Algorithms in Silicon

D. de Leeuw Duarte

Parallel and Distributed Systems Group
Delft University of Technology
Abstract

FPGA-based Custom Computing Machines are application-specific computing devices that can outperform conventional software implementations of algorithms by several orders of magnitude. Developing these CCMs is a time consuming creative process that requires expert knowledge and experience. This thesis describes a systematic approach to the synthesis of FPGA-based CCM designs from conventional algorithm descriptions, based on common design patterns found in efficient hand-crafted CCM designs. It presents the imperative programming language ‘Julia’ that is based on this design method. The compilation process of Julia is described, and static analysis techniques are presented to predict the runtime and circuit growth of Julia programs. Performance experiments are presented for a number of sequential and parallel algorithms, which show that Julia is capable of delivering the expected speedups in practice.
Preface

The idea for this project emerged out of a conversation between Sidney Cadot of Science & Technology BV and myself, in October 2004. He had shown me the work he had been doing with Arjan van Gemund on FPGA-based micro-controllers for the new Embedded Realtime Systems course at Delft University of Technology, when he suggested I’d borrow one of their FPGA development boards for experimentation. In the weeks that followed, I experienced first-hand that hardware design in VHDL was quite a radical departure from programming in the conventional sense. When I returned the board, we discussed the possibility of creating a conventional imperative programming language for the development of FPGA-based, algorithm-specific computing machines. Shortly afterward, I was offered the opportunity to develop such a language at S&T, as a combined research assignment and MSc. project, under Arjan’s and Sidney’s supervision.

During my research assignment, I studied existing work on imperative programming languages for FPGA development, and, in a broader sense, reconfigurable computing systems. I learned that the development of an FPGA-targeted compiler for an imperative language was by no means to be a trivial matter. Many attempts had been made, with mildly disappointing results. For this reason, we agreed upon a more modest goal for my MSc. project: I was to obtain quantitative evidence for the potential of FPGA-based custom computing machines for a number of applications, and I was to investigate the possibility of creating a language using a ‘bottom-up’ approach, i.e., by analyzing existing circuit designs, and synthesizing a language based on that knowledge. Initially, we (well, I at least) did not expect to see an actual, working language within the timeframe of this project, but the bottom-up approach quickly proved to be very fruitful.

First and foremost, I want to thank prof. dr. ir. Arjan van Gemund (Delft University of Technology) and ir. Sidney Cadot (Science & Technology BV) for their support, enthusiasm, and useful advice during this project. Over the past fifteen months, they taught me many valuable lessons about scientific- and technical writing, embedded systems engineering, and scientific research. I want to thank Science & Technology BV in Delft for kindly providing the financial and physical means to carry out this project, without expecting any direct benefit from my work. Finally, I want to thank my parents, Paulien van Katwijk, Mark Dufour, and all people at S&T for their support and/or aid during the process of writing this thesis.

D. de Leeuw Duarte

Delft, The Netherlands
16th May 2006
# Contents

Preface v

1 Introduction 1
   1.1 Problem Statement ........................................... 1
   1.2 Contributions ................................................ 2
   1.3 Related Work .................................................. 3
   1.4 Thesis Outline ................................................ 4

2 A Survey of Reconfigurable Computing 5
   2.1 Reconfigurable Hardware ...................................... 5
      2.1.1 Reconfigurability .......................................... 5
      2.1.2 Coupling ..................................................... 6
      2.1.3 Granularity ................................................ 6
      2.1.4 Reconfiguration Time ...................................... 6
   2.2 Development Tools ............................................ 7
      2.2.1 The Semantic Gap .......................................... 7
      2.2.2 Development Tools ......................................... 7
   2.3 Chapter Summary ............................................... 9

3 Designing CCMs Manually 10
   3.1 A Simple Problem: The Collatz Conjecture .................... 10
      3.1.1 Problem Statement ......................................... 10
      3.1.2 A Collatz Verification Procedure ......................... 11
      3.1.3 Implementation ............................................. 11
   3.2 An Intermediate Problem: Transitive Closure ............... 15
      3.2.1 Problem Statement ......................................... 16
      3.2.2 A Parallel TC Algorithm .................................. 16
      3.2.3 Implementation ............................................. 16
   3.3 A Complex Problem: Boolean Satisfiability .................. 20
      3.3.1 Problem Statement ......................................... 20
      3.3.2 The DPLL Algorithm ....................................... 20
      3.3.3 Implementation ............................................. 23
      3.3.4 Further Reading ............................................ 25
   3.4 Chapter Summary ............................................... 26
# 4 Julia: an Imperative Programming Language for Hardware Design

4.1 Design Rationale .................................................. 27
4.2 An Example Program .............................................. 28
4.3 The Execution Model .............................................. 29
4.4 Overview of the Compilation Process ......................... 31
4.5 Lexical Analysis and Parsing .................................. 31
4.6 Semantic Analysis ................................................ 31
  4.6.1 Scoping ......................................................... 32
  4.6.2 AST Traversal Order ........................................ 32
  4.6.3 Type Checking, Type Annotation and Initialization Detection ...... 32
  4.6.4 Recursion Detection ........................................ 33
4.7 Syntax Tree Optimization ....................................... 34
4.8 Execution Unit Transformation ................................. 34
  4.8.1 The Execution Unit Model ................................ 34
  4.8.2 The EU Hierarchy ........................................... 35
  4.8.3 AST to EU Transformation ................................ 37
4.9 Execution Unit Optimization .................................... 39
4.10 Code Generation ................................................ 39
  4.10.1 The Structure of the Generated Code ....................... 40
  4.10.2 Synthesis of the Execution Units ......................... 40
4.11 Chapter Summary ............................................... 41

# 5 Performance Evaluation

5.1 The Test Setup ................................................ 42
5.2 Runtime Calculation ........................................... 43
  5.2.1 Calculating the Runtime of Julia Programs .................. 43
  5.2.2 Validation of the Runtime Calculation Model .............. 45
  5.2.3 Summary ..................................................... 49
5.3 Hardware Usage ................................................ 49
  5.3.1 Estimating Hardware Costs ................................ 50
5.4 Julia Performance in Practice ................................. 52
  5.4.1 Binary GCD .................................................. 52
  5.4.2 Transitive Closure ......................................... 53
  5.4.3 Satisfiability ................................................ 55
5.5 Chapter Summary ............................................... 57

# 6 Conclusions and Future Work

6.1 Summary and Conclusions ..................................... 59
6.2 Future Work ..................................................... 60

# A The Julia Language Specification

A.1 Lexical Elements ............................................... 64
  A.1.1 Source Alphabet and Character Encoding .................. 64
  A.1.2 Line Terminators .......................................... 64
  A.1.3 White Space ................................................ 64
  A.1.4 Comments ................................................... 65
  A.1.5 Keywords ................................................... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1.6 Special Symbols</td>
<td>65</td>
</tr>
<tr>
<td>A.1.7 Literals</td>
<td>65</td>
</tr>
<tr>
<td>A.1.8 Separators</td>
<td>66</td>
</tr>
<tr>
<td>A.1.9 Identifiers</td>
<td>66</td>
</tr>
<tr>
<td>A.2 Types, Constants and Storage Locators</td>
<td>66</td>
</tr>
<tr>
<td>A.2.1 Types</td>
<td>66</td>
</tr>
<tr>
<td>A.2.2 Constants</td>
<td>66</td>
</tr>
<tr>
<td>A.2.3 Storage Locators</td>
<td>67</td>
</tr>
<tr>
<td>A.3 Operators and Expressions</td>
<td>67</td>
</tr>
<tr>
<td>A.3.1 Built-in Operators</td>
<td>67</td>
</tr>
<tr>
<td>A.3.2 Expressions</td>
<td>67</td>
</tr>
<tr>
<td>A.4 Type Specifiers and Declarations</td>
<td>68</td>
</tr>
<tr>
<td>A.4.1 Type Specifiers</td>
<td>68</td>
</tr>
<tr>
<td>A.4.2 Storage Locator Declarations</td>
<td>68</td>
</tr>
<tr>
<td>A.4.3 Function Declarations</td>
<td>68</td>
</tr>
<tr>
<td>A.4.4 Operator Declarations</td>
<td>69</td>
</tr>
<tr>
<td>A.5 Statements and Statement Blocks</td>
<td>69</td>
</tr>
<tr>
<td>A.5.1 Assignment Statements</td>
<td>69</td>
</tr>
<tr>
<td>A.5.2 Conditional Statements</td>
<td>70</td>
</tr>
<tr>
<td>A.5.3 Loop Statements</td>
<td>70</td>
</tr>
<tr>
<td>A.6 Programs</td>
<td>70</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

*Everything should be built top-down, except the first time.*

— Alan Perlis

Since its popularization in the mid 1940’s, the concept of the *stored-program computer* has defined the way we design computing machines. The power and versatility of the idea — a computing machine that can be adapted to different tasks by changing the instruction sequence in its memory — quickly drove researchers away from alternative architectures. Amongst the few exceptions was Gerald Estrin, a researcher at UCLA, who pioneered the concept of a *fixed plus variable* computer in 1959: a stored-program machine with modular hardware, designed to be reconfigured by the programmer to better suit the task at hand (Figure 1) [11].

Because of the recent emergence of *Field Programmable Gate Array* technology, it is now economically feasible to extend Estrin’s approach by creating true *Custom Computing Machines*: computing devices that are designed to solve one specific problem, or even one specific instance of a problem. These FPGA based CCMs have the potential to outperform equivalent solutions in software by several orders of magnitude.

An obstacle to the widespread adoption of FPGA-based CCMs is that it is much harder to design these devices than it is to write conventional programs. To create a machine that is optimally suited to perform a particular task is still a creative art, perhaps much like programming was when the stored-program computer first emerged. But programming has since become a craft; a creative process that can be learned and taught, and a process for which we have tools and established practices.

This thesis is about the development of tools and established practices for the creation of custom computing machines. It is about a systematic approach to transforming algorithms into efficient silicon devices. Ultimately, it is about creating a tool that can perform this transformation automatically.

1.1 Problem Statement

FPGA-based custom computing machines have the potential to be significantly faster at executing computationally intensive algorithms than conventional software implementations. At present, CCMs are designed and implemented using *hardware description languages*. Due to the ‘semantic gap’ between these languages and the way we describe algorithms, i.e. the dichotomy between *structural* and
behavioral descriptions, the implementation of CCMs is a complicated process. Several attempts have been made to create a conventional imperative programming language for FPGA-based CCM design, but these efforts have generally resulted in poor performance results or poor language designs.

In this thesis, we explore the possibility of developing a systematic approach to creating FPGA-based custom computing machines from imperative pseudocode that preserves the performance characteristics of the input algorithm. In addition, we explore the possibility of creating a purely behavioral programming language using these techniques.

1.2 Contributions

In this thesis, we make the following contributions:

- We present a systematic approach for the design of FPGA-based custom computing machines that uses conventional algorithm descriptions as a starting point.
- We present ‘Julia’, an imperative programming language based on this systematic design approach, that can be used to transform algorithms into efficient CCM designs automatically.
- We provide several original CCM designs for algorithms of varying complexity, in particular, a novel implementation of the DPLL algorithm for the Boolean Satisfiability problem.
The bottom-up approach of our research distinguishes our effort from related work. We developed and studied CCM designs for different problems, and we retrospectively created formal descriptions of the computational steps that these devices performed. This allowed us to create a design method and a programming language based on common design patterns, found in actual, efficient CCM designs.

