = case regfile of
25                        (Vector32 b31 b30 b29 b28 b27 b26
26                                  b25 b24 b23 b22 b21 b20
27                                  b19 b18 b17 b16 b15 b14
```

```
28                  b13 b12 b11 b10 b9  b8
29                  b7  b6  b5  b4  b3  b2  b1  b0) ->
30          case reg of
31            R0 -> b0
32            R1 -> b1
33            R2 -> b2
34            R3 -> b3
35            R4 -> b4
36            R5 -> b5
37            R6 -> b6
38            R7 -> b7
39            R8 -> b8
40            R9 -> b9
41            R10 -> b10
42            R11 -> b11
43            R12 -> b12
44            R13 -> b13
45            R14 -> b14
46            R15 -> b15
47            R16 -> b16
48            R17 -> b17
49            R18 -> b18
50            R19 -> b19
51            R20 -> b20
52            R21 -> b21
53            R22 -> b22
54            R23 -> b23
55            R24 -> b24
56            R25 -> b25
```

```
57                              R26 -> b26
58                              R27 -> b27
59                              R28 -> b28
60                              R29 -> b29
61                              R30 -> b30
62                              R31 -> b31
63
64
65 setReg :: (Reg,RegVal) -> RegFile -> RegFile
66 setReg rrv rfile = case rfile of
67     (Vector32 b31 b30 b29 b28 b27 b26 b25 b24 b23 b22 b21 b20
68               b19 b18 b17 b16 b15 b14 b13 b12 b11 b10 b9  b8
69               b7  b6  b5  b4  b3  b2  b1  b0) ->
70     case rrv of
71         (R0,_) -> rfile --R0 remains unchanged always at zero
72         (R1,b1) -> (Vector32 b31 b30 b29 b28 b27 b26
73                              b25 b24 b23 b22 b21 b20
74                              b19 b18 b17 b16 b15 b14
75                              b13 b12 b11 b10 b9  b8
76                              b7  b6  b5  b4  b3  b2  b1  b0)
77         (R2,b2) -> (Vector32 b31 b30 b29 b28 b27 b26
78                              b25 b24 b23 b22 b21 b20
79                              b19 b18 b17 b16 b15 b14
80                              b13 b12 b11 b10 b9  b8
81                              b7  b6  b5  b4  b3  b2   b1  b0)
82         (R3,b3) -> (Vector32 b31 b30 b29 b28 b27 b26
83                              b25 b24 b23 b22 b21 b20
84                              b19 b18 b17 b16 b15 b14
85                              b13 b12 b11 b10 b9  b8
```

```
86                                  b7   b6   b5   b4   b3   b2    b1   b0)
87          (R4,b4) -> (Vector32 b31 b30 b29 b28 b27 b26
88                                  b25  b24  b23  b22  b21  b20
89                                  b19  b18  b17  b16  b15  b14
90                                  b13  b12  b11  b10  b9   b8
91                                  b7   b6   b5   b4   b3   b2    b1   b0)
92          (R5,b5) -> (Vector32 b31 b30 b29 b28 b27 b26
93                                  b25  b24  b23  b22  b21  b20
94                                  b19  b18  b17  b16  b15  b14
95                                  b13  b12  b11  b10  b9   b8
96                                  b7   b6   b5   b4   b3   b2    b1   b0)
97          (R6,b6) -> (Vector32 b31 b30 b29 b28 b27 b26
98                                  b25  b24  b23  b22  b21  b20
99                                  b19  b18  b17  b16  b15  b14
100                                 b13  b12  b11  b10  b9   b8
101                                 b7   b6   b5   b4   b3   b2    b1   b0)
102         (R7,b7) -> (Vector32 b31 b30 b29 b28 b27 b26
103                                 b25  b24  b23  b22  b21  b20
104                                 b19  b18  b17  b16  b15  b14
105                                 b13  b12  b11  b10  b9   b8
106                                 b7   b6   b5   b4   b3   b2    b1   b0)
107         (R8,b8) -> (Vector32 b31 b30 b29 b28 b27 b26
108                                 b25  b24  b23  b22  b21  b20
109                                 b19  b18  b17  b16  b15  b14
110                                 b13  b12  b11  b10  b9   b8
111                                 b7   b6   b5   b4   b3   b2    b1   b0)
112         (R9,b9) -> (Vector32 b31 b30 b29 b28 b27 b26
113                                 b25  b24  b23  b22  b21  b20
114                                 b19  b18  b17  b16  b15  b14
```

