UNIVERSITY OF MALTA

An Embedded Language for the Definition and Refinement of Synchronous Circuits

Christine Vella

A dissertation submitted in fulfilment
of the degree of Master of Science

Department of Computer Science and A.I.
Faculty of Science
University of Malta
2006

Supervisor: Dr. Gordon J. Pace

External Examiner: Dr. Koen Claessen

Other Examiners: Mr. Michael Rosner
Dr. Adrian Francalanza

In memory of my father George Vella

Abstract

SharpHDL is a language for designing, specifying and verifying hardware. It is embedded in the object-oriented programming language C# and therefore hardware definitions are treated as first-class objects in the host language. Thus, in developing our hardware objects, we are able to use object-oriented features like polymorphism and inheritance, as well as the various tools developed for C#. Being a structural hardware description language, it provides means to describe circuits by specifying the components and their interconnections. It also supports higher-order circuits, which are referred to as Generic Circuits. These are circuit descriptions which take other circuits as their input.

The language also allows definition of circuit specification by describing observers expressing safety properties. Observers are circuits that take the inputs and outputs of a circuit under observation and output a single wire stating whether the combination of inputs and outputs satisfy the given specifications. In addition, circuit descriptions can be translated to verification tools, thus enabling a designer to verify circuit specifications. Presently, SharpHDL supports translation to SMV, which is a standard model checker. The language can generate other types of description formats, including translations to other hardware description languages.

Using these features, SharpHDL was used to design and verify the equivalence of two circuit implementations of Fast Fourier Transform (FFT) algorithms — the radix-2 FFT and the radix-2^2 FFT. FFTs are efficient algorithms to compute the Discrete Fourier Transform which is widely used in digital signal processing and other related fields.

We also discuss the embedding of a simple language over SharpHDL that caters for modular verification and refinement. Modular verification allows a verification problem to be decomposed into smaller manageable sub-problems and each sub-problem is verified individually. Refinement allows the verification of an implementation against an abstract specification defining its legal behaviour. We analyze how the characteristics of SharpHDL give more user-control over the verification process, thus highlighting the various contributions given by the embedded approach.

Acknowledgments

Completion of my thesis would have been impossible without the help I received constantly from so many people. I would like to thank first and foremost my supervisor, Dr. Gordon J. Pace, for accepting to supervise this dissertation and for his constant support and constructive feedback relating to my work.

I would also like to thank Dr. Adrian Francalanza, Mr. Joseph Cordina and Mr. Sandro Spina for their constructive comments during the various stages of this work.

Thanks also goes to Mr. Michael Rosner, Head of the Computer Science and Artificial Intelligence (CSAI) Department at the University of Malta, for his help and support.

I would also like to thank the administrative staff members at the Faculty of Science and the CSAI department at the University of Malta, namely Mr. Vincent Sammut, Ms. May Lawrence and Mr. Carmelo Vella who provided practical help and answers to the many questions relating to administrative affairs.

Last but not least, I would like to thank my family and friends for their constant support and encouragement in my academic studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Hardware Description Languages (HDLs) are computer languages which cater for defining operations and design of circuits, and are normally associated with other tools like simulators. Undoubtedly, the most used HDLs are Verilog [47] and VHDL [5], both introduced in the early 1980s. Though having different syntax and semantics, the languages provide similar tools and approaches for designing circuits.

Both languages allow *structural* and *behavioural* descriptions of circuits. A structural description defines a circuit in terms of the components it uses and the interconnections between them. On the other hand, a behavioural description is given when a circuit's functionality is algorithmically defined using conventional programming language constructs.

Like most HDLs, the principle aims of Verilog and VHDL was originally to provide simulation and it was only later that these language were extended to allow hardware synthesis. Simulation allows a designer to analyze and test a design without using the actual hardware, whilst hardware synthesis allows the generation of hardware from a circuit specification.

Nevertheless, these languages suffer from a number of problems. One major problem is that circuit definitions tend to be verbose and difficult to understand and maintain. Another problem is that circuits are not treated as first-class objects. This forbids them to parameterize circuits by the used components and thus they provide poor support for higher-order circuits, i.e. circuits that use other circuits to build regular-patterned circuits. The patterns that can be described are limited to linearly-shaped networks using

specific components, whilst recursive patterns, like trees, are only described for particular sizes. Moreover, the synthesis tools are very limited — they can be only applied to a small subset of circuits. Having a complex semantics also makes the languages unsuitable to use other tools, including formal verification tools.

Formal verification techniques provide a way for specifying the correct behaviour of a system and provide an exhaustive method for determining that the designed model obeys the set specifications for any combination of inputs. Such techniques are very useful in the testing of hardware — besides reducing the cost factor of correcting errors in implemented hardware, they improve the time and cost of the design-stage testing when compared to using simulation. Using simulation, one has to create a set of inputs that are sufficient to test a system properly, and also determine the correct output which is used to countercheck the simulation output [27].

Given these simulation drawbacks and the lack of expressiveness and proper semantics exhibited by the standard HDLs, the need was felt to build simple HDLs that, besides offering the usual tools, provide means for verification. Such HDLs include μFP [79], Lava [9], Ruby [50], Lustre [38], Hawk [54] and many others [11, 23, 77]. SharpHDL [84, 85] was our solution to this problem.

1.1 Problem Definition and Primary Aims

SharpHDL provides means to structurally describe circuits using the object-oriented paradigm. It is an embedded language and therefore circuits are first-class objects in the host language. This permits circuit descriptions to be passed around as parameters, thus supporting higher-order functions. SharpHDL circuits can also be translated to descriptions for verification or simulation.

Though a valid language in its own right, SharpHDL and OO do not seem to provide any apparent advantage (or disadvantage) over the existing HDLs. Nevertheless, we notice that we did not manage to sufficiently exploit the imperative nature of the language. Therefore we turn to investigate how this nature can be used in connection with verification and the refinement process.

Formal verification methods can be grouped into two general sets: automatic verification techniques and theorem proving methods. Automatic verification techniques are able to automatically verify a fairly broad class of properties. However, being that such methods are based on the exhaustive search of a model's state space which may grow exponentially to the number of processes, they suffer from the *state explosion problem*, making them unsuitable to verify large systems.

One way to avoid this condition is to make the verification problem smaller using *modular verification*. Modular verification allows a system to be broken down into smaller parts and have each part verified separately, assuming that the other parts behave correctly. Such a technique is also useful when unimplemented sub-components with known specifications are used, such that verification of such systems assumes that the unknown sub-components conform to their specifications. These components can later be *refined* to an implementation which is tested for correct behaviour.

Various HDLs and verification languages allow modular verification using different approaches [1, 23, 39, 62]. Unfortunately, the existing languages provide little or no user-control over the verification process. In this dissertation we attempt to analyze how the characteristics of SharpHDL can help us provide more control and support in this area. Once again we use the language-embedding approach to build a simple refinement language over SharpHDL that will provide us with the necessary refinement tools.

1.2 Objectives of Dissertation

The work we present here experiments with the SharpHDL language to analyze and highlight the contributions of an embedded object-oriented language. The principal contributions are three-fold:

- **SharpHDL 2** — We present an enhanced version of SharpHDL [84, 85]. SharpHDL is a hardware description language embedded in the object-oriented language C#. Besides providing the necessary constructs to describe a circuit structurally, it also provides a set of higher-order circuits and allows the generation of descriptions to external tools like verification tools and other hardware description languages.

While keeping the same functionalities, the new SharpHDL version — SharpHDL 2 — exploits more the hierarchical and modularity characteristics of the object-oriented paradigm such that a circuit object is treated as an individual component made up of its direct sub-circuits. The first SharpHDL, on the other hand, flattened the circuit descriptions, such that circuit objects were manipulated and stored in terms of the basic boolean gates they use, thus losing all the individual component information. The drawback of this approach was largely felt in the generated code for external tools. Such descriptions were not modular and lacked reusability of component definitions, resulting in long, unreadable code which, at times, was impossible to load in the respective applications.

SharpHDL 2 solves this problem by keeping information about circuit hierarchy, such that a circuit is considered to have a number of sub-components and itself is a sub-component to a *parent* component. This approach helps in producing modular and reusable descriptions for external tools. SharpHDL 2 also eliminates some verbosity exhibited in the first version.

- **Case Study: designing and verifying the equivalence of two FFT circuits** — FFTs (Fast Fourier Transform) are efficient algorithms for computing particular Discrete Fourier Transforms (DFT) — an algorithm frequently used in various applications like telecommunications, and signal and image processing. To study the effectiveness of circuit description in SharpHDL[1], we describe two circuits, each defining a different FFT algorithm — the radix-2 FFT algorithm and the radix-2^2 FFT algorithm. Their butterfly-like structure makes them easy to implement using higher-order circuits. The equivalence of these two algorithms is then verified by defining an *observer* circuit that compares their output given the same input. The description of the whole circuit is generated and given to a verification tool which verifies their equivalence.

- **Modular verification and refinement** — We explore the use of SharpHDL to enable modular verification and refinement. We present

[1]Unless otherwise specified, the term *SharpHDL* refers to the new version of the language, i.e. SharpHDL 2

a simple refinement language over SharpHDL which allow a verification problem to be decomposed into smaller manageable sub-problems which can be verified individually. Refinement allows the verification of an implementation against an abstract specification defining its legal behaviour. By building small examples, we analyze how the imperative nature of the language provides us with a scripting language which allows us to control and document the refinement process.

1.2.1 Document organization

This dissertation is divided into three main chapters, each discussing one of the three aims outlined above.

Chapter 3 describes the new version of SharpHDL, its various tools and how they can be used. A case study is given in chapter 4 where two circuits describing different FFT algorithms are build and their equivalence verified. Chapter 5 describes the new refinement language we developed over SharpHDL to provide for modular verification and refinement and outlines the various advantages given by the embedded approach and the language's characteristics.

Finally, chapter 6 outlines some points we intend to develop in the coming future. It also discusses the major contributions of the whole research. Before we start with the main chapters, we present an overview of some background notions in the next chapter.

Chapter 2

Background Notions

The principle aim of this dissertation is to highlight the advantages an embedded language gives when applied to the hardware description language domain. It also concentrates on providing efficient verification tools based on model checking. In this chapter, we shall briefly outline these two areas.

2.1 Embedded Languages

The traditional way of developing a new language is to define its syntax and semantics and implement the appropriate programs that will process them, including lexical parsers, compilers and many others. Alternatively, one may embed the new language in an existing language. Embedded languages are basically a set of libraries in an existing language, called the *host language*, which allow us to define programs in our new language as objects in the host language [17]. This makes the new language a domain-dependent one.

A library describing an embedded language typically includes an abstract datatype, a number of primitive programs, more complex combinators and functions that manipulate the embedded programs. To better explain this we describe a simple language for specifying processor programs.

A processor takes a list of commands and processes them to give a result. Therefore, a typical processor program is made up of basic *expressions* which are manipulated by *combinators*. To process the program, the processor also needs a *run* function. Therefore, the library defining the embedded language includes the following:

abstract datatype, which is the type all the embedded processor programs are elements of.

primitive programs, which consist of the basic expressions used by the processor. These consist of data values and variables, and simple mathematical operations like subtraction, addition and multiplication:

```
data Expression =
      Val Data
    | Var Variable
    | Expression :-: Expression
    | Expression :+: Expression
    | Expression :*: Expression
```

combinators, which are elements of the abstract datatype. These use the primitive programs to build more complex programs. They include variable declaration, assignment, conditional statements and loops. The datatype `Program` can thus be defined as follows:

```
data Program =
      | Declare Variable
      | Variable := Expression
      | IfThenElse Expression (Program, Program)
      | While Expression Program
```

***run* function,** which is the function that evaluates a given `Program` and outputs the result. We name this function `simulate`. This takes a `Store` (basically, a lookup table relating variable names and values) and a `Program` and outputs an updated `Store`:

```
simulate :: Store -> Program -> Store
```

The biggest advantage of embedding a language is *reusability*. An embedded language reuses the host's

- syntax and semantics;
- parsers, interpreters, compilers and other tools;
- and the users too. Since the language uses the same syntax, a user who is familiar with the host language will not find it difficult to use the embedded language.

Another advantage is that programs are first class objects in the host language. This allows generation, manipulation and analyzes of programs using the same host language. Nevertheless, like anything else, this approach also exhibits some disadvantages. One major disadvantage is the mismatch that arises between the embedded and host languages since features or syntax of the host language may not match the desired attributes of the embedded language.

The embedded approach has frequently been used to develop domain-specific languages[1] (DSLs) — Elliot et al. [29] embedded Pan in Haskell to provide an image-synthesis and manipulation language. Also, Elliot applied the embedded approach to the multimedia animation domain and implemented Fran [28] in Haskell. The latter functional language was also used as host language to Haskore [44], a language that describes music notation. Another picture-manipulation language is FPIC [51] which provides facilities for drawing simple two-dimensional pictures using types and functions defined in Standard ML.

The embedded approach has been frequently used in the hardware-description domain too. Some embedded languages include JHDL (embedded in Java) [46], PamDC (embedded in C++) [12] and the many Haskell-embedded languages [9, 54, 70] amongst others.

In this domain, this approach was also used to combine two HDLs so as to support both structural and behavioural descriptions of circuits. For example, in [75], Sharp embeds Magma, a Lava-style structural language in the functional programming language ML and then embeds it in the behavioural HDL SAFL [76]. This enables him to use a functional language to specify hardware across different levels of abstraction. Claessen and Pace [19] also present a framework to merge structural and behavioural descriptions by embedding behavioural languages in the structural HDL Lava. Using similar examples, SharpHDL, also a HDL embedded in C#, was used as host language to another language that provided behavioural descriptions of circuits for regular languages [84].

[1] A **domain-specific language** is a programming language tailored for a particular domain [43].

2.2 Model Checking

Model checking is one of the most widely used verification techniques. It accepts a system's specification and its implementation model and, using an efficient search procedure, checks that the implementation satisfies the specification for any input. If this is not the case, the model checker generates a counter-example which shows why and when the specification was not obeyed (cf. Figure 2.1) [22, 32].

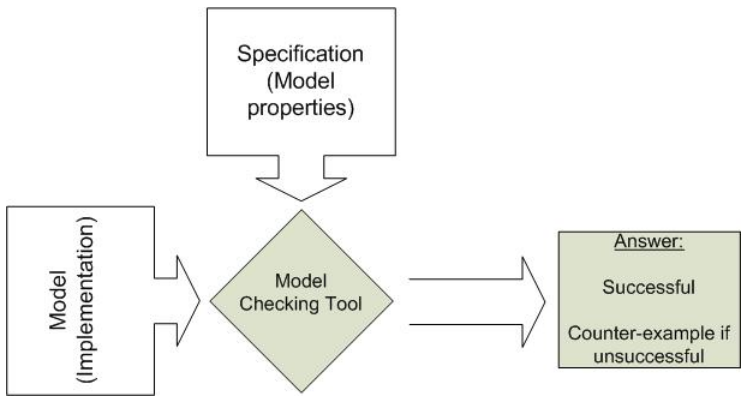

Figure 2.1: The framework of a model checking tool

An implementation model is usually expressed as a state-transition system or *Kripke structure*.

Definition 2.2.1. *A* Kripke structure *is defined as a 4-tuple* $M = \{S, I, R, L\}$ *where*

- S *is the set of states,*
- I *is the set of initial states,* $I \subseteq S$*,*
- R *is the transition relation,* $R \subseteq S \times S$*, specifying the possible transitions from state to state,*
- L *is a function that labels states with the atomic propositions from a given language.*

Specifications are often written as a *temporal logic* formula. Therefore, a verification problem can be defined as follows:

Definition 2.2.2. *Given a desired property expressed as a temporal logic formula p, and a model M having initial state s, decide if $M, s \vDash p$*

Temporal logic allows reasoning about propositions in terms of time, such that statements about the past, present and future can be expressed. Originally, this logic was developed by philosophers to investigate the use of time in natural language arguments.

One useful temporal logic is the **Computation Tree Logic (CTL)**. It is a branching time temporal logic such that every instance has a unique past but a non-deterministic future. Therefore, this logic is particularly suited for defining the semantics of non-deterministic programs. Given a Kripke structure K corresponding to the system model, a model checker examines K to check if it satisfies the property, such that a property is true for K if it is true at the initial states [65]. CTL formulas are made up of a *state operator* followed immediately by a *path operator*:

State operators, which work on paths of a given state. Given p to be a proposition, the state operators are defined as follows:

1. `A`, where `A`p is true for all paths of the current states.
2. `E`, where `E`p is true iff there exists at least one path starting from a given state where p holds.

Path operators, which work on states of a given path:

1. `F`, where `F`p is true in some state in the future, given p is true now.
2. `G`, where `G`p is true in every moment in the future.
3. `X`, where `X`p is true if p holds in the immediate successor of the given state.

Besides model checking, various other verification methods exists including theorem provers, term rewriting systems and proof checkers. However, these techniques require human intervention and therefore makes them time consuming. On the other hand, model checking is completely automatic. One

problem of this technique is the *state explosion problem* — the number of states in a model can grow exponentially with the number of concurrent components.

There have been various attempts to solve this problem. One idea is to represent a state space symbolically instead of explicitly using binary decision diagrams (BDDs) [16, 22, 65].

A **BDD** [4, 14] is a canonical tree which represents a function in *if-then-else normal form* (INF). An INF is a boolean expression built entirely from the if-then-else operator and the constants 0 and 1, such that all tests are performed only on variables and not on states [4].

Definition 2.2.3. *An* if-then-else *operator* $x \rightarrow y_0, y_1$ *is defined as*

$$(x \wedge y_0) \vee (\neg x \wedge y_1).$$

Using expressions in this form, a decision tree is built where each node has two outgoing edges: one edge (represented by a dashed line) corresponds to the cases where the variable evaluates to 0, whilst the other edge (represented by a solid line) corresponds to when the variable is 1. So, for example, $(x \Leftrightarrow y)$ can be written as follows:

$$x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$$

This is represented as the decision tree in figure 2.2.

A BDD is built when all equal sub-expressions are substituted by one expression. A BDD is an Ordered BDD (OBDD) if the variables obey a certain order, i.e. every path from the root to a leaf encounters the variables in a preset order.

A model checker based on OBDDs is McMillan's Symbolic Model Verifier (SMV) [64, 66, 67], which uses these graphs as the basis for a search algorithm to determine whether a system defined in the language satisfies the specifications.

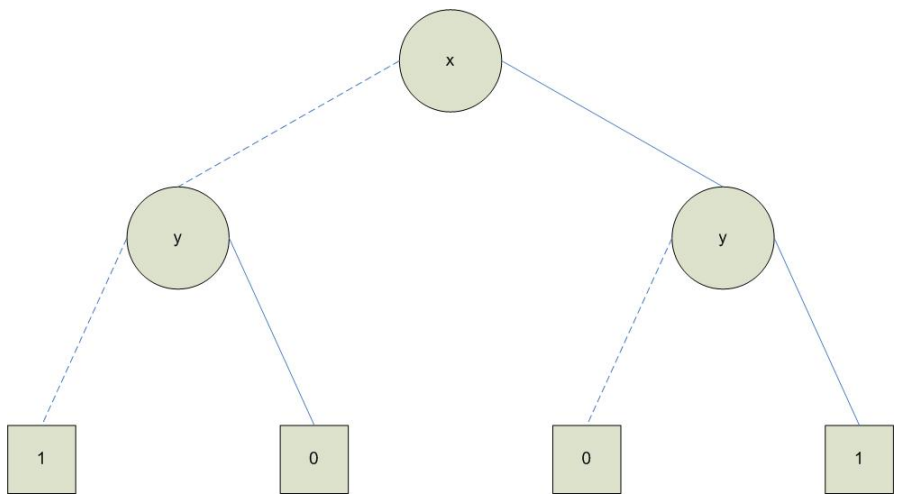

Figure 2.2: A DECISION TREE FOR $x \Leftrightarrow y$

Chapter 3

SharpHDL — A Hardware Description Language Embedded in C#

Embedded domain-specific languages have been shown to be useful in various domains. One particular domain to which this approach has been applied is hardware description languages. In this chapter we present such a language embedded in C#, enabling us to describe structurally-large, regular circuits in an intuitive way. These descriptions can then be automatically used in simulators and verification tools.

3.1 Introduction

Hardware Description Languages (HDLs) are programming languages used to describe a circuit structurally or behaviourally. Some languages, including the standard HDLs Verilog and VHDL allow both forms of circuit descriptions.

A structural HDL provides means for a designer to specify the components and connections needed to build a circuit. On the other hand, a behavioural HDL allows a higher-level description using conventional programming language constructs like *if-then-else* statements and loops. The difference between these two descriptions can be illustrated using a *multiplexer* example. A *multiplexer* (cf. Figure 3.1) is a digital circuit with multiple signal inputs, one of which is selected and outputted depending on an input condition signal. Therefore, a 2-bit multiplexer accepts input signals *input*0 and *input*1 and outputs the value of one of them depending on the value of a third input signal *select* — if *select* is true the circuit outputs the value of *input*1, otherwise it outputs the value of *input*0.

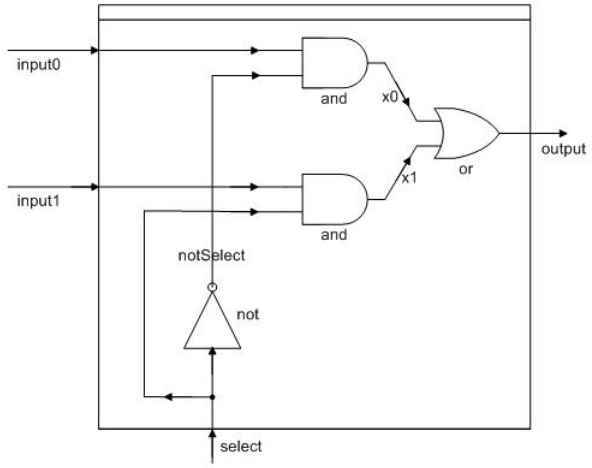

Figure 3.1: A MULTIPLEXER CIRCUIT.

The circuit can be implemented by describing the internal sub-circuits needed and their interconnections. This will require the initialization of modules or functions that describe the individual sub-circuits using the wires with which

they are connected. Therefore a structural definition of the 2-bit multiplexer will have the following format:

```
notSelect = not(select)
x0 = and(input0, notSelect)
x1 = and(input1, select))
output = or(x0, x1)
```

Alternatively, the circuit can be described behaviourally using an if-then-else statement, which definition will have the following format:

```
if (select == true)
    then input1
    else input0
```

SharpHDL [84, 85] is a structural HDL embedded in the object-oriented (OO) language C#, such that circuits are expressed as objects having a set of values and which perform a set of operations. Therefore, using SharpHDL, we can describe the structure of a circuit using C# classes and objects. Being an embedded language, it is a meta language which allows a hardware designer to generate regular circuits. SharpHDL also allows generation of circuit descriptions to other tools for verification, simulation and testing.

This chapter gives a brief introduction to SharpHDL and highlights its characteristics and features.

3.2 Object-Oriented Programming and C#

C# is based on the object-oriented paradigm (OOP), which paradigm has a number of characteristics. For a better understanding of SharpHDL, in this section we will briefly describe such characteristics and give an overview of the host language C#.

3.2.1 The object-oriented paradigm

OO languages [31, 42] started to emerge in the 1990s. Until that time, the most popular programming languages were based on the *procedural* concept. A procedure is a named block of code. In this style of programming, a developer writes one or more procedures and works with a set of independent

variables which can be manipulated by any piece of code. C and Pascal are two programming languages that are based on this paradigm.

Smalltalk and Simula were amongst the first programming languages that introduced the object-oriented paradigm. The inventors stated that humans do not express ideas as procedures but, instead, they express them in terms of objects. *Objects* are entities that have a set of states and a set of behaviours based on a *class*. A class dictates the states and behaviours a set of objects have. Therefore, a class can be considered as a template. On the other hand, an object is a concrete instance of a class. For example, a `Circuit` class has a string state representing `name`. One instance, or object, of `Circuit` has the `name` state set to `"or_gate"` whilst another one has the `name` state set to `"circuit2"`. A user works with objects that are based on classes which can be user-defined.

Object-Oriented Programming Characteristic

OOP provides a number of important characteristics [31], most of which were key elements for the successful and efficient upgrading and later extensions of SharpHDL. These are listed and briefly explained here.

Composition states that a complex object is made up of several smaller and simpler objects, thus employing the divide-and-conquer concept. It is important to understand all the small objects and their relationship to one another. This feature allows us to reuse existing objects to build different objects. There are two forms of composition — ***Association*** and ***Aggregation***. Association does not allow the internal objects to be externally visible; Aggregation allows them to be visible and usable by other external objects. These forms of composition are often used together.