1.3 Related Work

Several programming languages and compilers exist for the development of FPGA-based custom computing machines. These systems fit in the following categories:

**Hardware Description Languages** are low level languages that are used for digital circuit design. HDL descriptions can be compiled into FPGA configuration bit streams using commercial synthesis tools. These languages cannot be used to express algorithms in the traditional sense. Hence, the transformation from algorithm to hardware design has to be performed manually by the application developer. The dominant HDLs are VHDL and Verilog. HDLs are the most common languages for CCM design.

**Dusty Deck Compilers** are compilers that accept programs written in conventional programming languages and emit synthesizable HDL designs (the term ‘dusty deck’ is hacker jargon for legacy code.) These languages require the compiler to find an efficient mapping from source code to silicon. The intimate relation between conventional languages and the stored-program computer makes it difficult to achieve this mapping. Systems like DEFACTO [7] rely on ‘heroic’ compilation techniques to identify implicit parallelism, in order to achieve acceptable performance. The dusty deck approach has largely been abandoned due to limited success.

**Hybrid Languages** occupy the middle ground between HDLs and the dusty deck approach. This category, the largest, contains conventional languages with special restrictions that allow the compiler to generate better code. The restrictions have been designed with hindsight on dusty deck compilers. SA-C [20], a static single assignment version of the C language, is a prominent member of this family, as is Streams-C [12], a subset of C with special library routines for efficient calculations on streams. Both these systems are special purpose languages, primarily tailored towards regular, data driven applications such as signal/image processing, compression, and encryption. At the other end of the spectrum are languages that extend conventional languages with hardware description features. These include the modeling language SystemC [25] and the Machines system [23], that respectively extend C++ and Java with class libraries for hardware design and simulation. Another language of this kind is Lava [2]. It uses the functional programming language Haskell as a host for circuit design. The company Celoxica sells a compiler for a C-like programming language with few restrictions, named Handel-C [4]. It is based on unknown, proprietary technology.

The compilation techniques used in these and other languages will be discussed in more detail in Chapter 2.
1.4 Thesis Outline

This thesis is structured as follows. In Chapter 2, we survey existing work in the field of reconfigurable computing, including programming languages for FPGA-based CCMs, and other reconfigurable architectures. In Chapter 3, we present a systematic approach to the design of FPGA-based CCMs through a series of examples of progressing complexity. In Chapter 4, we present Julia, an imperative programming language for FPGA-based CCM design, based on the techniques developed in Chapter 3. The performance of Julia is discussed extensively in Chapter 5. In Chapter 6, we summarize the conclusions of this work, and provide recommendations for future research. Appendix A contains the full specification of the Julia language.
Chapter 2

A Survey of Reconfigurable Computing

In this chapter we present an overview of reconfigurable computing systems. FPGA-based custom computing machines, the subject of this thesis, are part of this field. Section 2.1 focuses on reconfigurable hardware architectures. Section 2.2 provides an overview of the programming languages and compilers that are used by these systems.

2.1 Reconfigurable Hardware

Reconfigurable computing systems are computing devices that consist, in whole or in part, of hardware that is designed to be modifiable by a programmer/application developer. There exist a variety of such systems, which may be classified by their degree of reconfigurability, coupling, granularity and reconfiguration time. We discuss these properties, along with several representative systems, in the following sections.

2.1.1 Reconfigurability

Reconfigurable computing systems consist of ‘fixed’ and ‘variable’ subsystems. The variable subsystem, which is commonly referred to as the reconfigurable fabric, is a VLSI circuit with flexible routing and logic components.

The reconfigurability is an indication of the relative size and importance of the variable subsystem with respect to the fixed part of the architecture. On partially reconfigurable systems, the reconfigurable fabric is used in conjunction with a conventional CPU and I/O hardware. On fully reconfigurable systems, the reconfigurable fabric performs all computational tasks, i.e. there is no conventional CPU, and the fixed subsystem exists only to provide I/O facilities to the reconfigurable fabric.

Field Programmable Gate Arrays (FPGA) are reconfigurable fabrics that consist of small logical elements (‘Configurable Logic Blocks’), and a flexible routing mesh. FPGAs are often sold on development boards that provide basic I/O facilities to the reconfigurable fabric (e.g., ethernet ports, VGA connectors, etc.) The FPGA is reprogrammed from a remote workstation, typically by means of a serial (e.g., JTAG) interface. These basic FPGA development boards are examples of fully reconfigurable architectures. Several vendors offer FPGAs (on development boards) that share a die with a conventional CPU [27]. These are partially reconfigurable systems.
2.1.2 Coupling

Partially reconfigurable systems are further classifiable by the nature of the connection between the fixed sequential processor and the reconfigurable fabric.

On loosely coupled systems, the reconfigurable fabric is used as a separate processing unit that has a large degree of autonomy. The CPU is used to delegate computational tasks to the fabric. Loosely coupled systems are characterized by lower bandwidth and high latency connections between the fabric and the host CPU.

Tightly coupled systems are characterized by high bandwidth, low latency connections between the fixed and variable logic. In these systems, the CPU and the reconfigurable fabric typically exist on the same die. The fabric is used to implement custom instructions (e.g., VLIW operations) that are used within the context of an otherwise conventional stored program.

The Garp architecture [3] is a fixed MIPS [1] core, combined with a loosely coupled, FPGA-like reconfigurable fabric on a separate die. The CPU dispatches small tasks to the fabric, and is notified of their completion through an interrupt mechanism. The Chimaera architecture [15] also combines a fixed core with an FPGA-like fabric, but the subsystems are tightly coupled, and exist on a single die. The fixed core shares registers with the reconfigurable fabric. The fabric is used to implement application-specific VLIW instructions.

2.1.3 Granularity

Reconfigurable fabrics can be classified according to the (logical) size of the smallest configurable elements. In the context of FPGAs, these elements are referred to as Configurable Logic Blocks, or CLBs. We will use this term to refer to the elementary units of reconfigurable fabrics in general. On extremely fine-grained systems, the programmer might have control over the interconnections between individual logic gates. On coarse grained systems, the CLBs may be specialized components (e.g., RAMs, multipliers) or even simple CPU cores.

The Raw architecture [26] consists of 16 or more MIPS cores, connected by a flexible routing mesh. The programmer has explicit control over the routing mesh, and computations can be optimized and delegated to minimize communication overhead. The Raw architecture is an extremely coarse grained fabric. The CLBs of FPGAs consist of small lookup tables (e.g., 4x4 bits). On some models, the CLBs also contain flip-flops. FPGAs often contain additional specialized blocks, such as multipliers and RAMs (details vary for different vendors and models.) The fabric of Garp [3] consists of CLBs with two bit operands, i.e., they are extremely fine-grained.

2.1.4 Reconfiguration Time

The last distinguishing system property that we consider here is the reconfiguration time. While some reconfigurable fabrics have to be reconfigured statically, i.e., before use, there exist fabrics that are partially reconfigurable during execution. Amongst dynamically reconfigurable systems, we can further distinguish between systems with reconfiguration times in the microsecond range and systems that have reconfiguration times of several seconds. The faster the fabric, the more it is suitable for ‘hardware virtualization’ schemes that allow fitting of larger designs on a fabric with a limited amount
of CLBs. Another goal of runtime reconfigurability is the ability to perform dynamic rerouting of signals between CLBs. Depending on the granularity of the architecture, this can be a time consuming operation.

Most low-budget FPGAs currently on the market are statically reconfigurable. Several vendors offer high end models that allow runtime reconfiguration, but the reconfiguration times are long. The PipeRench system [14] is a coarse grained fabric that was designed for regular computations on long data streams. It consists of pipelined reconfigurable ‘stripes’ that have an extremely short configuration time (one clock cycle). The PipeRench hardware can accommodate virtual pipelines that are longer than the number of available stripes through hardware virtualization. The routing mesh of the Raw architecture [26], discussed earlier, is partially switchable under control of the running program.

2.2 Development Tools

Reconfigurable computing systems are programmed using special languages and compilers. In this section, we discuss important properties of these languages and their implementations.

2.2.1 The Semantic Gap

From the programmer’s perspective, reconfigurable fabrics are a radical departure from stored-program computers. Unlike the machine instruction sequence of the stored-program computer, which is analogous to the description of an algorithm in an imperative programming language, the reconfigurable fabric’s configuration bit stream is of a completely different nature. It is a structural description of CLBs and their interconnections, rather than a behavioral one.

To implement an algorithm on a reconfigurable computer, one needs to translate the behavioral description of the algorithm to the structural model that is the configuration bit stream. The ‘semantic gap’ between these models forms a challenge, to application programmers and compiler developers alike. Ideally, we would like to have a powerful, flexible, behavioral language to express algorithms, and a compiler that synthesizes efficient configuration bit streams from these programs. In practice, there needs to be a certain amount of intellectual load balancing between the application developer and the compiler designer to cross the semantic gap effectively. The goal of our research, and of related efforts, is to find a satisfying compromise.

2.2.2 Development Tools

In this section, we discuss existing programming languages/compilers for reconfigurable computing systems, varying from hardware description languages to imperative programming languages.

Hardware Description Languages

Hardware description languages, or HDLs, are languages for digital circuit design, simulation, and verification. Subsets of these languages are used for hardware synthesis (synthesis of FPGA and CPLD configuration bit streams). The most popular HDLs are VHDL and Verilog. We will use the former throughout this work.

In a HDL, one defines the signal interfaces and the internal structure of digital components. Complex
systems can be composed by instantiating and connecting elementary systems, i.e., through purely structural design. The languages offer limited support for behavioral modeling. A typical FPGA-based custom computing machine design consists of a network of connected finite state machines that cooperatively perform a task. The behavioral constructs of HDLs are sufficiently powerful to model these state machines. At present, HDLs are the most commonly used tools to implement FPGA-based custom computing machines.

**Dusty Deck Compilers**

For obvious reasons, there has been a considerable amount of research interest in compilers that can generate CCM designs from legacy (‘dusty deck’) code. These compilers face the monumental task of having to transform typical stored-program computer idioms, e.g., sequential programs, pointers, and dynamically indexed arrays, to efficient CCM designs. Since FPGA-based CCMs achieve most of their performance advantage through parallel execution, a good compiler must automatically discover implicit parallelism, which is a notoriously difficult problem in compiler design.

The DEFACCTO [7] system is a dusty deck high-level language compiler (for C or MatLab) that generates code for a loosely coupled, partially reconfigurable system. The transformation process involves automatic parallelization using array index analysis techniques, and an iterative compilation cycle that includes successive profiling and recompilation. Quantitative data on the performance of the resulting code is unavailable.

GCC/Chimaera is a C compiler for the Chimaera architecture [28]; a tightly coupled, partially reconfigurable system. The Chimaera compiler generates code for a Custom Instruction Set Architecture, i.e., a conventional instruction set with a number of efficient, program specific VLIW instructions, that are implemented on the reconfigurable fabric. The compiler identifies blocks of code that are suitable for optimization using custom instructions. While compiling code for a custom instruction set architecture is considerably easier than generating code for a more flexible system, the drawback of the approach is that only bit-level parallelism can be exploited. As a result, Chimaera is capable of delivering modest speedups of a constant factor 2 to 3, compared to programs that only use the fixed instruction set.

**Hybrid Languages**

The last and largest class of programming systems that we discuss here is a diverse group of languages that try to achieve a good compromise between programmer friendliness and quality of the generated code. This is achieved either by restricting the language to constructs that are straightforward to compile, or by forcing the programmer to supply more contextual information regarding the algorithm under consideration. In some languages, the programmer is exposed directly to the nature of the underlying platform, i.e., he or she will have to deal explicitly with structural design issues.