```
115                          b13 b12 b11 b10 b9  b8
116                          b7  b6  b5  b4  b3  b2   b1  b0)
117          (R10,b10) -> (Vector32 b31 b30 b29 b28 b27 b26
118                          b25 b24 b23 b22 b21 b20
119                          b19 b18 b17 b16 b15 b14
120                          b13 b12 b11 b10 b9  b8
121                          b7  b6  b5  b4  b3  b2   b1  b0)
122          (R11,b11) -> (Vector32 b31 b30 b29 b28 b27 b26
123                          b25 b24 b23 b22 b21 b20
124                          b19 b18 b17 b16 b15 b14
125                          b13 b12 b11 b10 b9  b8
126                          b7  b6  b5  b4  b3  b2   b1  b0)
127          (R12,b12) -> (Vector32 b31 b30 b29 b28 b27 b26
128                          b25 b24 b23 b22 b21 b20
129                          b19 b18 b17 b16 b15 b14
130                          b13 b12 b11 b10 b9  b8
131                          b7  b6  b5  b4  b3  b2   b1  b0)
132          (R13,b13) -> (Vector32 b31 b30 b29 b28 b27 b26
133                          b25 b24 b23 b22 b21 b20
134                          b19 b18 b17 b16 b15 b14
135                          b13 b12 b11 b10 b9  b8
136                          b7  b6  b5  b4  b3  b2   b1  b0)
137          (R14,b14) -> (Vector32 b31 b30 b29 b28 b27 b26
138                          b25 b24 b23 b22 b21 b20
139                          b19 b18 b17 b16 b15 b14
140                          b13 b12 b11 b10 b9  b8
141                          b7  b6  b5  b4  b3  b2   b1  b0)
142          (R15,b15) -> (Vector32 b31 b30 b29 b28 b27 b26
143                          b25 b24 b23 b22 b21 b20
```

```
144                         b19 b18 b17 b16 b15 b14
145                         b13 b12 b11 b10 b9  b8
146                         b7  b6  b5  b4  b3  b2   b1  b0)
147         (R16,b16) -> (Vector32 b31 b30 b29 b28 b27 b26
148                         b25 b24 b23 b22 b21 b20
149                         b19 b18 b17 b16 b15 b14
150                         b13 b12 b11 b10 b9  b8
151                         b7  b6  b5  b4  b3  b2   b1  b0)
152         (R17,b17) -> (Vector32 b31 b30 b29 b28 b27 b26
153                         b25 b24 b23 b22 b21 b20
154                         b19 b18 b17 b16 b15 b14
155                         b13 b12 b11 b10 b9  b8
156                         b7  b6  b5  b4  b3  b2   b1  b0)
157         (R18,b18) -> (Vector32 b31 b30 b29 b28 b27 b26
158                         b25 b24 b23 b22 b21 b20
159                         b19 b18 b17 b16 b15 b14
160                         b13 b12 b11 b10 b9  b8
161                         b7  b6  b5  b4  b3  b2   b1  b0)
162         (R19,b19) -> (Vector32 b31 b30 b29 b28 b27 b26
163                         b25 b24 b23 b22 b21 b20
164                         b19 b18 b17 b16 b15 b14
165                         b13 b12 b11 b10 b9  b8
166                         b7  b6  b5  b4  b3  b2   b1  b0)
167         (R20,b20) -> (Vector32 b31 b30 b29 b28 b27 b26
168                         b25 b24 b23 b22 b21 b20
169                         b19 b18 b17 b16 b15 b14
170                         b13 b12 b11 b10 b9  b8
171                         b7  b6  b5  b4  b3  b2   b1  b0)
172         (R21,b21) -> (Vector32 b31 b30 b29 b28 b27 b26
```