Generalization allows us to identify common elements in objects, thus allowing an entity to work as generally as possible. This makes it possible to work with a different number of inputs or input of different kinds. Generalization comes into two forms — ***Hierarchy*** and ***Polymorphism***.

Hierarchy (or ***Inheritance***) allows objects to be organized into tree structures, such that the root holds the attributes and behaviour common to all descendants. In OO languages this entity is known as the *base class*. As the tree is traversed top-down the descendants are more specialized. This feature allows ***Extensibility*** — new specializations can be added at any level

and new attributes and behaviours can be easily added to the right subset of specialization. This is done without having to rewrite a new class from scratch.

Polymorphism enables a programmer to treat a collection of classes derived from the same base class in the same way, i.e. the derived classes are treated as the base class but the correct operation implemented by the derived class is invoked. The operation in the derived class should be an implementation of an operation recognized by the base class.

Abstract data types are another feature offered by OO languages. In such languages, types are associated to objects and therefore they are also with their attributes and behaviour. Hence, in defining a class, a user will automatically be defining a new type.

Separation is a technique which allows us to separate *what* an object does from *how* it is doing it. In more technical terms, OOP allows us to separate the *interface* from the *implementation*. The interface is what the user sees and has to understand to be able to use a particular object. On the other hand, the implementation is the hidden part of the object which usually only interests the implementor. An implementation satisfies an interface if the behaviour specified by the interface is reflected in the implementation.

3.2.2 C# — the host language

C# [31], pronounced C Sharp, is an OO language designed by Microsoft. It combines the power and control of C and C++, which are amongst the most widely used languages for software development. It also has great similarities to Java [42], not only in its syntax but also in its purpose for web-application development and automatic memory management. It was designed to work with Microsoft's .NET platform, a solution which enables developers to build various applications, using the same tools and skills.

C# is easy to learn and use. A comprehensive description of the language and its syntax can be found in [31]. Further description of the language's syntax is out of the scope of this documentation.

3.3 Introduction to SharpHDL

The SharpHDL language consists of a set of libraries in C# which provide the necessary tools to describe and manipulate circuits. The main functionalities are given by three libraries:

- A core library which controls the internal structure of a circuit;
- Another library consisting of a set of implementations of higher-order circuits;
- A third library which provides ready-made circuits.

SharpHDL descriptions can be output to other external tools:

- verification tools, which allow us to test circuit specifications;
- other HDLs, which allows us to use the various tools based on them.

To date, two versions of SharpHDL have been developed. The first version was built in 2004 and presented in [84, 85]. We shall now give an outline of this version and highlight the various areas of improvement that were tackled in the second implementation. The latter is the version used in the rest of this dissertation.

3.3.1 The first version

[84, 85] introduce, discuss and test the first implemented version of SharpHDL. The development was largely influenced by two embedded HDLs: Lava [9, 20] and JHDL [45, 46].

Lava is embedded in the functional programming language Haskell. It provides possibilities of simulation and formal verification of circuits, and also includes a set of higher-order circuits referred to as *connection patterns*. Lava's most attractive features are possibly its simplicity and elegance, much aided by its host language.

JHDL is embedded in the OO language Java. Therefore it treats circuit descriptions as objects and classes. It provides simulation and analysis tools and

also boasts of an attractive CAD suite. Unfortunately, it does not support formal verification and also lacks efficient support for connection patterns.

It was therefore our primary objective to design an OO HDL that provides means of designing circuits using short and sweet descriptions and which also provides means for verifying circuits. C# was selected because it is object-oriented, easy-to-use and documentations exist to help the user.

Using SharpHDL, we successfully embedded a behavioural HDL thus creating a hierarchy of languages (cf. Figure 3.2). The new language, which we called the *Regular Expressions Language*, provides constructs with which circuits that describe regular expressions can be designed. Being embedded, circuit descriptions in the new language can use all the tools provided by SharpHDL and C# and therefore, they can also be analyzed and verified.

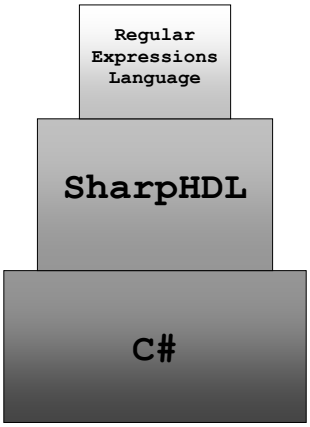

Figure 3.2: A hierarchy of languages is created by embedding SharpHDL in C# and the *Regular Expressions* language in SharpHDL.

However, as more complex examples were being implemented in SharpHDL, some problems started to emerge. A major problem arose because SharpHDL maintained flat descriptions of circuits — internally, a circuit was decomposed and stored in terms of the most primitive gates or circuits (e.g. *and*, *or*, *not* etc.). So, for example, if a user defined a *half-adder* circuit (cf. Figure 3.3(a)) as having an *and*-gate and a *xor*-gate and then defined a *full-adder* circuit (cf. Figure 3.3(b)) in terms of two *half-adder* circuits and a *xor*-gate, SharpHDL decomposed each *half-adder* into the *and*-gate and the *xor*-gate and stored the *full-adder* definition as three *xor*-gates and two *and*-gates. This approach

lost the compositional and reusability characteristics of SharpHDL, which characteristics inherited from embedding the language in C#. This greatly affected the descriptions generated for external tools since such descriptions consisted of one flattened long piece of code that was difficult to understand and at times impossible to load in the respective applications.

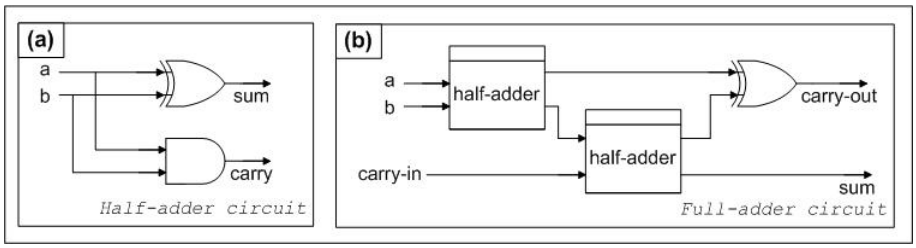

Figure 3.3: (a)Half-adder circuit (b)Full-adder circuit.

Another problem was the external-code-generation algorithm. For each primitive circuit used in a description, the algorithm added the appropriate code to the output being generated. However, the algorithm was written to process only a defined set of primitive circuits such that it had to be manually modified when a new primitive circuit was implemented. In other words, the algorithm was not extendible to consider later defined primitive gates.

One other problem was the declaration of a circuit's interface. The interface, or the set of input and output ports that allows a circuit to communicate, was defined as a *static* array of `CellInterface` objects and therefore several instances of the same circuit object shared the same interface objects. Since this approach created confusion in circuit-traversing algorithms, declaring interfaces was avoided. This made redundant the concept of the `CellInterface`. Besides, interface objects had to be assigned and referred to by a unique name which was not always possible to know, especially when the implementation was hidden.

There are also a number of other features that could have been implemented in this primitive version of SharpHDL but which were not possible due to time constraints. These include multiple-bit wires and their respective ports, and implementation of other circuits. These weak points and other enhancements were all taken into consideration during the design and implementation stages of the new SharpHDL.

3.3.2 The new SharpHDL

To preserve the compositional and reusable nature of the language, the internal structure of the second implementation of SharpHDL introduces the idea of having a hierarchy of circuits such that a circuit is the child of a parent if it is one of its direct sub-components. Referring to the *full-adder* circuit described in the previous section, the *full-adder* is the *parent* of two *half-adder* circuits and a *xor*-gate, where each *half-adder* circuit has an *and*-gate and a *xor*-gate as its children. Therefore, the new SharpHDL does not decompose the definitions to the primitive boolean circuits but instead keeps the relationships between the circuits and treats each circuit as an individual circuit in its own right.

This approach solves the problem of the big-file-generation output for external tools. This is possible because, given that the individual information of each circuit is not lost, the generated output can be divided into modules such that a module is created for each used component type. Each module defines a component's structure in terms of its immediate descendants and it can be invoked by other modules whenever needed. This made the generated files modular and therefore more user-friendly and much shorter.

To allow greater extensibility, new abstract classes are introduced to the general class structure to provide a wider and better spectrum of different kinds of circuits, wires and ports. Such introductions include classes for multiple-bit wires and their ports which allow the wires to be connected to the circuits, and also new types of circuits. Other changes were made such that the interface declaration is no longer static and instead an interface is created for every circuit. This declaration is optional and when not defined the ports are created automatically according to the type of wires connected to the circuit. The `CellInterface` concept was completely eliminated from the system. This approach gives greater abstraction power to circuits, especially higher-order circuits since their interface type is created according to the input circuit.

The next sections give an overview of the main SharpHDL libraries and how they can be used.

3.4 SharpHDL Features

SharpHDL is an HDL embedded in C# and therefore one can describe a circuit by writing a C# program. It consists of three main libraries (cf. Figure 3.4)

1. `SharpHDL`, which is the core library. It manages the internal structure of circuits and provides classes of primitive logic gates;
2. `GenericCircuits`, which contains a number of classes defining higher-order circuit descriptions which, given another circuit, build a regular-structured circuit;
3. `LogicGates` where classes of compound logic gates are implemented.

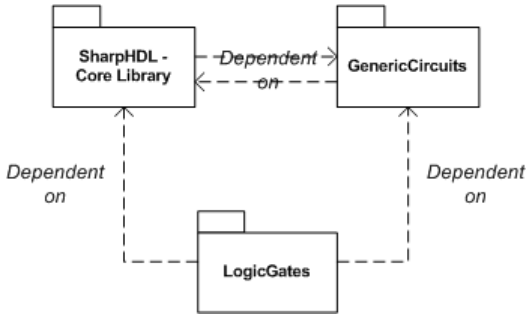

Figure 3.4: The relationships between the three main libraries of SharpHDL.

This section gives an overview of these libraries and the various functionalities they provide.

3.4.1 Managing SharpHDL circuits

The main function of the `SharpHDL` library is to provide and manage the underlying structure of SharpHDL circuits.

A circuit can be defined as a structure having a set of input and output ports to which wires are connected. For example, an *and*-gate has two input ports and one output port (cf. Figure 3.5).

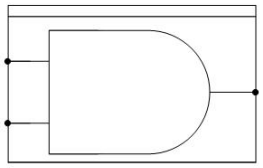

Figure 3.5: An AND-gate has two input ports and one output port.

Furthermore, a circuit's operation can be defined by internal circuits interconnected with wires, as we have seen in the multiplexer and adder circuits on pages 14 and 20 respectively. Therefore, a circuit is based upon three elements — ports (which allow a circuit to communicate with its environment), wires (which connect a circuit to other circuits or the external environment) and internal circuits (which define the circuit's operation). Based on this definition, SharpHDL objects inherit from either of the three main classes:

1. `Wire` is the class that represents a wire, or signal. It provides methods to access information about the wire object and its connections but does not provide for the connection itself. For every class that inherits `Wire`, the type of port to which it can be connected to has to be specified. SharpHDL provides three types of `Wire`s:

   (a) `LogicWire`, which represents a one-bit signal;

   (b) `BusWire`, which represents a multiple-bit signal. Basically, it defines a list of the same `Wire` type. An example of this is a wire carrying an integer. In a digital system, numbers are stored as lists of bits. In SharpHDL this is implemented as `Integer`, which is a `BusWire`-inheriting class which accepts `LogicWire` objects as its elements (cf. Figure 3.6).

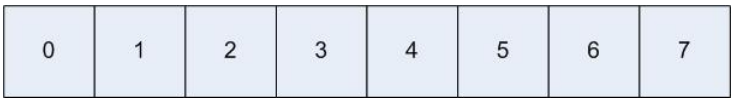

Figure 3.6: An Integer is a list of LogicWire objects.

   (c) `CompoundWire`, which represents a signal that is made up of different types of wires. This can be illustrated using the implementation of a wire carrying a floating-point number. A floating-point number (FP) is defined as follows:

$$FP = +/- \ Mantissa \times 2^{Exponent}$$

The *sign* $+/-$ is represented by a one-bit signal, whilst the *mantissa* and *exponent* are integers signals (cf. Figure 3.7) [10, 71, 80]. SharpHDL implements this type of number using class `Float`, which inherits from `CompoundWire`.

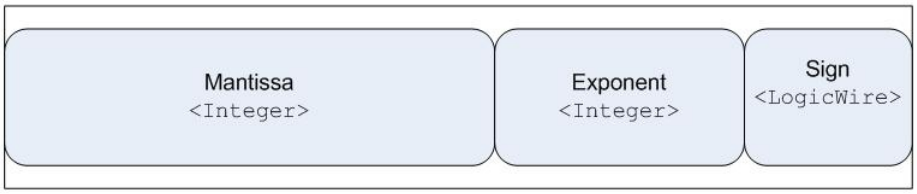

Figure 3.7: A FLOAT IS MADE UP OF THREE WIRE COMPONENTS — AN INTEGER MANTISSA, AN INTEGER EXPONENT AND A LOGICWIRE SIGN.

2. `Port` is the general representation of a port — the object which allows the connection of a wire to a circuit. Ports can be of two types:

   - *Input ports* allow wires carrying input to be connected to a circuit.
   - *Output ports* allow wires carrying output to be connected to a circuit.

   Class `Port` provides the means to specify the type of the port. It also provides methods that return information about the wire and circuit connected to it. Classes that extend the `Port` class have to define the type of `Wire` that can connect to them.

3. `Circuit` is the basic representation of a circuit. It provides methods to add other `Circuit` objects that together form part of the internal structure of the circuit being defined. It also provides the methods to generate descriptions to external tools. One of the latest implementations to this class is the definition of the *parent* of the circuit, i.e. the user is allowed to specify the circuit to which the circuit being defined is a direct sub-circuit. `Circuit` inherits from class `Cell` which provides methods to define the interface of the circuit and to manage connections.

All these classes inherit from the top-most class `Nameable` which gives a unique name to every element created in the circuit definitions.

Another class worth mentioning is the `BooleanLogic` class. This class inherits from the `Circuit` class and represents the primitive circuits. Therefore this class is extended by:

- classes representing conventional logic gates (e.g. `And`, `Or`, `Not`, etc.),
- two delay gates which allow the definition of sequential circuits — `DelayTrue` which initializes to 1 and `DelayFalse` which is initialized to 0;
- the *constant* signals — `ConstantTrue` which produces a signal that has a value of 1 throughout the circuit; and `ConstantFalse` which produces a signal having value 0.

3.4.2 Generic circuits

Although two circuits may use different gates, their structure may be identical. Other circuits have a regular structure which can be easily extended to larger circuits. In SharpHDL such circuits are captured in *generic circuits*. Generic circuits are higher-order circuits, such that when given a circuit, they build another circuit. These implementations, together with their respective *interfaces*[1], are stored in the `GenericCircuits` library. The interfaces define the methods which should be implemented by circuits used as input to the generic circuits.

All generic circuits inherit from class `GenericCircuit`. This class imposes a validation method on the implementation of each generic circuit. This method, `VerifyCircuit()`, checks the interface of the input component and decides whether it can be used by the generic circuit. The `GenericCircuit` class inherits from the abstract class `Circuit`.

SharpHDL provides various generic circuits some of which are discussed here. Chapter 4 discusses others that were used for building FFT circuits.

[1] An **interface** is not a class but a set of requirements for classes that have to conform to the interface. It is a way of describing what classes should do, without specifying how they should do it [42].

`Map` Generic Circuit

Consider the case where we want to apply a *not* operation to every signal in a list (cf. Figure 3.8). We can do this by iterating through the list and applying the operation to each element.

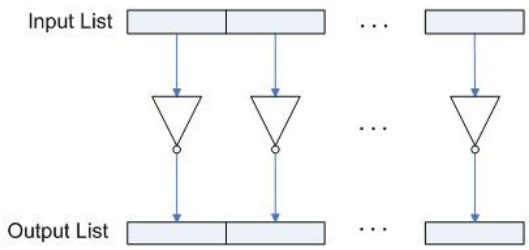

Figure 3.8: Applying a `Not` operation to every wire in an input list.

Alternatively, we can use the `Map` generic circuit. Given a list of wires, it constructs a regular-structured circuit that applies an input component to every wire in the list (cf. Figure 3.9).

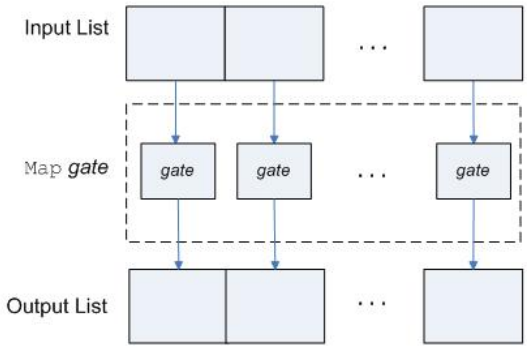

Figure 3.9: `Map` applying logic component `gate` to every wire in a given input list.

SharpHDL implements three variations of `Map`: `MapOne` accepts a one-input-port circuit, `MapTwo` accepts a two-input-port circuit, and `MapThree` accepts

a three-input-port circuit. Consequently, these variations accept one list of wires, two lists of wires and three lists of wires respectively.

`Sequence` Generic Circuit

Consider the situation where we need to apply a *not* operation repeatedly (cf. Figure 3.10). We can implement this using a loop that iterates for a user-input n times, such that an iteration uses the output of the previous iteration as input.

Figure 3.10: Applying a `Not` operation repeatedly for n times.

Otherwise, we can use a `Sequence` generic circuit. As its name suggests, a `Sequence` applies a circuit operation sequentially over a set of input wires. Therefore, for $i = 1 \ldots n$, n instances of the input component C are created and placed near each other, such that the output from the ith instance of C, C_i, is fed to the neighboring instance C_{i+1}. C_1 is fed with the list of wires input to the `Sequence` circuit whilst C_n outputs its final result (cf. Figure 3.11).

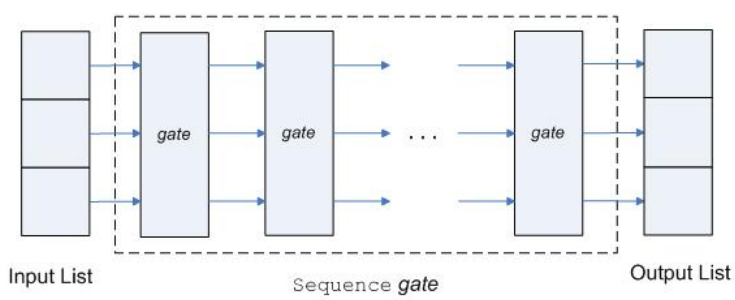

Figure 3.11: Logic component `gate` repeated for n times using generic circuit `Sequence`.

This generic circuit accepts a circuit having an equal number of input and output ports, and also accepts the number of times the circuit has to be applied.

`Row` Generic Circuit

The `Row` generic circuit resembles a combination of the `Map` and `Sequence` structures. Besides a component, it accepts a list of wires and a one-bit wire. It creates instances of the input component for the number of wires in the input list.

`Row` is built such that the nth instance of the input component takes the nth wire from the list and the one-bit wire output from the $(n-1)^{th}$ component. Each component produces two wires: one is saved in the output list of wires and the other is fed to the neighboring circuit. Thus, the final output of this generic circuit is a one-bit wire and a list of wires having the same size of the input list (cf. Figure 3.12).

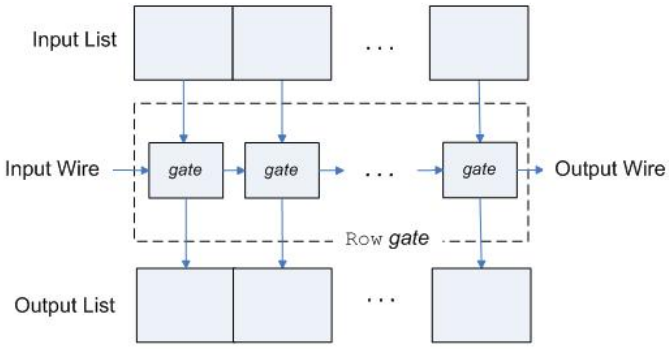

Figure 3.12: A `Row` generic circuit using component `gate`.

An application of this type of generic circuit is an *incrementor* which increments a number represented by a list of `LogicWires`. Using a *half-adder* circuit as the input component to the generic circuit, it creates copies of the component for each wire in the input list. The input one-bit signal is a constant wire with value 1. The output list of the generic circuit is the sum of the *incrementor* whilst the one-bit output is its carry-out (cf. Figure 3.13).

SharpHDL implements three variations of `Row`: `RowOne` accepts a two-input-port circuit, `RowTwo` accepts a three-input-port circuit, and `RowThree` accepts a four-input-port circuit. Consequently, they accept one list of wire, two lists of wires and three lists of wires as input respectively, together with a one-bit wire.

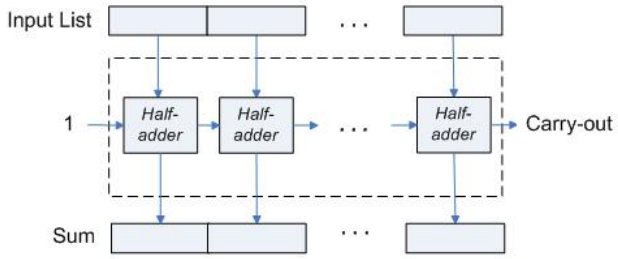

Figure 3.13: An incrementor can be implemented using a `Row` generic circuit.

`Tree` Generic Circuit

There are circuits that can be built recursively. This means that to build a given circuit, they call themselves one or more times to deal with closely related sub-circuits. These circuits typically follow a *divide and conquer* approach — they break the circuit into several sub-circuits that are similar to the original circuit but smaller in size, build the sub-circuits recursively and then combine these circuits to create the final circuit [26].

An example of such a circuit is the `Tree` generic circuit. The input component should accept two inputs and produce one output. The generic circuit creates multiple instances of the component and uses these as nodes of a binary tree.

Organizing a set of circuits of the same type in a tree structure reduces the length of the circuit when compared to a linearly-formed circuit — applying an operator on n wires using a `Tree` structure will have a path of approximately $\log_2 n$. On the other hand, if a circuit is generated linearly, the length of the longest path is $n - 1$ (cf. Table 3.1).

3.4.3 Interpretations for verification

One of the important reasons for developing SharpHDL was to build a HDL that catered for verification. Verification is advantageous because it can save time from tedious debugging, as well as avoid expensive hardware being implemented incorrectly.

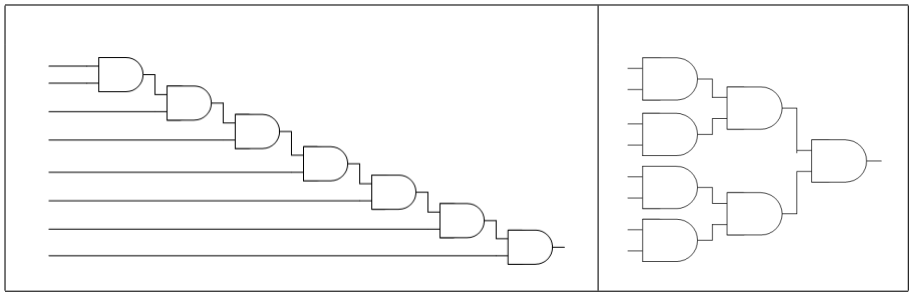

Table 3.1: Applying an And-gate to 8 wires — if applied linearly (left) the longest path is 7 gates. If the circuit is built in a tree structure (right) the longest path is 3 gates.

Safety Properties and Observers

SharpHDL circuits can be translated to other descriptions such that they can be verified against specifications using formal verification tools. In SharpHDL, specifications are defined as *safety properties*. Such properties state that a condition is always true or never false. As an example consider a train control system. A safety property we might want to verify on this system is that a train always stops at a stop signal.

To prove that a circuit obeys a specified property we write an *observer* circuit which takes as input the inputs and outputs of the circuit under test, and outputs a signal *ok* stating whether it abides with the property or not. In the train control system, the observer will take the inputs and outputs of the control system and checks whether for a given input the train stops at the stop signal.

More generally, consider circuit C, having input signals I_C and output signals O_C such that $S = I_C \cup O_C$. An observer Ω_P of safety property P on signals S is a circuit which takes S of the circuit C and outputs an alarm signal α, where $\alpha \notin S$. (cf. Figure 3.14). On using this approach we say that the safety property is changed into an *invariant* [34, 36, 39].