The SA-C [20] language is a variation on the C programming language. It is a single assignment language, i.e., it allows a variable to be assigned a value only once. To manipulate a value, the programmer needs to declare a new variable, and assign to it the result of an expression that contains the old variable. The single assignment restriction was introduced to allow a simple mapping of variables to signals (‘wires’). SA-C further discards pointers and recursion from the C language, and it adds arbitrary precision integers and true multidimensional arrays. Streams-C [12] is another variation on
the C language. It extends the C language with a small set of commands in which streams, processes, and signals can be defined. Using these commands, the programmer creates networks of processes that operate on regular data streams. Both SA-C and Streams-C are capable of generating efficient CCM designs for problems in the image processing and signal processing domain.

The Machines system [23], SystemC [25], and Lava [2] are systems that offer features of hardware description languages within the context of existing programming languages. These languages do not extend the syntax of their host language, but rather provide a set of library routines and/or classes that allow the programmer to design and simulate systems. In the Machines system, for example, a programmer designs systems in Java by extending the Machine base class. This class has an abstract step() method, which is called on every clock cycle. The standard Java compiler can be used to simulate these designs, whereas a special compiler can generate VHDL code from it. Similarly, SystemC and Lava provide library utilities for C++ and Haskell to create structural models of systems. Apart from a more familiar syntax, these languages offer no real abstractions over HDLs.

### 2.3 Chapter Summary

In this chapter, we provided an overview of the field of Reconfigurable Computing, to which FPGA-based custom computing machines, the subject of this thesis, belong. In Section 2.1 we discussed various approaches to hardware design, and their distinguishing features. In Section 2.2, we discussed the programming languages and compilers of reconfigurable computing systems. We also discussed the ‘Semantic Gap’, the discrepancy between the way we describe algorithms and the way we design hardware.

In Section 2.2, we have seen that three types of programming tools currently exist for FPGA-based CCM design: Hardware Description Languages (HDL), Dusty Deck Compilers, and hybrid approaches. The main objection to HDLs for CCM design is that they do not aid in crossing the semantic gap. The main objection to Dusty Deck compilers is that they generate poor code, i.e., they are incapable of exploiting the full potential of the underlying hardware. The main objections to current hybrid approaches are poor abstractions (HDL functionality in the language), overly restrictive semantics (e.g., the single assignment restriction), or a limited application domain.
Chapter 3

Designing CCMs Manually

In this chapter, we present a systematic approach to the design of FPGA-based custom computing machines from conventional algorithm descriptions. This approach is based on the manual development and subsequent analysis of efficient CCM designs. It provides the basis for our ‘bottom-up’ development of the Julia language in Chapter 4.

We will introduce the design method by means of three example problems of progressing complexity. Throughout these examples, we use the hardware description language VHDL as a target. As FPGA synthesis tools support only a limited subset of VHDL for hardware synthesis, the objective is to transcribe the algorithms under consideration in terms of simple state machines and combinational logic.

3.1 A Simple Problem: The Collatz Conjecture

As an introductory example, we will consider a trivial algorithm, derived from a problem in number theory. The focus of this example will be on elementary circuit design techniques, rather than algorithmic complexity. We will use this example to illustrate:

- How CCMs communicate with their environment.
- How computations are started and stopped.
- How simple program loops are encoded in hardware as state machines.

In the later examples, we will build on these techniques to produce implementations of more interesting algorithms.

3.1.1 Problem Statement

Consider the function $f : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ defined by:

$$f(n) = \begin{cases} n/2, & n \equiv 0 \pmod{2} \\ 3n + 1, & n \equiv 1 \pmod{2} \end{cases}$$

(3.1)

In 1937, the German mathematician Lothar Collatz conjectured that any sequence produced by repeated application of this function, first to an initial input and subsequently to its own result, will eventually produce the number 1, regardless of the initial input. The conjecture is verified here for the first ten positive integers:
As of March 29, 2006, the Collatz conjecture has been verified for all positive integers up to $9 \times 2^{58}$ [10], but a formal proof has not been found.

In this example, we design a component that can verify the Collatz conjecture for an integer up to a certain size. On an FPGA with sufficient capacity, such a component can be instantiated many times, in order to conduct a massively parallel search for potential counterexamples.

### 3.1.2 A Collatz Verification Procedure

Algorithm 3.1.2 describes a procedure for testing the Collatz conjecture on an arbitrary positive integer. The procedure takes two parameters: a candidate number $n$ and a positive bound $b$, and returns the index of the first 1 in the Collatz sequence starting at $n$. If the sought index exceeds the given bound, a zero is returned. Note that we ignore overflow.

**Algorithm 1** A procedure for verifying the Collatz conjecture.

```plaintext
1: procedure COLLATZ(n,b)
2:     i = 1
3:     while n > 1 and i <= b do
4:         if $n \equiv 0 \pmod{2}$ then
5:             $n = n/2$
6:         else
7:             $n = 3n + 1$
8:         end if
9:     i = i + 1
10: end while
11: if $n > 1$ then
12:     i = 0
13: end if
14: return i
15: end procedure
```

### 3.1.3 Implementation

The implementation of Algorithm 3.1.2 will consist of a synthesizable description of a digital component that has data inputs for the formal parameters $n$ and $b$, and an output for the computed return
value. Such a component can be instantiated inside a larger circuit, which occasionally provides new input data, waits for the computation to finish, and does something useful with the calculated result. A constraint to our implementation is thus that the enclosing circuit must be able to initiate a computation, and detect its termination.

Typically, a CCM and its enclosing circuit will progress on the rising edges of some clock signal. It is common, especially in larger designs, that these systems are asynchronous, i.e., the clock signals of the CCM and of the enclosing circuit may be mutually skewed or different in frequency. This introduces a second constraint, namely that the component must be able to communicate its input and output values robustly across different clock domains.

![Figure 3.1: Asynchronous data transfer with the four phase handshake.](image)

Figure 3.1 illustrates an elegant communication protocol that can be used to meet both these constraints: the four phase handshake. Consider the input parameter \( n \), for which we create the following signals in the interface of the Collatz component:

- **n.DATA** An input signal of an integer type.
- **n.AVAIL** An input signal of a two valued logic type.
- **n.ACK** An output signal of a two valued logic type.

The transmission of the parameter \( n \) from the enclosing circuit to the CCM proceeds in four phases. During the initial idle phase, the sender and receiver keep their synchronization signals low. The sender may initialize the second phase, a data transmission, by putting data on the data channel and driving the n.AVAIL signal high. Upon detecting the transition, the receiver registers the content of the data channel and acknowledges the transfer by driving n.ACK high, thus entering the third phase. Upon seeing the acknowledgement, the sender enters the final phase by driving n.AVAIL low. The idle phase is reentered when the receiver drives n.ACK low.
When used for all input parameters and the return value of the algorithm, the four phase handshake mechanism ensures reliable data communication between the CCM and its enclosing circuit. As a useful convention, we will design CCMs in such a way that they start processing once they receive new values for all formal parameters. Termination of the algorithm is detectable when the component transmits its return value.

As shown in Figure 3.2, the control flow of the Collatz component can be modeled as a finite state machine. The initial state being \texttt{WAIT\_INPUT}. A state transition occurs on each rising edge of the internal clock of the component. Apart from the shown named states, the component has to maintain additional state in the form of registers for each variable that is used in the main loop of the algorithm. During each transition from the \texttt{EXECUTE\_LOOP} state onto itself, the values in these registers are updated to reflect one iteration of the loop.

It is straightforward to encode state machines with the described properties in synthesizable VHDL code. For reference in future examples, we will provide the complete source code for the Collatz verifier here.
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity collatz is
port ( clock : in std_logic;
    n_DATA : in signed( 31 downto 0 );
    n_AVAIL : in std_logic;
    n_ACK : out std_logic;
    b_DATA : in signed( 31 downto 0 );
    b_AVAIL : in std_logic;
    b_ACK : out std_logic;
    retval_DATA : out signed( 31 downto 0 );
    retval_AVAIL : out std_logic;
    retval_ACK : in std_logic );
end collatz;

architecture collatz_arch of collatz is
    type state_t is ( WAIT_INPUT,
    ACK_INPUT,
    EXECUTE_LOOP,
    SEND_RESULT,
    WAIT_RESULT_ACK );
begin
    -- The state machine.
    collatz_proc: process( clock )
        variable state : state_t := WAIT_INPUT;
        variable n : signed( 31 downto 0 );
        variable b : signed( 31 downto 0 );
        variable i : signed( 31 downto 0 );
        begin
            if rising_edge( clock ) then
                -- Set default control signal values.
                n_ACK <= '0';
                b_ACK <= '0';
                retval_AVAIL <= '0';

                case state is
                    when WAIT_INPUT =>
                        if n_avail = '1' and b_avail = '1' then
                            n := n_DATA;
                            b := b_DATA;
                            n_ack <= '1';
                            b_ack <= '1';
                            state := ACK_INPUT;
                        end if;
                    when ACK_INPUT =>
                        n_ack <= '1';
                        b_ack <= '1';
                        if n_AVAIL = '0' and b_AVAIL = '0' then
                            n_ack <= '0';
                            b_ack <= '0';
                            i := to_signed( 1, 32 );
                            state := EXECUTE_LOOP;
                        end if;
                    when EXECUTE_LOOP =>
                        -- Stop condition and body of the while loop.
                end case;
            end if;
        end process;
end collatz;
if ( n > 1 ) and ( b > i ) then
  if n(0) = '0' then
    n := n srl 1;
  else
    n := to_signed( 3 * to_integer( n ), 32 ) + 1;
  end if;
  i := i + 1;
else
  if ( n = 1 ) then
    retval_DATA <= i;
  else
    retval_DATA <= to_signed( 0, 32 );
  end if;
  state := SEND_RESULT;
end if;
when SEND_RESULT =>
  retval_avail <= '1';
  if retval_ack = '1' then
    retval_avail <= '0';
    state := WAIT_RESULT_ACK;
  end if;
when WAIT_RESULT_ACK =>
  retval_avail <= '1';
  if retval_ack = '0' then
    state := WAIT_INPUT;
  end if;
end case;
end if;
end process collatz_proc;
end architecture collatz_arch;

In this example, we have seen the transformation process of a trivial iterative algorithm from pseudocode to state machine to hardware description. We have demonstrated techniques for asynchronous transmission of parameter data and return values to and from a CCM. In the subsequent examples, we assume that the reader is familiar with these techniques, as we focus on other aspects of the source-to-silicon transformation process.

3.2 An Intermediate Problem: Transitive Closure

In this example, we shift our attention from elementary control flow and data transfer to the transformation of nontrivial algorithms. This example will illustrate:

- How subroutine calls are implemented.
- How parallel loops are implemented.
- How sequential loops are expanded either spatially or temporally.

In addition, the example shows the potential of reconfigurable hardware as a cost effective platform for parallel matrix multiplication algorithms.
3.2.1 Problem Statement

Let \( G(V, E) \) be a directed multigraph with vertices \( V = \{v_1, \ldots, v_n\} \) and edges \( E \). The transitive closure of \( G \), denoted by \( G^*(V, E^*) \), is a graph such that \((v_i, v_j)\) is an edge in \( E^* \), if and only if there is a path from \( v_i \) to \( v_j \) in \( G \) of length one or more. The objective of the transitive closure problem is to find \( G^* \) for a given \( G \).

Figure 3.3: The transitive closure problem.

Formulated more intuitively, the transitive closure of a graph describes which vertices are reachable from some starting vertex in the original graph, either directly or indirectly. This is illustrated in Figure 3.3, where the black edges denote the original graph, and the dashed arrows denote edges that are added to form the transitive closure. The transitive closure problem arises in logistics and computer networking.