```
173                             b25 b24 b23 b22 b21 b20
174                             b19 b18 b17 b16 b15 b14
175                             b13 b12 b11 b10 b9  b8
176                             b7  b6  b5  b4  b3  b2   b1  b0)
177         (R22,b22) -> (Vector32 b31 b30 b29 b28 b27 b26
178                             b25 b24 b23 b22 b21 b20
179                             b19 b18 b17 b16 b15 b14
180                             b13 b12 b11 b10 b9  b8
181                             b7  b6  b5  b4  b3  b2   b1  b0)
182         (R23,b23) -> (Vector32 b31 b30 b29 b28 b27 b26
183                             b25 b24 b23 b22 b21 b20
184                             b19 b18 b17 b16 b15 b14
185                             b13 b12 b11 b10 b9  b8
186                             b7  b6  b5  b4  b3  b2   b1  b0)
187         (R24,b24) -> (Vector32 b31 b30 b29 b28 b27 b26
188                             b25 b24 b23 b22 b21 b20
189                             b19 b18 b17 b16 b15 b14
190                             b13 b12 b11 b10 b9  b8
191                             b7  b6  b5  b4  b3  b2   b1  b0)
192         (R25,b25) -> (Vector32 b31 b30 b29 b28 b27 b26
193                             b25 b24 b23 b22 b21 b20
194                             b19 b18 b17 b16 b15 b14
195                             b13 b12 b11 b10 b9  b8
196                             b7  b6  b5  b4  b3  b2   b1  b0)
197         (R26,b26) -> (Vector32 b31 b30 b29 b28 b27 b26
198                             b25 b24 b23 b22 b21 b20
199                             b19 b18 b17 b16 b15 b14
200                             b13 b12 b11 b10 b9  b8
201                             b7  b6  b5  b4  b3  b2   b1  b0)
```

```
202         (R27,b27) -> (Vector32 b31 b30 b29 b28 b27 b26
203                                b25 b24 b23 b22 b21 b20
204                                b19 b18 b17 b16 b15 b14
205                                b13 b12 b11 b10 b9  b8
206                                b7  b6  b5  b4  b3  b2  b1  b0)
207         (R28,b28) -> (Vector32 b31 b30 b29 b28 b27 b26
208                                b25 b24 b23 b22 b21 b20
209                                b19 b18 b17 b16 b15 b14
210                                b13 b12 b11 b10 b9  b8
211                                b7  b6  b5  b4  b3  b2  b1  b0)
212         (R29,b29) -> (Vector32 b31 b30 b29 b28 b27 b26
213                                b25 b24 b23 b22 b21 b20
214                                b19 b18 b17 b16 b15 b14
215                                b13 b12 b11 b10 b9  b8
216                                b7  b6  b5  b4  b3  b2  b1  b0)
217         (R30,b30) -> (Vector32 b31 b30 b29 b28 b27 b26
218                                b25 b24 b23 b22 b21 b20
219                                b19 b18 b17 b16 b15 b14
220                                b13 b12 b11 b10 b9  b8
221                                b7  b6  b5  b4  b3  b2  b1  b0)
222         (R31,b31) -> (Vector32 b31 b30 b29 b28 b27 b26
223                                b25 b24 b23 b22 b21 b20
224                                b19 b18 b17 b16 b15 b14
225                                b13 b12 b11 b10 b9  b8
226                                b7  b6  b5  b4  b3  b2  b1  b0)
227
228
229 writeback_ :: (Monad m) => Maybe (Reg,RegVal) -> ReT (Maybe
        (Reg,RegVal)) RegFile (StT RegFile m) ()
```

```
230 writeback_ i = do
231             s <- lift get
232             let sp = case i of
233                         Just inp -> setReg inp s
234                         Nothing  -> s
235             lift (put sp)
236             inpp <- signal sp
237             writeback_ inpp
238
239 writeback :: (Monad m) => ReT (Maybe (Reg,RegVal)) RegFile m ()
240 writeback = do
241                extrude (writeback_ (Nothing)) zeroFile
242                return ()
```

Listing B.6: The DLX Writeback phase implemented in Haskell

B.7 Combining DLX Phases to a Processor

```
1 module Redux.Proc where
2
3 import Redux.Types
4 import Redux.Fetch
5 import Redux.Decode
6 import Redux.ALU
7 import Redux.Memory
8 import Redux.Writeback
9
10
11 flatten (a,(b,(c,(d,e)))) = (a,b,c,d,e)
12 fanOut ((a,b,c),(d,e,f,g),h,(i,j,k,l,m,n),o) =
```

