Alex Horsman

Mantle: Building HDL Abstractions from an RTL Model in Haskell

Computer Science Part II Project

Homerton College

May 17, 2013

Proforma

Name:	Alex Horsman
College:	Homerton College
Project Title:	Mantle: Building HDL abstractions from an RTL model in Haskell
Examination:	Computer Science Part II Project, July 2013
Word Count:	9660
Project Originator:	Alex Horsman
Supervisor:	Dr Simon Moore

Project Aims

The goal of the project was to create a hardware description language, as an embedded DSL in Haskell.

Work Completed

A hardware description language has been produced, which is capable of describing simple synchronous designs, and producing corresponding Verilog. Two extensions were then developed, introducing an abstraction for propagating clocks, and one for size-typed vectors.

Special Difficulties

None

Declaration of Originality

I Alex Horsman of Homerton College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

I give permission for my dissertation to be made available in the archive area of the Laboratory's website.

Signed

Date

Contents

Chapter 1

Introduction

The goal of my project was to produce a hardware description language, as an embedded DSL in Haskell, and then explore new abstractions using the capabilities of the host language. The resulting language "Mantle" achieves this. It provides a convenient syntax for RTL designs, similar to that of Verilog, using Haskell's `do`-notation for monads. This is then used as a base to create several new abstractions for commonly occurring design patterns, including automatic clock propagation, guarded channels, and most significantly a generalised notion of signal groups which provides a solid basis for future abstractions. Finally, the effectiveness of the language is demonstrated by using it to create an increasingly complex series of design examples.

Here is a brief example of Mantle code:

```
counter :: SyncComp (Output Int)
counter out = do
    count <- reg 0
    onClock $ do
        count <=: rd count + 1
    out =: rd count
    endComponent
```

Although simple, this example demonstrates most of the key features used in a typical synchronous design, and also makes use of several of the abstractions mentioned above.

Outline

The Preparation chapter gives an overview of existing HDLs, and then examines some of the more interesting ones in detail, highlighting some key features, and also

identifying some flaws that might be avoided in a new language. The choice of Haskell as the DSL host language is then discussed, detailing some of the features which make it particularly suited for this role. Finally the last section of the chapter gives an overview of the requirements of the project.

The Implementation chapter builds up the various elements of the DSL, from typed expressions up to a complete RTL hardware model. This basic language is then extended with a series of abstractions, one for propagating clocks around synchronous designs, one for building structured bidirectional interfaces and finally one for size-typed vectors.

The Evaluation chapter introduces a series of examples, and uses them to confirm that the project meets the requirements specified in the Preparation chapter, both in terms of the concrete properties of the implementation, and the more subjective language quality.

Chapter 2

Preparation

Background

Currently, the only widely supported hardware description languages are Verilog and VHDL. While these languages offer a vast improvement on older techniques of manual circuit design, as hardware designs become more complex, the limited abstraction capabilities of these older languages are no longer sufficient.

There has been a lot of work developing new hardware description languages in the past decade, which can be roughly divided into three broad categories. Firstly, as so much of the industry is based around Verilog and VHDL, there has been a significant amount of effort spent on extending these languages to accommodate new features. Secondly a lot of HDL research has focused on dataflow models of hardware, to take advantage of existing work on functional programming languages. Finally there have been recent developments using a model of *Guarded Atomic Actions* to describe hardware.

In order to explain the advantages and disadvantages of each model, the next three sections will each discuss a different HDL, chosen to represent the state of the art in each category. These languages are SystemVerilog, Lava, and Bluespec respectively.

SystemVerilog

SystemVerilog is a superset of Verilog-2005, created by Accellera to incorporate and standardise several proprietary Verilog extensions[1]. It adds many new features, including new datatypes and even classes. However the majority of these features are only intended for simulation and verification, and cannot be synthesised. Since we are interested in new abstractions for hardware, we will only discuss those features which are synthesisable.

Specialised Always Blocks

While Verilog's `always` theoretically provide a lot of flexibility to describe hardware in different ways, in practice a majority of the possible designs cannot be synthesised by existing tools. In order to create a design which is guaranteed to be synthesisable, one must use certain patterns which correspond to particular hardware implementations.

SystemVerilog makes these patterns explicit by adding specialised `always` blocks for each pattern. The new keywords `always_comb`, `always_ff` and `always_latch` correspond to combinational, flip-flop and latched logic respectively. Tools can then check these blocks to ensure that the relevant pattern is being used correctly.

Interfaces

For large designs, enumerating every individual port of a complex module can become rather tedious and error prone, especially when these often follow a common pattern. In order to simplify this process, SystemVerilog introduces the concept of an interface. These are hierarchical structures of related wires, which can then be instantiated as a group:

```
interface Channel;
    wire [31:0] data;
    wire enable;
    wire ready;
endinterface

module ChannelUser;
    Channel chan();
endmodule
```

In addition to raw wire access, there are two other ways of using interfaces to connect modules together. The first requires the user to declare a set of `modport`s for the interface. These specify a subset of the wires inside, and allocate each a direction, which can then be used like a normal port in a module header. In the above example this would mean:

```
interface Channel;
    ...
    modport out (
        output value,
        output valid,
```

```
        input ready
    );
    modport in (
        input value,
        input valid,
        output ready
    );
endinterface
```

Typically one defines two `modport`s in this way, each corresponding to "direction" of the interface, but more complex structures are also possible.

The second way interfaces can be accessed involves defining a set of `task`s. These are procedural actions using interface's wires, which can be used in `always` blocks. For example:

```
interface Channel;
    ...
    task push(input [31:0] x);
        value = x;
        valid = 1;
    endtask
    ...
endinterface
```

This allows the user to define interfaces in terms of an abstract protocol, rather than exposing the details of its implementation. This eliminates duplication of these details, simplifying the resulting code, and reducing the chance of bugs.

Shortcomings

While the new simulation and verification features are useful additions to the design process, SystemVerilog adds relatively few features which improve the actual hardware description capabilities of the language. The concept of `task` based interfaces is a promising idea, but on its own represents a relatively small improvement in expressiveness over Verilog.

Lava

Lava is an experimental HDL, originally created at Chalmers University[2], which has served as a basis for a variety of research into language design. Variants exploring different possibilities have been developed by other institutions[3][4], and even

FPGA manufacturer Xilinx[5]. The language is based on a dataflow model of hardware, which is composed from gate level primitives using a Haskell DSL. A similar technique is used by this project, and indeed this is the origin of the name "Mantle".

Dataflow

Lava uses a dataflow model of hardware, representing components as functions over an abstract `Signal` type. For example, the `and` function of type `Signal -> Signal -> Signal` implements an AND gate. From primitives like these more complex components can be composed. State is introduced using a special `delay` function which delays value of the input `Signal` by one clock cycle. This model allows most common functional combinators from the Haskell standard library to be used without modification. This allows experienced functional programmers to apply familiar patterns to the domain of hardware design.