3.2.2 A Parallel TC Algorithm

For the sake of this example, we will limit ourselves to a subset of the problem domain, where the number of vertices in \( G \) is \( 2^k \), for some positive integer \( k \).

Let \( M \) denote the incidence matrix of \( G \), i.e., we have that \( M[i, j] = 1 \) for each \( i, j \) where \((v_i, v_j) \in E\), and 0 otherwise. Note that, for any positive \( i, j, k \), the value \( M^k[i, j] \) indicates whether the vertex \( v_j \) is reachable from \( v_i \) in exactly \( k \) steps. Note also that the shortest path from \( v_i \) to \( v_j \) in a graph with \( n \) vertices can be at most \( n - 1 \) steps. The incidence matrix \( M^* \) of the transitive closure of \( G \) is therefore given by:

\[
M^* = I \lor M^1 \lor \cdots \lor M^n = (I \lor M)^{2k}
\]  

(3.2)

Equation 3.2 shows that \( M^* \) can be computed with \( k = \log n \) matrix squaring operations. The complete procedure is given in Algorithm 2.

3.2.3 Implementation

For the implementation of the TRANSITIVECLOSURE() routine, we follow a strategy that is similar to the one used for the Collatz verification procedure in Section 3.1. Due to the subroutine call inside the body of the procedure, it will be necessary to include additional states in the control flow mechanism,
Algorithm 2 A parallel transitive closure algorithm for graphs with $n = 2^k$ vertices.

1: procedure TRANSITIVECLOSURE($M$)
2: $M = I \lor M$
3: for $k \in [1, \log n]$ do
4: $M = $ MATRIXSQUARE($M$)
5: end for
6: return $M$
7: end procedure

8: procedure MATRIXSQUARE($M$)
9: parfor $i, j, k \in [1, n]$
10: $Y[i, j, k] = M[i, k] \land M[k, j]$
11: end parfor
12: for $h \in [1, \log n]$ do
13: parfor $i, j, k \in [1, n/2^h]$
14: $Y[i, j, k] = Y[i, j, 2k - 1] \lor Y[i, j, 2k]$
15: end parfor
16: end for
17: parfor $i, j \in [1, n]$
18: $Z[i, j] = Y[i, j, 1] \lor Y[i, j, 1]$
19: end parfor
20: return $Z$
21: end procedure
i.e., to send data to the \textsc{MatrixMultiply()} component, wait for its result, and continue processing. The state transition diagram for the \textsc{TransitiveClosure()} procedure is shown in Figure 3.4. As transitions from one state onto itself have no side effects in this example, these have been omitted from the description.

Figure 3.4: State transition diagram of the \textsc{TransitiveClosure()} routine.
The body of the `MatrixSquare()` routine is comprised of three loops: the parallel loops on lines 9 through 11 and 17 through 19, and a sequential loop on lines 13 through 16. Given that the computations inside a `parfor` loop are by definition assumed to be independent across loop instances, and that the total number of instances is statically computable, the parallel loops can be implemented by instantiating the appropriate number of AND or OR gates for the respective loop bodies, and connecting the signals as necessary.

For the sequential `for` loop on lines 13 through 16, we could again adopt the strategy of creating a state machine that performs successive iterations of the loop body, as we did with the main loop of `TransitiveClosure()`. We will however use the opportunity to discuss an alternative technique that is useful in some algorithms: (spatial) loop unrolling.

One can observe that the number of iterations of the `for` loop of lines 13 through 16 is statically computable, and that during one iteration of the loop two matrices of the array of matrices $Y$ are added (or-ed) element wise, as illustrated in Figure 3.5. A state machine implementation of the loop would consist of one parallel matrix addition circuit which is reused during successive loop iterations. An alternative is to expand the loop spatially by creating a pipeline of $\log n$ successive matrix adders. While costly in terms of hardware utilization, a spatially expanded loop can significantly reduce the average throughput of an algorithm, especially if the pipeline can be kept full. If, as in this case, the loop body is a combinational circuit, the registers between the pipeline stages may even be omitted, allowing for all the loop iterations to be performed in a single clock cycle. The downside of this is that the maximum clock frequency at which the design will operate correctly is bound by the length of the longest path in the circuit, and that it will be necessary to lower the frequency as the matrix gets bigger. This tradeoff is beyond our current scope, but is an interesting topic for further study.

In this example, we have demonstrated how a simple parallel algorithm can be transformed into a structural hardware design. We have seen how the state machine concept from Section 3.1 can be expanded to allow for the execution of subroutines inside a procedure or loop body, and we have seen an alternative method for implementing loops by unrolling them into pipelined circuits. In the third and final example, we will apply and expand upon these techniques to solve a realistic and computationally hard problem.
3.3 A Complex Problem: Boolean Satisfiability

In the third and final example, we will study the application of CCMs in a more demanding and realistic environment. This example will demonstrate:

- How the techniques developed in the earlier examples are applied in the context of a larger problem.
- How, and to what extent, the practical limitations of reconfigurable hardware (e.g., the lack of large, random access storage) can be addressed when implementing larger systems.

Importantly, this example demonstrates the potential of reconfigurable hardware as a powerful and cost effective alternative to conventional distributed computing systems.

3.3.1 Problem Statement

Let $F$ be a boolean formula in conjunctive normal form (CNF) which contains a number of unknowns $U = \{A, B, \ldots\}$. The objective of the Boolean Satisfiability problem (SAT) is to decide whether there exists a valuation for each of the unknowns, such that $F$ evaluates to TRUE.

As an example, consider the formula:

$$F = (A \lor D) \land (A \lor \neg B \lor C \lor D) \land (\neg A \lor \neg B \lor \neg D) \land (\neg C)$$ \hspace{1cm} (3.3)

The valuation $V : \{A = D = \text{TRUE}, B = C = \text{FALSE}\}$ is one of several valuations that satisfies $F$, hence $F$ is satisfiable. $V$ is called a certificate for the instance $F$.

In 1971, Stephen Cook proved that any nondeterministic polynomial time decision problem can be reduced to the Satisfiability problem in polynomial time. This established SAT as the first proven NP complete problem. The best known algorithms for SAT have exponential worst case time complexity (whether there exists a polynomial algorithm for SAT is one of the central open questions in complexity theory.) However, heuristic solvers with reasonable average case performance find common use in many areas of application, including circuit design verification systems, model based diagnosis systems, and logistical planners.

3.3.2 The DPLL Algorithm

Most modern SAT solvers are based on a procedure that was developed by Davis, Putnam, Logemann and Loveland in the early 1960s [5] [6]. We will first describe the classic procedure, after which we present a fine-grained parallel variation for implementation in hardware.

Before we describe the main algorithm, we shall present one of its component parts: a generic procedure for the simplification of CNF formulae. Consider again the example from Equation 3.3:

$$F : (A \lor D) \land (A \lor \neg B \lor C \lor D) \land (\neg A \lor \neg B \lor \neg D) \land (\neg C)$$ \hspace{1cm} (3.4)

Observe that the last clause in $F$ contains a singleton literal. It is evident that this clause, and thus $F$, can only be satisfied by valuations that include $\{C = \text{FALSE}\}$. We say that the valuation of $C$ is
implied in this formula. In order to decide on the satisfiability of $F$, we can thus resort to deciding on the satisfiability of the following simplified formula:

$$F' : (A \lor D) \land (A \lor \neg B \lor D) \land (\neg A \lor \neg B \lor \neg D)$$

(3.5)

The elimination of singleton clauses in this manner is known as unit propagation.

Further examination of the formula $F'$ of Equation 3.5 reveals that the variable $B$ only occurs in negated form. The literal $\neg B$ can thus be satisfied without causing conflicts, which reduces the formula even further, to $F'' : (A \lor D)$. This transformation is known as the pure literal rule.

Unit propagation and the pure literal rule mark the beginnings of a collection of deterministic transformations that can be used to simplify Boolean formulae. Since the successful application of one rule may lead to the (re)applicability of another, we arrive at a general procedure for the simplification of CNF formulae, which is described in Algorithm 3.

**Algorithm 3** A simplification procedure for CNF formulae

1: procedure SIMPLIFY($F$)
2:     repeat
3:         RuleApplied = FALSE
4:     if UNITPROPAGATION($F$) then
5:         RuleApplied = TRUE
6:     end if
7:     if PURELITERALRULE($F$) then
8:         RuleApplied = TRUE
9:     end if
10:    . . . ▷ Optional additional rules.
11:   until !RuleApplied
12: end procedure

The procedure of Algorithm 3 can be used to simplify CNF formulae, as long as one of the reduction rules in its body is applicable. On problems of reasonable size, the routine is likely to terminate well before a satisfying valuation is reached. A complete algorithm for the SAT problem thus requires an additional strategy to deal with undecided formulae that cannot be simplified any further. The DPLL procedure is such an algorithm.

Algorithm 4, at last, shows a generic version of the classic DPLL procedure. The first step of the algorithm is to perform a simplification procedure such as the one described in Algorithm 3. This will yield a formula that is simplified to the extent that it is at least devoid of singleton clauses. If the formula is simplified to the point where it contains no more clauses (line 4.3), the DPLL procedure concludes that the original formula was satisfiable. If, on the other hand, the formula contains an empty clause after simplification, the original formula will be concluded to be unsatisfiable (line 4.6).
If simplification does not yield a decision, the DPLL procedure will ‘guess and test’. It will select a free literal \( l \) from the formula (this is known as the \textit{branch literal}, line 4.9), and an attempt is made to satisfy \( F \) with an added singleton clause containing \( l \). Adding a singleton clause to the formula will have the effect of satisfying its only literal immediately, due to application of the unit propagation rule. If the attempt to solve the augmented problem fails, the algorithm retries with an added singleton clause containing \( \neg l \). If this attempt also fails, the formula is unsatisfiable.

**Algorithm 4** The DPLL procedure

1: \textbf{procedure} \textsc{Solve}(\( F \))
2: \hspace{1em} \textsc{Simplify}(\( F \))
3: \hspace{1em} \textbf{if} \( F = \emptyset \) \textbf{then}
4: \hspace{2em} \textbf{return} \textit{satisfiable}
5: \hspace{1em} \textbf{end if}
6: \hspace{1em} \textbf{if} \( \{\} \in F \) \textbf{then}
7: \hspace{2em} \textbf{return} \textit{unsatisfiable}
8: \hspace{1em} \textbf{end if}
9: \hspace{1em} \( l := \text{ChooseBranchLiteral}(\ F \) \)
10: \hspace{1em} \textbf{if} \textsc{Solve}(\( F \cup \{l\} \)) = \textit{satisfiable} \textbf{then}
11: \hspace{2em} \textbf{return} \textit{satisfiable}
12: \hspace{1em} \textbf{else}
13: \hspace{2em} \textbf{return} \textsc{Solve}(\( F \cup \{\neg l\} \))
14: \hspace{1em} \textbf{end if}
15: \hspace{1em} \textbf{end procedure}

Algorithm 4 is a parametric version of the DPLL procedure. A complete algorithm requires a full specification of the two subroutines, i.e., a set of simplification rules\(^1\), and a branching policy. For the classic DPLL procedure, the set of simplification rules consists of the two rules discussed earlier, unit propagation and the pure literal rule, and a branching routine that picks the first literal of the first unsatisfied clause in the formula.

Modern DPLL based SAT solvers differ from the classic algorithm by their use of better branching heuristics, and, to a lesser extent, more elaborate simplification rules. Lookahead solvers, e.g., SATZ [17], MARCH [16], and KCNFS [9], tentatively calculate the result of satisfying each possible branch literal, and assign fitness scores to the resulting simplified formulae. Conflict driven solvers, such as zCHAFF [19], and GRASP [18], are based on a slightly modified version of the DPLL algorithm. They analyze the root cause of conflict after a failed branch, in order to make better branch predictions during subsequent attempts.