Given these circuits, we can use formal verification techniques to verify that, for any given input, the circuit always abides with the property defined by the observer circuit. In other words, we prove that the combination of the circuit and the observer outputs a *true* signal on the alarm signal α for any given input. We will further discuss this in section 5.2.

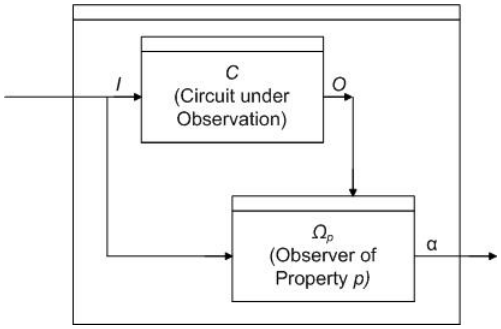

Figure 3.14: Using an observer to verify a property over a given circuit.

Generating Descriptions for Verification

Given a list of wires in a circuit, a description can be generated that allows a designer to test whether a design conforms to a set of specifications. The list of wires is assumed to be the output wires of the observers of each specification which a circuit-under-test is being verified against and thus assuming that the whole circuit consists of the circuit-under-test and the observers defining the properties. Therefore, given to the appropriate verification tool, it verifies that the composition of the observer outputs is true for any input.

A SharpHDL method which provides such a functionality accepts a list of wires and outputs the verification result[2] and the generated code. The verification result is of type `VerificationReport` which is an *enumerated type*[3]. It can have two values:

1. `FALSE_VERIFICATION` when verification fails, and
2. `COMPLETE_VERIFICATION` when the circuit verifies correctly.

SharpHDL also defines the interface `IVerification`, which specifies rules which have to be implemented by classes implementing algorithms that generate output for specific verification tools. Such an interface makes Sharp-

[2]It should be pointed out that no direct connection exists between SharpHDL and any verification tool as yet. The type representing the verification result was created as a framework for when the connection method is actually implemented.

[3]An **enumerated type** defines a group of constants under a common name. Such types are used when a variable should be combined to a specific set of values [31].

HDL extendible to generate output to different verification tools. These *rules* are:

- `Circuit.VerificationReport ProduceCode(LogicWire [] inputWires, out string code)`
  — This method should implement the algorithm for producing the circuit description for the verification tool being represented by the class. It accepts a set of wires, which are considered to be the signals to verify. The method outputs the result of the verification tool. The description is outputted via the string parameter `code`.

- `Circuit CodedCircuit{get;}`
  — This is a property accessor which returns the circuit being represented by the code object.

SMV and the SMV Output Code Format

In this dissertation, we target the generation of SMV descriptions for verification. SMV [65, 67] is a standard model checker based on OBDD symbolic model checking. Choosing this particular tool was completely random and, therefore, the decision was not based on any particular reason. This section briefly explains the syntax and format of the generated SMV descriptions.

SMV supports modularity and therefore a generated description is divided into *modules*. A module describes the circuit's structure in terms of the boolean expressions and other modules which it uses directly. It also defines a set of formal parameters such that, when creating an instance of the module, correct signals are fed in for each specified parameter. The module also declares whether the parameters are input or output types by using the keywords `INPUT` and `OUTPUT` respectively. Other used variables are declared under the keyword `VAR`. Therefore, a module has the following format:

```
MODULE <module_name>(wire1, wire2, .... wireN)
{
    INPUT <formal parameters which are input signals> : boolean;
    OUTPUT <formal parameters which are output signals> : boolean;

    VAR
        <list of non-formal-parameter signals> : boolean

    --Circuit Logic--
    <invocations of expressions and module instances>
}
```

The whole SMV circuit description is controlled by one `main` module, which is the one evaluated by the SMV interpreter. Besides defining the structure of the top-most circuit, this module also defines the properties which should be verified by the model checker. These are defined using *assertions*. Assertions, declared using the keyword `assert`, are written in temporal logic. Since SharpHDL specifications are defined as safety properties, only the *global* temporal operator `G` is used in the generated SMV description. Given a property p, the formula `G`p states that p is true in every moment in the future.

For example, given a circuit `circ` and specifications `p1` and `p2` described using observers `observer1` and `observer2` respectively, verification consists of checking that the composition of the outputs from the two observers, `ok1` and `ok2` respectively are true for all inputs. The following is the generated SMV description:

```
MODULE main()
{
   VAR

       -- Signals in the circuit --
       input1, input2, ..., output1, ..., outputN, alarm : boolean;

       -- Circuit under test
       circ : circuit(input1, input2, ..., output1, ..., outputN);

       -- Observer 1 taking inputs and outputs of circ and outputs ok1
       p1: observer1(input1, input2, ..., output1, ..., outputN, ok1);

       -- Observer 2 taking inputs and outputs of circ and outputs ok2
       p2: observer2(input1, input2, ..., output1, ..., outputN, ok2);


       --PROPERTIES to prove--
       assertMain := ok1 & ok2;
       mainProperty : assert G (assertMain);

       --Extra Variable--
       assertMain: boolean;
}

MODULE circuit(wire1, wire2, ..., output1, ..., outputN)
{
      -- ... module body ...
}

MODULE observer1(wire1, wire2, ..., output1, ..., outputN, ok)
{
   INPUT wire1, wire2, ..., output1, ..., outputN : boolean;
   OUTPUT ok : boolean;

      -- ... module body ...
}

MODULE observer2(wire1, wire2, ..., output1, ..., outputN, ok)
{
   INPUT wire1, wire2, ..., output1, ..., outputN : boolean;
   OUTPUT ok : boolean;

      -- ... module body ...
}

...
<Other modules needed>
```

Generating SMV Descriptions

SMV descriptions are generated using method `ToSMV()`. Every type of circuit that is used in the SharpHDL definition is represented by an SMV module

where the circuit calling the `ToSMV()` method is defined as the `main` module.

Internally, the `ToSMV()` method creates an `SMV` object of the circuit. `SMV` is a class representing SMV code. SharpHDL circuits implement interface `ISMVSerializable` so that they can be processed by the `SMV` object. The interface defines the following methods:

- `SMV.SMVComponent SMVType{get;set;}`
  — This defines how the circuit wants the `SMV` object to consider it. If the circuit is a boolean gate or the user knows the SMV expression representing the circuit, then the `SMVType` property should return `BOOLEAN` otherwise it should return `DEFAULT`, both of which are members of the enumerated type `SMVComponent` defined in the `SMV` class.

- `bool DefineModule{get;set;}`
  — This property allows the user to specify whether he wants the particular circuit to be represented as a separate module or not. In the case where he prefers not to have an individually defined module for a circuit, the module representing the parent of the particular circuit will define the structure of the circuit as if it was its own.

- `string UserDefinedLogic{get;}`
  — This property returns a user-defined SMV expression representing the circuit. It is invoked if the circuit returns `BOOLEAN` as its `SMVType` property. The user must be careful to define correct SMV code.

- `string SMVModuleDescription{get;}`
  — The string returned by this property is used as a comment placed just before the start of the module representing the circuit.

3.4.4 More about interpretations

Descriptions of SharpHDL circuits can also be generated to other HDLs. We target the standard HDL Verilog.

Method `ToVerilog()` generates the Verilog description of a circuit. One should point out that, due to time constraints, the algorithm to produce Verilog descriptions has not been updated to produce a modular format, as done to the algorithm producing SMV. Nevertheless, the idea used in the SMV-generation algorithm could be used to produce a neater and modular

Verilog representation. Since the research concentrated mostly on verification techniques, updating the Verilog-generation algorithm was deemed as less important.

3.4.5 Other SharpHDL libraries

`LogicGates` is another important SharpHDL library. It uses the core library to construct compound gates. Such gates are those which need to use a number of primitive gates and other compound gates. Some circuits provided include the *nand, nor, xor,* and *equivalence* gates amongst others. Other circuits can easily be implemented in this library.

The SharpHDL language was further extended with more libraries which capture classes of circuits, wires and ports doing related jobs. Such libraries are based on the three main SharpHDL libraries — `SharpHDL`, `GenericCircuits` and `LogicGates`.

One extra library created is the `Arithmetic` library where the implementations of a group of circuits performing arithmetic operations are stored. Such circuits include adders and multipliers which work on integers and floating-point numbers. It also includes classes which inherit from class `Wire`, representing different types of numbers and `Port` classes which accept these wires.

3.5 Using SharpHDL

This section gives a brief overview about how to build circuit descriptions using SharpHDL. It takes the form of a progressive tutorial from building simple circuits to using advanced SharpHDL tools. It is assumed that the reader has a basic knowledge about C# programming.

3.5.1 Defining a simple circuit

We shall introduce SharpHDL syntax by defining a half-adder circuit. A half-adder adds two one-bit numbers and outputs their sum and carry. Therefore, a half-adder circuit takes two input wires — `InputA` and `InputB` — and outputs two wires: `Sum` and `Carry`. The `Sum` result is produced by applying a `Xor` operation over the two inputs; the `Carry` result is produced by applying an `And` operation (cf. Figure 3.15).

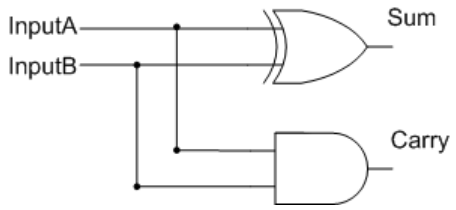

Figure 3.15: A HALF-ADDER CIRCUIT.

To be able to use the main C# functionalities a call to the `System` namespace must be included. The `System` namespace contains classes that implement basic programming constructs. The `SharpHDL` library is also invoked to be able to make use of its tools:

```
using System;
using SharpHDL;
```

Since we are going to code a circuit, the `HalfAdder` class should inherit from the class `Logic`, an abstract class representing the general logic circuit. This will also enable us to have access to protected methods which will otherwise be hidden to the circuit:

```
public class HalfAdder : Logic {...}
```

When a C# object is created it is initialized by calling the class *constructor*. When initialized, a SharpHDL circuit must also specify its parent circuit. The parent of a circuit is the circuit to which the new circuit is a direct sub-circuit. The `HalfAdder` constructor is specified as follows:

```
public HalfAdder(Circuit parent):base(parent)
{
    ...
}
```

In the constructor we specify functions that an object has to carry out on creation. Optionally, we can specify the set of input and output ports using methods `AddInputPort()` and `AddOutputPort()` respectively, thus defining the interface of the circuit (cf. Table 3.2 Figure (a)):

```
//Input Ports
Port[] input = {new LogicPort(), new LogicPort()};
this.AddInputPort(input);

//Output Ports
Port[] output = {new LogicPort(), new LogicPort()};
this.AddOutputPort(output);
```

`LogicPort` is a class representing a port that connects a 1-bit wire to a circuit.

The structure of the circuit is defined in the method `gate_o()`[4]. It accepts four wires, two of which are the inputs and the other two are the outputs. The keyword `ref` is used for the output parameters to indicate that they are *reference parameters*. In other words, the values are supplied by reference and can be read and modified.

```
public void gate_o(LogicWire InputA, LogicWire InputB,
                   ref LogicWire Sum, ref LogicWire Carry)
{
    ...
}
```

The first compulsory step in this method is to connect the wires to the ports by using the method `Connect()` (cf. Table 3.2 Figure (b)). It is important to supply the wires in the same order as the ports were declared; though in this example this doesn't make a difference since the wires are of the same type. When the ports are not declared, they are created automatically according to the type of wires passed as parameters to the `Connect()` method.

```
Wire[] input = {InputA, InputB};
Wire[] output = {Sum, Carry};
this.Connect(input, output);
```

As mentioned previously, the half-adder requires two circuits, an `And`-gate and a `Xor`-gate. An instance of each is created by calling their respective constructors and, using the keyword `this`, we specify that the `HalfAdder` is their parent circuit. The `gate_o()` method of each circuit is invoked, passing it the appropriate parameters (cf. Table 3.2 Figure (c)).

[4]In SharpHDL, the name `gate_o()` is a standard method name for the method that specifies the structure and behaviour of the given circuit, given the necessary input and output wires. Likewise is the method `gate()` which, although having the same function, creates new output wires.

```
new And(this).gate_o(InputA, InputB, ref Carry);
new Xor(this).gate_o(InputA, InputB, ref Sum);
```

The first line invokes the method `gate_o()` of the `And`-gate object. It accepts three wires: the first two being the input wires and the third being the output wire. Therefore the two input wires to the half adder and the wire `Carry` are passed to this method. The same is done for the `Xor` instance, but this time the two input wires and the `Sum` wire are passed to its `gate_o()` method.

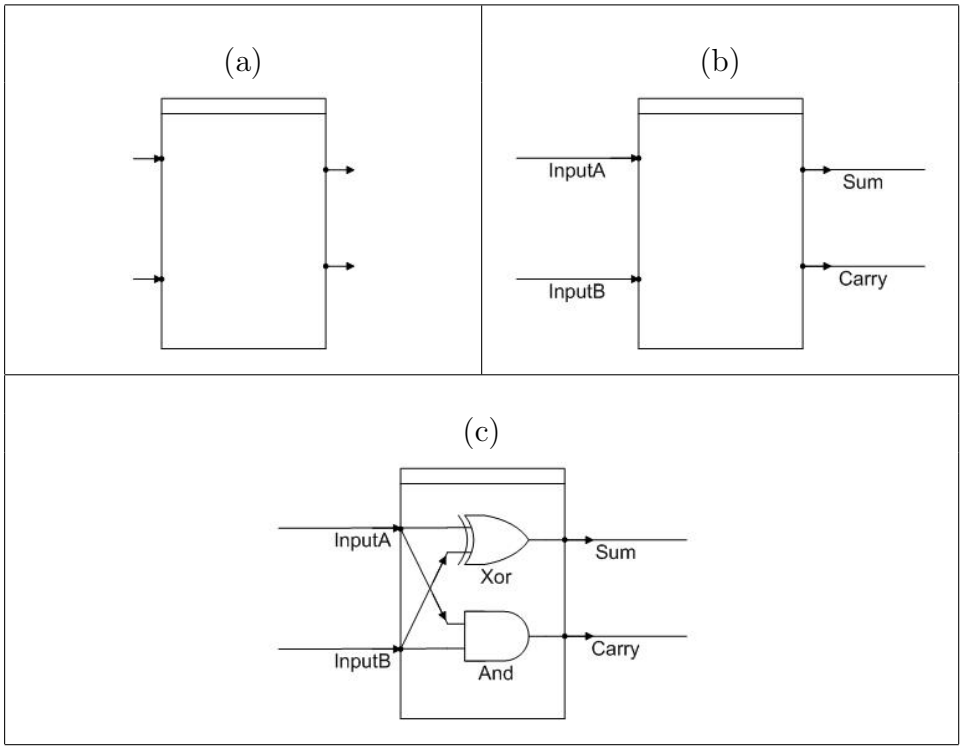

Table 3.2: Steps to create a SharpHDL circuit: (a) Define the interface to the circuit (optional step); (b) Connect wires to the circuit; (c) Define the internal structure of the circuit.

We also define a `gate()` method which calls the method `gate_o()` such that it carries out the same operations as the latter method. The difference lies in that the new output wires are created in the method and not passed as input by the user.

```
        public LogicWire gate(LogicWire InputA, LogicWire InputB,
                              out LogicWire Sum)
        {
            Sum = new LogicWire();
            LogicWire Carry = new LogicWire();
            gate_o(in0, in1, ref Sum, ref Carry);
            return Carry;
        }
```

The keyword `out` indicates that the value of the parameter `Sum` is set in the method and returned to the calling method. Any value assigned to it before the `gate()` method is called is lost. The `Carry` wire is created and returned by the method. The complete code for the class representing a half-adder circuit is the following:

```
using System;
using SharpHDL;

public class HalfAdder: Logic
{
    public HalfAdder(Circuit parent):base(parent)
    {
        Port[] input = {new LogicPort(), new LogicPort()};
        this.AddInputPort(input);
        Port[] output = {new LogicPort(), new LogicPort()};
        this.AddOutputPort(output);
    }
    public LogicWire gate(LogicWire InputA, LogicWire InputB,
                          out LogicWire Sum)
    {
        //Assign a new LogicWire to Sum
        Sum = new LogicWire();

        //Create a new Carry Wire
        LogicWire Carry = new LogicWire();

        //Call gate_o
        gate_o(InputA, InputB, ref Sum, ref Carry);
        return Carry;
    }

    public void gate_o(LogicWire InputA, LogicWire InputB,
                       ref LogicWire Sum, ref LogicWire Carry)
    {
        Wire[] input = {InputA, InputB};
        Wire[] output = {Sum, Carry};
        this.Connect(input, output);

        //And gate
        new And(this).gate_o(InputA, InputB, ref Carry);

        //Xor gate
        new Xor(this).gate_o(InputA, InputB, ref Sum);
    }
}
```

3.5.2 Using circuits to build other circuits

After defining a circuit we can use it either as a stand-alone circuit or as part of a more complex circuit.

The half-adder circuit can be used to add two one-bit numbers. We create an instance of the `HalfAdder` class by invoking its constructor. Then, taking `in0` and `in1` to be the wires representing the input numbers, and `sum1` and `carry1` to represent the sum and carry outputs respectively, we call method `gate()` or `gate_o()` to build the structure of the circuit. The names of the variables are left to the user.

```
//Create an instance of HalfAdder
HalfAdder hA = new HalfAdder(this);

//Build HalfAdder circuit
hA.gate_o(in0, in1, ref sum1, ref carry1);
```

A *full-adder* circuit uses half-adder circuits to add two one-bit numbers, `a` and `b`, and a carry-in bit, `carryIn`, and outputs a sum, `sum`, and carry-out bit, `carryOut` (cf. Figure 3.16).

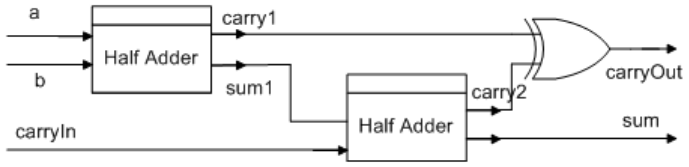

Figure 3.16: A full-adder circuit.

Using the diagram as a guide, we can see that the first half-adder takes wires `a` and `b` and outputs the sum, `sum1`, and the carry, `carry1`:

```
//Create new LogicWires for sum1 and carry1
LogicWire sum1 = new LogicWire();
LogicWire carry1 = new LogicWire();

//Create and build a half-adder circuit
new HalfAdder(this).gate_o(a, b, ref sum1, ref carry1);
```

The second half-adder takes wire `sum1` and the carry-in wire, `carryIn`, and outputs the full-adder circuit's sum wire, `sum`, and a carry-out wire, `carry2`.

```
//Create new LogicWire for carry2
LogicWire carry2 = new LogicWire();

new HalfAdder(this).gate_o(sum1, carryIn, ref sum, ref carry2);
```

The final carry-out answer `carryOut` is determined by using a `Xor`-gate and feeding it the two carry-outs produced by the half-adders `carry1` and `carry2`.

```
new Xor(this).gate_o(carry1, carry2, ref carryOut);
```

Therefore, the `gate_o()` method of a `FullAdder` class representing the full-adder circuit is the following:

```
public void gate_o(LogicWire a, LogicWire b, LogicWire carryIn,
                   ref LogicWire sum, ref LogicWire carryOut)
{
    //Connect wires to ports
    Wire[] input = {in0, in1, carryIn};
    Wire[] output = {carryOut, sum};
    this.Connect(input, output);

    //Create intermediate wires
    LogicWire sum1 = new LogicWire();
    LogicWire carry1 = new LogicWire();
    LogicWire carry2 = new LogicWire();

    //First half-adder instance
    new HalfAdder(this).gate_o(in0, in1, ref sum1, ref carry1);

    //Second half-adder instance
    new HalfAdder(this).gate_o(sum1, carryIn, ref sum, ref carry2);

    //Xor-gate
    new Xor(this).gate_o(carry1, carry2, ref carryOut);
}
```

3.5.3 C# constructs for defining SharpHDL circuits

Being embedded in C#, SharpHDL circuits can be described using constructs provided by the host language. These include iteration constructs like `for-do` and `while-do` loops and the recursion mechanism. We will describe how more complex circuits than the ones already described can be built using these constructs.

Defining a Ripple-Carry Adder by Iteration

A *ripple-carry adder* [40] performs addition on n number of wires. It uses full-adder circuits, defined in section 3.5.2, and tiles them as illustrated in figure 3.17. Therefore, it accepts two lists of wires and a one-bit carry-in wire and outputs a list of sum wires and a single carry-out wire.

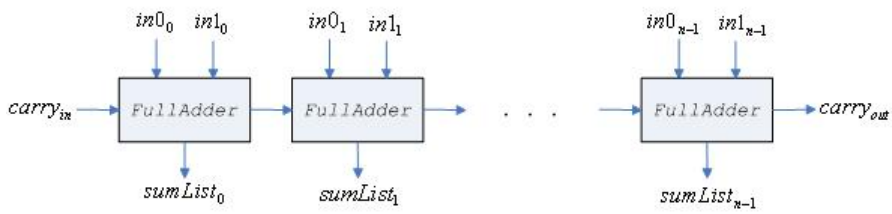

Figure 3.17: A RIPPLE-CARRY ADDER CIRCUIT.

Using a `for-do` loop we can iterate through the two input lists of wires simultaneously and apply to them a full-adder circuit using the resulting carry from the previous application. Therefore for $i = 0 \ldots (n-1)$ wires in each list, we create a full-adder and feed it with the ith wire from the two lists together with the carry-out of the $(i-1)^{th}$ iteration. The final output is a list of sums of length n and the carry-out from the $(n-1)^{th}$ full-adder. The `RippleCarryAdder` class, representing the ripple-carry adder is defined as follows:

```
using SharpHDL; ...

public class RippleCarryAdder : Circuit
{
    //Class constructor
    public RippleCarryAdder(Circuit parent):base (parent){}


   public void gate_o(BusWire in0, BusWire in1, LogicWire carryIn,
                      ref BusWire sumList, ref LogicWire carryOut)
    {
        //...Connect wires to ports

        //ASSUMPTION: Equal length BusWires
        int n = in0.Count - 1;
```

```
        //For-do loop from 0 till the wire-before-the-last
        for(int i=0; i < (n); i++)
        {
            LogicWire sum;
            LogicWire carryOut = new FullAdder(this).gate(in0[i], in1[i], carryIn,
                                                  out sum);

            //The carryIn for the next iteration is
            //the carryOut of this iteration
            carryIn = carryOut;

            //Add the sum to the sum list
            sumList.Add(sum);
        }

        //Last iteration that will output the final carry-out
        LogicWire sum;
        new FullAdder(this).gate_o(in0[n],
                            in1[n], carryIn, out sum, ref carryOut);
        sumList.Add(sum);
    }
}
```

Defining a Carry-Select Adder by Recursion

Another technique supported by C# is recursion. We shall define a recursively built n-bit adder circuit.

A *carry-select adder* [3] is a faster adder implementation than the ripple-carry adder described in the previous section. An n-bit carry-select adder (cf. Figure 3.18) accepts two lists of wires of length n and sub-divides them in two such that the lower half has wires 0 to $\frac{n}{2} - 1$ and the upper half has wires $\frac{n}{2}$ to $n - 1$. The two subdivisions are added recursively using a $\frac{n}{2}$-bit carry-select adder until the length of the lists is 1. In this case it applies a simple full-adder operation.