Observable Sharing

One problem with using functions to represent dataflow, is that it doesn't provide a way to express feedback loops. One solution offered by Lava is a special `loop` combinator which takes a function of type `(a,Signal) -> (b,Signal)` and feeds back the signals to give just the `a -> b`. However this is rather awkward, so Lava also provides an alternative solution.

As a result of it's lazy evaluation semantics, Haskell allows values to be defined recursively, creating infinite structures. If we consider the `Signal` type to represent the sequence of values a wire takes over time, we can use this lazy evaluation to describe feedback:

```
tFlipFlop :: Signal -> Signal
tFlipFlop en = val
  where
    val = xor en prev
    prev = delay val
```

Unfortunately, while this model is conceptually sound, it cannot be used to produce finite hardware. Indeed as a result of Haskell's referential transparency, it is impossible to determine the original recursion pattern, even if an abstract structure is used. This is because there is no way to determine whether two positions in this structure are represented by the same location in memory, or merely equal. In order to avoid this problem, Lava uses a Haskell extension called Observable Sharing to allow these checks to be made.

Shortcomings

While the Observable Sharing extension is convenient, it also necessarily breaks referential transparency. The original Lava paper defines a constrained call-by-need semantics, which avoids any problems this might cause. It then argues that most implementations respect these constraints, and further that they are typical properties of compiler design in practice. Nonetheless, as these constraints are not part of the standard, there is no guarantee that future implementations will never choose to break them. This is a particular concern given the range and complexity of extensions that are being added by some implementations.

Another more practical problem with Lava is that modelling components as functions can make it awkward to construct interfaces with multiple bidirectional subinterfaces. For example, if we consider a protocol using a feedback signal to request new data, this can be modelled on its own as a function `Signal -> Signal`, but it is less clear how to model an interface which combines two of these. `(Signal,Signal) -> (Signal,Signal)` is sufficient, but doesn't make the relation between the signals clear. New datatypes could be used, but without a way to compose them from smaller interfaces, these would each need to be constructed specially.

Bluespec

Bluespec is a commercial hardware description language based on Haskell. It was originally created at Sandburst for designing 10Gbit routers, and is now being developed by a spin-off company Bluespec, Inc[6]. The language has shown to be effective tool for architectures research at Cambridge[7].

Bluespec deserves special mention here, as my experiences using the language (both positive and negative) as part of a research internship were the inspiration for this project.

Guarded Atomic Actions

Bluespec uses a model of *Guarded Atomic Actions* to describe the behaviour of hardware. These are transactions on the state of the system, guarded with particular preconditions for their operation. For example, a rule for an enabled counter might be:

```
rule increment (enable);
    count <= count + 1;
endrule
```

These transactions preserve the usual ACID semantics as one might expect from a database, that is, they are conceptually considered to operate sequentially. This model makes reasoning about correctness far easier than the unconstrained RTL model used in Verilog and VHDL.

The language itself describes a hierarchy of modules, each containing registers, and a set of these behavioural rules. These are then expanded and scheduled by the compiler, to give a synchronous hardware design implementing them.

Methods

In Bluespec, communication between modules is done in a somewhat object oriented fashion. Each module declares a predefined interface type, which contains a set of methods, and possibly sub-interfaces. The module then defines each of these methods in terms of its registers, and the methods of its sub-modules.

Methods can return pure value types, constructed from the current state of the module, or alternatively they can return `Action` values to allow state modifications. These are fragments of transactions, which can be incorporated into the behavioural rules of other modules to make the relevant state changes.

Like rules, methods can also include preconditions required for their operation. For example, a method to get a result could be conditional on an operation having completed:

```
method Int#(32) getResult() if (complete);
    return result;
endmethod
```

These preconditions automatically propagated and added to any rule or method which uses them. This allows complex operations requiring the synchronisation of multiple modules to be written very easily.

Imperative Sub-language

In practice many designs use the transaction semantics to implement explicit state machines, with a register holding the state, and separate rules conditioned on each of the values it can take. Furthermore these state machines often describe typical control flow sequential logic. This results in common patterns being used to implement concepts such as `while` or `for` loops. In order to simplify the creation of these kinds of state machines, and to make the flow of control clearer, Bluespec includes a

simple imperative sub-language. This can be used to generate these same patterns of state registers and rules.

For example, while a counter can be adequately described using rules, one could use the imperative sub-language instead like so:

```
mkAutoFSM(seq
    while (True) seq
        count <= count + 1;
    endseq
endseq);
```

Shortcomings

While Bluespec's imperative sublanguage is one of its strengths, it also highlights some limitations of the language. Combined with the implicitly synchronised methods, it seems like it should allow designs to be written in a communicating process style. However, it turns out this is far less practical than one might imagine, due to two properties of the language.

Firstly, as modules' interfaces are implemented as methods, the flow of control is inverted. External communication must be done in reaction to external requests, rather than being driven by the module. This can be partially solved by passing interfaces into modules, but this ultimately moves the problem somewhere else. The only real way to have a local process push and pull from external channels is to bridge the gap with a FIFO or similar. Not only does this require boilerplate to define and connect, it also introduces cycle latency which may not be desired.

This leads on to the other problem. One might suggest that the latency could be avoided by using a channel module with a directly connected pair of `Get` and `Put` interfaces, but it turns out that such a module is impossible to create in Bluespec. This is due to the atomicity of the transactional model. Since each method call is a separate transaction, they must be performed in some order. However, since each method needs to know whether the other is being called, they cannot be ordered. The closest possible design is the built in `BypassFIFO`, which passes a result straight through if both sides are ready, or stores it otherwise. This shows an example of reasonably sensible behaviour which is restricted by the hardware model of Bluespec.

Another problem with Bluespec is that the implicit conditions of methods can only be used in a very limited way. They are automatically propagated to rules that use them, but they cannot be directly accessed. This prevents designs from changing their behaviour based on the availability of a method. For example, one might want to dynamically select from several methods based on their availability. Obviously one can use explicit signals for the same purpose, but these must be propagated

manually. Furthermore, many library modules do not provide such signals, and would need to be either extended or replaced to add this functionality.

One other problem with Bluespec is that its syntax has been altered quite drastically from that of Haskell. The intention being to make the language more familiar to users of other HDLs. Many of these changes are cosmetic, but in some cases important syntax sugar has been removed. Most notably, Haskell's `do`-notation for monads has been replaced with a special case syntax for its `Module` monad. As a result, there are many highly useful abstractions in Haskell which cannot be used effectively in Bluespec.

Haskell for DSLs

It might seem like a rather strange choice to use Haskell, a high level pure functional language, as a basis for the inherently low level domain of hardware description. Indeed, the semantics of Haskell itself are entirely unsuited to describing hardware. Most notably the language is garbage collected, and reducing this to a finite state implementation would be extremely difficult, not to mention undecidable. The trick here is that we will be using Haskell as a meta-language to construct descriptions of hardware in a more appropriate model.

In order to allow users to construct these descriptions in a familiar way, we create an embedded DSL which more closely matches existing HDLs. Haskell has some relatively unique features which make it particularly effective at creating DSLs in a variety of styles, which still fit neatly into the idioms of the language. The rest of this section will discuss these features.