Empirical evidence shows that DPLL based solvers spend around 90 percent of their time applying simplification rules [8]. While good branching heuristics reduce the execution time by avoiding unnecessary searches, there is much performance to be gained by creating a fast parallel implementation of the simplification procedure itself. In fact, the linear amount of work required for the application of a simplification rule with respect to the length of the formula can be performed in constant time using such an implementation, leading to speedups that grow linearly with the problem size. This will be

\(^1\) Note that the unit propagation rule is required as a bare minimum, in order to ensure termination of the algorithm.
the main focus of our implementation.

3.3.3 Implementation

Due to its recursive nature, the DPLL procedure is considerably more challenging to implement in hardware than the previous examples. The complexity arises from the fact that a partial valuation is stored ‘on the stack’ for each recursive instance of the procedure. We need to deal with the stack problem explicitly, because we cannot dynamically allocate physical storage space at runtime.

The following pseudocode implements a stack for partial valuation vectors. This implementation requires a fixed amount of storage for the valuation vector, regardless of the number of items on the stack. For \( k \) variables, the algorithm requires a maximum stack depth of \( k \) items, hence the timestamp vector that is used is also of fixed size.

```c
enum tristate_bool_t { TRUE, FALSE, UNDEFINED };
tristate_bool_t valuation[NUMBER_OF_VARIABLES];
int timestamp[NUMBER_OF_VARIABLES];

void pushValuation()
{
    parfor( i in 0 to NUMBER_OF_VARIABLES )
        if ( valuation[i] != UNDEFINED )
            timestamp[i]++;
}

void popValuation()
{
    parfor( i in 0 to NUMBER_OF_VARIABLES )
        if ( valuation[i] != UNDEFINED )
            {
            timestamp[i]--;
            if ( timestamp[i] == 0 ) valuation[i] = UNDEFINED;
            }
}
```

The stack mechanism works by keeping timestamps for each variable in the partial valuation. The `pushValuation()` operation pushes all defined variables on the stack by incrementing their timestamps. The `popValuation()` primitive decrements all timestamps, and clears all variables for which the timestamp reaches 0. For the DPLL algorithm, this mechanism is sufficient, as a defined variable can only be changed by the call instance that defined it, hence it is invariant under the push operation.

The valuation stack has the useful property that it consists of a fixed amount of storage that resides on a fixed physical location. The remainder of the solver consists of two parts: a collection of fast operations for manipulation of the valuation vector (i.e., the top element of the stack), and a ‘main’ routine that determines which operation is to be performed next, based on the current valuation, and some additional global variables.

We start by defining the `branch` operation. This operation examines the valuation vector, and assigns a value to exactly one of the values that are currently undefined. The branch operation is a fine-grained parallel routine that is specific to the SAT instance at hand. Its code is generated for each instance with a script. For our solver, we implement a simple branching heuristic that satisfies the left most unassigned literal of the left most unsatisfied clause. This can easily be accomplished by using
one priority encoder per clause to find the left most unassigned value, and one priority encoder to
find the left most unsatisfied clause. The branch operation requires a linear amount of hardware with
respect to the problem size, and \(O(\log(n))\) time. It is the slowest operation in the solver. Fortunately,
is is not executed frequently.

The simplify operation is the most frequently used operation in the solver. This operation satisfies
all variables that are directly implied by the current valuation. Repeated applications of the operation
are necessary to resolve indirect implications. Our solver contains implication rules for unit propa-
gation only. It can easily be extended with additional rules. The simplification operation is another
instance specific routine. It might look as follows:

```c
void simplifyISP()
{
    tristate_bool_t impliedValues[NUMBER_OF_VARIABLES];
    // Unit propagation on A for the formula (A v !B v C)(!A v D)
    if ( valuation[0] == UNDEFINED )
    {
            impliedValues[0] = TRUE;
        if ( (valuation[3] == FALSE) )
            impliedValues[0] = FALSE;
    }
    else impliedValues[0] = UNDEFINED;
    // ... (implication code for other variables)
    // Update valuation
    parfor ( i in 0 to NUMBER_OF_VARIABLES )
    {
        // Pick any _defined_ value.
        valuation[i] = any( impliedValues[i], valuation[i] );
    }
}
```

Note that the code for each variable can be extended with additional implication rules beyond unit
propagation, and that the individual entries of the impliedValues vector can be computed con-
currently. As the size of the code for the individual variables grows very slowly with the problem
size, the execution time of the simplification procedure will approximately be constant; a significant
improvement over the linear performance of a sequential implementation for the critical inner loop of
the algorithm.

With the valuation stack, the branch operation, and the simplify operation, it is possible to create
a non recursive reformulation of the DPLL algorithm:

```c
bool solve()
{
    bool resultUndetermined = true;
    // The search process.
    while( resultUndetermined )
    {
        switch( nextAction() )
        {
            case SIMPLIFY:
                simplifyISP();
                break;
```
case BRANCH:
    pushValuation();
    branchISP();
    pushValuation();
    break;

case UNROLL:
    popValuation();
    break;

case RETRY:
    parfor( i in 0 to NUMBER_OF VARIABLES )
    {  // Invert valuation at top of stack.
       if ( retry[i] && timestamp[i] == 1 )
       {
           valuation[i] = !valuation[i];
           retry[i] = false;
       }
    }

case HALT:
    resultUndetermined = false;
    break;
}

return satisfyingValuationFoundISP();
}

The procedure uses the nextAction() routine to examine the current state of the valuation, and to
determine the next course of action. The nextAction() routine contains calls to several instance
specific routines that determine whether the current valuation satisfies or conflicts with the formula
or leaves it undecided, whether simplification rules would be applicable on the current formula, and
whether the current branch variable (i.e., the only variable with a timestamp of 1) has or has not been
tried in negated form (indicated by the retry vector). The five possible courses of action are: apply
simplification rules, activate a branch variable, unroll to the last branch variable, retry the negation of
the last branch variable, and halt.

At this point, we have reduced the problem of implementing the DPLL algorithm to that of im-
plementing the constructs used in the earlier examples. We have shown how a complex, dynamic
algorithm can be implemented within the limitations of reconfigurable hardware. In Chapter 5, we
will show that this solver, despite its primitive branching heuristics, is capable of outperforming state
of the art software based solvers by orders of magnitude.

3.3.4 Further Reading

Hardware satisfiability solvers were pioneered by Suyama et al. in 1996 [24]. Like most of the later
architectures, Suyama’s solver consists of a conventional program that generates HDL source code
for an instance specific solver, much like our design. The Suyama architecture is based on an algo-

rithm that considers full valuations, in contrast to DPLL solvers, which are characterized by the use
of partial valuations during the search process. The Suyama algorithm is fundamentally slower than
DPLL based solvers.

The solver of Zhong et al. [29] (1998) was the first hardware based solver to use partial valuations.
The design is based on the DP algorithm [6], a slightly older variation of the DPLL procedure. Due
to clock frequency problems with the original design, Zhong completely and successfully revised the architecture in 1999. A problem shared by all the Zhong designs is the static ordering of branch variables, which effectively prohibits the implementation of advanced branching heuristics. The Zhong designs are limited to instances with a fixed number of variables per clause. Strong points of the later revisions are the addition of simple conflict analysis/nonchronological backtracking, inspired by the GRASP [18] solver, and the proposed non-instance specific variation of the solver, which allows for dynamic clause addition, and effectively eliminates the synthesis overhead, at the cost of some runtime overhead.

The work of Zhong inspired several derivate architectures, most notably the work of Platzner et al [21]. Other architectures are based on the PODEM [13] algorithm, rather than the DP algorithm. Sklarova and De Brito Ferrari have compiled an excellent overview of existing work on hardware based SAT solvers up to 2003 [22].

3.4 Chapter Summary

In this chapter, we presented a collection of techniques for the translation of algorithms, expressed in imperative pseudocode, to efficient custom computing machine designs, through a series of progressing examples. With these techniques, we can transcribe the following constructs:

- Conditional statements
- Sequential loops
- Parallel loops
- Subroutines

These techniques form the basis of the imperative programming language ‘Julia’, which is presented in Chapter 4. We have suggested in passing that the implementations described in this chapter can be used to achieve significant speedups over conventional implementations in software. In Chapter 5, we will substantiate these claims with the results of a number of performance experiments.
Chapter 4

Julia: an Imperative Programming Language for Hardware Design

In this chapter we present ‘Julia’ 1, an imperative programming language with a clean, C-like syntax, explicit parallel loops, implicit fine-grained parallelism, and a straightforward mapping onto reconfigurable hardware. The design of the language was inspired by the collection of manual problem solving techniques developed in Chapter 3.

4.1 Design Rationale

The Julia language is a lightweight imperative programming language, built on top of the hardware description language VHDL. The language was designed using a bottom-up approach: it consists of conventional imperative constructs that are needed to describe the functionality of a variety of efficient, hand-crafted custom computing machine designs. As such, the language is a true abstraction of common CCM design patterns. In this section, we provide an overview of the rationale behind the key design choices that constitute the language.

Syntax, Semantics and Grammar

The syntax, semantics, and grammar of Julia are based on the C language, and to a lesser extent, on C++. The widespread use of C, C++, Java, and other C-like languages ensures instant familiarity with Julia to a large number of developers.

Julia stays very close to the C language, to the extent that non-trivial Julia functions are often equally valid and semantically identical C functions. The converse is however not generally true, as Julia is more strict on several issues. Some of these restrictions were necessary to aid compilation for the FPGA platform, in particular with regard to the omission of \texttt{goto}, and the inhibition of \texttt{return} statements in the middle of a function. Other restrictions were added out of discontentment with some of the oddities of the C language, and should not be of practical hindrance to most programmers.

Execution Model

The Julia compiler creates directed, feed-forward data-flow graph representations of input programs. The vertices of these graphs represent functional units (operators, etc.), while the

---

1The Julia language was named in honor of the author’s niece, whose birth coincided with the development of the prototype compiler. Any allusions to French mathematicians, living or dead, or fractal families of the same name, are unintentional.
edges represent handshaked data connections. The compiler back-end generates HDL code from these graphs. Julia’s data-flow model includes special functional nodes that encapsulate subgraphs for conditional and/or iterative computations. The data-flow model, augmented with the special control nodes, is directly motivated by the structure of the hand-crafted CCM designs considered in Chapter 3.

**Parallelism**  Julia deals with parallelism in two ways. At the statement level or expression level, parallelism is discovered and exploited by the compiler automatically. For mutually independent statements or (sub)expressions, the compiler generates concurrent execution paths in the data-flow graph. For parallelism at the block/loop level, Julia has an explicit construct: the `parfor` loop. With this double approach, we provide support for fine-grained parallelization, whilst avoiding the known difficult problem of automatic loop parallelization.

**Abstraction Level**  Julia was designed as a thin abstraction layer on top of FPGA hardware. As performance is the primary motivator for the use of FPGA-based custom computing machines, we chose to avoid high level abstractions that have no direct equivalent in reconfigurable hardware. The most prominent omission, in this respect, is the lack of dynamic array indexing. In Julia, array indices must be known at compile-time. This restriction was introduced to force the programmer to deal with the lack of random access memory explicitly, as the alternatives—propagating large arrays integrally across the data flow network, or inferring RAM blocks—would introduce unpredictable circuit growth or performance.

In summary, we designed Julia to offer the best behavioral abstraction over HDLs possible, without sacrificing transparency. We designed it to encourage the use of algorithmic constructs that are, in a sense, suitable for implementation on the FPGA platform, and we decided not to afford the implementation of constructs that are fundamentally unsuitable for the platform.