The addition of the two sub-divided lists are done in parallel, so the carry-in of the upper half, $carry_{in\_upper}$ is unknown. Therefore the addition of the upper subdivision is done twice: once using $carry_{in\_upper} = 0$ and another time taking $carry_{in\_upper} = 1$. Once the carry-out of the lower half is known, a *multiplexer*, defined in section 3.1 on page 14, is used to select the correct sum of the upper sub-division. A SharpHDL implementation of the carry-select adder algorithm is the following:

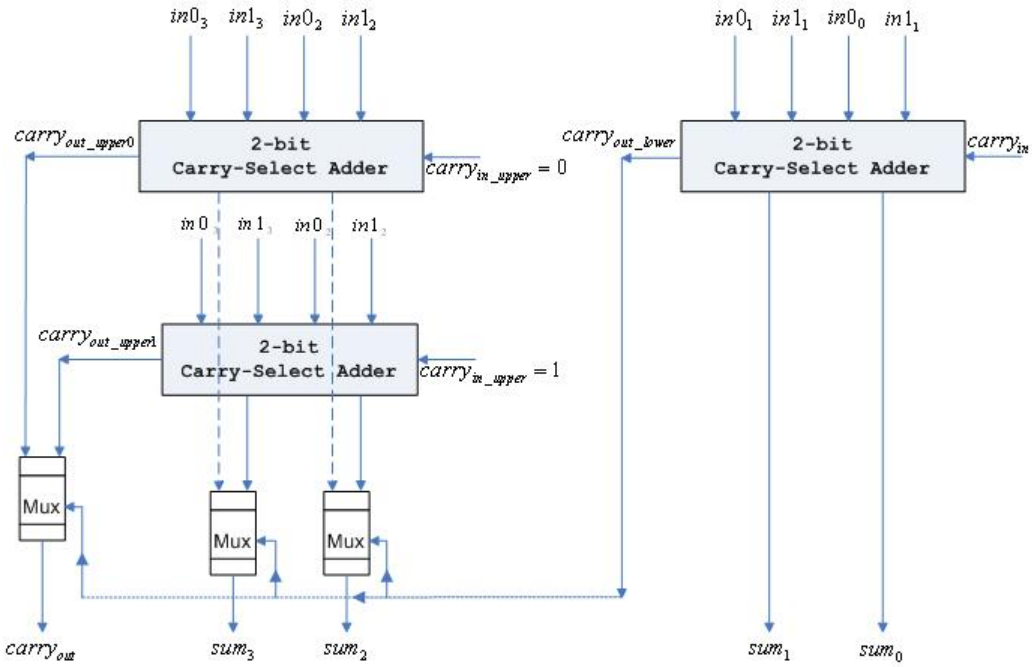

Figure 3.18: A 4-bit carry-select adder circuit, using a 2-bit carry select adder to perform additions on the two subdivided lists of wires.

```
private void gate_o(WireList in0, WireList in1, LogicWire carryIn,
                ref WireList sum, ref LogicWire carryOut)
{
    //Stopping Condition for Recursion
    if (in0.Count==1)
    {
        LogicWire sumWire = new LogicWire();
        new FullAdder(this).gate_o(in0[0], in1[0],carryIn,
                            ref sumWire, ref carryOut);
        sum.Add(sumWire);
    }
    else
    {
        //Phase 1 : Dividing the lists of wires & other initializations
        //Split the two inputs in two
        int newSize = in0.Count/2;
        WireList lowerList0 = new WireList();
        WireList lowerList1 = new WireList();
        WireList upperList0 = new WireList();
        WireList upperList1 = new WireList();

        //Split in0 into lowerList and upperList using newSize
        in0.Split(newSize, ref lowerList0, ref upperList0);
        //Split in1 into lowerList and upperList using newSize
        in1.Split(newSize, ref lowerList1, ref upperList1);
```

```
        //Initialize the carryIns
            //a) carry_in_upper=0
        LogicWire cInUpper0 = new ConstantFalse(this).gate();
            //b) carry_in_upper=1
        LogicWire cInUpper1 = new ConstantTrue(this).gate();

        //Initializing CarryOuts
        LogicWire cOutUpper0 = new LogicWire();
        LogicWire cOutUpper1 = new LogicWire();
        LogicWire cOutLower = new LogicWire();

        //Initializaing Sums
        WireList sumUpper0 = new WireList();
        WireList sumUpper1 = new WireList();
        WireList sumLower = new WireList();


        //Phase 2: Recursive calls
        //Addition of upper part using carry_in_upper = 0
        gate_o(upperList0, upperList1, cInUpper0, ref sumUpper0, ref cOutUpper0);

        //Addition of upper part using carry_in_upper = 1
        gate_o(upperList0, upperList1, cInUpper1, ref sumUpper1, ref cOutUpper1);

        //Addition of lower part
        gate_o(lowerList0, lowerList1, carryIn, ref sumLower, ref cOutLower);


        //Phase 3: Use Multiplexer to choose the sum
        WireList sumUpper = new WireList();
        for (int i = 0; i < sumUpper0.Count; i++)
        {
            LogicWire sum_i = new Multiplexer(this).gate(sumUpper0[i],
                                        sumUpper1[i], cOutLower);
            sumUpper.Add(sum_i);
        }

        //Phase 4: Answer
        //i. Sum - Concatenate sums of upper half with sums of lower half
        sum.Concatenate(sumLower, sumUpper);
        //ii. CarryOut
        new Multiplexer(this).gate_o(cOutUpper0, cOutUpper1, cOutLower, ref carryOut);
    }

}
```

3.5.4 Using generic circuits

One of the major benefit of generic circuits is that they reduce the number of lines of code needed to construct a complex circuit, thus helping the user to generate more elegant code. Looking at the ripple-carry adder defined in section 3.5.3 on page 43 we realize that it has a regular structure which can

be captured by a generic circuit. We therefore try to improve the definition of this circuit by applying the full-adder circuit to a `RowTwo` structure.

To use a circuit as input to a generic circuit, some minor extensions have to be implemented. The most important is the implementation of the appropriate interface. Being that the `RowTwo` circuit will used, the full-adder class should implement the `IRowTwoStructurable` interface. The interface dictates that the `FullAdder` class should have two extra methods implemented:

- `void gate_o(Wire in0, Wire in1, LogicWire inputWire, out Wire out0, ref LogicWire outputWire)`
- `LogicWire gate(Wire in0, Wire in1, LogicWire inputWire, out Wire out0)`

These are the names of the methods that describe the structure of a circuit. Therefore, the new `FullAdder` will be the following:

```
using System;

    public class FullAdder : Logic, IRowTwoStructurable
    {
        //Class constructor
        public FullAdder(Circuit parent):base(parent)
        {
           ...
        }

        //Returns the carryOut wire
        public LogicWire gate(LogicWire in0, LogicWire in1, LogicWire carryIn,
                              out LogicWire sum)
        {
            //... Calls gate_o(...)
        }

        public void gate_o(LogicWire in0, LogicWire in1, LogicWire carryIn,
                           ref LogicWire sum, ref LogicWire carryOut)
        {
            //Connection of wires to ports
            //Structure definition
        }
```

```
        //New implementations ...
        #region IRowTwoStructurable Members

        LogicWire SharpHDL.GenericCircuits.IRowTwoStructurable.gate(Wire in0, Wire in1,
                                        LogicWire inputWire, out Wire out0)
        {
            //Calls gate(..);
        }

        void SharpHDL.GenericCircuits.IRowTwoStructurable.gate_o(Wire in0, Wire in1,
                        LogicWire inputWire, out Wire out0, ref LogicWire outputWire)
        {
            //Calls gate_o(...)
        }

        #endregion
    }
}
```

Using this new `FullAdder` class, the ripple-carry adder can be constructed using the `RowTwo` generic circuit:

```
using System;
using SharpHDL.GenericCircuits;

    public class NBitAdder : Logic
    {
        public NBitAdder(Circuit parent):base (parent){}

        public void gate_o(WireList in0, WireList in1, LogicWire carryIn,
            ref WireList sum, ref LogicWire carryOut)
        {
            //Connect wires to ports...

            //Create RowTwo structure,
            //passing a full-adder circuit to it
            RowTwo row2 = new RowTwo(this, new FullAdder(null));

            //Build circuit
            row2.gate_o(in0, in1, carryIn, ref sum, ref carryOut);
        }
    }
```

This proves that, using generic circuits a designer can write elegant and short descriptions even for complex circuits. Generic circuits also increase code reusability since a pattern can be used for several components without having to rewrite any code.

3.5.5 Verifying implementations

One important use of verification is to check that a new implementation is correct. This can be done by comparing it to a another implementation that is guaranteed to work correctly and checking that, for any input, they give out the same output.

As an example we will check the correctness of the carry-select adder defined in section 3.5.3 on page 44 by verifying that, given the same input, both the ripple-carry adder and the carry-select produce the same result. We assume that the ripple-carry adder is correct. An observer is created that feeds the same input into the two adders, compares their results and outputs whether they are equivalent or not through a one-bit wire. Therefore equivalence is satisfied if the sum and carry-out results are equal for any input list of wires (cf. Figure 3.19):

$$
\begin{aligned}
sum^n &= (sum^n_{ripple\text{-}carry} \Leftrightarrow sum^n_{carry\text{-}select}) \\
Equivalent_{sums} &= \bigwedge_{n=0,\ \ldots,\ size\_of\_outputs} (sum^0,\ sum^1,\ \ldots,\ sum^n) \\
Equivalent_{carrys} &= (carryOut_{ripple\text{-}carry} \Leftrightarrow carryOut_{carry\text{-}select}) \\
Equivalence &= Equivalent_{sums} \bigwedge Equivalent_{carrys}
\end{aligned}
$$

The SharpHDL description of the observer is the following:

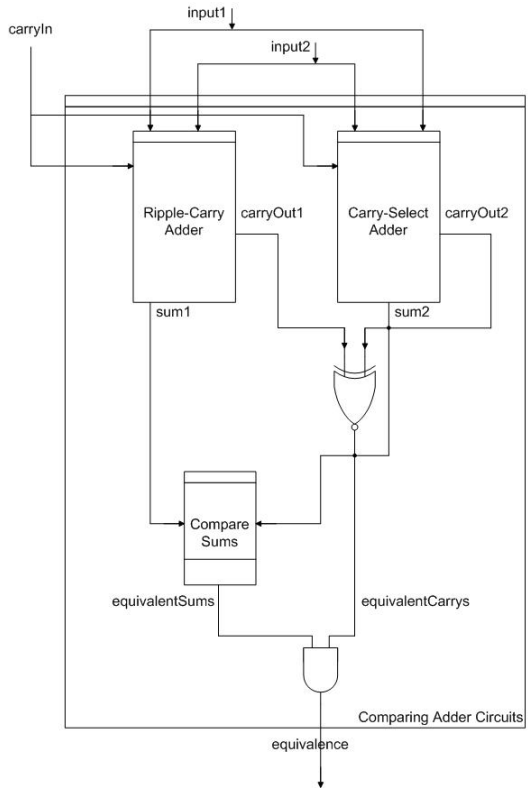

Figure 3.19: The observer feeds the same input to the two adder circuits and checks their equivalence by comparing their output sums and carrys.

```
//Call the two adders using lists input1 and input2
BusWire sum1, sum2;
LogicWire carryOut1 = new RippleCarryAdder(this).gate(input1, input2, carryIn,
                                          out sum1);
LogicWire carryOut2 = new CarrySelectAdder(this).gate(input1, input2, carryIn,
                                          out sum2);

//Use MapTwo to apply NXOR-gate to the sums
BusWire sumN = new MapTwo(this, new NXor(null)).gate(sum1, sum2);

//Use Tree to apply AND-gate to check that equalSums is all 1
LogicWire equivalentSums = (LogicWire)new Tree(this, new And(null)).gate(sumN);

//Compare carry-out
LogicWire equivalentCarrys = new NXor(this).gate(carryOut1, carryOut2);

//Equivalence = equivalentSums & equivalentCarrys
new And(this).gate_o(equivalentSums, equivalentCarrys, ref equivalence);
```

We use wire `equivalence` as an input to the `ToSMV()` method so that the SMV code produced will make the model checker verify that, for any input, `equivalence` is always true.

```
LogicWire [] prove = {equivalence};
this.ToSMV(prove, "C:\\", "CompareAdders");
```

Chapter 4 will discuss a more complex case study using circuits describing FFT algorithms.

3.6 Related Work

medskipIn this section some HDLs that are most related to the work described here are highlighted.

3.6.1 Embedded object-oriented HDLs

The idea of object-oriented HDLs has already been thought of and experimented with. Two such languages are JHDL [45, 46] and PamDC [12, 69].

JHDL is, one of the most renowned languages. This object-oriented hardware description language is embedded in Java and given that it is based on the same paradigm, JHDL considers every element of a circuit to be an object. The two main JHDL classes are:

`Logic`, which is sub-classed by every circuit,

`Wire`, which offers a rich set of methods to enable the user to create and manipulate wires.

One major difference is how objects representing logic gates are accessed. The `Logic` class, which is one of the two main classes of JHDL, provides a quick-hardware method library to build circuits efficiently. Therefore, boolean gates are invoked by calling a method rather than by instantiating an object by a constructor call.

One of the main problems of this approach is that user-created `Logic` classes are treated differently from hard-coded ones. This may affect the reusability and extensibility factors provided by the object-oriented paradigm. It also

makes it difficult to use forms of generic circuits since methods cannot be used as input as can be done with an object. In fact, JHDL provides little or no support for such circuits.

An interesting feature in JHDL is the provision for relative placement information. Studies have shown that user-specified placement information often improves the performance of FPGAs when compared to automatic placement tools [81]. The language allows such specification through two methods, `map()` and `place()`, which map gates to atomic FPGA cells and set the relative placement of such cells respectively. Such a feature can be considered as a future extension to SharpHDL.

JHDL provides a rich CAD suite, which includes a simulator, a waveform viewer where a user can view the wave output by a signal, a Command Line Interpreter (CLI) Console which holds a listing of all commands performed by the user, a graphical circuit browser and a netlister amongst other tools. Unfortunately the language does not provide explicit verification tools. Debugging and verifying a JHDL design is done through the *design window* which was developed with the intention to provide a visualization aid for debugging.

JHDL's main concentration is to develop an efficient CAD suite for designing and testing hardware. Being that the main intention of SharpHDL concentrates round verification, the difference in the approaches and tools offered by the two languages is understandable.

Another OO HDL is PamDC which is embedded in C++. When run, a PamDC program creates a Xilinx netlist file which can be passed to a placement tool. The main targets of this language is to provide control over placement to enhance circuit performance. In fact, it gives the designer full control over the design. Nevertheless this requires relatively high efforts to create structural designs on a very low level.

To simplify this process, in [69] Mencer et al. introduced levels of abstraction over PamDC by implementing an object-oriented hierarchy. These consists of two libraries: `PamBlox` which consists of a set of parameterizable simple templates of objects, normally containing carry chains; and `PaModules` which holds complex, fixed circuits built from `PamBlox` circuits. The whole set of libraries is called PAM-Blox.

PamDC descriptions are quite different from JHDL or SharpHDL ones. All data and methods are declared as public to allow maximal visibility during

simulation. Unlike the other two languages, input and output wire parameters are not defined in the method describing the internal structure of the circuit. Instead inputs are passed to the constructor of the class whilst the outputs are passed to the `out` method. The latter is the method defining the internal logic of the circuit. If any more parameters are required these are passed by declaring the formal parameters to be type reference, while optional parameters are passed as pointers.

Names are also important in PamDC — the language supports a hierarchical naming scheme that creates a unique name for each wire in the design. The names are stored in a format similar to file paths that describe all the ancestors of the wire, i.e. the circuit where it is being used and the circuits that are parent to the circuit. One must note that, like JHDL and SharpHDL, PamDC stores information about circuit hierarchy. Nevertheless, one must point out that he naming mechanism is only applied to the wires.

3.6.2 Other embedded HDLs

There are various HDLs that are embedded in a functional language. The most influential one to this work is Lava [9, 20]. Lava is a HDL embedded in the functional programming language Haskell [83] and therefore circuits are defined in terms of Haskell functions. Although, like SharpHDL, it is not possible to describe circuits in such a depth as the standard HDLs, the descriptions are short and sweet.

By using higher-order functions, it also offers the facilities of *connection patterns*, which are basically the generic circuits found in SharpHDL. Such circuits were used to design and analyze various complex circuits including a sorter core [21], Fast Fourier Transforms [8, 9] and others.

One of Lava's unbeatable features is its elegance. A half-adder circuit description in the language is the following:

```
import Lava
halfAdd(a, b) = (sum, carry)
                where
                  sum = xor2(a, b)
                  carry = and2(a, b)
```

Notice the lack of verbosity in the description — a feature which SharpHDL is not as good at as this Haskell-embedded language. Other Lava tools include

the possibility of analyzing circuits by simulation, verification, and generation of code for input to verification tools and other HDLs that are connected to it. Lava uses the same approach for verification, such that the main kind of properties of circuits Lava deals with are *safety properties* which are tested using *observer* circuits, as described by Halbwachs using the synchronous programming language Lustre [36, 39].

Also embedded in Haskell is the language developed by Launchbury et al, Hawk [54, 58]. It is mainly used to design and verify modern microprocessors and so offers the necessary mechanisms to specify their components. This differs from SharpHDL since our intentions are to model circuits at bit-level.

It considers circuits and components to be functions from signals to signals. As a library it is based on two main abstractions — the `Signal` abstract data type, which holds the state of a wire for every clock cycle; and the `Transaction` abstract data type, which is used to encapsulate the state of an instruction as it is being processed. Verification in Hawk takes a more algebraic form. They use the approach proposed by Burch and Dill [15] which state that the states produced by running the specification should be the same to those resulting after running the implementation, given the same input.

O'Donnell has also developed Hydra [70] — a Haskell library for describing hardware. The main purpose of this langauge is to serve as a teaching tool for the design and testing of a microprocessor. It also uses higher-order functions. The language is capable of describing circuits at different levels levels of abstraction. It defines a `Signal` class which represents the value carried by a wire such that a circuit is defined as a function from `signal` to `signal`. However, Hydra does not allow users to define composite signal types, such as signals of integers.

3.6.3 Other important HDLs

Two languages worth mentioning are μFP [79] and Ruby [50]. These languages are considered to be the stepping stones which led to the development of the functional-based languages we have described in the section before.

μFP is a structural HDL based on Backus' FP developed back in the beginning of the 1980s. A program in FP is an expression representing a function that maps objects into objects. Such functions can be either primitive

functions like arithmetic functions, predicates and sequence manipulators, or combining forms which map functions into functions.

Whilst functions in FP take a single input and produce a single output, μFP programs take a sequence of inputs and produce a sequence of outputs. The nth value of a sequence corresponds to the value of a signal at time $t = n$. This made the representation of sequential circuits possible. This was done by adding a new combining form over Backus' FP which represents state. This was called μ.

Using μFP one can describe not only a circuit's behaviour but also its layout by giving simple geometric interpretations for each of the combining forms. Besides, the process of designing μFP programs can be considered as a program transformation process from an initial abstract specification to an efficient implementation which must obey a set of algebraic rules. Thus, the first notions of circuit verification is suggested here.

Later Sheeran and Jones refined μFP into Ruby [50]. In this language, circuits are defined in terms of binary relations between input and output goals. Placement is an important issue in this language and therefore it defines a set of components which define both the structure of the circuit and its location in relation to other circuits and other external signals.

3.7 Conclusions

In this chapter we have seen an overview of the new SharpHDL language and the tools it provides. By embedding the language into C#, we provided an OO HDL with which circuits can be described structurally. The OO characteristics inherited from the host language allows us to define a set of classes which abstract the general structure of a circuit. Such classes can be extended to define concrete components which given their polymorphism nature, can still be treated at their higher-level description. Also, using modularity, a circuit can be manipulated as an individual component having other circuits, and also itself making up part of a parent circuit. SharpHDL does not flatten the descriptions in terms of the basic boolean gates.

The embedded approach also provides a meta-language which allows the definition of higher-order circuit descriptions. We refer to such descriptions as Generic Circuits and these describe circuits which use other circuits to build another complex circuit having a regular structure.

Using SharpHDL, a designer can define specifications of a circuit in terms of safety properties by building an observer circuit that takes the input and output of a circuit-under-test and decides whether it obeys the property. Therefore we can express both circuits and their specifications using the same language.

SharpHDL can generate other descriptions of the circuit which can be used by verification tools and other HDLs. Until now, we target the SMV model checker and the standard HDL Verilog. Using the SMV descriptions, a designer can verify a circuit with its specifications; the Verilog descriptions permits the use of the various tools that exist for this language including simulators.

In the next chapter we see a complex case study using SharpHDL, whereby we verify the equivalence of two FFT circuits. Further on we further analyze the contributions of embedding a language when we use SharpHDL to build a language for modular verification and refinement.

Chapter 4

Describing and Verifying FFT Circuits using SharpHDL

Fourier transforms are critical in a variety of fields but in the past, they were rarely used in applications because of the big processing power required. However, the Cooley's and Tukey's development of the Fast Fourier Transform vastly simplified this. A large number of FFT algorithms have been developed, amongst which are the radix-2 and the radix-2^2 FFT algorithms. These are the ones that have been mostly used for practical applications due to their simple structure with constant butterfly geometry. Most of the research to date for the implementation and benchmarking of FFT algorithms have been performed using general purpose processors, Digital Signal Processors and dedicated FFT processor ICs but as FPGAs have developed they have become a viable solution for computing FFTs. In this chapter, SharpHDL will be used to describe the circuits implementing the above-mentioned FFT algorithms and verify their equivalence.

4.1 Introducing FFTs

A *Fast Fourier Transform* (FFT) is an efficient algorithm to compute the *Discrete Fourier Transform* (DFT) and its inverse. A DFT, also referred to as a *Finite Fourier Transform*, is a Fourier Transform widely used in *Digital Signal Processing* (DSP) and related fields to analyze the frequencies of a sampled signal, solve partial differential equations and for other operations such as convolutions.

A DFT's problem is defined over an integral domain. We shall discuss DFTs defined over complex numbers. Given such a domain, the DFT's arithmetic consists of the computation of the sequence $\{X(k)\}$ of N complex numbers given a sequence of N $\{x(n)\}$, according to the formula:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \qquad 0 \leq k \leq N-1$$

where

$$W_N^{kn} = e^{-j2\Pi/N} = \cos(\frac{2\Pi}{N} \cdot nk) - j\sin(\frac{2\Pi}{N} \cdot nk)$$

The latter is the principal n-th root of unity, also known as the *twiddle factor*.

One can notice that for each input value of k, $X(k)$ takes N complex multiplications and $N-1$ complex additions. Therefore, to compute all the N values of the DFT N^2 complex multiplications and $N^2 - N$ complex additions are required. This makes a DFT a $\bigcirc(n^2)$ problem. If the *symmetry* and *periodicity* properties of the twiddle factor W_N^k are taken into consideration, a more efficient computation is carried out. The properties specify that

$$\begin{aligned} W_N^{k+\frac{N}{2}} &= -W_N^k \quad \text{(Symmetry property)} \\ W_N^{k+N} &= W_N^k \quad \text{(Periodicity property)} \end{aligned}$$

FFTs use these properties, amongst others, making them efficient algorithms for computing DFTs, such that the problem is reduced to a $\bigcirc(n \log n)$ one [25, 48].

The general idea of FFTs was made popular by a publication of J. W. Cooley and J. W. Tukey in 1965, who re-invented the algorithm of Carl Friedrich Gauss and described how to perform it efficiently on a computer. Gauss observed that a Fourier series of width $N = N_1 N_2$ can be broken up into a computation of the N_2 sub-sampled DFTs of length N. He used it to interpolate the trajectories of the asteroids Pallas and Juno in the year 1805 but this work was not recognized, though various limited forms were also rediscovered several times throughout the 19th and 20th century [74].

4.1.1 Radix-2 decimation in time FFT algorithm

Various forms and variations of FFTs have been developed but the most well-known textbook example is Cooley and Tukey's radix-2 FFT. It works on an input sequence having length to the power of two. It splits the input into the odd-indexed numbers and the even-indexed numbers, hence making it a *decimation-in-time* algorithm. Therefore,

$$\begin{aligned} f_1(n) &= x(2n) \\ f_2(n) &= x(2n+1) \qquad n = 0, 1, \ldots, \frac{N}{2} - 1 \end{aligned}$$

It follows that

$$\begin{aligned} X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{kn} \\ &= \sum_{n\,even} x(n) W_N^{kn} + \sum_{n\,odd} x(n) W_N^{kn} \\ &= \sum_{m=0}^{(\frac{N}{2})-1} x(2m) W_N^{2mk} + \sum_{m=0}^{(\frac{N}{2})-1} x(2m+1) W_N^{k(2m+1)} \end{aligned}$$

But $W_N^2 = W_{\frac{N}{2}}$

Therefore,

$$
\begin{aligned}
X(k) &= \sum_{m=0}^{(\frac{N}{2})-1} f_1(m) W_{\frac{N}{2}}^{km} + W_N^k \sum_{m=0}^{(\frac{N}{2})-1} f_2(m) W_{\frac{N}{2}}^{km} \\
&= F_1(k) + W_N^k F_2(k) \qquad k = 0, 1, \ldots N-1
\end{aligned}
$$

where $F_1(k)$ and $F_2(k)$ are the $\frac{N}{2}$-point DFTs of the sequences $f_1(m)$ and $f_2(m)$ respectively.