Overloading

One of the most useful properties of Haskell for creating DSLs is that almost every aspect of the language can be overloaded. While this might initially sound like a recipe for disaster, the language's concept of type classes allows most of this overloading to be done in a very principled way.

Type classes are related groups of functions, parametrised with a type variable, which can be independently implemented by any type in the language. As a simple example, we will consider the `Show` type class, used for generating textual representations of structures:

```
class Show a where
    show :: a -> String
```

This declares a new type class with a single function `show`, which takes a member of the class (the type parameter `a`) and returns a `String`.[1] We can now implement this function for booleans:

```
instance Show Bool where
    show True  = "True"
    show False = "False"
```

We can also implement polymorphic functions which work for any `Show` instance:

```
print :: Show a => a -> IO ()
print x = putStrLn (show x)
```

We can also use this technique to define `show` recursively for structures of `Show` instances:

```
instance Show a => Show (Maybe a) where
    show (Just x) = "Just " ++ show x
    show Nothing  = "Nothing"
```

Many functions in the standard library are implemented in terms of type classes, and as such they can be overloaded for arbitrary types simply by defining the relevant instances. However, fixing the types these instances must use helps to enforce reasonably sensible overloaded implementations. In addition the documentation of some type classes include extra rules which instances should respect.

One other property of Haskell which makes this technique particularly effective, is that it allows arbitrary symbolic infix operators to be defined. This allows common operators that are part of the grammar in other languages, to be defined in the standard library, and take advantage of the same overloading technique:

```
class Eq a where
    (==) :: a -> a -> Bool

instance Eq Bool where
    True  == True  = True
    False == False = True
    _     == _     = False
```

Some extensions even allow some aspects of Haskell's syntactic sugar to be overloaded, by redefining the equivalent functions. For example, this allows Haskell's `if-then-else` syntax to be overloaded by defining the `ifThenElse` function.

[1] The actual definition has a few more methods, but these are irrelevant in most cases and are omitted here for simplicity.

Monads

Monads are a very general abstraction for composing computations which have additional context that needs to be handled. They can be defined in various equivalent ways, but perhaps the most familiar to type theorists is in terms of effect arrows of the form `a -> m b`, where `m` is the monadic type constructor. This is similar the more typical notation $a \xrightarrow{F} b$, although here the effect is represented as a concrete algebraic datatype which wraps the result type. A monad then provides a way to compose these arrows, and an identity for this composition:

```
class Monad m where
    identity :: (a -> m a)
    compose  :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

These functions are called `return` and `(>=>)` respectively in Haskell's standard library, but we have renamed them here for readability. As a more practical example, we will consider Haskell's option type `Maybe`. The effect this represents in the context of an arrow, is the possibility of producing no result. Arrows of the form `a -> Maybe b` can be composed by short-circuiting the `Nothing` result:

```
instance Monad Maybe where
    identity = Just
    compose f g x = case (f x) of
        Just y  -> g y
        Nothing -> Nothing
```

This is a simple example, but the same pattern can be applied to much more complex effect systems. Probably the most well known example of this is their use in Haskell to allow access to IO without sacrificing the purity of the language[8]. This is done by constructing a representation of IO actions, which can then be used as an effect arrow and composed to create complex programs. To make this arrow composition convenient, Haskell has a special syntax for working with monads, called `do`-notation. Essentially this allows a chain of effect arrows to be written on separate lines. Using this syntax, code using the `IO` monad can be written in a familiar imperative style:

```
main = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn "Hello, " ++ name
```

Furthermore, since these blocks are first class values in the language, they can be manipulated by functions, allowing many typical control flow structures like `while` loops to be implemented as functions. Another way of looking at this, is that the the IO system in Haskell is *itself* an embedded DSL for describing imperative programs.

The same notion which makes IO convenient can also be used with a wide variety of other monads, each forming a small DSL for a specific purpose. It also allows more extensive DSLs to be created, by defining custom monads to provide any effects needed, and then creating functions matching the desired syntax. This technique will be used in the creation of Mantle.

Requirements

There are two aspects to the design of this project, each with their own requirements: The underlying model of hardware and the language built on top of it.

The hardware model should be sufficient to describe simple synchronous designs. The system should then be able to translate these designs to Verilog code which matches the expected behaviour. Finally this Verilog code should be reliably synthesisable.

The language should be at least as expressive as Verilog, and hopefully better in this respect. It should have a relatively familiar syntax for users of traditional RTL languages. Finally, it should enforce type safety properties on the hardware designs it describes.

Chapter 3

Implementation

This chapter will describe the techniques used in the construction of each element of the Mantle DSL and its underlying model, as well as some abstractions built on top of this system.

The first three sections introduce a basic syntax roughly equivalent to that of Verilog. The next section then discusses the underlying model used to represent this, and the monadic framework used to construct it. At this point, the work discussed comprises a simple, but nonetheless complete, language and implementation.

The remaining sections use some advanced Haskell features to extend this simple language with more powerful abstractions. This starts with the a simple abstraction for propagating clocks around synchronous circuits. Then a far more general abstraction is produced, for describing hierarchical bidirectional interfaces. Finally a system of size-typed vectors is added, to allow synthesisable collections wires and registers.

Note: In some cases the code examples given in this document differ from the actual source code. Some types and functions are given different names to make their purpose clearer, and in a few cases details are omitted where they complicate an explanation.

Expressions

Probably, the simplest part of hardware description to model in a functional language are the purely combinational expressions. As we have seen in the Preparation chapter, we can overload common operators by implementing certain type classes. This includes basic arithmetic operators such as the `+` used in the `counter` example. So to get a familiar expression syntax, all we need to do is create a new type representing synthesisable combinational logic. We will implement these as abstract

expression trees, using the same constructs as are available in the target language Verilog. This tree structure can be defined easily as an algebraic datatype:

```
data Expr = Add Expr Expr
          | Sub Expr Expr
          | ...
          | And Expr Expr
          | Or  Expr Expr
          | ...
```

This expression structure provides no type information about the values it represents. There are a couple of reasons this simpler structure has been chosen. Firstly, it allows expressions of different types to be stored together in containers without complex type machinery, and secondly it exposes all the capabilities of Verilog to users who might want to do bit level manipulation.

Nonetheless, it would be useful at the higher level to be able to restrict how expressions are constructed. We can do this in Haskell using a technique called "Phantom Typing".

The usual way of indicating what the type of an expression structure would be to parametrise the structure's type with this information:

```
data Expr a = Add (Expr a) (Expr a)
            | ...
```

This ensures that functions over this structure cannot combine expressions of different types, and also allows us to restrict which expression types certain type classes are implemented for:

```
instance Num a => Num (Expr a) where
    x + y = Add x y

fails = 2 + (5 < 10)
```

However, there is no requirement to actually use a type variable within the structure. This means we can create a wrapper type which has a type parameter, but internally just uses a non-parametrised expression type:

```
data Signal a = Signal Expr
```

We can now create functions over this new type which enforce the type safety properties we desire:

```
instance Num a => Num (Signal a) where
    (Signal x) + (Signal y) = Signal (Add x y)
```

And if we do not expose the internal structure of `Expr` by default, the user can only use the type safe functions to construct it.

Wires

While abstract expressions alone can implement the majority of combinational logic, there are a few features of hardware structures which they are insufficient to describe.

Firstly, we cannot use the same expression multiple times without duplicating it. For example if we were to write the following code:

```
addAndDouble x y =
    let sum = x + y in sum + sum
```

We would expect this to do only two additions. However, because we are using an abstract expression, the result is a tree containing three. While this might be noticed and fixed by an optimiser, we would prefer to have more precise control.

Another problem, which we have already seen in Lava, is that the functional style provides no way to describe feedback. While this has little use in purely combinational logic, it will be very important once we introduce state.

The cause of both of these problems is that the expressions have a tree structure, whereas hardware can involve arbitrary graph structures. The solution in Verilog is to allow the declaration of `wire`s which can be used in expressions, and then bound later in the code. To provide this familiar structure in Haskell, we can use the do-notation discussed in the Preparation chapter. To do this, we need to construct a custom `Circuit` monad, which will be used to store the various parts of the hardware model. Since we have not yet discussed all the aspects of this model, we will leave the precise details of this monad until after registers have been introduced. For now, it suffices to say `Circuit` provides a way to store tables of declarations and assignments.

Using this `Circuit` monad, and an abstract `Wire` type, we can now implement the desired syntax by defining functions for declaration and assignment, and one to convert `Wire`s into `Signal`s so they can be used in expressions:

```
newWire :: Circuit (Wire a)
(=:)    :: Wire a -> Signal a -> Circuit ()
rd      :: Wire a -> Signal a
```

The implementations of these functions are mostly trivial, making the relevant changes to the declaration and assignment lists. However there is one aspect which is a little more complex. In order to declare a wire in Verilog, we need to know its bit width, but this depends on the type of the `Wire`. To get this information we can use a type class to provide a function which is dependant on the type of its argument:

```
class Bits a where
    bitWidth :: a -> Int

instance Bits Bool where
    bitWidth _ = 1
```

However, since the type in question is a phantom type, we don't actually have a value to call this function with. Fortunately, Haskell's call-by-need semantics allow us to work around this. Since the function does not use its argument, it is non-strict, and we can safely pass $\perp$. This is provided in Haskell as the value `undefined`. This allows the `newWire` function to be defined something like this:

```
newWire :: Bits a => Circuit (Wire a)
newWire = do
    ref <- addDeclaration (bitWidth (undefined :: a))
    return (Wire ref)
```

Here the type of `undefined` is ambiguous, as it can validly take the type of any `Bits` instance, and so we have used a type annotation to clarify this.

Using this concept of wires we can now implement the problematic example from above as follows:

```
addAndDouble x y = do
    sum <- newWire
    sum =: x + y
    return (rd sum + rd sum)
```

Registers

One way we could implement state would be to combine Lava's notion of dataflow delay, with the mechanisms used for wires above to introduce registers as delayed wires. This would provide a relatively familiar interface of declaration and assignment, and would be sufficient to describe many designs. For example, we could implement a counter like so:

```
counter = do
    count <- newRegister
    count =: rd count + 1
    return (rd count)
```

However in more complex designs we would like to control when updates occur more precisely. For example to implement a counter with an enable signal, we would have to do something like:

```
    count =: if enable
        then rd count + 1
        else rd count
```

Not only is this slightly tedious, it also has subtlety different semantics from those desired. While the externally visible behaviour is as expected, this implementation includes an unnecessary feedback path.

Furthermore, this model would limit the language to expressing synchronous designs with a single global clock. However, the use of multiple clocks is relatively commmon, and while completely asynchronous designs are rare, many practical designs involve some instances of asynchronous behaviour. For example, many synchronous designs include an asynchronous reset signal.

At the other extreme, Verilog allows its `always` blocks to be triggered by changes in the value of arbitrary expressions. This is very flexible, but not all designs created in this way can be synthesised. Since one of our goals is reliable synthesis, we will have to use a slightly more conservative model.

We have seen that SystemVerilog provides special constrained `always` blocks which ensure that the behaviour they describe is synthesisable. Of particular interest is the `always_ff` block which covers the common case of synchronous designs. One of the restrictions it enforces, is that only non-blocking assignments can be used within these blocks. This simplifies the timing, and makes the behaviour relatively simple to translate to hardware. We will use the same restriction in our register model.

Furthermore, in order to ensure that the signals used to trigger blocks are reasonably sane, we will only allow them to be created from external inputs. This covers a large number of use cases, notably including designs which make use of multiple clock signals. Nonetheless, this is a very limiting constraint, and it may be relaxed in future.

We can implement declaration and assignment in the same way as for wires, using new functions `newReg` and `(<=:)`, and overloading the same `rd` function. But as a familiar syntax for this model, we would also like to have a construct similar to Verilog's `always` blocks. To do this, we introduce a function `onTrigger` which takes a monadic value as its last argument. If we write this argument with `do`-notation, this gives the sub-block syntax we desire:

```
    onTrigger trigger $ do
        x <=: 1
        y <=: 2
```

The other feature we would like to provide a syntax for is conditional assignment. We can do this by overloading Haskell's if-then-else syntax as mentioned in the Preparation chapter. However, as this syntax is normally functional, the else clause cannot be omitted. In order to work around this, we also provide a function `iff` which uses the same trick as `onTrigger` to provide a conditional block without an else clause.

Since the actions allowed within this block are different from the top level `Circuit` block, we must use a different monad for these actions. The details of this monad (and of `Circuit`) will be discussed in the next section.

Using these constructs, we can now completely implement the counter example:

```
counter :: Trigger -> Circuit (Signal Int)
counter clock = do
    count <- newReg
    onTrigger clock $ do
        count <=: rd count + 1
    return (rd count)
```

Hardware Model

Throughout the previous two sections we have mentioned various elements of the hardware model stored by the `Circuit` monad, namely the declarations and assignments of wires and registers. This section will discuss the construction of this monad in more detail.

We will first add a concept of external ports to this model. Most existing languages simply use the interface of their top level module to provide external ports. However, this requires that external ports in sub-modules have to be explicitly routed all the way up the hierarchy. Instead, Mantle allows modules to create external ports at arbitrary points in the hierarchy. These are created using `newExtInput/Output` functions, and accessed in the same way as `Wires`.

First, we will need a way to include all of these variable types in the expression structure we defined earlier. This can be done by extending the sum type with another case, containing a reference type which indexes the relevant part of the hardware model. We could use simple list indices for this purpose, but this requires that we have separate reference types for each type of variable, when they are all

used the same way in expressions. Instead we provide a single source of unique references, which can be used for all types of variables. In order to preserve type safety for the user, these are then supplied in different wrapper types depending on their origin. In order to provide these unique references, we will use a construction called the `State` monad.

The usual way state changes are handled in functional languages is by taking in the initial state as an argument and returning the new state. Combined with regular arguments this gives the type `a -> s -> (b,s)` where `s` is the state argument. Looking carefully, we see this type is of the form `a -> m b` required for a monadic effect arrow. Furthermore, if we rearrange the type to the equivalent form `(a,s) -> (b,s)`, we see that the identity and composition operations for this arrow are just the same as those for pure functions. This structure is provided in the Haskell standard library along with `get`, `put` and `modify` functions for working with it.

We can use this monad to provide a supply of unique references, by storing an `Int`, and using a function `newRef` which increments this value, and returns the previous value. This ensures that each access gets a unique reference. We could also use this monad to store the rest of the structure, but as we will see shortly, there is another more appropriate monad we can use for this purpose.

Using these reference values we can now construct the hardware model structure. This is a new data type `RTL`, which consists of three tables, for declarations, combinational assignments and register assignments. Although there are four different types of declaration, they all store essentially the same information, namely their associated reference, and their bit width. As a result, these can all be stored in the same table, with their types distinguished by a tag value. The combinational assignment table covers both wire and output assignments. No tag types are required here, as the syntax for `assign` statements is the same for both. This table simply maps references to expressions.

However, the register assignment table is somewhat more complicated, involving two new structures. Firstly, `Trigger` a representation of transition events, which is simply a set of references with edge directions. Secondly, `Block` a tree structure mapping conditions (`Expr`s) to then and else sub-blocks, and also containing lists of register assignments at each node.

An interesting property of this hardware model, is that it is entirely compositional. That is, all language features operate by adding entries to the structure and we never make any destructive edits. To be more precise, the model forms a monoid: It has an associative composition operation, and an identity under this composition. This allows us to use the slightly more constrained `Writer` monad to construct it, rather than the `State` monad we used for references.

The effect arrows in the `Writer` monad have the form `a -> (w,b)` where `w` is a monoidal value. These arrows are composed using the monoid composition operator to combine each of the extra results, and the identity arrow simply returns the monoid identity as its extra argument.

This additive process ensures that nothing is overwritten by accident. This is especially important with the recursive `Block` structure, where it might be possible to replace an entire subtree by accident.

In order to combine these two monads together we use a monad transformer. This is a type constructor which takes another monad as an argument, and includes it in its structure. The monad definition for the overall structure then composes the submonad as well as its own features. In this case, the overall structure is:

```
data Circuit a = Circuit (StateT Int (Writer RTL) a)
```

Finally we need to translate this model into Verilog code. This turns out to be relatively trivial, as the structure matches that of a Verilog program pretty closely. The generator merely needs to deconstruct the tree, and compose corresponding code. In order to give slightly nicer output we use a pretty printing combinator library[9], which use a set of combinators to construct a document structure. From this, neatly formatted source can be generated.

Synchronous

Although the possibility of asynchronous behaviour is useful in some cases, in practice the most common use case will still be globally clocked synchronous designs, possibly with an asynchronous reset signal. These forms a common pattern where each module takes the clock and reset signals as arguments, and uses them to create an `onTrigger` block of the correct structure. Ideally we would like some way to abstract away this boilerplate.

The pattern of a set of functions each needing access to a common argument can be abstracted by using a `Reader` monad. The monad arrows here have the form `a -> r -> b`, where `r` is the type of the common argument, and are composed by passing the same argument into each subarrow. The identity arrow merely ignores the extra argument, and returns the first.

However, we are already using the `Circuit` monad to store a hardware model. We could make a global clock accessible in this monad, but this would restrict the user to using this abstraction with a single clock. Instead we can layer a new monad on top of `Circuit`, by using another monad transformer.

We can now define a function `onClock`, which creates a triggered block from the implicit clock and reset signals. Register assignments within this block will occur at the clock edge. For resets, the situation is a little more complex. We could just use an equivalent `onReset` block, and indeed one is provided, but in most cases such a block should only contain constant valued assignments. In order to help enforce this, we instead use a trick from Bluespec, and provide a function `reg` which takes

a reset value, and constructs a register with a corresponding reset assignment. This also puts register declaration and initialisation in the same place, making the code easier to understand.

This now allows us to write the counter example like so:

```
counter :: Synchronous (Signal Int)
counter = do
    count <- reg 0
    onClock $ do
        count <=: rd count + 1
    return (rd count)
```

Interfaces

Using functions as modules works reasonably well for these simple examples, but as we have seen in Lava, it becomes less practical for larger modules. The addition of wires helps to solve the feedback problem, and by constructing structures of these we can get something like the bidirectional hierarchies we would like.

However, it can become very difficult to keep track of which wires should and should not be assigned to when they travel beyond the scope of a single module. We can help solve this problem by splitting wires into a `Signal` and a separate `Input` type, which can only be assigned to.

The model of interfaces use in Mantle extends this solution to provide hierarchical structure of directional wires. These are created in opposing pairs, where each wire in one structure is the opposite direction to the corresponding one in the opposing structure. This is similar to the `modports` used by SystemVerilog, but specialised to the common case.

In order to create these, we first define two uninhabited types which we use to indicate direction:

```
data Outer
data Inner
```

We then extend the `Signal` type we have used so far with an extra type parameter which will hold one of these types to indicate its direction. Since the data contained in this structure will need to depend on this direction type, we use Haskell feature called data families to allow this. These are similar to type classes, but where a type constructor is defined instead of a group of functions:

```
data family   Signal d a
data instance Signal Outer a = Output Expr
data instance Signal Inner a = Input Ref
```

We can now create new data types, which propagate a direction type parameter to several `Signal`s:

```
data Pair d a = Pair (Signal d a) (Signal d a)
```

We can also use these composite interfaces to create new ones hierarchically:

```
data TwoPairs d a = TwoPairs (Pair d a) (Pair d a)
```

However, at the moment all these `Signal`s are in the same direction. This is useful, but ideally we would like to construct bidirectional interfaces. To allow this we use another Haskell feature called type families. Similar to data families, these are type dependent synonyms. They can be used to define what are essentially type level functions. We define a type family called `Flip` which maps each direction type to its inverse:

```
type family   Flip d
type instance Flip Inner = Outer
type instance Flip Outer = Inner
```

We can now use this to define bidirectional interfaces, and even flip sub-interfaces hierarchically:

```
data InOutPair d a =
     InOutPair (Signal (Flip d) a) (Signal d a)
data InOutQuad d a =
     InOutQuad (InOutPair d a) (InOutPair (Flip d) a)
```

Finally, we need to provide a way to create a connected pair of opposing interfaces. To do this, we create a type class called `Interface`, which provides a function `newIfc` to do this. In order to give this a type, we also provide a type family `FlipIfc` which maps interfaces to their opposing type:

```
type family FlipIfc ifc

class Interface ifc where
    newIfc :: Circuit (ifc, FlipIfc ifc)
```

This is then defined for the `Signal` type, by creating a new wire, and returning corresponding `Input` and `Output` values. We can then create other instances by simply recursing over every subinterface like so:

```
type instance FlipIfc (Pair d a) = Pair (Flip d) a

instance Interface (Pair d a) where
    newIfc = do
        (outerX,innerX) <- newIfc
        (outerY,innerY) <- newIfc
        return (Pair outerX outerY, Pair innerX innerY)
```

Originally one planned extension was to eliminate this boilerplate using Template Haskell, a meta-programming system. However this system turned out to be rather complicated, and since this wasn't necessary to continue, it was abandoned in favour of other extensions.

Components

Using this construct we can now write create modules with arbitrary interfaces using the following pattern:

```
example :: Circuit (ExampleIfc)
example = do
    (ext,inner) <- newIfc
    ... inner ...
    return ext
```

This is already much better than using individual wires, but we can abstract this further. Here the body of the module is essentially a function of type `FlipIfc ExampleIfc -> Circuit ()` taking `inner` as its argument. By defining a function called `make` which takes functions of this form, and implements the pattern above, we can eliminate this boilerplate and just define the function for the body. This turns out to be a very convenient interface:

```
enCounter :: FlipIfc (Input Bool, Output Int) -> Synchronous ()
enCounter (enable,out) = do
    count <- reg 0
    onClock $ do
        iff (enable) $ do
            count <=: rd count + 1
    out =: rd count
```

In order to simplify the type signatures we also define some type synonyms for functions of this form:

```
type Component ifc = FlipIfc ifc -> Circuit ()
type SyncComp  ifc = FlipIfc ifc -> Synchronous ()
```

This allows `enCounter`'s type to be written as simply `SyncComp (Input Bool, Output Int)`.

There is however, one issue with this construction. As `FlipIfc` is a type synonym, rather than a real type constructor, the type `Component ifc` does not actually involve the type `ifc` at all. This can cause problems when we want to make the behaviour of a function depend on this type. In order to get around this, we introduce a special `VoidIfc ifc` phantom type, and change the type of `Component` to `FlipIfc ifc -> Circuit (VoidIfc ifc)`. This fixes the problem, but now requires that all of our components return `VoidIfc ifc`. To make this slightly neater, we create a function `endComponent` which can be included at the end of each component definition to do this. There may be a better alternative to this, but this was the least awkward compromise of the various structures tried.

Channels

Often in circuit designs, a particular operation cannot be completed in a single cycle. This may be because it has to wait for an external input, or it may simply be that the propagation delays of the combinational logic required are too long for the target clock frequency. In these cases, a common solution is to include a signal indicating when the data is valid. Similarly, a receiving component may need a signal to indicate when it is ready to take new input. Creating and handling these extra signals involves extra logic, which will often involve very similar patterns. To avoid this, we would like to combine these signals into a single structure, and use functions to provide these patterns.

Using the interface abstraction discussed earlier we can create a new `Channel` interface which includes these extra ready signals:

```
data Channel d ifc = Channel {
    channel :: ifc,
    valid   :: Signal d Bool,
    ready   :: Signal (Flip d) Bool
}
```

Here we have chosen to parametrise the structure with an interface type, rather than just the value type of a signal. There is a reason for this beyond simply making the

structure more general, and this will be explained shortly. We will also define type synonyms for some common cases:

```
type OutChan a = Channel Outer (Output a)
type InChan  a = Channel Inner (Input a)
```

We can now create functions which compose these channels in various ways. One simple pattern is mapping a function over the result of a channel. This is easy to implement in hardware, by adding the extra compositional logic to the result signal. Since this does not alter the validity of the result, we can simply pass through the original control signals:

```
chanMap :: (Output a -> Output b) -> OutChan a -> OutChan b
chanMap f (Channel wire valid ready) =
    Channel (f wire) valid ready
```

In Haskell, data structures which can be mapped over are called *Functors*, and there exists a correspondingly named type class for these types. The `Functor` type class has a single method `fmap` of type `(a -> b) -> f a -> f b` where `f` is the functor. Therefore, we would prefer to make `Channel` an instance of `Functor` rather than implementing mapping as a separate `chanMap` function. As the functions we would like to map are of the form `Output a -> Output b` rather than simply `a -> b`, we have chosen to parametrise the `Channel` data type with an interface. We can therefore define `fmap` in the same way as the `chanMap`:

```
instance Functor (Channel Outer) where
    fmap f (Channel wire valid ready) =
        Channel (f wire) valid ready
```

Another common composition is to combine the results of two channels with a merging function. In this case we will have to use some logic to combine the control signals as well. The `valid` signals can simply be combined with `&&`, but the `ready` signals are a little more complicated. If we just feed the same signal into both of them, this will allow either channel to register a successful transfer when the other channel is not valid. So we need to make each `ready` signal conditional on the `valid` signal of the other channel.

More generally we would like to merge arbitrary numbers of channels, with functions of the corresponding arity. There is another type class in Haskell called `Applicative` (short for applicative functor) which abstracts this pattern nicely, although the way it does so is not immediately obvious. `Applicative` provides the infix method `<*>` of type `f (a -> b) -> f a -> f b`. This is similar to the type signature of `fmap`, but this time the function is also wrapped by the functor. While it may seem slightly strange to have a channel interface containing a function, this allows us to store a

partially evaluated result along with some control signals. The definition of `<*>` then applies this function, and combines these control as discussed above. Combined with `fmap`, this allows us to apply functions of arbitrary arity in a relatively convenient way:

```
mergedChan = fmap merge3 chan1 <*> chan2 <*> chan3
```

In fact it is more typical to use an infix version of `fmap` with the symbol `<$>` to make this pattern clearer:

```
mergedChan = merge3 <$> chan1 <*> chan2 <*> chan3
```

Finally, we will briefly mention one other useful pattern which also fits into an existing Haskell type class. We would often like to choose between several channels based on their availability. The method `<|>` of the `Alternative` type class is intended for this purpose. Once implemented, it allows us to compose several channels of the same type into a single channel, which exposes the leftmost valid channel of those currently available. This is a simple implementation which doesn't provide any fairness guarantees, but is still useful in many cases. A more complex function could no doubt be made for those cases where fairness is necessary.

We can now construct a counter which returns its values through a channel:

```
chanCounter :: SyncComp (OutChan Int)
chanCounter (Channel value valid ready) = do
    count <- reg 0
    onClock $ do
        iff ready $ do
            count <=: rd count + 1
    valid =: true
    value =: rd count
    endComponent
```

Vectors

We would often like modules to provide a number of identical interfaces in parallel. In order to do this, we need to make an `Interface` instance for a collection type, but none of the usual ones are suitable. Since they can vary in size, we do not know how many wires to instantiate for any particular collection interface. In order to fix this, we need a collection type which encodes its size.

In order to do this, we must first create a type level representation of natural numbers. There are various ways to do this, but the simplest is to use Peano arithmetic. To do this we define two empty types representing zero, and the successor function:

```
data Zero
data Succ n
```

We can then use a type class to expose this numeric information in the value domain:

```
class Natural n where
    valueOf :: n -> Int

instance Natural Zero where
    valueOf _ = 0

instance Natural n => Natural (Succ n) where
    valueOf _ = valueOf (undefined :: n) + 1
```

Finally we just need to include this type as a phantom parameter for a special collection type:

```
data ListN n a = ListN [a]
```

As this number now at the type level, we can then define an `Interface` instance for `ListN` which uses valueOf to work out how many subinterfaces to fill it with.

Mantle actually uses an existing library for these structures, in order to take advantage of the higher performance `Vector` type, rather than plain linked lists. This also provides all the standard collection operations, instead of having to reimplement all of these manually. For example, we can declare a vector of registers like so:

```
    regVec :: Vec D8 (Reg Int) <- replicateM (reg 0)
```

Since the `replicateM` function could be used for any size of vector, we have to specify this size by providing a type signature. Here we have used a type synonym `D8` rather than writing out `Succ (Succ ...` manually. A library in Mantle provides these up to D999 for convenience. The latest Haskell compiler also includes an extension allowing plain numbers to be used to represent numeric types, but this feature has not yet been adopted by the library used here.

Chapter 4

Evaluation

There are two aspects of this project which must be evaluated: The concrete requirements of the implementation, and the more subjective qualities of the language design. In order to examine each of these, a series of examples has been created, which exhibit the majority of Mantle's features. The first section of this chapter briefly explains each of the examples, and then the second and third use them to evaluate the quality of the implementation and of the language design respectively.

Examples

All of the examples have a clock and reset input in addition to the ports mentioned below.

Counter

This simple module has a single integer output which increments with the rising edge of the clock.

ChanCounter

This extends the previous counter example with a channel interface. Abstractly this module provides an incrementing sequence of integers. More concretely, the valid signal should remain high permanently, and when the ready signal is high, the value should increment on the rising clock edge.

Fibonacci

This module provides the Fibonacci sequence in the same way as the ChanCounter above.

FIFO

This module has input and output channels, and stores a single value at a time. This is a two state FIFO which accepts values and outputs them in separate cycles.

SumNetwork

This module has eight integer input channels, and when all of these are valid, it takes in these values and outputs their sum. In order to test this, a fifo has been added to the end, as otherwise this module is entirely combinational. This example was chosen to demonstrate channel merging and tree folding.

MovingAverage

This module has one integer input and one integer output. It stores the values seen on the input at the last four clock edges, and outputs their average. That is, it outputs a four point moving average of its input. This example was chosen to demonstrate potential for DSP applications.

Implementation

There are three key requirements the language implementation must fulfil in order to be considered a success.

Firstly, it must take designs created using Mantle, and produce Verilog code matching their intended behaviour. This will be confirmed using a set of simulated test cases corresponding to each of the examples.

Secondly, the Verilog code produced must be synthesisable. This will be confirmed by attempting to synthesise each design using a commercial tool.

Finally, the synthesised designs must be comparable in hardware utilisation efficiency to equivalent ones created using existing tools. This will be confirmed by duplicating each example in SystemVerilog, and comparing the results on the basis of several design metrics.

Simulation

Each program was connected up to a test bench written in SystemVerilog, which manipulates the inputs, including the clock signal, and checks the outputs with assertions. These were then compiled and simulated using *ModelSim Altera 10.1b*.

It was decided to use an existing mature verification tool, to provide a reliable basis for testing, rather than developing a testing framework within Mantle.

Each of the tests runs through a typical usage cycle, checking the state of the outputs after every modification of the inputs. This included ensuring that no changes occurred except during the clock cycle. These cycles are then repeated a number of times to ensure consistent results.

It is perhaps notable that only a few bugs were discovered during these tests, and almost all of these were due to misunderstandings of Verilog syntax or semantics, which were quickly resolved. This success can most likely be attributed to Haskell's strong type system catching most bugs during type checking. After these initial bugs were fixed, all the examples behaved as expected.

Synthesis

The Verilog output for each design was synthesised using *Quartus II 12.1*, for the Cyclone IV chip present on the Terasic TPad. All examples compiled with no errors, or significant warnings.

A few of these synthesised designs were subsequently tested on the corresponding hardware, by connecting their ports to the board's IO devices, and manipulating them manually. This process was not practical for extensive testing, but served to ensure that a complete design flow was possible, from Mantle code to working hardware. The Counter and FIFO examples were tested in this way, and both behaved as expected.

Each of the examples was then duplicated in SystemVerilog, and the logic elements used, and the Fmax of each example compared. In every case these measurements were close to identical. However, a larger example may be necessary to show any differences.

Language Design

There were also three key requirements for the language design to be considered a success.

Firstly, it must provide a similar level of expressiveness to Verilog. The resulting language has equivalent features for most of the synthesisable constructs of Verilog, as it copies the structure of SystemVerilog's synthesisable `always` blocks. There are a few more obscure features that are missing, but these could likely be added by defining a few new functions at such time as they become necessary.

Secondly, it should provide a familiar syntax to traditional RTL languages. The syntax of Mantle is mostly structurally similar to Verilog, except in a few cases where it eliminates complexity. There are a few warts, like the necessity of `$` before the introduction of some `do` blocks, but these are reasonably easy to learn.

Finally, the language should enforce type safety properties of the designs it allows. Mantle provides and checks types for all combinational expressions, and even uses more advanced type structures to provide directioned signals, and even sized vector types.

Comparisons

Versions of each of the above examples were produced in SystemVerilog for the analysis above, and a further version was produced in Bluespec. The number of lines of code used for each was recorded.

Lines of Code	Mantle	SystemVerilog	Bluespec
Counter	7	14	n/a[1]
ChanCounter	9	17	8
Fibonacci	11	19	10
FIFO	14	28	16
SumNetwork	6	35	9
MovingAverage	4[2]	15	7

There are far too many complicating factors to draw any real conclusions from this data, not least the dubious worth of the metric itself, but it does at least suggest that Mantle may be able to produce more concise hardware descriptions than existing languages in some scenarios.

[2]Creating a non-guarded output in Bluespec would have been relatively complex, and is unlikely to have been a very useful example.

[2]This relies on an relatively specific library function, and is probably not a fair comparison.

Chapter 5

Conclusion

The goals of this project were to design a model of hardware, produce a DSL to construct it, and explore new abstractions for hardware description. All of these were successful achieved. The hardware model developed is simple but flexible, and can be reliably translated into synthesisable Verilog. The DSL Mantle provides a familiar syntax for traditional RTL programmers, while still allowing advanced users to take advantage of Haskell's features. And a new system of interface types has been created, which allows low level protocols to be abstracted and controlled with higher level operations.

Overall the design demonstrates a solid language basis with lots of potential for developing new abstractions. One possible extension would be a control flow sub-language much like Bluespec's, this time allowing it to be used as part of a communicating process style.

Obviously, this is a proof of concept and more effort would be required to create a complete toolchain and comprehensive library. Indeed, I intend to continue development beyond the work shown here, hopefully as part of a Masters or PhD course. It is possible that these longer term goals may have steered focus towards practical concerns outside the scope of the immediate project. For example, developing a model with support for asynchronous behaviour, where a simpler model may have allowed even more interesting abstractions to be developed.