### 4.2 An Example Program

We provide a first look at the Julia language by means of a short example: a reimplementation of the Collatz conjecture tester from Section 3.1.

```c
/**
 * Input: two positive integers, ‘n’ and ‘b’.
 * Return value: index of first 1 in Collatz sequence of n, or 0 if index exceeds b.
 */
int collatz( int n, int b )
{
    int iterations;
    int result;
    result = 0;
    iterations = 0;

    while( ( n > 1 ) && ( iterations < b ) )
    {
        if( ( n & 1 ) == 1 ) n = 3 * n + 1;
        else n = n >> 1;
    }
    if ( n == 1 ) result = iterations;
    return result;
}
```
As the example shows, the syntax of Julia is similar to that of C, but there are notable differences. The most important ones are:

- Julia programs have no ‘main’ routine. Each function compiles into a separate VHDL entity, which can be instantiated inside a device specific top level circuit or a simulation environment.
- Julia has a `parfor` loop.
- All mutually independent statements inside a block are executed concurrently.
- Recursion, direct or mutual, is not permitted.
- Variables may be declared on any line, but must be initialized (unconditionally) before use.
- Julia has no pointers or dynamically indexed arrays.
- A function must contain exactly one return statement, which must be the last statement in the function body.
- All functions in Julia are ‘pure’. There are no global and/or static variables, and functions can have no side effects.

The specification of the Julia language (Appendix A) reveals a number of additional differences, but as a general rule, the language is so defined that a valid Julia function is also a valid and equivalent ANSI C99 function. The converse is not generally true, as Julia is more strict than C99 on many occasions (e.g., booleans and ints are truly distinct types, assignment symbols have no return value, etc.) The main exception to this rule is the `parfor` loop, which has no C99 equivalent.

### 4.3 The Execution Model

From the programmer’s perspective, the statements inside a block of code are executed sequentially, exactly as they would be in C. With the exception of variable declarations, each statement is compiled into an ‘execution unit’, which has data inputs for all the variables that are used as right hand side values in that statement, and data outputs for all the values that are used on the left hand side, if such exist. The end result of compilation is a network of connected execution units that implements the functionality of the compiled code block. Statements and operations that have no mutual data dependencies are executed concurrently by the network, resulting in the highest possible degree of fine-grained parallelism.

The details of Julia’s execution model are presented later in this chapter, but the sequential loop semantics of the language deserve special mention here. In Julia, sequential loops are always implemented as actual, temporal loops that require a fixed amount of hardware, regardless of the number of iterations. The compiler achieves this by transforming the loop body and the stop condition into separate execution units, and packing these in a ‘repetition unit’. The repetition unit contains registers for all the external variables that are used inside the loop body and the stop condition, and repeatedly sends the register data to these units, updating the registers after each iteration. Hence, a sequential loop can be used by the programmer to explicitly force hardware reuse.

Figure 4.3 shows a schematic representation of the execution unit network that is generated by the Julia compiler during the compilation of the Collatz program of Section 4.2. The image shows a large
composite execution unit, which carries the name of the compiled function ('collatz'). The function unit has two data inputs, b and n, which are immediately connected to the repetition unit that implements the main loop. The solid arrows denote direct connections, whereas the dotted arrows represent connections between 'special' execution units and their children. These special units are
units that perform loops or conditional statements. The interior of this unit, in turn, shows the two nested units labeled LoopCondition and LoopBody. The reader is encouraged to study the image carefully, as a good understanding of the example will be of aid throughout the following chapters.

4.4 Overview of the Compilation Process

In the remainder of this chapter, we will discuss the techniques that are used in the compilation of the Julia language. We assume that the reader is familiar with the basics of compiler theory, hence, we will focus on the transformations that are unique to Julia.

The compilation process consists of a number of stages:

1. Lexical analysis and parsing.
2. Semantic analysis.
4. Execution unit transformation.
5. Execution unit optimization.

The discussion that follows is not specific to any implementation of the language, but we will frequently highlight implementation details of `hwc`, our prototype implementation the language, as an example of how the theory is put to practice.

4.5 Lexical Analysis and Parsing

The compilation of a Julia program starts with conventional lexical analysis and parsing of a source file. The product of this stage is a tree representation of the code, in which each node represents a different grammatical construct (e.g., a function declaration, a binary operator expression, etc.) The Julia language imposes no special requirements on the lexer and parser, hence we will only use this section to provide details on the AST structure of the hwc compiler for later reference.

The AST nodes of the hwc compiler have a simple structure. Each node has a nodetype attribute, which indicates which type of language construct the node represents. Each node has a line number, an optional list of children, and a nodedata attribute for optional string data. Depending on the node type, it can contain the name of an identifier, a constant expression, an operator symbol, etc. Several node types have additional attributes, which are used during semantic analysis and code generation. These will be introduced as as appropriate, during the discussion of the subsequent stages.

4.6 Semantic Analysis

After parsing, the correctly constructed AST has to be analyzed for semantic correctness and annotated with contextual information. To this end, the compiler has to traverse the AST, keep track of the current scope in a stack-like data structure, and perform a number of local operations on the visited
nodes. The intimate relation between the semantic analysis stage and the special properties of the Julia language warrants a detailed look at this process. In the subsequent sections, we will discuss traversal order, scoping, type checking/annotation, enforcement of the variable initialization rules and recursion detection in the Julia compiler.

4.6.1 Scoping

In the Julia language, a scope is a set of (name, declaration) tuples. These tuples are called definitions. Definitions are unique within a scope with respect to the name field. A scope can have multiple child scopes, which implicitly contain all the elements of the parent scope. Redefinitions are not allowed.

In the hwc compiler, scoping is implemented as an hierarchical associative array that associates strings with AST nodes that represent declarations of functions, types, storage locators, or operators. The declarations of functions, storage locators (variables) and types can be retrieved from the associative array by their respective identifiers (the function name, the name of the storage locator, the name of the type.) Operator declarations are identified by a concatenation of the operand type names and the operator symbol, e.g., “int+int”, “!bool”, etc.

4.6.2 AST Traversal Order

The Julia language specification (Appendix A) prescribes that a Julia source file is parsed as a declaration list, and that the declarations in this list are either function declarations, operator declarations or type declarations. These declarations exist in the program’s global scope. They may be referenced from inside a function body from any point in the file. This implies that all the top level declarations have to be added to the global scope before the bodies of functions, operator declarations, etc. are analyzed. After all the top level declarations are added to the global scope, the bodies of operator declarations and function declarations are visited in depth first order.

4.6.3 Type Checking, Type Annotation and Initialization Detection

After declaration of the types, functions, and operators in the global scope, the compiler performs a depth first visit of the bodies of the functions declared at the top level, entering and leaving scopes according to the rules defined in Appendix A. As variable declarations are encountered, the type names are resolved in the current scope. If the used type name exists, the AST node containing the variable declaration is annotated with a reference to the AST node that contains the type declaration. The declared variable is itself added to the current scope. Failure of the type declaration lookup indicates a semantic error in the program (‘undefined type’ error.) Prior existence of a variable with the same name is also a semantic error (‘illegal redeclaration’ error.)

An important part of the traversal process is the annotation of expressions with type information. When variables and function names are encountered during the traversal of expressions, their types are retrieved through a lookup in the current scope. The AST nodes of these expressions are annotated with a reference to the AST node that contains the declaration of their type.

Unary and binary operator expressions are analyzed and annotated in a similar manner. Once the operand expressions are analyzed, the declarations of operators can be found in the current scope using the type names of the operand expressions and the operator symbol itself. In the hwc compiler,
this is implemented by joining the operand type names with the string representation of the operator to form the name of the type, e.g., “\texttt{int+int}”. The AST node of the operator expression is annotated with a reference to the return type of the operator.

Apart from type checking and type annotation, the compiler has to check whether variables are initialized unconditionally before they are used as rvalues. The \texttt{hwc} compiler implements this by maintaining records of all the initialized variables up to the current AST node inside the scope data structure. The matter of initialization can be addressed more elegantly in future revisions of the language by forcing the initialization of variables upon declaration, i.e., by dropping declarations and initialization of the form

\begin{verbatim}
int x;
x = 3;
\end{verbatim}

in favor of a syntax that requires immediate initialization:

\begin{verbatim}
int x = 3;
\end{verbatim}

4.6.4 Recursion Detection

Due to the nature of the target architecture, recursive programs are not permitted in the Julia language. Hence, the compiler must detect all forms of recursion and issue appropriate error messages.

Recursion detection can be implemented as follows. During the traversal of the AST, the compiler can annotate function declarations with a list of all functions that are called from the body of each function. These ‘callee lists’ form a directed call graph of the application. The call graph can be analyzed with a cycle detection algorithm, such as Algorithm 5.

\begin{algorithm}
\caption{A procedure for detecting cycles in directed graphs.}
\begin{algorithmic}[1]
\Procedure{HasCycle}{\textit{node}}
\If{\textit{node}.flagged} \Return{True} \EndIf
\State{\textit{node}.flagged = True}
\For{\texttt{1} to \texttt{k}}
\If{HasCycle{\textit{node}.callees[\texttt{k}]}} \Return{True} \EndIf
\EndFor
\State{\textit{node}.flagged = False}
\Return{False}
\EndProcedure
\end{algorithmic}
\end{algorithm}

Algorithm 5 visits each of the function declarations in the call graph depth-first, and ‘flags’ each of the nodes it visits. When a flagged node is encountered twice, a cycle is detected in the graph. After all callees of a node are visited, the flag is removed to ensure that the algorithm does not falsely detect recursion when the node is visited via an alternative but valid route. The algorithm can be optimized by caching results in each node after traversal of all its children, thus ensuring that each edge has to be visited only once.
4.7 Syntax Tree Optimization

The semantic analysis phase is followed by the first of two optimization phases. During this phase, the syntax tree is optimized using a variety conventional AST optimization techniques. We briefly describe a number of possible optimizations here:

**Constant folding** The simplification of constant expressions, e.g., replacing $12 + 3 \times 32$ with $108$.

**Dead code elimination** Disposal of blocks of code that are never executed. This includes e.g., the inlining of if statements with a conditional expression that has the constant value `true`.

**Common subexpression elimination** Replacing multiple identical computations with one computation and sharing the result, e.g., replacing $a = x + y; b = x + y; u = x + y; a = u; b = u;$. The hwc compiler performs constant folding only. Dead code elimination is performed only on conditional statements with a constant condition.

4.8 Execution Unit Transformation

After semantic analysis and AST optimization, the compiler transforms the semantically correct, annotated AST into an intermediate representation: the Execution Unit model. This transformation marks the most significant phase in the compilation process, as it marks the transition from the behavioral algorithm representation that is the source code to a purely structural representation, suitable for implementation in hardware. In this section, we describe the Execution Unit model and the process of extracting it from the AST.

4.8.1 The Execution Unit Model

![Figure 4.2: The basic structure of an Execution Unit.](image)

Execution Units are abstract representations of physical data processing devices. Figure 4.2 shows a schematic representation of a generic Execution Unit (specific unit types will be discussed later). The shown unit has a number of inputs and outputs, collectively called the ‘IO points’ of the unit, from which it can send or receive data. I/O points have data types (e.g., `int`, `bool`, etc.), which correspond to the types of the Julia language.

The I/O points of an Execution Unit have names. I/O point names are unique within a unit with
respect to their orientation (input or output), i.e., an input point may have the same name as an output point of the same unit, but no pair of input points (or output points) may have the same name.