Using the symmetry property, we know that $W_N^{k+\frac{N}{2}} = -W_N^k$. We also know that $F_1(k)$ and $F_2(k)$ are periodic, having period $\frac{N}{2}$. Therefore,

$$
\begin{aligned}
F_1(k + \frac{N}{2}) &= F_1(k) \\
&\text{and} \\
F_2(k + \frac{N}{2}) &= F_2(k)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
X(k) &= F_1(k) + W_n^k F_2(k) \qquad k = 0, 1, \ldots, \frac{N}{2} - 1 \\
X(k + \frac{N}{2}) &= F_1(k) - W_n^k F_2(k) \qquad k = 0, 1, \ldots, \frac{N}{2} - 1
\end{aligned}
$$

This final derivation is also known as the *radix-2 FFT Butterfly*, better illustrated in figure 4.1.

The decimation of the data is repeated recursively until the resulting sequences are of length two. Thus, each $\frac{N}{2}$-point DFT is computed by combining two $\frac{N}{4}$-point DFTs, each of which is computed using two $\frac{N}{8}$-point DFTs and so on. The whole radix-2 FFT network is given in figure 4.2.

One should note that the indexes of the input sequence are re-ordered. The technique used is called *bit reversal*. This basically consists of reversing of

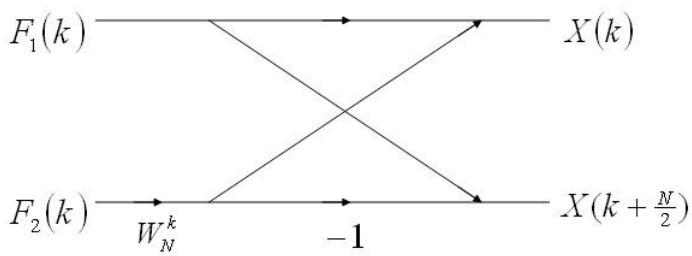

Figure 4.1: Radix-2 FFT butterfly — crossing lines symbolize addition and the numbers on the wires are multiplication.

the bits representing the index of a number in the sequence such that the Most Significant Bit (MSB) becomes the Least Significant Bit (LSB) and vice-versa. Therefore, bit-reversed indexes are used to combine FFT stages [25, 52].

4.1.2 Radix-2^2 decimation in frequency FFT algorithm

Decimation in frequency (DIF) is another FFT algorithm developed by Sande, in the same time as Cooley and Tukey. This technique decomposes the input sequence using a first-half/second-half approach.

One such algorithm is the radix-2^2 FFT algorithm [7], which is a less popular algorithm than the radix-2 FFT algorithm described in the previous section. It is used by an input sequence of length to the power of 4, so that the N-point DFT formula can be broken down into four smaller DFTs.

Therefore, the DFT is calculated as follows:

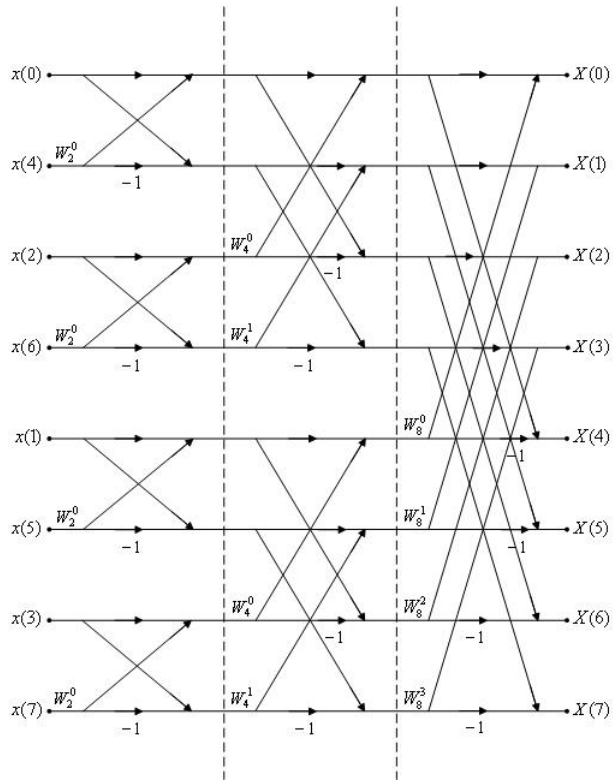

Figure 4.2: A radix-2, decimation in time FFT of size 8.

$$
\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(n) W_N^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} x(n) W_N^{kn} + \sum_{n=\frac{N}{4}}^{\frac{N}{2}-1} x(n) W_N^{kn} + \sum_{n=\frac{N}{2}}^{\frac{3N}{4}-1} x(n) W_N^{kn} + \sum_{n=\frac{3N}{4}}^{N-1} x(n) W_N^{kn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} x(n) W_N^{kn} + W_N^{\frac{Nk}{4}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\frac{N}{4}) W_N^{kn} \\
&\quad + W_N^{\frac{Nk}{2}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\frac{N}{2}) W_N^{kn} + W_N^{\frac{3Nk}{4}} \sum_{n=0}^{\frac{N}{4}-1} x(n+\frac{3N}{4}) W_N^{kn}
\end{aligned}
$$

We know that

$$\begin{aligned} W_N^{\frac{kN}{4}} &= (-j)^k \\ W_N^{\frac{kN}{2}} &= (-1)^k \\ W_N^{\frac{3kN}{4}} &= (j)^k \end{aligned}$$

Hence,

$$X(k) = \sum_{n=0}^{\frac{N}{4}-1} [x(n) + (-j)^k x(n + \frac{N}{4}) + (-1)^k x(n + \frac{N}{2}) + (j)^k x(n + \frac{3N}{4})] W_N^{nk}$$

To get the radix-2^2 DIF DFT, we subdivide the DFT sequence into four $\frac{N}{4}$-point sub-sequences:

$$\begin{aligned} X(4k) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) + x(n + \frac{N}{4}) + x(n + \frac{N}{2}) + x(n + \frac{3N}{4})] W_N^0 W_{\frac{N}{4}}^{kn} \\ X(4k+1) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) - jx(n + \frac{N}{4}) - x(n + \frac{N}{2}) + jx(n + \frac{3N}{4})] W_N^n W_{\frac{N}{4}}^{kn} \\ X(4k+2) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) - x(n + \frac{N}{4}) + x(n + \frac{N}{2}) - x(n + \frac{3N}{4})] W_N^{2n} W_{\frac{N}{4}}^{kn} \\ X(4k+3) &= \sum_{n=0}^{\frac{N}{4}-1} [x(n) + jx(n + \frac{N}{4}) - x(n + \frac{N}{2}) - jx(n + \frac{3N}{4})] W_N^{3n} W_{\frac{N}{4}}^{kn} \end{aligned}$$

Note that the property $W_N^{4kn} = W_{\frac{N}{4}}^{kn}$ is used. A radix-2^2 FFT stage is illustrated in figure 4.3. This procedure is repeated for $\log_4 N$ times.

The corresponding network for this algorithm can be seen in figure 4.4. One can note that the output needs to be re-ordered using a bit-reversal permutation.

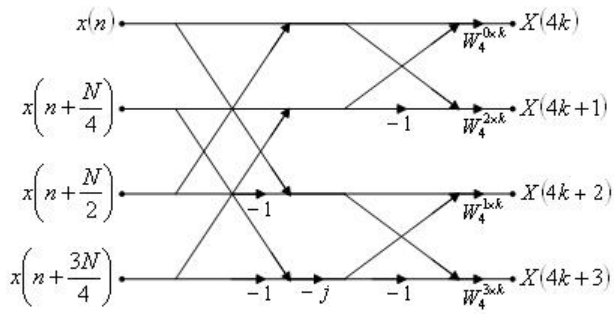

Figure 4.3: A RADIX-2^2 FFT STAGE.

4.2 FFT Descriptions in SharpHDL

In this section we explain how circuits implementing the two FFT algorithms described above were implemented in SharpHDL. Before this was possible, we defined new circuits and other components.

4.2.1 New structures for FFT circuits

Given that the FFTs we consider are based on complex numbers, we define a set of new classes that help us define circuits implementing complex number operations. We also analyze the algorithm networks given in figures 4.2 and 4.4 to capture common patterns and components.

`ComplexArithmetic` — A Library for Complex Numbers

`ComplexArithmetic` is a new library in SharpHDL that holds the objects that represent or manipulate complex numbers. It therefore holds a set of classes that inherit from `Wire`, `Port` or `Circuit`, which are the three main SharpHDL classes that represent a wire, a port and a circuit respectively:

- In general, a complex number is written in the form $a+bi$, where a and b are real value numbers and a is the *real* part and b is the *imaginary* part of the number. So, for example, given the complex number $2+3i$, 2 is the value of the real part of the number whilst 3 is the imaginary-part value [13].

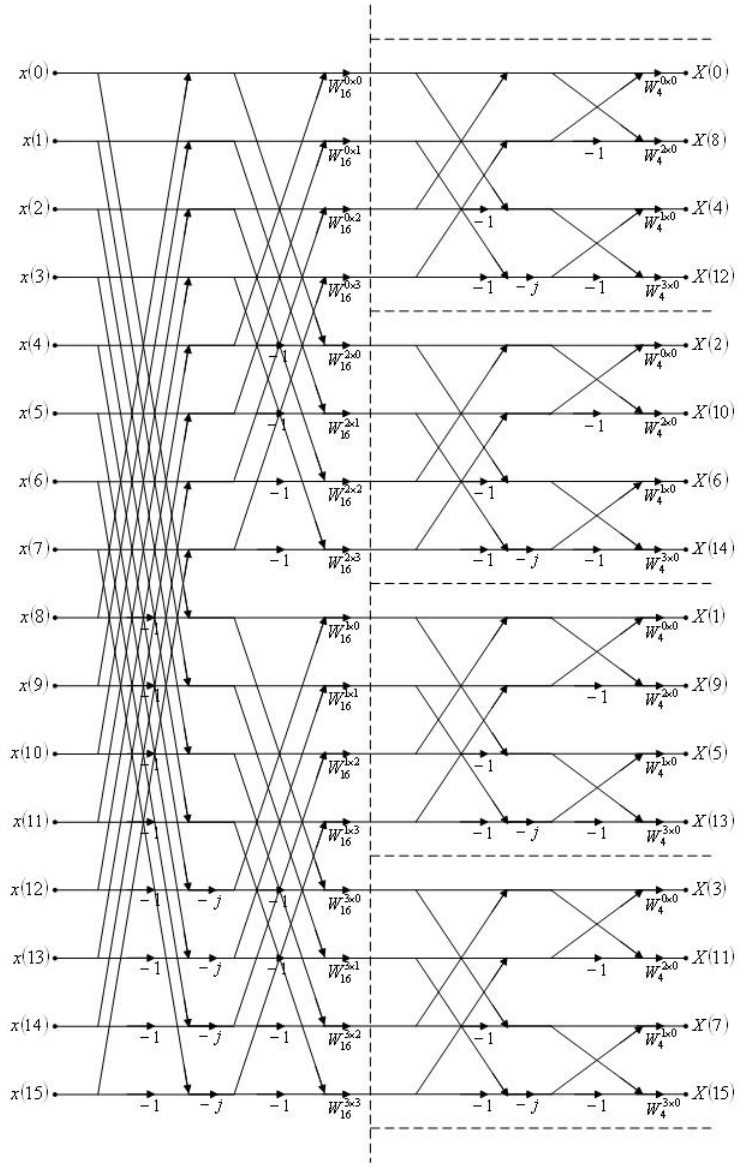

Figure 4.4: A radix-2^2, decimation in frequency FFT of size 16.

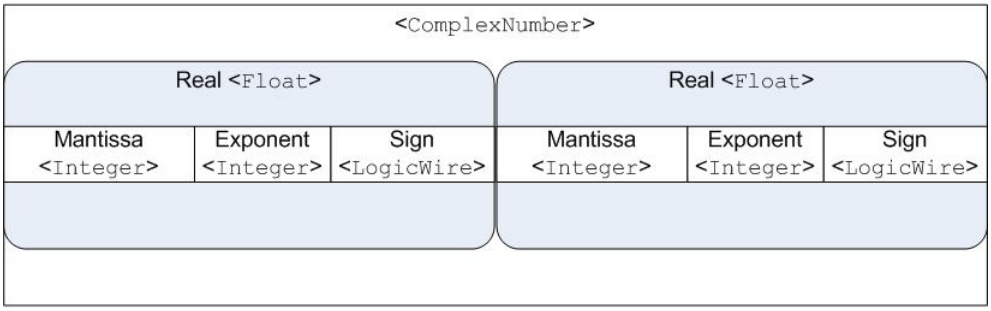

Figure 4.5: ComplexNumber wire is made up of two Float wire objects representing the real and imaginary parts of a complex number.

`ComplexNumber` is the class that captures the signal that carries this type of wire. It is made up of two `Float` objects, representing the real and imaginary parts and thus inherits from `CompoundWire`. The latter class type is the `Wire`-inherited class abstracting signals made up of different types of signals. `Float` represents floating-point numbers which encode real-valued numbers. It is made up of an `Integer` mantissa, an `Integer` exponent and a `LogicWire` sign. Figure 4.5 illustrates the structure of the `ComplexNumber` wire.

A list, or bus of `ComplexNumber` wires is captured by the `ComplexList` class.

- `ComplexPort` and `ComplexListPort` are the `Port`-inheriting classes that allow a `ComplexNumber` and `ComplexList` respectively to connect to a circuit.
- The `ComplexArithmetic` library also has `Circuit` objects that define circuits that operate on complex numbers. These include circuits implementing complex number addition (`ComplexAdder`) and complex number multiplication (`ComplexMultiplier`).

New Generic Circuits

Analyzing the networks illustrated earlier on, we notice that the algorithms are built on regular patterns which can be defined as higher order circuits

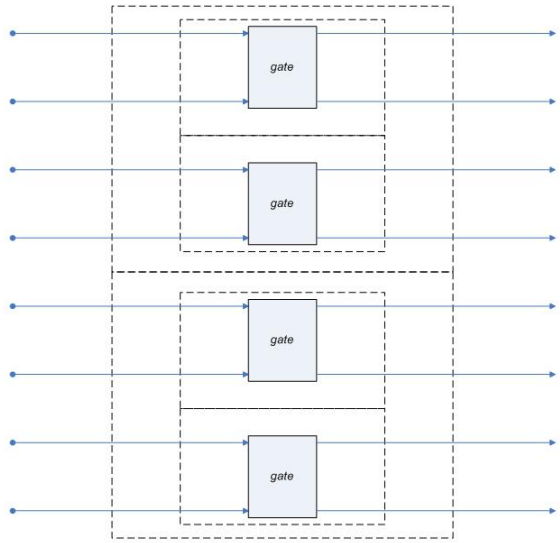

Figure 4.6: `TwoN(GATE, 2)` — THE GENERIC CIRCUIT RECURSIVELY DIVIDES THE INPUT LIST IN TWO FOR $N = 2$ TIMES AND APPLIES COMPONENT `GATE` TO EACH SUBDIVISION.

in the `GenericCircuits` library. Three generic circuits were defined for the FFT implementations [9]:

- The generic circuit `TwoN` recursively divides a list of wires in two for N times and applies the input circuit to each subdivision (cf. Figure 4.6). The input component must have $\log_2 N$ input ports of the same type.
- `OneN` is similar to `TwoN` but instead of applying the circuit to each resulting subdivision, it applies it to the most bottom sub-division only (cf. Figure 4.7).
- A `Butterfly` circuit is a circuit that can be built recursively. The valid input circuit is a component having two inputs and two outputs. After riffling the list, it uses the `TwoN` generic circuit to divide a given list of wires for $\log_2 N$ where N is the size of input list. It then applies the input circuit to the resulting subsets of wires. At the end, the list is unriffled. By riffling we mean perfectly shuffling two half-lists, while unriffling is the reverse operation i.e. the list is shuffled such that the resultant list's first elements are the even-indexed elements of the input

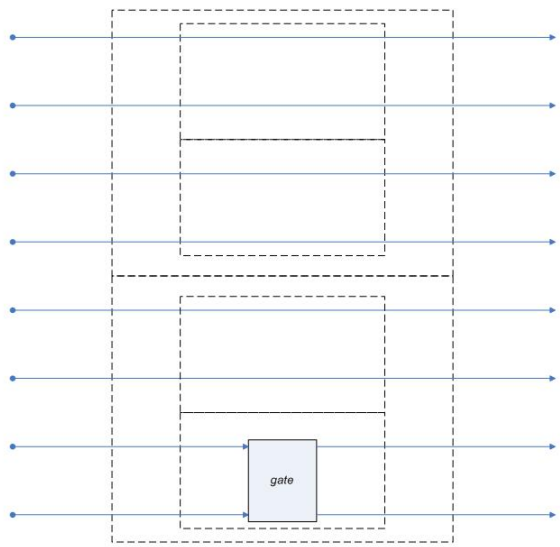

Figure 4.7: `OneN(gate, 2)` — the generic circuit recursively divides the input list in two for $N = 2$ times and applies component `gate` to the bottom subdivision.

list, followed by the odd-indexed elements [9, 49]. Figure 4.8 illustrates this circuit.

Common Components

Although being two different approaches to solving FFTs, the algorithms have some common operations:

- From the network designs one can notice the basic *FFT operation*. This takes two inputs x_1 and x_2, and returns $x_1 + x_2$ and $x_1 - x_2$ (cf. Figure 4.9). This is the main operator of the algorithms as it calculates the 2-point DFT.

- Both algorithms use the *bit-reversal permutation*, though at different times. This is carried out by splitting an input list of wires in two and recursively applying the bit-reversal operation to each sub-division again. This is repeated until the sub-divisions hold only two wires, where in such a case the wires are riffled and returned.

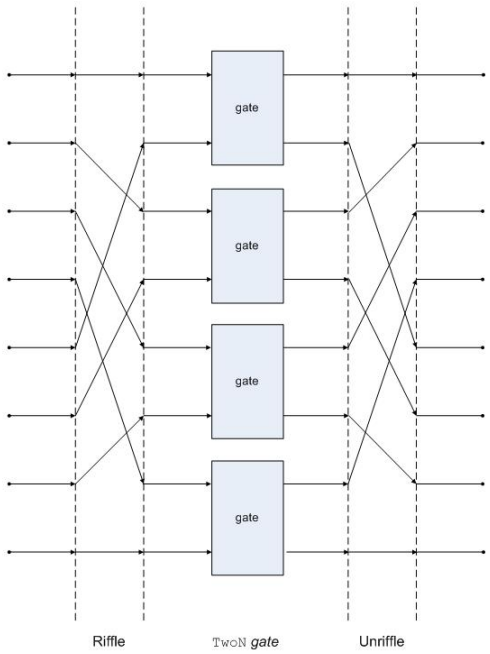

Figure 4.8: A **Butterfly** circuit using logic component **gate** on a list of eight wires.

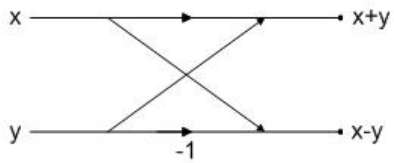

Figure 4.9: The basic FFT operator circuit — given two numbers, it outputs their addition and subtraction.

4.2.2 SharpHDL descriptions

Using the described circuits and components the two FFT algorithms are implemented in SharpHDL.

A radix-2 FFT algorithm first bit-reverses the input list of complex numbers and then uses class `Radix2Stage` as an input to the `TwoN` generic circuit:

```
...
//Bit Reversal Permutation
ComplexList in0 = (ComplexList)new BitReversal(this).gate(in0);

int exponent = (int) Math.Log(in0.Count, 2);

for (int i = 0; i < exponent; i++)
{
    int n = (exponent - 1) - i;

    //Use TwoN to divide the list for n times and apply
    //the Radix2FFTStage to each resultant set
    ComplexList output0 = (ComplexList)new TwoN(this,
                    new Radix2FFTStage(null),n).gate(in0);
    in0 = output0;
}
```

Class `Radix2Stage` defines the radix-2 stage, described in section 4.1.1. Therefore, the SharpHDL description of the `Radix2Stage` is given as follows:

```
...
//Apply twiddle factor using OneN generic circuit
BusWire one = new OneN(this, new FFT2Twiddle(null), 1).gate(in0);

//Call Butterfly using the FFTComponent
new Butterfly(this, new FFTComponent(null)).gate_o(one, ref outBus);
```

One the other hand, the radix-2^2 FFT algorithm uses the `TwoN` generic circuit to apply `Radix4FFTStage` for a number of times depending on the size of the input list, and then bit-reverses the output. This is described as follows:

```
...
int exponent = (int)Math.Log(in0.Count, 4);
int n;
for (int i = exponent; i > 0; i--)
{
    n = (exponent - i)*2;

    //Use TwoN to divide the list for n times and apply
    //the Radix4FFTStage to each resultant set
    ComplexList intermediate = (ComplexList)new TwoN(this,
                new Radix4FFTStage(null), n).gate(in0);
    in0 = intermediate;
}
//BitReversal
new BitReversal(this).gate_o(intermediate, ref output0);
```

`Radix4FFTStage` is the class that describes the circuit implementing a radix-2^2 stage, explained in section 4.1.2:

```
...
//Call Butterfly using the FFTComponent
BusWire bflys = new Butterfly(this, new FFTComponent(null)).gate(in0);

//Use OneN to multiply with -j
BusWire ones = new OneN(this, new MinusJ(null), 2).gate(bflys);

// Use TwoN to divide the list of wires and apply a Butterfly to it
// the Butterfly circuit uses the FFTComponent
ComplexList twoBflys = (ComplexList)new TwoN(this,
        new Butterfly(null, new FFTComponent(null)), 1).gate(ones);

//Multiply with twiddle constant
new FFT4Twiddle(this).gate_o(twoBflys, ref out0);
```

4.3 Verifying the Equivalence of the FFT Circuits

Using SharpHDL circuits, descriptions can be generated so as to allow verification. We use this tool to verify the equivalence of the two implemented FFT circuits. The two circuits are equal if their output is the same, given the same input [65]:

Definition 4.3.1. *Consider two circuits x and y. Given that $\gamma(c, i)$ is the function that determines an output from circuit c given input i then the equivalence relation R is such that*

$$\forall i \,.\, (\, \gamma(x, i) \;=\; \gamma(y, i)\,)$$

This can be implemented by constructing an observer circuit which calls the two FFT circuits using the same input, and compares their outputs. The circuit outputs `true` if all the FFT outputs are equal [36]:

$$\begin{aligned} output^n &= (output^n_{radix\text{-}2} \Leftrightarrow output^n_{radix\text{-}2^2}) \\ Equivalence &= \bigwedge(output^0,\ output^1,\ \ldots,\ output^n) \\ & \qquad n = 0,\ \ldots,\ number\_of\_outputs \end{aligned}$$

where $output_{radix-2}$ and $output_{radix-2^2}$ are the list of complex numbers output from the radix-2 FFT circuit and the radix-2^2 FFT circuit respectively (cf. Figure 4.10).

Following, is the SharpHDL description for the circuit:

```
//Radix 2 FFT
ComplexList outList2 = new ComplexList(); new
Radix2FFT(this).gate_o(inputlist, ref output_radix2);

//Radix 4 FFT
ComplexList outList4 = new ComplexList(); new
Radix4FFT(this).gate_o(inputlist, ref output_radix2_2);

//MAP NXOR to check that the nth output of each circuit are equal
WireList equalityList = new NXorMap(this).gate(output_radix2, output_radix2_2 );

//Use TREE to check that ALL NXOR outputs are true
new AndTree(this).gate_o(equalityList, ref equivalence);
```

Using the one-bit wire `equivalence` as input, we generate a description which, given to a verification tool, verifies that it is true for any input, thus proving equivalence. Successful testing was carried out on size 4 FFT circuits.

4.4 Related Work

Much work has been done on FFTs using different hardware description languages and various approaches.

The approach described here can be easily related to the FFT implementations in Bjesse et al. using Lava in [9]. Here the verification of the equivalence

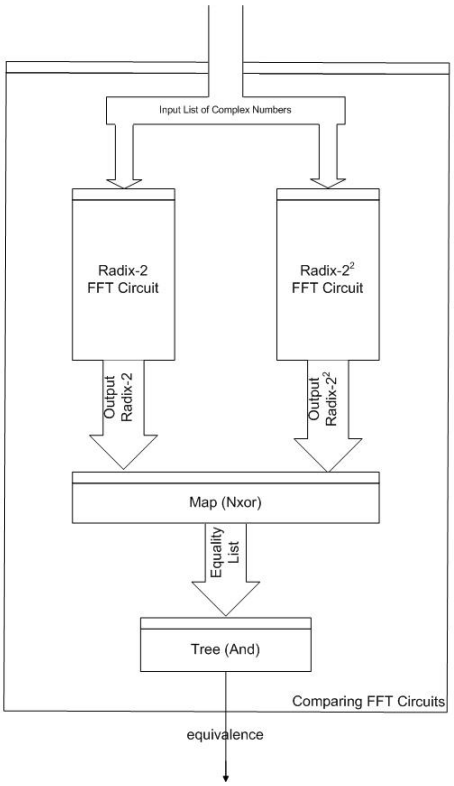

Figure 4.10: The observer feeds the same input to the two FFT circuits and checks their equivalence by comparing their outputs.

of two algorithms is also used as a case study for the language. The algorithms used are also the Radix-2 and the Radix-2^2.