```
      (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o)
13 pack (a,b,c,d,e) = (a,(b,(c,(d,e))))
14 packIn (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o) =
      ((a,b),(c,d,e,f),(g,h,i,j,k),(l,m),(n,o))
15
16
17 alu_stall = refoldT
18                id
19                (\_ (s,f) -> case s of
20                                H -> Nothing
21                                L -> Just f
22                             )
23                alu
24
25 decode_stall = refoldT
26                 id
27                 (\_ (s,f) -> case s of
28                                 H -> Nothing
29                                 L -> Just f
30                 )
31                 decode
32
33 fetch_stall = refoldT
34                 id
35                 (\_ (s,f) -> case s of
36                                 H -> Nothing
37                                 L -> Just f
38                 fetch
39 devOut :: (NextInst,
```

```
40            Instr ,
41            Redux.Fetch.PC,
42            Opcode ,
43            Reg ,
44            (Reg , RegVal) ,
45            (Reg , RegVal) ,
46            ALUO,
47            Maybe Data ,
48            Address ,
49            Stall ,
50            Redux.Memory.Flush ,
51            Maybe (Reg , RegVal) ,
52            Maybe Redux.Memory.PC,
53            RegFile) -> (NextInst , Maybe Data , Address)
54 devOut (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o) = (a,i,j)
55
56 devOutTR :: (NextInst ,
57            Instr ,
58            Redux.Fetch.PC,
59            Opcode ,
60            Reg ,
61            (Reg , RegVal) ,
62            (Reg , RegVal) ,
63            ALUO,
64            Maybe Data ,
65            Address ,
66            Stall ,
67            Redux.Memory.Flush ,
68            Maybe (Reg , RegVal) ,
```

```
69          Maybe Redux.Memory.PC,
70          RegFile) -> (NextInst, Maybe Data, Address,RegFile)
71 devOutTR (a,b,c,d,e,f,g,h,i,j,k,l,m,n,o) = (a,i,j,o)
72
73 dlx = refold
74          (\x -> devOut (fanOut (flatten x)))
75          (\out -> \inp -> let outp = (fanOut (flatten out))
76                           in let fe   = fetchIn outp inp
77                           in let dc   = decodeIn outp inp
78                           in let al   = aluIn outp inp
79                           in let me   = memIn outp inp
80                           in let wb   = wbIn outp inp
81                           in pack (fe,dc,al,me,wb))
82          (parI fetch_stall
83           (parI decode_stall
84            (parI alu (parI mem writeback))))
85
86 dlx_testreg = refold
87          (devOutTR . fanOut . flatten)
88          (\out inp -> let outp = (fanOut (flatten out))
89                           fe   = fetchIn outp inp
90                           dc   = decodeIn outp inp
91                           al   = aluIn outp inp
92                           me   = memIn outp inp
93                           wb   = wbIn outp inp
94                       in pack (fe,dc,al,me,wb))
95          (parI fetch_stall
96           (parI decode_stall
97            (parI alu_stall (parI mem writeback))))
```

```
98
99  fwd2 :: Maybe (Opcode, RegDest, RegVal, RegVal)
100         -> Maybe (Reg, RegVal)
101         -> (Reg, RegVal) -> RegVal
102 fwd2 aluo memo otro = case aluo of
103                          Nothing           -> fwd1 memo otro
104                          Just (_,R0,_,_)   -> fwd1 memo otro
105                          Just (_,frd,frv,_) ->
106                           case otro of
107                            (rd,rv) -> case regEq frd rd of
108                                         --ALU FWD is a hit
109                                         H -> frv
110                                         --ALU FWD is a miss, try Mem FWD
111                                         L -> fwd1 memo otro
112
113 fwd1 :: Maybe (Reg, RegVal) -> (Reg,RegVal) -> RegVal
114 fwd1 memo otro = case memo of
115                        --No matches in the current inst
116                        Nothing -> (snd otro)
117                        Just (frd,frv) -> case otro of
118                                            (rd,rv) ->
119                                             case regEq frd rd of
120                                                  H -> frv
121                                                  L -> rv
122
123 fetchIn :: (NextInst, Instr,
124         Redux.Fetch.PC,
125         Opcode,
126         Reg,
```