The I/O points of an Execution Unit can be connected to the I/O points of other execution units to form complex systems. The transmission of data over such a connection is called an I/O event, or simply an event. An output point can drive multiple receivers. Input points can be driven by at most one sender.

An Execution Unit starts processing once it has received exactly one event on each of its input points. When processing is finished, the unit fires exactly one event from each of its outputs, and only then will it have to be ready to accept new input events.

4.8.2 The EU Hierarchy

The Execution Unit model consists of a collection of specialized classes of execution units, each of which possess the basic properties described in Section 4.8.1. What follows is a description of each of these classes.

Operator Units

The Operator Unit class (Figure 4.3) describes a processing component that performs an atomic computation on its input data. Atomic computations are those computations corresponding to the built in operators of the Julia language, as described in Section A.3.1.

Operator Units have a varying number of input points, depending on the arity of the implemented operation, and exactly one output point. The name of this output point is always return_value.

Composite Units

The Composite Unit class (Figure 4.4) is a container for a network of connected Execution Units. The inputs of the composite unit feed directly into the inputs of its children. Similarly, the outputs are aliases of the outputs of (some of) its children. The Composite Unit is used in the model in order to treat networks of units as a single entity during the compilation process. It is flattened during code generation, hence no performance or hardware penalty is incurred when it is used.

Conditional Units

The Conditional Unit class (Figure 4.5) describes a component that can perform one of two alternative operations on its input signals, based on the evaluation of a predicate. Conditional Units encapsulate three child units: one unit to evaluate the predicate (the ‘condition expression unit’), one unit to execute when the predicate succeeds (the ‘ifcode unit’), and one unit to execute when the predicate fails.
The condition expression child unit can be any Execution Unit with one or more data inputs. It must have exactly one output point, named return_value, of type bool. No restrictions apply to the ifcode and elsecode units.

A Condition Unit has a varying number of I/O points. The unit has one input point for each input of its child units. If two children have an input with the same name and type, e.g., the condition expression and the ifcode unit both have an input for the value $x$, the Condition Unit will have only one input for both values. In a similar manner, the set of output points of a Conditional Unit is the union of all outputs of the ifcode and elsecode units.

**Repetition Units**

The Repetition Unit class (Figure 4.6) describes a component that performs iterative computations on its input data. The component has two child units: a unit to evaluate the stop condition of the loop (the ‘condition expression unit’) and a unit that implements the computation that is performed on the input data (the ‘loop body unit’). The condition expression unit has at least one input and exactly one output of type bool, named ‘return_value’.

The Repetition Unit has a variable number of input points. The set of input points is the union of the input points of the two children of the unit. If both children have an input with the same name, they are considered the same element, and the Repetition Unit will have only one input with this name.
The output points of the Condition Unit correspond to the set of outputs of the loop body unit.

The Repetition Unit stores all its input data in registers. It then sends the registered data to the conditional expression unit, and sends the data to the loop body unit if the conditional expression unit returns `true`. The contents of the registers are updated with the output values of the loop body unit, and the process is repeated until the predicate fails. At this point, the content of the registers will be transmitted from the Repetition Unit’s output points.

Additional Unit Classes

Up until this point, we have discussed the most important classes of Execution Units in the model. Depending on the implementation of the language, the model can be extended with other useful unit classes. The `hwc` compiler defines, amongst others, the following additional units:

- **Identity Unit** A unit that propagates its input value unchanged.
- **Constant Unit** A unit that generates (events with) constant values.
- **Bitbucket Unit** A unit that consumes and discards input events.

Most of these units can be discarded during optimization, depending on the capability of the compiler to do so. Optimizations of the Execution Unit representation are discussed in Section 4.9.

4.8.3 AST to EU Transformation

The compiler transforms the annotated and semantically correct syntax tree into an Execution Unit network by visiting all nodes in the tree. During this traversal, all type declarations and variable declarations are ignored. For the remaining AST node types, the following transformations apply:

- **Function Declarations and Operator Declarations** Functions and user defined operators are transformed into Composite Units. The input points of the generated unit correspond to the function’s formal parameters, or the operator’s operands. The Composite has exactly one return value. Its type is the return type associated with the function or operator. Its name is always `return_value`. The bodies of functions and user defined operators are generated by transforming the statement list in the body to a Composite Unit, renaming the relevant output point of the statement list to `return_value`, and connecting any unconnected outputs to Bitbucket Units.
**Statement Lists**  The lists of sequential statements that comprise the bodies of functions, operator definitions, loops, or conditional statements are transformed into Composite Units, in the following manner:

1. An empty Composite Unit is instantiated (we call this the ‘list unit’).
2. Each individual statement in the list is transformed into an Execution Unit, and added to the list unit.
3. For each input point of an individual statement unit, an output with the same name is sought in the list of units of the preceding statements, starting at the end of the list. If an output is found, a connection is established. If not, the input is connected to an input of the list unit itself. If no such input exists, it is created.
4. When each of the individual statement units are generated, the list unit is finalized. During this stage, any unconnected outputs of the statement units are either connected to the output of the list unit, or to a Bitbucket Unit, in case the generated value is no longer live.

**Assignment Statements**  Assignment statements are transformed into Composite Units. This is accomplished by transforming the expression on the right hand side, and renaming the ‘return value’ output of the resulting Composite Unit to the name of the storage locator on the left hand side.

**Expressions**  Expressions are transformed into Composite Units. An expression containing unary and binary operators, constants, variables and function calls is traversed, creating a network of Operator Units that is isomorphic to the parse tree of the expression. Constant Units are generated and added to the expression unit for each constant. Input points are created in the expression unit for all the variables that are encountered. These are connected to the inputs of the Operator Units that require their values. Function calls are inlined. A Composite Unit that is generated by an expression always has exactly one output, named return value, whose type corresponds to the return type of the top level operator, variable or constant in the expression.

**Conditional Statements**  Conditional statements (if and if/else) are transformed into Conditional Units. The conditional expression, the if block and the optional else block are each transformed into individual Execution Units. These are then used to instantiate the Conditional Unit, as described in Section 4.8.2.

**Sequential Loop Statements**  The sequential loop statements (while and for) are transformed into Repetition Units. The loop condition expression and the loop body are each transformed according to the procedures described for expressions and statement lists, and used to instantiate the Repetition Unit, as described in 4.8.2.

**The parfor Statement**  The parallel loop statement is transformed into a Composite Unit. A duplicate Execution Unit is created for each value of the statically computable loop range variable. The values of all the variables that are used in the loop body are sent to all duplicates of the loop. It is assumed that different instances of the loop do not assign new values to the same variables (the loop instances are assumed to be independent.) The behavior of programs that violate this rule is undefined, but it is conceivable that a resolution policy is defined in future language updates to simulate a ‘concurrent write’ PRAM model (e.g., ‘write adding’ or ‘write any’).
4.9 Execution Unit Optimization

After the Execution Unit transformation phase, the program is represented as a list of execution units. Each unit in this list is a Composite Unit (i.e., a network of units) that contains the implementation of a function or user defined operator. At this point, the compiler may perform a second series of optimizations. The optimizations on the Execution Unit representation are largely beyond the scope of this work, but they are critical to future efforts. Hence, we briefly describe possible optimizations here.

**Combinational Grouping** The Execution Unit transformation generates Execution Units for each operator and for each assignment statement in a program. During code generation, state machines are created that perform the non-phase handshake on each connection that exists between two execution units. Hence, an expression of the form \((a + b) + (c + d)\) causes three Execution Units to be generated, each of which has two input points and one output point, resulting in a total of six input capture state machines, three output transmission state machines, and three adders. For statements such as these, it is more economical to generate a single Execution Unit, containing four input points, three adders, and one output point. More importantly, by eliminating the handshakes between subexpressions, the execution time is reduced significantly. **Combinational Grouping** is the process of identifying subnetworks of execution units that are suitable for implementation in a single combinational circuit. We think that the development and implementation of combinational grouping is the most important next step in Julia development.

**Elimination of unnecessary Execution Units** During the Execution Unit transformation, the analysis of an expression must always return an Execution Unit, even if the expression does not perform any actual work. e.g., consider the statement \(x = y\), which causes the compiler to generate a unit that captures the value of \(y\) and outputs the new value \(x\). Rather than generating an actual physical unit that propagates the value of \(y\) untransformed, the compiler should replace connections to the value \(x\) with a direct connection to \(y\). Similarly, binary operators with one constant operand should be implemented in a single combined unit, as opposed to generating a separate Constant Unit and Operator Unit.

**Elimination of unnecessary output points** The Execution Unit transformation generates output points for some Execution Units that remain unused throughout the rest of the computation. The approach of the hwc compiler is to connect Bitbucket Units to these outputs, to capture the outgoing I/O events on these output points, to ensure progress of the computation. A mature compiler should analyze the Execution Unit network, remove Bitbucket Units, and recursively discard the output points from the Execution Units to which they are connected.

4.10 Code Generation

So far, we established a body of techniques for the transformation of a behavioral Julia program to an equivalent structural representation in the EXECUTION UNIT framework. In this section, we describe the final stage of the compilation process: the generation of synthesizable VHDL code for each of the classes in the Execution Unit hierarchy.
4.10.1 The Structure of the Generated Code

There is a close resemblance between the units of the Execution Unit framework and the entity construct of the VHDL language. Both are abstract representations of a physical processing component that can receive and transmit data through typed I/O points. The most obvious way to represent execution units in VHDL is to generate one entity for each unit, and to generate component instantiations for any child units that the unit may have. The generation of a separate entity for each execution unit is however not strictly necessary. An alternative approach is to generate a single entity for top level units (functions, user defined operators), and to declare the I/O signals, processes and combinational code of all child units inside the top level architecture. We call these two approaches ‘hierarchical code generation’ and ‘flat code generation’.

The primary benefit of hierarchical code generation is that it results in output code that is easier to read and debug. Unfortunately, it also has many disadvantages. Firstly, hierarchical code is approximately ten times the size of flat code. This is a significant problem, as designs can easily become quite large. A second disadvantage manifests itself when code has to be generated for deeply nested Composite Units. If an output of a composite unit drives multiple receivers, it will be necessary to generate a fan-out component of some kind. If composites are nested, it may occur that an event passes several fan-out components, resulting in unnecessary delays. When generating flat code, it is straightforward to connect inputs directly to their signal source, resulting in more efficient hardware usage and faster code. Hence, we have chosen for flat code generation in the hwc compiler.

4.10.2 Synthesis of the Execution Units

The generation of synthesizable VHDL code for each of the Execution Units in the analyzed program proceeds in three phases: generation of code for the input points, generation of code for the output points, and generation of code for the internal functionality of the unit. Note that these phases do not apply to Composite Units, because these are flattened/inline during code generation.

Input Point Code Generation For each input point in an Execution Unit, the compiler generates DATA, AVAIL, and ACK signals. These are used to perform the four phase handshake, as described in Section 3.1. In addition, a small state machine is generated that implements the handshake protocol for each input point. The state machine buffers the data value that is received during the input transaction in a register. It then sets a flag to indicate that the buffer is full. The logical AND of the BUFFER_FULL bits of all input state machines forms a signal called ALL_INPUT_BUFFERS_FULL, which indicates to the interior of the Execution Unit that an input event has been received on all inputs. When an input state machine has a full buffer, it waits for a rising edge on another internal control signal, the ALL_INPUT_CONSUMED signal. This signal is driven by the interior of the Execution Unit, and it indicates that the input state machines may clear their BUFFER_FULL bits, and accept a new input event.