Nonetheless I feel that a lot has been achieved in the time allowed, and the technique of constructing DSLs in Haskell shows a lot of promise for future HDL research, as well as a more general tool for tackling problems outside the scope of typical programming models.

References

[1] D.I. Rich, *IEEE Design Test of Computers* **2003**, *20*, 82.

[2] K. Claessen, *Embedded Languages for Describing and Verifying Hardware*, Chalmers University of Technology, **2001**.

[3] A. Gill, in *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Acm, New York, NY, USA **2009**, 117.

[4] M. Naylor, C. Runciman, *The Reduceron Reconfigured*, **n.d.**

[5] S. Singh, in *2000 IEEE Symposium on Field-Programmable Custom Computing Machines*, **2000**, 145.

[6] P. Hudak, J. Hughes, S.P. Jones, P. Wadler, in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, Acm Press**2007**, 1.

[7] S.W. Moore, P.J. Fox, S.J. Marsh, A.T. Markettos, A. Mujumdar, in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, **2012**, 133.

[8] S.P. Jones, in *Engineering Theories of Software Construction*, Press**2001**, 47.

[9] D. Leijen, *University of Utrecht* **2001**, *10*.

Computer Science Project Proposal

Building HDL abstractions from an RTL model in Haskell

Alex Horsman, Homerton College

11th October 2012

Project Supervisor: Dr. Simon Moore

Director of Studies: Dr. Bogdan Roman

Project Overseers: Dr. Peter Robinson & Dr. Robert Watson

Introduction

Currently, the only widely supported hardware description languages are Verilog and VHDL. While these languages are a vast improvement on older techniques of manual circuit design, as designs become ever more complex, more powerful languages will be needed.

Existing alternatives these low level languages usually provide a higher level model which is then compiled to Verilog. However in many cases, while the model is effective at designing certain kinds of system, it can be awkward to use for others. Additionally, the compiler may not be able to make use of microarchitectural tricks that would usually be used by a designer at a lower level.

In order to provide multiple powerful abstractions, while still allowing microarchitectural freedom where it is desired, the aim of this project is to start with a low level model, and build higher level models on top of it. These higher level models can then be connected together through a common interface, such that each component of a system can be created in an appropriate abstract model.

Haskell has many properties useful for this goal. Its syntax is very flexible, allowing arbitrary binary operators to be defined, and providing a notation for monadic operations. In addition, its type system allows very powerful abstractions and generalisations to be made, while maintaining type safety.

Starting Point

There are a few of relevant libraries in the Haskell package database which I may use in my project. In particular there are some libraries for working with VHDL and Verilog. It is likely I will also make use of other more general libraries from this package database as well. In addition, during the summer, I created a library for abstracting connections in Bluespec, and I may use a similar design in my language's library.

Substance and Structure

The core of the project will consist of three components. A representation for RTL circuit models, a generator to transform the representation to Verilog code, and a series of library functions for working with the representation, which should be designed to form an embedded language.

The RTL representation will essentially represent a subset of Verilog, indicating what registers exist, what logic connects them, and what signal edges trigger an update. Other features of Verilog may be included if it turns out their reimplementation at a higher level is either less efficient, or cumbersome. For example, it may prove useful to be able to describe a module hierarchy, to assist the Verilog compiler by providing synthesis boundaries. This representation will exist as a data type in Haskell.

As the representation will correspond closely to Verilog, generating this code will involve a relatively simple transformation. Each register is given a name and defined, similarly intermediate logical values must be named and assigned, and finally register update logic is produced as a set of "always" blocks representing the different triggers.

While it would be possible to use the constructors of the internal representation as a language to describe hardware directly, this is unlikely to be particularly convenient, and so a set of library functions will be written to make it easier to build circuit representations.

One key feature of the library will be to provide a common interface for communicating between circuits described at different levels of abstraction. The suggested interface is structural groups of input and output wires, which can be constrained and abstracted as appropriate in each higher level model. For example, a model based on guarded channels could constrain its wire interfaces to have appropriate guard signals, and expose these as channels internally.

Once the library is created, the project will, as an extension, explore various higher level abstractions built on top of this base. For example, one simple model which covers many use cases, is single clock synchronous circuits, where all register updates are triggered by the same clock transition.

Evaluation

During the creation of the system, a series of hardware examples will be created. This will be used to test that the system produces hardware matching specified behaviour, using a series of test cases. These hardware examples will also be used to compare with existing implementations using other languages, such as SystemVerilog and Bluespec.

The main intention of the system is that the code needed to create the hardware will be simpler, so this will be the main aspect of the comparison. This is of course, somewhat subjective, so in addition to a qualitative comparison, a few metrics will be compared, such as lines of code.

It is of course also important that any simplicity in creation does not come at a significant cost of performance in the final design. In order to test this, comparisons will also be made of logic elements used, and maximum frequencies of the designs produced in the different languages.

Success Criteria

1. An internal representation of RTL circuit description must be designed and implemented as a data type in Haskell.

2. A program must be written to transform this representation into valid Verilog source code.

3. A set of functions must be written, which provide an embedded language for creating RTL circuit descriptions, in the internal representation.

4. Simple hardware examples should be created in the language to demonstrate the project works, and that the resulting hardware designs match the expected behaviour, as defined by test cases.

5. Hardware implemented using the language should be compared with implementations in other languages, in terms of code length as well as logic elements used.

Plan of Work

1. **19th October - 2nd November:** Read papers on Lava, and other related languages. Research Verilog synthesis semantics. Set up computers with appropriate simulation and synthesis software, and familiarise myself with its usage.

2. **2nd November - 16th November:** Design and implement the underlying RTL representation. Use raw constructors to demonstrate simple toggle, counter and FIFO examples.

3. **16th November - 1st December:** Create the Verilog generator, and test with the previous examples. Demonstrate simulated or synthesised hardware.

4. **1st December - 15th December:** Implement a simple set of functions for building RTL circuits, and use them to simplify the previous examples. Construct a FIFO sequence with a fold to demonstrate use of higher level functions.

5. **15th December - 31st December:** Implement an abstraction for groups of signals. Use this to describe the FIFO's channel interface.

6. **31st December - 18th January:** Write progress report. At this stage, the project should be capable of describing simple RTL systems, and producing correct Verilog to implement them.

7. **18th January - 25th January:** Prepare progress presentation.

8. **25th January - 22nd February:** Implement a simpler interface for creating synchronous systems. Demonstrate simpler code for the examples. Explore further extensions.

9. **22nd February - 22nd March:** Write first draft of dissertation, and send to supervisor for review.

10. **22nd March - 5th April:** Break to focus on revision, and await feedback.

11. **5th April - 21st April:** Any further work as suggested. Revise and print dissertation.

Resources Required

I will be using my own computers for the majority of the project work. For backup, I will keep my project in a git repository, and regularly push it to a number of locations, including Github, Dropbox and my external hard drive.