The design approach taken in this work is similar to our work. Lava was first extended with new definitions including the introduction of a complex number datatype and the definition of the butterfly circuit (`bflys`). Other defined components also include a circuit that multiplies a given input with a twiddle factor (`wMult`) and another circuit that carries out the bit-reversal permutation (`bitRev`).

The description of the Radix-2 FFT circuit in Lava is the following:

```
radix2 n =
    bitRev n >-> compose [ stage i | i <- [1..n] ]
where
    stage i = raised (n-i) two
            $ twid i
            >->bflys(i-1)

twid i = one (decmap (2^(i-1)) (wMult(2^i)))
```

`raised` is a function that, in this case will repeat the function `two` for $n - i$ times. Function `two` works in the same way as circuit `TwoN` described earlier on in this chapter.

One can notice that the Lava and SharpHDL definitions are quite similar, though different in style. However, one must point out that the SharpHDL code displayed omits statements that are needed to define a regular C#/SharpHDL class, like constructor definitions and method names.

The Lava approach differs in the verification system used — it targets the first order theorem prover Otter [59]. Also, the logical descriptions of the circuits of a fixed size are automatically generated and given to the theorem prover together with proof options consisting of laws about the twiddle factors. Therefore the twiddle factors are abstracted and the verification process uses relationships and theories of such. In our case, the twiddle factor values are generated whilst the SMV code is being produced, thus the values are "hard-coded" in the generated code.

Based on this work, Bjesse also conducted work in proving the equivalence between two combinational FFT circuits and a pipelined FFT circuit [8] using an induction principle.

Gamboa defines an FFT circuit using powerlists in the theorem prover ACL2 and verifies its correctness [33]. Powerlists are lists constructed using parallel

operators on other powerlists. The proof requires user interaction since the level of abstraction is very high. The approach consists of proving mathematical theorems about the algorithms rather than showing the equivalence of two automatically generated logical descriptions of circuits.

4.5 Further Work and Conclusions

In this chapter we discussed the implementation and verification of two circuits describing different FFT algorithms using SharpHDL.

After a good analyzes of the derivations and networks of these algorithms, we extended the language with a new library that captures complex numbers and operations that work on them. We also defined new generic circuits and other useful components. Having defined the circuits for the radix-2 and radix-2^2 FFT algorithms, we built an observer circuit that compared their outputs given the same inputs. Using the code-generation tool of SharpHDL, we generated a description of the circuits and the observer. This description allowed us to verify that, for any input, the two circuits give the same output, thus confirming their equivalence.

The experiment was conducted using the SMV model checker and was successful for size-4 FFT circuits. Model checking techniques suffer from the state-explosion problem (i.e. the exponential growth in the state space that needs to be searched) and this made it impossible for us to verify the equivalence for larger circuits. We plan to consider methods that will allow us to verify larger circuits. One such method is modular verification, which allows a verification problem to be decomposed to smaller problems which can be verified individually, thus reducing the states. We will discuss modular verification and refinement in SharpHDL in the next chapter.

Chapter 5

Modular Verification and Refinement in SharpHDL

Modular verification and refinement methods are important techniques to reduce the state-explosion problem of verifying large and complex systems. They allow a verification problem to be decomposed into smaller manageable sub-problems and prove each sub-problem against its predefined property, assuming that the rest of the system is correct. We embed a simple refinement language over SharpHDL which captures constructs that allow the definition of the environment and specifications of circuits. It also includes a special construct that allows a user to define a circuit in two ways: either by defining the observer that defines its specification or by defining its actual implementation. Using such a structure we can use the specification to assume correct behaviour and then refine it to the correct implementation. The object-oriented nature of the language allows these extensions to be included with great ease. We also see how the imperative nature of SharpHDL gives the user more control over the verification process.

5.1 Introduction

Verifying hardware before fabrication is an important step when developing hardware as this helps to reduce the high cost of finding faults during or after the actual implementation. Most HDLs today provide for formal verification where, in many cases, descriptions are converted and input to a verification tool: the Haskell-embedded HDL Lava [9] interfaces with various verification tools including the propositional tautology checker Prover, the first order logic theorem provers Otter and Gandalf, and the model checker SMV amongst others. Hawk [58], another Haskell-based HDL, enables verification of the correctness of microprocessors through the mechanical theorem prover Isabelle. Lustre [39], a synchronous data-flow language has the symbolic, BDD-based model-checker Lesar associated with it.

It was our prime interest and objective to provide for circuit verification in SharpHDL. The language is able to convert SharpHDL descriptions into a description which can be fed to verification tools to verify that the circuit conforms to a specification. In SharpHDL, specifications are described as a *safety property*. Such a property states that a given situation should never occur or a condition is always true.

In chapter 3 we discussed how a SharpHDL designer can verify a circuit against a safety property by expressing it using an *observer* circuit; such a circuit observes the inputs and outputs of the circuit under observation and outputs a single wire stating whether they satisfy the property.

A step further than this introduces us to modular verification and refinement. *Modular verification* allows the verification problem of a compound system to be broken down into smaller parts and have each part verified separately. Each part assumes that the other parts are implemented correctly when it is being verified. This technique is a particulary useful technique, especially in complex systems or when there are structurally-unknown components. In the latter scenario, verification is carried out assuming that the unknown components abide with their specifications. In other words, the verification of the system sees the specification properties as *environment assumptions*. Environment assumptions are a set of conditions under which a system or circuit works correctly.

Furthermore, an unknown component may have its implementation updated, or *refined*. The new updates have to obey to the set of specification rules

defined for the specific component so that the verification of the system as a whole is not affected. This process is called *refinement*.

Such verification techniques has been experimented with using different types of languages [1, 23, 32, 36, 53, 62, 65] but these lack in providing the user with control facilities over the verification process mainly due to the language syntactic and semantic structure. In this chapter, we investigate whether an OO approach will enable us to build a system whereby a designer is able to specify and document the verification process by taking advantage of the imperative style of the language. We do this by embedding a simple language over SharpHDL, where one of its classes defines a construct that allows the description of both the implementation and the specification observer of a circuit. The circuit, therefore, can be handled in two ways during verification: either using the observer as an assumption that the circuit behaves according to the specification or using the implementation, which can also be refined and verified against the observer.

This chapter will start by introducing some important definitions on which the development is based. The mappings of these definitions into the new language are described in section 5.3 whilst section 5.4 describes some examples of how this language can be used. In section 5.5, we show the advantages of modular verification by experimenting with an integer squarer. A section of related work and some conclusions on this work follows at the end.

5.2 Formal Definitions

In this section we will formally define the important terms that will be used throughout this chapter[1].

5.2.1 A circuit

Definition 5.2.1. *A* circuit C *is defined as a 5-tuple* $(Q_C, q0_C, I_C, O_C, \delta_C)$ *where*

- Q_C *is a set of states,*

[1] Definitions are taken from [36] and [41]

- *$q0_C$ is the initial state, $q0_C \in Q_C$,*
- *I_C, O_C are disjoint sets of input and output signals respectively,*
- *δ_C is the transition function, where $\delta_C \subseteq Q_C \times E_{I_C} \times E_{O_C} \times Q_C$. E_X is the set of events on signals X where $E_X = 2^X$. $(q, i, o, q') \in \delta_C$ will be abbreviated to $q \rightarrow^i_o q'$ when there is no ambiguity.*

Given a sequence $(i_1, i_2, \ldots, i_n, \ldots)$ of input events, a circuit C returns the sequence $(o_1, o_2, \ldots, o_n, \ldots)$ of output events.

For a sequence $(q_0, q_1, \ldots, q_n, \ldots)$ of states, where $q_0 = q0_C$ and $n \geq 1$, $q_{n-1} \rightarrow^{i_n}_{o_n} q_n$.

Definition 5.2.2. *For a circuit C, the trace of C, written $\tau(C)$, is defined as*

$$((i_1 \cup o_1),\ (i_2 \cup o_2),\ \ldots,\ (i_n \cup o_n))$$

Therefore, for a sequence of states $(q_0, q_1, \ldots, q_n, \ldots)$ where $q_0 = q0_C$, the transition relation from $q0_C$ to q_n, given a finite trace σ is noted as $q0_C \rightarrow^\sigma q_n$. Finally, for $q \in Q$, $traces(q)$ is the set of traces produced for the transition from the initial state to q, defined as the set $\{\, \sigma \mid q0_C \rightarrow^\sigma q \,\}$.

5.2.2 Safety properties and observers

Definition 5.2.3. *A property on a set of signals S is a set of traces on S.*

Definition 5.2.4. *A circuit C satisfies a property P if and only if each trace σ of C belongs to P, such that*

$$\forall \sigma \ \cdot\ \sigma \in \tau(C) \ \Rightarrow\ \sigma \in P$$

written

$$\tau(C) \ \Rightarrow\ P$$

For a circuit C, let S be the disjoint set of input signals I_C and output signals O_C, such that $S = I_C \cup O_C$. An observer Ω_P of safety property P on signals S is a circuit which takes S of the circuit C and outputs an alarm signal α, $\alpha \notin S$:

Definition 5.2.5 (Observer). Observer Ω_P *of property* P *on signals* S *is a circuit defined as*

$$\Omega_P = (\, Q_{\Omega_P},\, q0_{\Omega_P},\, S,\, \{\, \alpha \,\},\, \delta_{\Omega_P} \,)$$

where

- Q_{Ω_P} *is the set of states;*
- $q0_{\Omega_P}$ *is the initial state,* $q0_{\Omega_P} \in Q_{\Omega_P}$*;*
- S *is the disjoint set of the input signals* I_C *and output signals* O_C *of the circuit under observation* C*;*
- $\{\, \alpha \,\}$ *is the set of output signals. The set is empty if* $S \in P$*;*
- δ_{Ω_P} *is the transition function.*

Given a finite trace σ on signals S, $S \in P$, q_σ is the state that Ω_P reaches after reading σ. Given a state q, input event e, $e \in E_S$, and output o,

$$\delta^O_{\Omega_P} = \begin{cases} \emptyset & \text{if } \sigma.e \in \mathrm{P} \\ \{\alpha\} & \text{otherwise} \end{cases}$$

Satisfying a Safety Property

Verifying that a circuit C satisfies a safety property P consists in checking that their *parallel composition* $C \parallel \Omega_P$ never outputs α. Parallel composition combines two circuits into one circuit whose behaviour captures the intersecting behaviour of both circuits.

Definition 5.2.6 (Synchronous Parallel Composition). *For circuits X_1 and X_2, the* parallel composition $X_1 \parallel X_2$ *is defined to be circuit C defined as*

$$C = (\, Q_C,\, q0_C,\, I_C,\, O_C,\, \delta_C \,)$$

where:

- $Q_C = Q_{X_1} \times Q_{X_2}$, *is the set of states of C, resulting from the product of states of X_1 and X_2.*
- $q0_C = (\, q0_{X_1},\, q0_{X_2} \,)$, *is the pair of initial states of each circuit.*
- $I_C = (I_{X_1} \setminus O_{X_2}) \cup (I_{X_2} \setminus O_{X_1})$, *is the set of input signals consisting of the input signals of X_1 that are not output signals of X_2, and the input signals of X_2 that are not output signals of X_1.*
- $O_C = O_{X_1} \cup O_{X_2}$, *is the disjoint set of output signals of the two circuits.*
- δ_C *is the transition function.*

$$((\, q_1,\, q_2 \,),\, i,\, o,\, (q_1',\, q_2')) \in \delta_C \quad \Leftrightarrow \quad \begin{array}{l} (\, q_1,\, (\, i \cup o \,) \cap I_{X_1},\, o \cap O_{X_1},\, q_1' \,) \in \delta_{X_1} \\ \textit{and} \\ (\, q_2,\, (\, i \cup o \,) \cap I_{X_2}, o \cap O_{X_2},\, q_2' \,) \in \delta_{X_2} \end{array}$$

A transition of C therefore involves a transition from each circuit which is triggered by the global input and the output signals from the other circuit.

To verify a safety property P over circuit C consists in checking that $C \| \Omega_P$ never reaches an invalid state defined as $Q_C \times (Q_{\Omega_P} \setminus \{q_\alpha\}$ where Ω_P is the observer of P and q_α is the state reached after outputting the error alarm α. In other words, the safety property is translated into an *invariant* (cf. Figure 5.1). This is done without modifying the circuit C [34, 36, 37].

5.2.3 Defining the environment

Like any reactive system, circuits interact with a specified environment which must be taken into consideration during design and verification. Environment properties make restrictions on both the input and output of a circuit.

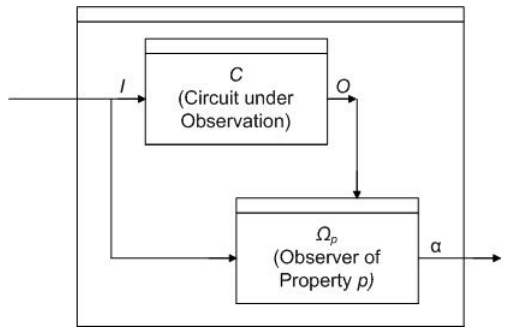

Figure 5.1: Translating a safety property into an invariant.

Therefore, given these constraints, one concludes that not all traces of a circuit are real. Using safety properties to define environment constraints, we can restrict the circuits to react only to real situations.

Given an environment assumption A, the restricted circuit C' has the same behaviour as circuit C composed with A, i.e. the set of traces of C' is the intersection of the set of traces of C with A.

Definition 5.2.7 (Restricted Circuit). *Given that circuit Ω_A is the observer of safety property A on signals $S = I_C \cup O_C$ of circuit C, and $C' = C \| \Omega_A$, the* restricted circuit *C/Ω_A is the circuit $(\, Q_{C'},\, q0_{C'},\, I_C,\, O_C,\, \delta'\,)$ where $\delta' = \{\, (\, q,\, i,\, o,\, q'\,) \in \delta_{C'} \mid \alpha \notin o \,\}$.*

Given these definitions, it is possible to verify a safety property P on a circuit C having environment assumptions A by proving that $(C/\Omega_A) \,\|\, \Omega_P$ never emits an alarm α_{Ω_P}.

5.2.4 Modular verification and refinement

Modular verification allows decomposition of a verification problem such that the verification of a component consists of individually verifying each subcomponent, considering the rest of the system as part of its environment. This technique is referred to as the *assume-guarantee* reasoning [41, 62].

Definition 5.2.8 (Assume-Guarantee Proof). *Let P be a safety property defined on circuit C, where C is the composition of circuits C_1 and C_2 such*

that $C = C_1 \parallel C_2$. If C_2 holds property A and C_1/Ω_A satisfies P then $C_1 \parallel C_2$ also satisfies P:

$$\frac{C_1,\, \Omega_A \vDash \Omega_P \qquad C_2 \vDash \Omega_A}{C_1 \| C_2 \vDash \Omega_P}$$

To verify a subcomponent we can *assume* that the other subcomponents abide with their specification properties. Then, to continue proving the rest of the subcomponents, the subcomponent is assumed to behave according to its specification. This is an extended idea of the assume-guarantee reasoning called *circular composition* [62].

A property can be considered a higher-level description of a circuit. A circuit can be *refined* to a lower-level version consisting of a deeper description of the circuit with the same behaviour. If a circuit P is a higher-level description of C, then C is a *refinement* of P, noted as $C \preceq P$.

Definition 5.2.9. *A circuit C is a* refinement *of property P if every trace σ_C of C is a trace σ_P of P.*

5.3 Embedding a Refinement Language

To allow modular verification and refinement in SharpHDL, we use once again a language-embedding approach and build a new language that provides the necessary functionalities. Therefore using this language, we can define SharpHDL circuits which can further be manipulated to allow the mentioned verification techniques. The refinement language contains a set of classes defining the new circuits and structures needed:

- `DualCircuit` — allows a designer to define both the specification and implementation of a circuit. It is made up of two objects (cf. Figure 5.2):
  - `Specification` object, which defines the specification observer;

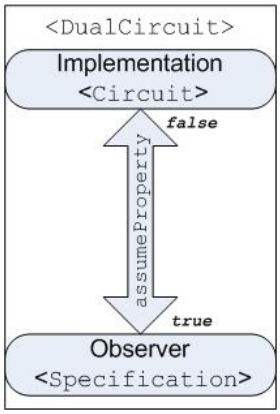

Figure 5.2: The structure of a DualCircuit component.

– `Circuit` object, which defines the implementation.

The latter has to implement the `IImplementation` interface to allow the circuit to be easily manipulated. `IImplementation` imposes the definition of two methods:

– `void gate_o(Wire [] inputs, ref Wire [] outputs)`
– `Wire [] gate(Wire [] inputs)`

These are the names of the methods that describe the structure of a circuit. Circuits implementing `DualCircuit` can be treated in either of two ways during verification: either using the implementation or using the specification observer to assume correct behaviour. A designer can toggle between these two options by alternating the value of this class property `AssumeProperty` — when set to `True`, verification will use the observer as an assumption; otherwise the implementation is used.

`DualCircuit` class also allows the refinement of a specification into an implementation by means of the method `Refine()`.

- `Observer` — represents the observer of a property. It outputs a `LogicWire` signal representing an observer's alarm signal which has to be verified or assumed. This class is extended by two classes:

- `Specification` — abstracts the observer of a specification property. The class allows the user to choose whether to assume or prove the specification during verification by setting the class' property `Type` to one of the `SpecType` values `ASSUME` and `PROVE` respectively.
- `Environment` — allows the description of observers defining environment conditions under which conditions a circuit state is considered to be valid.

These classes inherit from class `RefinementCircuit`, which class inherits from `Circuit` to provide extended options for generating verification descriptions and other methods. It implements a flag `Status` which describes the status of the circuit. This can have the following values:

`REFINED`, when a `DualCircuit` circuit has just had its implementation defined through the `Refine()` method;

`ASSUMED`, when a `DualCircuit` circuit is using the specification observer as an assumption of correct behaviour;

`VERIFIED`, when the circuit has been successfully verified;

`FALSE_VERIFICATION`, when the circuit has not been unsuccessfully verified;

`VERIFIED_WITH_ASSUMPTIONS`, when the circuit has been successfully verified, assuming some conditions.

5.3.1 Generating circuit descriptions for verification

The `RefinementCircuit` class extends the description-generator method of class `Circuit` to provide an algorithm that allows efficient use of modular verification. The aim is to exploit the imperative nature of SharpHDL to provide a scripting language that allows user control and documentation possibilities over the verification process.

Given a circuit, the algorithm first checks whether it has already been successfully verified and whether the verification process considered behaviour assumptions of some sub-components in the given circuit, i.e. it checks whether in the previous verification there were any `DualCircuit` circuits that

used the specification observer as an assumption of correct behaviour. This check is done by accessing the `Status` flag of the input circuit which, in the presence of assumptions in the last verification task, will have the value `VERIFIED_WITH_ASSUMPTIONS`.

Given this situation, the algorithm checks for new refinements that have taken place since then. In other words, it checks whether the `DualCircuit` components that used their specification observer in the last verification task has been refined to an implementation, thus having their `Status` value set to `REFINED`. For each refined component found, the algorithm creates an *invariant circuit* that connects the refined implementation to the specification observer and produces their description to verify if the implementation is correct with respect to the specification. Thus, based on the modular verification concept, SharpHDL automatically produces descriptions that allow the verification of the refined implementations and not re-verification of the whole system. Following is the pseudo-code for this algorithm:

```
method generateCode()
{
  if (circuit.status == VERIFIED_WITH_ASSUMPTIONS)
    then
    {
        new_refinements = get list of newly refined circuits

        if new_refinements is empty
            then
            {
                circuit.status = DEFAULT
                return code
            }
            else
            {
                foreach (refined_circuit rc in new_refinements){
                invariant = build_Invariant(rc.implementation, rc.observer)
                invariant.generateCode()
            }
    }
    else
    {
        return code
    }
}
```

To cater for the new extensions, a new value, `VERIFICATION_WITH_ASSUMPTIONS` has been added to the enumerated type `VerificationReport`, defined in section 3.4.3. This value is returned by the description-generating methods when some sub-components of a circuit use correct-behaviour assumptions instead of their implementation.

5.3.2 A scripting language for refinement

We shall use a simple example to illustrate how the imperative nature of SharpHDL provides us with a scripting language for the verification process.

Consider a circuit `mainCircuit` made up of several components, some of which are implemented as `DualCircuit` classes. The latter components have both the specification observer and implementation parts defined. We want to verify a property over `mainCircuit`, defined using the observer `mainObserver` which outputs alarm signal `mainAlarm` (cf. Figure 5.3(a))

We can verify `mainCircuit` assuming that the `DualCircuit` subcomponents satisfy their specification, i.e. the verification process assumes that the specification observers defined over such subcomponents output `true` (cf. Figure 5.3(b)). If this satisfies `mainObserver`, we can be take one `DualCircuit` component at a time and switch it to its implementation version to verify its correctness assuming that the rest of the sub-circuits conform to their specifications (cf. Figure 5.3(c)).

Looking at the code, one can realize that the new refinement language, not only allows modular verification, but, by exploiting the imperative style of its host languages, the designer is presented with a scripting language that allows control and provides documentation of the verification process.

Figure 5.3: Case study for refinement.

5.4 Using the New Language

Using two examples, this section gives a brief overview about how the new refinement language is used. The first example verifies the correctness of a floating-point (FP) multiplier with respect to two simple mathematical laws, given that the FP inputs are of the correct format. The second example illustrates how a `DualCircuit` component is used to verify a squarer.

Example 1: Verifying the Implementation of a FP Multiplier

Any multiplier abides with the following mathematical laws:

1. $1 \times n = n$ (Identity law)
2. $0 \times n = 0$

We will verify whether the implementation of a FP multiplier is correct in respect to these laws.

We will assume that the FP numbers given to the circuit are always well-formed. A well-formed FP number is one that is neither *denormalized* nor *overflown*. A denormalized number is one where the number is too small to be represented by a normalized number and therefore the mantissa part of the FP number does not have a 1 in its most significant bit (MSB). An overflow occurs when all of the exponent bits are 1s [10, 71, 80].

Given that well-formed numbers are described by these two properties, we will use them as constraints during the verification of the multiplier. Therefore, we consider these as the environment specification of our example (cf. Figure 5.4).

This environment constraint is implemented using a class extending the `Environment` class. Its logical definition is defined in the class by overriding the method `PropertyDefinition()`. This method is imposed by the `Observer` class from which class `Environment` derives:

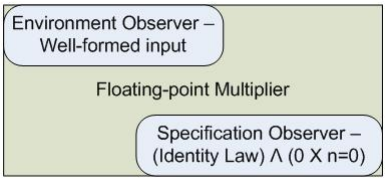

Figure 5.4: The floating-point multiplier is correct if, given well-formed numbers, it abides with the specified laws.

```
public class FloatFormat : Environment
{
    public FloatFormat(Circuit parent): base (parent) {}

    protected override void PropertyDefinition(Wire[] input, Wire[] output,
                                               ref LogicWire alarm)
    {
        //...Logical definition of the environment...
    }
}
```

The definitions of the two specifications against which the multiplier will be verified are defined in a class extending `Specification`.

```
public class Zeros : Specification
{
    public Zeros(Circuit parent):base(parent){}

    protected override void PropertyDefinition(Wire[] input, Wire[] output,
                                               ref LogicWire alarm)
    {
        //...Logical definition
    }
}

public class Identity : Specification
{
    public Identity(Circuit parent):base(parent){}

    protected override void PropertyDefinition(Wire[] input, Wire[] output,
                                               ref LogicWire alarm)
    {
        //...Logical definition
    }
}
```

Having the specifications and environment constraints defined, we build a circuit that extends class `RefinementCircuit`, having instances of the specification classes, the environment class and the multiplier itself:

```
public class MultiplierVerification : RefinementCircuit {
    public MultiplierVerification():base(null){}

    public void gate()
    {
        Float float1 = new Float(4,4);
        Float float2 = new Float(4,4);

        //Create an instance of the multiplier and build it
        FloatingPointMultiplier fmult1 = new FloatingPointMultiplier(this);
        Float ans1 = fmult1.gate(float1, float2);

        Wire [] inputs = {float1, float2};
        Wire [] outputs = {ans1};

        //Specifications
        LogicWire zeroProperty = new Zeros(this).gate(inputs, outputs);
        LogicWire identity = new Identity(this).gate(inputs, outputs);
        LogicWire [] properties = {zeroProperty, identity};

        //Environment Assumptions
        new FloatFormat(this).gate(inputs, outputs);
    }
}
```

Using list `properties` as input parameter to the method that generates circuit descriptions for verification, we generate a description with which we can verify that, given the inputs are well-formed, the two mathematical laws are obeyed. In more technical terms, we verify that the conjunction of the output alarms from the two specification observers is true for any input, given that the output alarm of the environment specification observer is also true:

$$\begin{aligned} \mathit{specification} &= \mathit{alarm}_{\mathit{zero\_law}} \cap \mathit{alarm}_{\mathit{identity}} \\ \mathit{environment} &= \mathit{alarm}_{\mathit{float\_format}} \\ \mathit{specification}_{\mathit{to\_prove}} &= G\,(\,\mathit{environment} \Rightarrow \mathit{specification}\,) \end{aligned}$$

Example 2: Using `DualCircuit` for Verification

An integer squarer is a component that takes an integer as input and uses a multiplier to multiply the input by itself [40, 56]. We want to verify that if the input to the squarer is 1 then its output is also 1.