```
127           (Reg, RegVal),
128           (Reg, RegVal),
129           ALUO,
130           Maybe Data,
131           Address,
132           Stall,
133           Redux.Memory.Flush,
134           Maybe (Reg, RegVal),
135           Maybe Redux.Memory.PC,
136           RegFile) -> (Instr, Data) -> (Bit,(Instr, NewAdd))
137
138 fetchIn (nextInst,instr,fetchPC,
139            dcOp,dcDreg,(regA,regAv),
140            (regB,regBv),aluO,rwData,
141            dAddr,stall,flush,mbWbReg,
142            mbPC,rfile) (instIn,dataIn) = (stall,(instIn,mbPC))
143
144
145 decodeIn :: (NextInst, Instr,
146           Redux.Fetch.PC,
147           Opcode,
148           Reg,
149           (Reg, RegVal),
150           (Reg, RegVal),
151           ALUO,
152           Maybe Data,
153           Address,
154           Stall,
155           Redux.Memory.Flush,
```

```
156           Maybe (Reg, RegVal),
157           Maybe Redux.Memory.PC,
158           RegFile) -> (Instr, Data) -> (Bit,(Vector32 Bit,RegFile,RegVal))
159
160 decodeIn (nextInst,instr,fetchPC,
161           dcOp,dcDreg,(regA,regAv),
162           (regB,regBv),aluO,rwData,
163           dAddr,stall,flush,mbWbReg,
164           mbPC,rfile) (instIn,dataIn) = (stall,(instr,rfile,fetchPC))
165
166 aluIn :: (NextInst, Instr,
167           Redux.Fetch.PC,
168           Opcode,
169           Reg,
170           (Reg, RegVal),
171           (Reg, RegVal),
172           ALUO,
173           Maybe Data,
174           Address,
175           Stall,
176           Redux.Memory.Flush,
177           Maybe (Reg, RegVal),
178           Maybe Redux.Memory.PC,
179           RegFile) -> (Instr, Data) -> (Bit,ALUI)
180
181 aluIn (nextInst,instr,fetchPC,
182           dcOp,dcDreg,regA,
183           regB,aluO,rwData,
184           dAddr,stall,flush,mbWbReg,
```

```
185             mbPC, rfile) (instIn, dataIn) =
186             (stall,(dcOp, flush, dcDreg,
187              fwd2 aluO mbWbReg regA, fwd2 aluO mbWbReg regB))
188
189 memIn :: (NextInst, Instr,
190               Redux.Fetch.PC,
191               Opcode,
192               Reg,
193               (Reg, RegVal),
194               (Reg, RegVal),
195               ALUO,
196               Maybe Data,
197               Address,
198               Stall,
199               Redux.Memory.Flush,
200               Maybe (Reg, RegVal),
201               Maybe Redux.Memory.PC,
202               RegFile) -> (Instr, Data) -> MemI
203
204 memIn (nextInst, instr, fetchPC,
205             dcOp, dcDreg, regA,
206             regB, aluO, rwData,
207             dAddr, stall, flush, mbWbReg,
208             mbPC, rfile) (instIn, dataIn) = (dataIn, aluO)
209
210
211 wbIn :: (NextInst, Instr,
212               Redux.Fetch.PC,
213               Opcode,
```

```
214             Reg,
215             (Reg, RegVal),
216             (Reg, RegVal),
217             ALUO,
218             Maybe Data,
219             Address,
220             Stall,
221             Redux.Memory.Flush,
222             Maybe (Reg, RegVal),
223             Maybe Redux.Memory.PC,
224             RegFile) -> (Instr, Data) -> Maybe (Reg, RegVal)
225
226 wbIn (nextInst,instr,fetchPC,
227           dcOp,dcDreg,regA,
228           regB,aluO,rwData,
229           dAddr,stall,flush,mbWbReg,
230           mbPC,rfile) (instIn,dataIn) = mbWbRegjk
```

Listing B.7: Combining the subcomponents of the DLX processor with support for stalling

VITA

Ian Graves was born in Kansas City, Missouri on July 11th, 1986, to Beverly and Leland Graves. He is an Eagle Scout and an alumnus of Lee's Summit Senior High School in the class of 2005. He currently resides in the Portland, Oregon area with his wife, Amanda Graves (B.S. 2011, M.A. 2013). He received his B.S. degree in Computer Science *cum laude* with a minor in mathematics in May of 2009 and he completed the Ph.D. degree in December, 2015.