There are two ways a squarer can be verified — the most straightforward way is to verify the specification over a squarer that uses a fully-implemented multiplier. This produces a very complex circuit which may be difficult to

verify especially when using automatic verification techniques. However, we can use another approach. By assuming that the multiplier abides with the identity law, we can simplify the circuit, thus making it possible to verify successfully (cf. Figure 5.5). Later, the multiplier can be refined to a proper implementation, which implementation can be verified to hold the identity law.

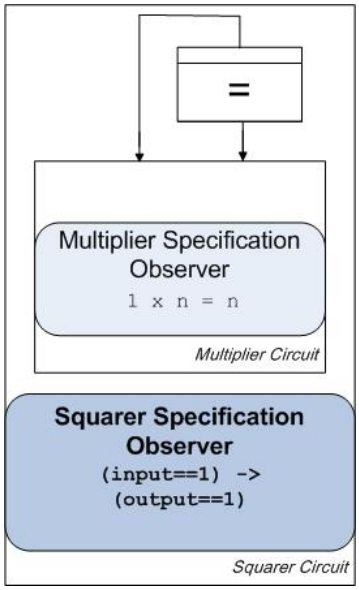

Figure 5.5: An integer squarer can be verified to produce 1 given 1 as input, assuming that the multiplier abides with the identity law.

Therefore we define a multiplier `Multiplier` using the `DualCircuit` class such that both its identity-law specification observer and implementation can be defined. The specification is defined by extending `Specification` class:

```
//Multiplier specification: 1*n = n
public class MultiplierProperty : Specification
{

    public MultiplierProperty()
    {
        this.Type = Specification.SpecType.ASSUME;
    }

    protected override void PropertyDefinition(Wire[] input, Wire[] output,
                                                ref LogicWire alarm)
    {
        //...Property Logic
    }
}

//Multiplier circuit, using MultiplierProperty as specification
public class Multiplier : DualCircuit
{
    public Multiplier(Circuit parent):base(parent, new MultiplierProperty()){}

    public Integer gate(Integer multiplicand, Integer multiplier)
    {
        //...calls gate_o()
    }

    public void gate_o(Integer multiplicand, Integer multiplier, ref Integer answer)
    {
        //..Undefined structure
    }
}
```

The squarer, represented by the class `Squarer`, uses an instance of class `Multiplier` to multiply the input by itself:

```
//Squarer circuit
public class Squarer : Circuit
{
    public Squarer(Circuit parent):base(parent){}

    public Integer gate(Integer in0)
    {
        Integer out0 = new Integer(in0.Size);
        gate_o(in0, ref out0);
        return out0;
    }
```

```
    public void gate_o(Integer in0, ref Integer out0)
    {
        Wire [] input = {in0};
        Wire [] output = {out0};
        Connect(input, output);

        //Create a new integer which is equal to the input
        Integer in1 = (Integer)new AssignmentMap(this).gate(in0);

        //Create an instance of multiplier to multiply the input by itself
        Multiplier mult = new Multiplier(this);
        mult.gate_o(in0, in1, ref out0);
    }
}
```

The specification observer of the squarer we want to prove is defined in the class `SquarerProperty` which inherits from `Specification`:

```
//Squarer specification: (input==1) -> (output==1)
public class SquarerProperty : Specification
{
    public SquarerProperty(Circuit parent):base(parent)
    {
        this.Type = Specification.SpecType.PROVE;
    }

    protected override void PropertyDefinition(Wire[] input, Wire[] output,
                                               ref LogicWire alarm)
    {
        //...Property Logic
    }
}
```

To carry out the verification process, we build a circuit which calls the squarer and its specification. Targeting the SMV model checker, we generate a description for verification. The algorithm detects the multiplier's use of its specification and therefore generates a description that verifies that the squarer obeys its specification assuming that the multiplier abides by the identity law:

```
    //Input
    Integer in0 = new Integer(4);

    //Squarer
    Squarer squarer = new Squarer(this);
    Integer out0 = squarer.gate(in0);

    //Specification
    Wire [] input = {in0};
    Wire [] output = {out0};
    SquarerProperty specification = new SquarerProperty(this);
    LogicWire mainAlarm = specification.gate(input, output);

    LogicWire [] prove = {mainAlarm};

    //Verify that squarer is correct assuming multiplier law
    VerificationReport report = this.ToSMV(prove, "C:\\", "Squarer");
```

We then define a proper implementation for the multiplier. Using method `UnresolvedAssumptionsList()`, we get the list of assumptions used in the last verification process to access the multiplier object. We call the multiplier's method `Refine()`, to refine it to the implementation defined in class `MultiplierImplementation`: besides defining the multiplier's circuit, this class also implements the `IImplementation` interface that allows it to be used as the `Implementation` component of a `DualCircuit`:

```
public class MultiplierImplementation : IImplementation
{
    public MultiplierImplementation(Circuit parent):base(parent){}

    public Wire[] gate(Wire[] inputs)
    {
        //...Implementation definition
    }

    public void gate_o(Wire[] inputs, ref Wire[] outputs)
    {
        //...Implementation definition
    }
}
```

On regenerating the whole circuit's description, SharpHDL detects the new refinement and, therefore, the description produced is such that allows verification of the multiplier implementation against its specification and not of the whole squarer circuit:

Size	Model Checking Timings		
	User Time	System Time	BDD nodes allocated
1	0.0701	0.03	6
2	0.0801	0.04	15
4	0.1902	0.06	105
8	0.66095	0.08	2555
16	20.199	0.21	38567

Table 5.1: Timing results for verifying an integer squarer using a full multiplier implementation.

```
//Get the Multiplier whose property was assumed
ArrayList assumptions = this.UnresolvedAssumptionsList();

DualCircuit unknown = (DualCircuit)assumptions[0];

//Refine it to a proper multiplier implementation
MultiplierImplementation mult = new MultiplierImplementation(null);
unknown.Refine(mult);

VerificationReport report = this.ToSMV(prove, "C:\\", "Squarer");
```

5.5 The Advantages of Modular Verification

Whilst being able to automatically verify a large range of properties, automatic verification techniques are very inefficient when verifying complex circuits [41, 63, 68], often due to the large state space that needs to be traversed. The number of states for multipliers, for example, can grow exponentially in the number of variables [14]. Various attempts were made to tackle this *state explosion problem*. These attempts range from improving algorithms and developing heuristics [15, 16, 57] to applying modular verification techniques to decompose a verification problem to smaller tasks [2, 23, 32, 41, 62, 63].

We have seen how modular verification enables the decomposition of a system. Using the squarer defined in section 5.4, we conducted an experiment whereby we could compare times for building and verifying a squarer using a full multiplier implementation with the times of building and verifying a squarer assuming the identity law over the multiplier.

Table 5.1 gives the timing measures for verifying the squarer with a multiplier

Size	Model Checking Timings		
	User Time	System Time	BDD nodes allocated
1	0.06	0.02	13
2	0.06	0.02	32
4	0.06	0.03	76
8	0.09	0.03	148
16	0.1	0.06	314
32	0.23	0.04	664
64	0.51	0.04	1398
128	1.1416	0.11	10192
256	3.44	0.09	12730
512	12.67	0.35	25626
1024	50.69	0.69	51898

Table 5.2: Timing results for verifying an integer squarer assuming that the multiplier obeys the identity law.

completely defined. It is important to point out that a 32, 64 and 128-bit squarer circuits were also generated but the verification process was still running after one hour when it was stopped. On the other hand, verifying all the eleven circuits using the multiplier assumption was much more efficient as shown in Table 5.2.

Therefore, modular verification techniques make it possible to verify circuits having components that are difficult to verify because of the exponential growth of their state graph representation. Nevertheless, one should point out that for successful and correct verification the specifications of the complex circuits should be carefully chosen as not all properties are relevant to verification being carried out. In our squarer case, for example, using the commutative law (i.e. $a \odot b = b \odot a$) is irrelevant to the property we are trying to prove on the squarer and verification will give a negative result. Also, the specifications should not produce bigger state graphs than the implementation itself.

5.6 Related Work

In this section we will discuss how other HDLs and formal verification tools provide modular verification and refinement.

A major influential work to our research was that conducted by Halbwachs et al. based on designing and verifying reactive systems using the synchronous dataflow language Lustre [35, 37, 38]. Using this language, the state-explosion problem is tackled by considering only states concerning only the properties to be proven together with a set of *assertions* during verification. Assertions are a means to describe environment constraints in Lustre.

CSML is one language that explicitly extended SML to provide modularity and reusability features for modular verification. SML was developed to specify complicated finite state machines such that SML programs represented complete synchronous circuits. CSML (Compositional SML) [23] provides two basic extensions to allow modularity and reusability: *process* and *processtype*. These constructs both describe a construct which represents a Moore machine. The difference between them is that the latter is a process which can be reused. The abstraction of a process is done by manually applying the *interface rule* to one of the processes, which abstraction the user thinks will reduce the state space most. The interface rule is a theorem which reduces the state of the model such that the model checking program verifies the CTL formula with respect to a reduced model. Therefore, a user has to choose which process to abstract after compilation and cannot define an abstract representation of a class of modules at the design level. This is largely due to the fact that, unlike SharpHDL circuits, the proposed constructs do not allow non-determinism.

Alur's and Henzinger's notion of *reactive module* [2] are a formal model of concurrent systems. They allow modular description of a system having both synchronous and asynchronous models and also supports non-determinism and temporal abstractions which allow the description of high-level or incomplete designs. They attempt to make use of both assume-guarantee concepts and refinement to verify large problems [41]. The refinement checking problem is approached by introducing the concept of *abstraction* and *witness* modules. Basically, these are reactive modules that observe the external and interface variables of other modules. *Abstraction modules* are used to describe environment constraints on aspects of a subcomponent that are relevant to the rest of the system. Therefore they act as an intermediate layer between the implementation and the specification. *Witness modules* are used when not all variables of a specification are present in an implementation, i.e. they make explicit how the private state of the specification depends on the state of the implementation. Though similar to the SharpHDL framework,

reactive modules do not provide the user with any direct control of the refinement process. Nevertheless, the implementation of this framework in the toolkit Mocha [1] produced an efficient, user-friendly tool which supports a range of compositional and hierarchical verification methodologies.

An efficient construct for compositional verification is presented by McMillan in [62]. He presents a system based on a generalized compositional rule which allows independent and efficient verification of refinements in an abstract environment. This system supports downward refinement maps and non-hierarchical abstraction. The latter allows an abstract specification to have a different structure from the implementation. He introduces *layers* where each layer represents an incremental change from a specification to the design. Therefore a user can verify each implementation update individually against the abstract representation. Incremental changes cannot be explicitly done in SharpHDL. The `layer` structure provides a way for managing *refinement maps* such that various implementations can be verified against an abstract interface. This layer structure was implemented over the symbolic model checker SMV [65].

5.7 Further Work and Conclusions

In this chapter we have discussed the approach we used to provide for modular verification and refinement. We have presented a new refinement language which provides various construct definitions which allow this functionality. By embedding the language into SharpHDL, we provided a meta-language with which a designer can have access to both the circuit definitions and the refinement process.

The major construct definition is one which allows a circuit to be defined and used in either of two ways: either through its implementation or through its specification, defined by an observer. The user can toggle between the two choices as needed. The implementation can be left unspecified and then refined later during the verification process. This is possible since SharpHDL accepts non-deterministic circuit, i.e. circuits whose structure is not known. Using this approach we are preparing to build a complex case study where we can verify the equivalence of larger FFT circuits. The large state space generated by the model checker made it impossible for us to verify FFT circuits of size larger than 4. We intend to apply modular verification to

sub-divide this complex problem and thus being able to verify larger FFT circuits.

However, our approach does not stop here. The embedding approach allows us to exploit the imperative nature of the host language C#. This provides us with a Turing-powerful scripting language that gives a designer full control and allows step-by-step documentation of the verification process. Though based on sound and efficient solutions to the discussed verification techniques, the existing languages only provide adhoc means to document the history of the verification process.

Chapter 6

Discussion, Further Work and Conclusions

In this chapter we first discuss how HDLs can benefit from the object-oriented paradigm. We then outline our future plans based on the research conducted and presented in this dissertation. In the end we highlight the overall conclusions of this work.

6.1 Discussion — OO Features in Hardware Description Languages

We have defined, implemented and worked with a HDL embedded in an object-oriented language. As pointed out earlier on in this document, the object-oriented paradigm offers a number of characteristics and benefits which are very popular in today's programming environments. Frequently in this dissertation, we have pointed out the benefits of some of these features when applied to SharpHDL. In this section, we shall highlight again these main OO features and discuss whether such a paradigm is suitable to the hardware-description area.

The basic idea of the OO ideaology is to think of a situation in terms of objects which are made up (or composed) of other simpler objects. In technical terms, this concept is called ***Composition***. We have treated circuits as complex objects made up of several smaller and simpler circuits and other components. In doing so, information about each circuit could be kept and

manipulated. By information we specifically mean the list of different components making up the circuit. The advantage of this characteristic can be seen if the two SharpHDL versions are compared. Using the first version, the designer could define an object for each circuit and reuse the definitions as much as needed. However, the underlying structure stored information in one common repository rather than having each circuit hold its own information. This made it impossible to manipulate individual circuits during run-time. This was amended in SharpHDL2, which gave the composition characteristic more importance. It is now possible for a designer to refer to a needed circuit and have access to its structure which can be manipulated without effecting the structure of other circuits.

OOP allows the organization of objects into tree structures, such that the root holds the attributes and behaviours common to all descendants. In other words, it provides ***Inheritance***. The internal class structure of SharpHDL is based on a tree structure where the top-most class `Nameable` assigns a unique name to all the different components making up a circuit, i.e. ports, wires and circuits. Figure 6.1 displays how the main classes of SharpHDL inherit from each other. For example class `And` inherits the properties and operations of class `Nameable` which provides a naming mechanism, `Cell` which defines the interface of a circuit block, `Circuit` which holds the repository of sub-circuits making up the circuit, `Logic` which abstracts the general definition of a circuit and `BooleanLogic` which abstracts primitive gates.

Applying inheritance correctly provides the possibility to extend the language with new types of circuits, wires or ports without the need to define them from scratch. We have seen how libraries of classes were easily added to the set of SharpHDL libraries by inheriting from classes defined in the main library and other libraries.

Another important OOP concept is ***Polymorphism*** which enables a programmer to treat a collection of classes derived from the same root class in the same way. This concept was used in the implementation of generic circuits. Generic circuits accept a circuit with which they build a regular-structured circuit. The input circuit should inherit from class `Circuit` and optionally implement operations declared in the latter class. The generic circuits treat the input circuit as a `Circuit` object although the operations invoked are those implemented by the circuit class itself.

OOP also provides the designer the possibility to separate the interface from

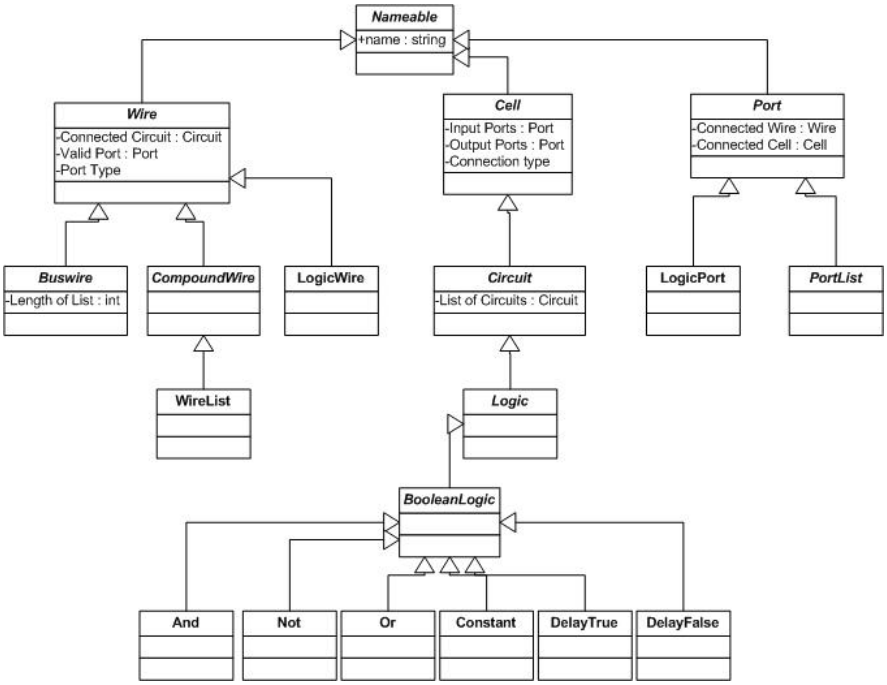

Figure 6.1: Class diagram of the main SharpHDL library. This diagram shows how the different classes relate to one other.

an object from its actual implementation. This ***Separation*** idea was greatly applied in the development of the `DualCirucit` structure. The latter provides the means to define a specification, or interface, and later refine it to the actual internal implementation. Besides OOP languages also provide an `Interface` mechanism where a set of functionalities that should be implemented by classes are defined. This mechanism was used to cater for future extensions, including possibilities of creating external output to other verification tools. Classes which provide processing of this kind have to implement the set of functions defined by the interface `IVerificationTool`. `Interfaces` were also used to define the functions for circuits which could be used as input to specific generic circuits.

Having defined a set of classes we also defined a set of hardware ***abstract data types*** and therefore a designer can conveniently work with packages of hardware components operations and attributes to hold and manipulate their state.

Having pointed out the various benefits gained from OOP we can say that ap-

plying this paradigm to hardware description languages makes perfect sense.

6.2 Further Enhancements

New ideas and areas of improvement are always coming up. Following are the main points which we plan to pursue.

6.2.1 Case study: verifying large FFT circuits

One of the case studies developed using SharpHDL and discussed in this dissertation is the definition of two circuits describing FFT algorithms. The equivalence of these descriptions was then verified using the model checker SMV. This tool is based upon building and traversing binary decision diagrams (BDDs) which represent the possible states and outcome of a circuit. Though very efficient, BDDs get very complex on encountering multipliers. Possibly due to this reason, we did not succeed to verify circuits that take more than four complex numbers as inputs.

Later on we also discussed how SharpHDL was extended to provide modular verification. Modular verification techniques allow a verification problem to be decomposed to smaller problems which can be verified individually. We plan to use this technique to verify larger FFT circuits. We intend to abstract the multipliers to a higher-level by using behavioural specifications instead of their actual implementation. We have already used this approach to verify a squarer circuit (refer to section 5.4).

One difficulty we might encounter is to find the right set of specifications that are relevant to the FFT circuits and which will enable us to reduce the state space of the circuits. In [8], Bjesse verifies the equivalence of FFT circuits implemented at arithmetic level, using some simple algebraic laws and axioms as input to the first-order logic theorem prover. These laws include:

- distributivity of multiplication over addition —

$$a \times (b + c) = a \times b + a \times c$$

- identity law —

$$1 \times n = n \times 1 = n$$

We consider the use of these mathematical laws as behavioural specifications of the multipliers used in our FFTs.

6.2.2 Placement extensions and description of other non-functional circuit properties

One disadvantage of FPGAs is the limited space available on a single chip and therefore placement consideration is very important. How resources are used in a circuit has an effect on all the circuits that make use of it. Placement control is also important for *reconfigurable circuits*[1] since correct placing can minimize reconfiguration time [60].

Although automatic placing tools exist, user-specified placement information sometimes produces more efficient placing. This is especially evident when circuits use regular structures since conventional algorithms do not exploit this characteristic to achieve a better implementation [81]. Nevertheless, providing explicit coordinate information for every component can be very tedious, cumbersome and error-prone. Defining *relative-placement information* which is then translated to explicit coordinates provides a higher-level means to specify placement, thus simplifying code and giving greater support to the development of run-time reconfiguration. This technique is implemented in various HDLs, all defining a different solution of how a user can do this.

JHDL, a Java-embedded HDL, implements two placement methods:

- `map()` which maps gates to atomic FPGA cells and
- `place()` through which relative placement of atomic cells is set.

The underlying interpretation is done by a special class called `Techmapper`, which checks the validity of the network, resolves all placement directives and reports any conflicts. This is done depending on the target platform [45].

Lava, a Haskell-embedded HDL, uses a set of combinators to define both the behaviour and the layout of circuits. Such include the *serial composition infix combinator* `>->` which, besides connecting the output of the circuit on its left

[1] **Reconfigurable systems** are systems having programmable logic that allows changes of the hardware circuits [30].

to the input of the circuit on its right, it also specifies that the circuits are to be laid out horizontally, from left to right. Other similar combinators are the *right-to-left serial composition* `<-<`, the *top-to-bottom serial composition* \/ and the *bottom-to-top serial composition* /\ operators [21]. Connection patterns are also used as a means to get layout information [9]. For example, the `row` pattern does not only create a number of the same connected circuits, but it also lays them out horizontally from left to right.

The same approach is adopted in Ruby [50, 82], a HDL based on defining relations. It allows the definition of both the behaviour and the topology of circuits. A Ruby component is considered to be a *tile* which can have ports on all four edges. The language provides various combinators to specify where the circuits are located in relation to other circuits and to external signals (c.f. table 6.1):

- `<->` is the *beside* combinator, such that `A<->B` means that `B` is horizontally beside `A`.
- `b` is the *below* combinator, such that `A b B` means that `A` is vertically below `B`

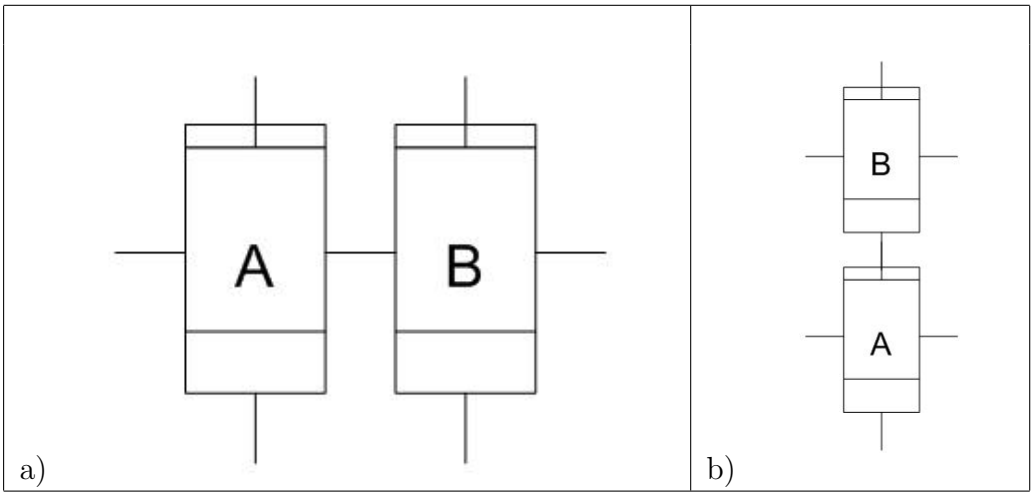

Table 6.1: Ruby placement combinators: a)`A<->B` b) `A b B`

Like Lava, it also uses generic descriptions to define placement. Examples include `row` — which replicates a cell horizontally; `col` — which replicates a cell vertically and other descriptions.

Yet another language that provides for both explicit and implicit placement definitions is Pebble [55]. The two main descriptions are `BESIDE`, which places two or more blocks beside each other, and `BELOW` which places blocks vertically.

To map the descriptions defined using these languages into hardware, VHDL or EDIF descriptions with explicit coordinates are generated. One difference between the Pebble and the rest of the discussed languages is that the former produces parametric VHDL descriptions. The others produce a flattened description [60, 61].

Given the importance of providing placement information and examined how various languages provide it, we consider the implementation of relative placement specifications in SharpHDL. Since our language already has a number of generic circuits and is able to translate descriptions to external tools we can adopt the approaches discussed in [9, 55, 82]. The code-producing algorithms can be extended to recognize generic circuits and generate code with explicit coordinates depending on the generic circuit used. Besides, one should point out that SharpHDL already produces parametric translations and therefore it can easily be extended to produce them with symbolic placement as proposed in [60].

Another extension worth considering is the provision of calculating other non-functional properties of a circuit like area, timing and power. Most of these properties are controlled by information found in wires. In [6], Axelsson et al. describe the Wired language which allows low-level control of wires. Again, the importance of combinators and generic circuits is highlighted since such have both functional and geometric interpretations. Wires are treated as circuits with explicit size and a distinction is made between circuits with or without geometry. Ports are also given their due importance. Given more thought, we think that extending the three basic classes — `Wire`, `Port` and `Circuit` — with properties that define non-functional features together with the generic circuits, the Wired approach is implementable in SharpHDL.

6.2.3 Using polymorphism to enhance generic circuit implementations

One of SharpHDL's features is its support for higher-order circuits. A set of such circuits are implemented in the `GenericCircuits` library. Two ex-

amples we have frequently used are the `Map` and the `Row` generic circuits — `Map` applies a given component to each element of an input list of wires (cf. Figure 6.2); `Row` uses two sources of input to a given component: the ith wire from an input list and the output from the $(i-1)^{th}$ component-application. Therefore, each component-application produces two outputs: one is saved in an output list of wires whilst the other is passed on as input to the $(i+1)^{th}$ wire of the input list (cf. Figure 6.3).

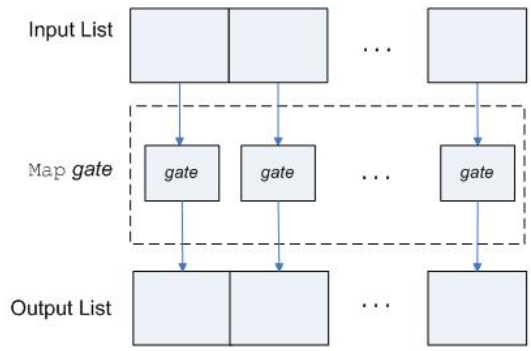

Figure 6.2: `Map gate`.

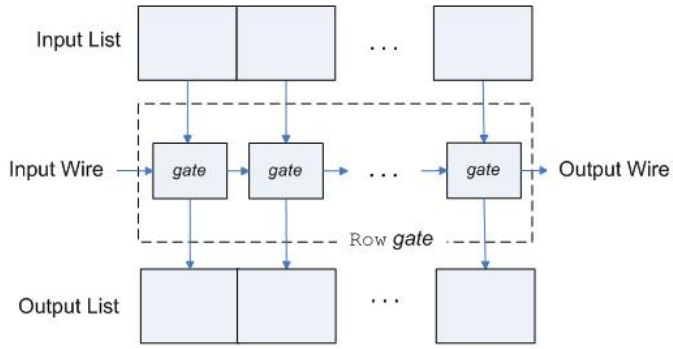

Figure 6.3: `Row gate`.

Presently there are three implemented variations for each of these generic circuits. For the `Map` generic circuit there are:

1. `MapOne` accepts one input list of wires;
2. `MapTwo` accepts two input lists of wires gate, and
3. `MapThree` accepts three input lists of wires.

Consequently, they accept a one-input component, a two-input component and a three-input component respectively.

The same approach is taken for the `Row` generic circuit:

1. `RowOne` accepts one input list of wires;
2. `RowTwo` accepts two input lists of wires gate, and
3. `RowThree` accepts three input lists of wires.

Consequently, they accept a two-input component, a three-input component and a four-input component respectively.

One enhancement to this approach is to use the polymorphism concept and implement only one type of circuit for each generic circuit. We know that SharpHDL defines class `BusWire` which represents a wire made up of a list of wires of the same type. Therefore, the generic circuit will accept an array of `BusWire`s as input and, given n `BusWire`s in the list, accepts a component that takes n inputs.

To enable this, an interface should be specified for the components that can be used by the generic circuits. The interface should specify that the methods that define the component's structure should accept all the inputs in an array of `Wire`s. The array will accept any type which extends the abstract class `Wire`. So, for example, the interface that allows a circuit to be used by `Map` will look as follows:

```
public interface IMapStructurable
{
    Wire gate(BusWire [] inputs);
    void gate_o(BusWire [] inputs, Wire output);
}
```

On the other hand, a `Row`-implementable circuit will implement the following interface:

```
public interface IRowStructurable
{
    void gate_o(BusWire [] inputs, LogicWire inputWire,
                out Wire out0, ref LogicWire outputWire);
    LogicWire gate(BusWire [] inputs, LogicWire inputWire, out Wire out0);
}
```

Using interfaces and polymorphism will therefore allow us to provide better `Map` and `Row` implementations that are more generic.

6.2.4 External tools facilities

SharpHDL allows the generation of circuit descriptions for verification and analyzes. However, the language does not provide a direct connection to the respective tools. This requires the user to manually open the generated descriptions in the correct application. It is ideal that SharpHDL automatically connects to the tool, feeds the generated descriptions and returns the result to the user. This way, the user sees only one language rather than having to manually control several of them.

SharpHDL allows generation of descriptions to other HDLs. The older version of SharpHDL generated descriptions that were not modular, thus creating very large files for complex circuits that at times were impossible to load. The intention was to implement an algorithm that produced modular definitions but due to the extra emphasize on verification and the time constraints this was not possible. We intend to do this in the coming future.

6.3 Conclusions

Throughout this dissertation we have used the HDL SharpHDL to investigate the contributions offered by the characteristics it is based upon.

In the beginning we have discussed the implementation of a new SharpHDL version. Though a sound language in its own right, the first version failed to fully exploit its modularity, inheritance and polymorphism features. Such features come free of charge from embedding the language in the object-oriented host language C#. Using the modularity feature, the new SharpHDL does not flatten the descriptions but instead keeps a hierarchical definition of them, where a circuit object is made up of a number of sub-circuit

and itself is a sub-circuit to a parent circuit. Modularity also enables the generation of modular descriptions for verification or other HDLs, such that descriptions are shorter and easier to understand. The language-embedding approach also provides a meta-language which allows the definition of higher-order circuits, which we called generic circuits.

Using the language, we can verify circuit properties. We do this by defining an observer circuit that takes the inputs and outputs of a circuit-under-observation and checks whether, for a given input, the latter abides by a given property. By generating a description of the circuit connected to the property observer we can verify that the circuit abides by the property for any input. Using this tool we are able to define and verify the equivalence of two circuits describing different FFT algorithms - the radix-2 FFT algorithm and the radix-2^2 FFT algorithm. However, though successful for size 4 FFTs, verification was not possible for more complex circuits.

We once again turn to the language-embedding approach and build a simple refinement language over SharpHDL. This language allows us to use modular verification and refinement techniques which permit the decomposition of a complex verification problem to smaller problems which can be verified individually. So using this language, we can verify circuits by abstracting complex sub-components to simpler versions defined by behavioural specifications and use them as assumptions in the verification of other parts of the system. Therefore, by embedding the refinement language in SharpHDL, we provide a means to define and verify large circuits using the same language. Moreover, by exploiting the imperative nature of SharpHDL we provided the designer means to specify and document the refinement process. To date, this was not possible in existing HDLs.

Using these tools, future intentions include the verification of equivalence of large FFT circuits which could be carried out if the large number of multipliers used in the circuits are simplified using behavioural specifications describing simple arithmetic laws.

Bibliography

[1] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani and Serdar Tasiran. "MOCHA: Modularity in Model Checking." In *Computer Aided Verification,* pages 521–525, 1998.

[2] Rajeev Alur and Thomas A. Henzinger. "Reactive Modules." *Formal Methods in System Design: An International Journal,* volume 15, number 1, pages 7–48, July 1999. Kluwer Academic Publishers.

[3] Behnam Amelifard, Farzan Fallah and Massoud Pedram. "Closing the Gap between Carry Select Adder and Ripple Carry Adder: A New Class of Low-Power High-Performance Adders." In *Proceedings of the 6th International Symposium on Quality of Electronic Design (ISQED 2005),* pages 148–152, 2005. IEEE Computer Society.

[4] Henrik Reif Andersen. "An Introduction to Binary Decision Diagrams." Lecture Notes for 49285 Advanced Algorithms E97, Department of Information Technology, Technical University of Denmark. April 1998.

[5] Peter J. Ashender."The VHDL Cookbook." Lecture Notes, Dept. Computer Science, University of Adelaide, Australia. First edition. 1990.

[6] Emil Axelsson, Koen Claessen and Mary Sheeran. "Wired — a Language for Describing Non-Functional Properties of Digital Circuits." In *Proceedings Int. Workshop on Designing Correct Circuits,* a satellite event of ETAPS 2004, Barcelona, March 2004.

[7] Dag Björklund and Henrik Enqvist. "Parallel Radix-4 DIF FFT Hardware — Implementation using VHDL." 24th October 2002.

[8] Per Bjesse. "Automatic Verification of Combinational and Pipelined FFT circuits." In *Computer Aided Verification,* July 1999. Springer, Verlag.

[9] Per Bjesse, Keon Claessen, Mary Sheeran and Satnam Singh. "Lava: Hardware Design in Haskell." In *Proceedings of the 3rd International Conference on Functional Programming,* 1998. ACM SigPlan.

[10] Pavle Belanovic and Miriam Leeser. "A Library of Parameterized Floating-Point Modules and Their Use." In *Field-Programmable Logic and Applications, FPL 2002,* Montpellier, France, September 2–4, 2002. Lecture Notes in Computer Science, volume 2438/2002. Springer Berlin/Heidelberg.

[11] G. Berry. "The Esterel v5 Languge Primer." Centre de Mathématiques Appliquées. Technical Report, version 5.21, release 2. April 1999.

[12] Patrice Bertin and Herve Touati. "PAM Programming Environments: Practice and Experience." In *IEEE Workshop on FPGAs for Custom Computing Machines,* pages 133-138, 1994. IEEE Computer Society Press.

[13] L. Bostock and S. Chandler. "Mathematics — The Core Course for A-level." 1st edition, 1981. ISBN 0-85950-306-2. Stanley Thornes (Publishers) Ltd.

[14] Randal E. Bryant. "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification." In *IEEE/ACM International Conference on Computer Aided Design, ICCAD,* pages 236–243, 1995. IEEE CS Press.

[15] Jerry R. Burch and David L. Dill. "Automatic Verification of Pipelined Microporcessor Control." In *Computer Aided Verification.* LNCS, volume 818, pages 68–79, 1994. Springer-Verlag.

[16] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. "Symbolic Model Checking: 10^{20} states and beyond." In *Proceedings of Fifth Annual IEEE Symposium on Logic in Computer Science,* Philadelphia, pages 428–39. June 1990.

[17] Koen Claessen. "Embedded Languages for Describing and Verifying Hardware." Ph.D thesis, Department of Computing Science, Chalmers University of Technology and Göteborg University, Sweden, April 2001.

[18] Koen Claessen and Gordon Pace. "An Embedded Language Approach to Teaching Hardware Compilation." In *Functional and Declarative Programming in Education, FDPE 2002,* 2002.

[19] Koen Claessen and Gordon Pace. "An Embedded Language Framework for Hardware Compilation." In *European Research Consortium in Informatics and Mathematics,* Grenoble, France, 2002.

[20] Koen Claessen and Mary Sheeran. "A Tutorial on Lava: A Hardware Description and Verification System." Available from `www.cs.chalmers.se/ keon/Lava`. August 2000.

[21] Koen Claessen, Mary Sheeran and Satnam Singh. "The Design and Verification of a Sorter Core." In *Correct Hardware Design and Verification Methods, CHARME.* LNCS, volume 2144, pages 355–369, 2001. Springer.

[22] E.Clarke, O. Grumberg and D. Long. "Verification Tools for Finite State Concurrent Systems." In *A Decade of Concurrency — Reflections and Perspective,* volume 803, pages 124–175, 1993. Springer-Verlag.

[23] E. M. Clarke, D. E. Long and K. L. McMillan. "A language for compositional specification and verification of finite state hardware controllers." In *Proceedings of the IEEE,* volume 79, number 9, pages 1283–92, September 1991.

[24] Byron Cook, John Launchbury and John Matthews. "Specifying superscalar microprocessors in Hawk." In *1998 Workshop on Formal Techniques for Hardware and Hardware-like Systems,* Marstrand, Sweden, 1998.

[25] Communication and Multimedia Laboratory. "Fast Fourier Transform (FFT)." Department of Computer Science and Information Engineering, NTU. `www.cmlab.csie.ntu.edu.tw.` Last viewed on January 11, 2005.

[26] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest. "Introduction to Algorithms." McGraw-Hill Book Company, 24th edition, 2000. ISBN 0-262-53091-0. The MIT Press.

[27] Stephen A. Edwards. "Design and Verification Languages." Columbia University, Computer Science Technical Report CUCS-046-04. November 2004.

[28] Conal Elliott. "Modeling interactive 3D and multimedia animation with an embedded language." In *First Conference on Domain-Specic Languages,* pages 285–296, October 1997.

[29] Conal Elliott, Sigbjorn Finne and Oege de Moor. "Compiling Embedded Languages." In *Semantics, Applications and Implementation of Program Generation workshop (SAIG 2000),* pages 9–27, 2000. Springer-Verlag.

[30] Rolf Enzler. "The current status of reconfigurable computing." Technical Report, Swiss Federal Institute of Technology (ETH) Zurich, Electronics Laboratory, 1999. `www.e-collection.ethbib.ethz.ch/show?type=bericht&nr=370`. Last viewed on March 29, 2006.

[31] Jeff Ferguson, Brian Patterson, Jason Beres, Pierre Boutiquin and Meeta Gupta. "C# Bible." 2002. ISBN 0-7645-4834-4. Wiley Publishing Inc.

[32] Orna Grumberg and David E. Long. "Model Checking and Modular Verification." In *ACM Transactions on Programming Languages and Systems,* volume 16, number 3, pages 843–871, May 1994. ACM Press.

[33] Ruben A. Gamboa. "Mechanically Verifying the Correctness of the Fast Fourier Transform in ACL2." In *Third International Workshop on Formal Methods for Parallel Programming: Theory and Applications,* 1998.

[34] N. Halbwachs. "About synchronous programming and abstract interpretation." In *International Symposium on Static Analysis, SAS'94,* Namur (Belgium). LNCS, volume 864, September 1994. Springer-Verlag.

[35] N. Halbwachs and F. Lagnier and C. Ratel. "Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre." In *IEEE Transactions on Software Engineering.*

Special issue on the Specification and Analysis of Real-Time Systems, September 1992.

[36] N. Halbwachs and F. Lagnier and P. Raymond. "Synchronous observers and the verification of reactive systems." In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93.* Workshops in Computing, June 1993. Springer-Verlag.

[37] N. Halbwachs. "Synchronous programming of reactive systems, a tutorial and commented bibliography." In *Tenth International Conference on Computer-Aided Verification, CAV'98,* Vancouver (B.C.). LNCS, volume 1427, June 1998. Springer-Verlag.

[38] N. Halbwachs and P. Caspi and P. Raymond and D. Pilaud. "The synchronous dataflow programming language Lustre." In *Proceedings of the IEEE,* volume 79, number 9, pages 1305–1320, September 1991.

[39] N. Halbwachs and P. Raymond. "Validation of Synchronous Reactive Systems: from Formal Verification to Automatic Testing." In *Asian Computing Science Conference, ASIAN'99,* Phuket (Thailand). LNCS, volume 1742, December 1999. Springer-Verlag.

[40] V. C. Hamacher, Z. G. Vranesic, S. G. Zaky. "Computer Organization." Computer Science Series, 4th Edition, 1996. ISBN 0-07-114323-8. McGraw-Hill International Editions,

[41] Thomas A. Henzinger, Shaz Qadeer and Sriram K. Rajamani. "You Assume, We Guarantee: Methodology and Case Studies." In *Proceedings of the 10th International Conference on Computer-aided Verification (CAV).* LNCS, volume 1427, pages 440–421, 1998. Springer-Verlag.

[42] Cay S. Horstmann and Gary Cornell. "Core Java 2, Volume I — Fundamentals." A Prentice Hall Title, 2001. ISBN 0-13-089468-0. Sun Microsystems Press.

[43] Paul Hudak. "Modular Domain Specific Languages and Tools." In *Proceedings of the Fifth International Conference on Software Reuse,* pages 134–142, June 1998. IEEE Computer Society Press.

[44] Paul Hudak, Tom Makucevich, Syam Gadde and Bo Whong. "Haskore Music Notation — An Algebra of Music." In *Journal of Functional Programming,* volume 6, number 3, pages 465–483, 1996.

[45] Brad Hutchings, Peter Bellows, Joseph Hawkins, Scott Hemmert, Brent Nelson, Mike Rytting. "A CAD Suite for High-Performance FPGA Design." In *IEEE Symposium on FPGAs for Custom Computing Machines,* 1999. IEEE Computer Society Press.

[46] Brad L. Hutchings and Brent E. Nelson. "Using general-purpose programming languages for FPGA design." In *Design Automation Conference,* pages 561–566, 2000.

[47] Daniel C Hyde. "CSCI320 Computer Architecture Handbook on Verilog HDL". Technical Report. Computer Science Department, Bucknell University. August 1997.

[48] Geriant Jones. "Deriving the fast Fourier algorithm by calculation." In *Functional Programming,* Glasgow 1989. Springer Workshops in Computing, 1990.

[49] Geriant Jones and Mary Sheeran. "The study of butterflies." In *Proceedings IVth Banff Workshop on Higher Order.* Springer Workshops on Computing, 1990.

[50] Geraint Jones and Mary Sheeran. "Circuit design in Ruby." In *Formal Methods for VLSI Design,* ed. J. Staunstrup, North Holland, 1990. Lecture notes on Ruby from a summer school in Lyngby, Denmark, September 1990.

[51] Samuel N. Kamin and David Hyatt. "A Special-Purpose Language for Picture-Drawing." In *Proceedings of the Conference on Domain-Specific Languages,* pages 297–310, 1997. USENIX.

[52] Alan K. Karp. "Bit Reversal On Uniprocessors." External HP Labs Technical Report, HPL-93-89, October 13, 1993.

[53] Sava Kristić, Byron Cook, John Launchbury and John Matthews. "Top-level Refinement in Processor Verification." 1998.

[54] John Launchbury, Jeff Lewis and Byron Cook. "On embedding a microarchitectural design language within Haskell." In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99),* Paris, France, September 27–29, 1999. ISBN 1-58113-111-9, volume 34(9), pages 60–69. ACM Press.

[55] Wayne Luk and Steve McKeever. "Pebble: A Language for Parametrised and Reconfigurable Hardware Design." In *FPL '98: Proceedings of the 8th International Workshop on Field-Programmable Logic and Applications, From FPGAs to Computing Paradigm,* pages 9–18, 1998. Springer-Verlag.

[56] M. Morris Mano. "Computer System Architecture." 3rd Edition, 1993. ISBN 0-13-175738-5. Prentice Hall International Editions, Inc.

[57] John Matthews and John Launchbury. "Elementary Microarchitecture Algebra." In *CAV'99: Proceedings of the 11th International Conference on Computer Aided Verification.* LNCS, volume 1633, pages 288–300, 1999. Springer-Verlag.

[58] John Matthews, Byron Cook and John Launchbury. "Microprocessor Specification in Hawk." In *1998 International Conference on Computer Languages,* 1998.

[59] William W. McCune and L. Wos. "Otter: The CADE-13 competition incarnations." In *Journal of Automated Reasoning,* volume 18, number 2, pages 211–220, 1997.

[60] Steve McKeever, Wayne Luk and Arran Derbyshire. "Compiling Hardware Descriptions with Relative Placement Information for Parametrised Libraries." In *FMCAD '02: Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design,* pages 342–359, 2002. Springer-Verlag.

[61] Steve McKeever, Wayne Luk and Arran Derbyshire. "Towards Verifying Parametrised Hardware Libraries with Relative Placement Information." In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03),* 2003. IEEE Computer Society.

[62] K.L. McMillan. "A compositional rule for hardware design refinement." In *Computer Aided Verification (CAV97),* pages 24–35, June 1997. O. Grumberg Ed.

[63] K. L. McMillan. "A methodology for hardware verification using compositional model checking." In *Science of Computer Programming*, volume 37, number 1–3, pages 279–309, 2000.

[64] K. L. McMillan. "Getting Started with SMV — User's Manual." Technical Report, Cadence Berkeley Laboratories, Berkeley. March 1999.

[65] K. L. McMillan. "Symbolic model checking: An approach to the state explosion problem." Ph.D thesis, Carnegie Mellon University, 1993. CMU-CS-92-131. Kluwer Academic Publishers.

[66] K. L. McMillan "The SMV Language." Technical Report, Cadence Berkeley Labs, Cadence Design Systems. 1999.

[67] K. L. McMillan, "The SMV System." Technical report, Cadence Berkeley Labs, Cadence Design Systems, number CMU-CS-92-131, 1992.

[68] K. L. McMillan. "Verification of Infinite State Systems by Compositional Model Checking." In *Conference on Correct Hardware Design and Verification Methods,* pages 219–234, 1999.

[69] Oskar Mencer, Martin Morf and Michael J. Flynn. "PAM-Blox: High Performance FPGA Design for Adaptive Computing." In *IEEE Symposium on FPGAs for Custom Computing Machines,* pages 167–174, 1998. IEEE Computer Society Press.

[70] John O'Donnell. "From transistors to computer architecture: Teaching functional circuit specification in Hydra." In *Symposium on Functional Programming Languages in Education,* July 1995.

[71] Stuart F. Oberman, Hesham Al-Twaijry and Michael J. Flynn. "The SNAP Project: Design of Floating Point Arithmetic Units." In *IEEE Symposium on Computer Arithmetic (ARITH-13 '97),* 1997. IEEE Computer Society.

[72] Gordon J. Pace. "Hardware Design Based on Verilog HDL." Ph.D Thesis, Computing Laboratory, Oxford University, 1998.

[73] Gordon J. Pace and Christine Vella. "Describing and Verifying FFT circuits using SharpHDL." In *Computer Science Annual Workshop 2005,* Department of Computer Science and AI, University of Malta, 2005.

[74] Daniel N. Rockmore. "The FFT – an algorithm the whole family can use." *Computer Science Engineering 2,* 60 (2000). Special issue on "Top Ten Algorithms of the Century".

[75] Richard Sharp. "Functional Design Using Behavioural and Structural Components." In *FMCAD 2002,* LNCS, volume 2517, pages 324–341, 2002. Springer-Verlag.

[76] Richard Sharp. "Higher-Level Hardware Synthesis." Ph.D thesis. April 2004. ISBN 3540213066. Springer Verlag.

[77] Robin Sharp and Ole Rasmussen. "The T-Ruby Design System." In *Formal Methods in System Design: An International Journal,* volume 11, number 3, pages 239–264, October 1997. Kluwer Academic Publishers.

[78] Mary Sheeran. "Hardware Design and Functional Programming: a Perfect Match." In *Journal of Universal Computer Science,* volume 11, number 7, pages 1135–1158, 2005.

[79] Mary Sheeran. "muFP, A language for VLSI Design." In *LISP and Functional Programming,* pages 104–112, 1984. ACM.

[80] Nabeel Shirazi, Al Walters and Peter Athanas. "Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines." In *FCCM '95: Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines,* 1995. IEEE Computer Society.

[81] Satnam Singh. "Death of the RLOC?" In *Proceedings of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM '00,* pages 145, 2000. IEEE Computer Society.

[82] Satnam Singh. "Architectural Descriptions for FPGA Circuits." In *IEEE Symposium on FPGAs for Custom Computing Machines,* pages 145–154, 1995. IEEE Computer Society Press.

[83] Simon Thompson. "Haskell: the craft of functional programming." ISBN 0-201-40357-9. 1996. Addison-Wesley.

[84] Christine Vella. "SharpHDL: A Hardware Description Language Embedded in C#." Final Year Project, University of Malta. June 2004.

[85] Christine Vella and Gordon J. Pace. "SharpHDL: A Hardware Description Language Embedded in C#." In *Computer Science Annual Workshop 2004,* Department of Computer Science and AI, University of Malta, 2004.