Output Point Code Generation The generated code for the output points of an execution unit is smaller than for the input points. For each output point, there exists again a DATA, AVAIL, and an ACK signal, but they are driven by one single state machine for all output points. This state machine waits for a rising edge on the ALL_OUTPUT_COMPUTED signal, which is driven by the interior of the unit. It indicates that the output state machine may initialize the four phase handshake. Because this state machine performs the handshake with multiple receivers simultaneously, it uses the AND and
the OR of all the individual ACK signals to perform the handshake. When the output transmission is complete, the state machine drives the ALL_OUTPUT_SENT signal high for one clock cycle. This indicates to the interior of the Execution Unit (which is in the same clock domain!) that all output has been transmitted.

**Interior Code Generation**  The generation of code for the implementation of the internal functionality of Execution Units differs for each class of the EU hierarchy. The simplest types of units are Operator Units, Constant Units, and Identity Units. These units have no stateful interior, i.e., their functionality can be implemented with combinational code. For these units, it is sufficient to connect the ALL_INPUT_BUFFERS_FULL and ALL_INPUT_CONSUMED signals of the input stage to the ALL_OUTPUT_COMPUTED and ALL_OUTPUT_SENT signals of the output transmission stage, and to generate signal assignments or combinational code to connect the input buffers to the output DATA signals. For Repetition Units and Conditional Units, the compiler generates a simple state machine that waits for all input to arrive, sends the data to the child unit that implements the condition, awaits the result, sends the data to the child unit that implements the loop body or conditional code block, waits for the result, and signals the output stage that all output is computed.

### 4.11 Chapter Summary

In this chapter, we presented ‘Julia’, an imperative programming language for FPGA-based CCM design. The language was designed to simplify the process of developing efficient CCMs. As such, we have focused on lean constructs and compilation techniques, and avoided overly powerful abstractions. We described the compilation process of the Julia language, using hwc, our proof-of-concept implementation, as a running example, and we suggested areas for further improvement in future versions of the language and the compiler. In the next chapter, we will evaluate the performance of Julia programs.
Chapter 5

Performance Evaluation

In this chapter, we evaluate the performance and hardware usage of FPGA-based CCMs created with the Julia language. We present static analysis methods for the calculation of runtimes and for the estimation of hardware usage, and we validate these calculations using \texttt{hwc}, our proof-of-concept implementation of the language. We compare the performance of Julia-generated CCMs to conventional software implementations and manually designed CCMs, and we conclude this chapter with an experiment that shows the potential of more advanced implementations of the Julia language.

5.1 The Test Setup

Unless stated otherwise, the timing measurements for the runtime experiments have been obtained through the following procedure:

1. The Julia program was compiled to a VHDL description using \texttt{hwc}, our prototype compiler.

2. The VHDL code was compiled and simulated on a workstation PC using \texttt{GHDL}, an open source compiler and simulator for VHDL designs, which resulted in an exact count of the number of clock cycles in which the algorithm reached completion.

3. The VHDL code was synthesized for a Xilinx Spartan-3 XC3S1000 FPGA, using Xilinx Inc.’s gratis proprietary synthesis tool \texttt{XST}, to obtain the maximum clock frequency at which the design would operate correctly.

4. The synthesized design was uploaded and executed on a Spartan-3 to verify correct operation of the design in actual hardware.

Throughout our experiments, the \texttt{XST} synthesis tool reported a maximum clock frequency of 94.976 MHz for all compiled Julia programs. Due to the asynchronous nature of the emitted code, we expect the maximum clock frequency to be constant, regardless of the size of the design. Hence, all timing figures were obtained by dividing the observed number of clock cycles from the GHDL simulation by 94.976 MHz.

For comparative performance experiments, a workstation with an AMD Sempron CPU was used, running at 1840.027 MHz. The algorithms were written in C. Whenever possible, source code has been shared between C and Julia implementations. The C code was compiled with \texttt{gcc} version 3.3.6 using \texttt{-O2} optimization. All experiments were performed under the Debian GNU/Linux operating
Timing information for the workstation implementations was obtained by reading the time stamp counter register before and after execution of the compiled function, to obtain the number of elapsed clock cycles. This process was repeated 1000 times for each algorithm, to compensate for cold caches. The minimum of all observations was divided by the CPU clock frequency to obtain the final result.

5.2 Runtime Calculation

5.2.1 Calculating the Runtime of Julia Programs

With an understanding of the compilation process and the Execution Unit model, it is straightforward to calculate the runtime of a Julia function through inspection of its source code and input data, provided that the number of iterations of each loop can be determined statically. For these calculations, we assume that there is a constant, model specific clock frequency $f$, at which all Execution Units operate. In addition, the following implementation-dependent constants are used in the runtime calculations:

- $T_c$: The overhead of capturing and buffering an input event and transmitting an output event, in clock cycles.
- $T_{rs}$: The startup overhead of a Repetition Unit, in clock cycles.
- $T_{ri}$: The overhead per iteration of a Repetition Unit, in clock cycles.
- $T_{cu}$: The startup overhead of a Conditional Unit, in clock cycles.

Assuming that no further optimization is performed, the runtimes of the various constructs of the Julia language are calculated as described below.

**Built-in Operators, Expressions and Assignment Statements**  Built-in operators are implemented as combinational circuits, contained in an Execution Unit shell that captures its operands and transmits the calculated result. Hence, the runtime of an Operator Unit is exactly $T_c$. The runtime of an expression composed of built-in operators is determined by the longest sequence of operators in the parse tree of the expression.

Consider the expression $(a + b) - - (c + d)$. The longest sequence of operators is encountered on the right hand side of the top level binary $-$ operator, where the data path includes the binary $+$ of the subexpression $c + d$, the unary $-$ and the top level binary $-$ itself. Hence, the execution time of this expression is $3T_c/f$. The runtime of an assignment statement is exactly the runtime of the expression on the right hand side of the assignment symbol.

**Statement Lists**  The runtime of a statement list, such as the body of a function or a loop, is determined by the runtimes of the individual statements in the list. Since independent statements are executed concurrently, whereas dependent statements are executed sequentially, the total runtime of a statement list is the maximum of the runtimes of all concurrent execution paths.

Consider the following fragment:
The code snippet defines:

\[
\begin{align*}
  x &= a + b \\
  y &= x + 1 \\
  z &= u + v
\end{align*}
\]

In this example, the third assignment statement can be executed independently from the first two statements, whereas the second statement depends on the first. The compiler generates two concurrent execution paths, equivalent to the expressions \((a + b) + 1\) and \((u + v)\), with respective runtimes of \(2T_c/f\) and \(T_c/f\). Hence, the concurrent execution path that defines the total runtime of this code has an execution time of \(2T_c/f\).

**Conditional Statements**

The runtime of a conditional statement depends on five factors:

1. The startup overhead of the Conditional Unit, \(T_{cu}\).
2. The input capture and output transmission overhead, \(T_c\).
3. The runtime of the conditional expression, \(T_{ce}\).
4. The result \(r\) of the evaluated conditional expression.
5. The runtimes of the \texttt{if} and \texttt{else} code, \(T_{if}\) and \(T_{else}\).

The runtime of a conditional statement is given by:

\[
T = T_{cu} + T_{ce} + T_c + \begin{cases} 
T_{if}, & r = true \\ 
T_{else}, & r = false 
\end{cases}
\]  

(5.1)

**Sequential Loop Statements**

The runtime of a \texttt{while}-loop depends on the following factors:

1. The startup cost of the Repetition Unit, \(T_{rs}\).
2. The control overhead per iteration of the Repetition Unit, \(T_{ri}\).
3. The overhead associated with capturing input data and transmitting output data, \(T_c\).
4. The runtime of the conditional expression, \(T_{ce}\).
5. The number of iterations of the loop, \(N\).
6. The runtime of the \(k\)-th iteration of the loop body, \(t_b(k)\).

The runtime of a \texttt{while} loop is given by:

\[
T = T_{rs} + T_c + (N + 1)T_{ce} + NT_{ri} + \sum_{k=1}^{N} t_b(k)
\]  

(5.2)

The runtime of the \texttt{for} loop is calculated by considering the functionally equivalent reformulation in terms of the \texttt{while} loop.
The `parfor` Statement  

The `parfor` statement of the Julia language is expanded into a sequence of mutually independent statements. The runtime of the construct is determined as for a normal statement list with mutually independent statements. E.g. consider a `parfor` loop with $N$ concurrent loop instances. Let $t_1, \ldots, t_N$ denote the runtimes of the individual loop instances. The runtime of the `parfor` statement then equals $\max_k(t_k)$.

Functions and User-Defined Operators  

The runtime of a function or user-defined operator is fully determined by the runtime of the statement list in the body of its declaration. Input parameter values that are used more than once have to be distributed to multiple sources inside the function body, as the signal interface of a compiled function allows only 1-to-1 handshakes. Hence, the compiler will introduce an Identity Unit for such parameters. In terms of runtime, this is equivalent to adding an additional assignment statement to the start of the function body:

```c
int func( int x )
{
    x = x + 0;  // Compiler generated identity.
    ...  
}
```

After insertion of identities, the runtime of a function or user-defined operator is calculated as described under statement lists.

5.2.2 Validation of the Runtime Calculation Model

In this section, we present the results of a series of runtime experiments with `hwc`, our proof-of-concept implementation of the Julia language. The purpose of these experiments is to validate the runtime calculation model, as described in Section 5.2.1. In addition, we will determine actual values of the compiler specific symbolic constants for the `hwc` compiler.

Expression Growth

First, we will investigate the runtime of expressions in the Julia language by evaluating programs of the form:

```c
int main( int A, int B, int C, ... )
{
    return EXPRESSION;
}
```

where `EXPRESSION` is an expression composed of the input variables. Recall from Section 5.2.1 that the runtime calculation model predicts that the execution time of the expression is a linear function of the depth of the parse tree. Specifically:

$$T_{expr} = dT_c$$

where $d$ is the depth of the tree and $T_c$ is the overhead due to capturing Execution Unit input and transmitting output. Table 5.1 shows the observed runtimes, in clock cycles and nanoseconds, of expressions of the form

```
A + B + C + D + ...
```

i.e., integer sums with exactly $k$ terms. Due to the fact that `hwc` does not insert brackets automatically, it is expected that the runtime of the expression grows with $T_c$ cycles ($T_c/f$ seconds) for each addition operator. Table 5.1 shows the observed runtimes for `hwc`, which support the predicted growth.
It can be inferred from the observed results that the compiler-dependent constant $T_c$, the communication overhead of an Execution Unit, equals 4 clock cycles for the hwc compiler. It should be noted that each of the measured times contains a program startup overhead of four clock cycles, due to initialization of the test harness.

The runtime of expressions can be improved dramatically by placing brackets to reduce the depth of the parse tree. An expression of the form
\[
( (A + B) + (C + D) ) + ( (E + F) + (G + H) )
\]
is predicted to grow logarithmically with respect to the number of terms. Hence, the runtime of a properly balanced expression is expected to be considerably shorter than the same expression evaluated on a Von Neumann machine, if the expression is sufficiently large.

### Table 5.1: Runtimes of unbalanced integer sum expressions.

<table>
<thead>
<tr>
<th>Terms</th>
<th>Clock cycles</th>
<th>Nanoseconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>84.231</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>126.34</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>168.46</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>210.57</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>252.69</td>
</tr>
<tr>
<td>7</td>
<td>28</td>
<td>294.81</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>336.92</td>
</tr>
</tbody>
</table>

### Table 5.2: Runtimes of balanced integer sum expressions.

<table>
<thead>
<tr>
<th>Terms</th>
<th>FPGA Runtime (ns)</th>
<th>CPU Runtime (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>84.231</td>
<td>14.673</td>
</tr>
<tr>
<td>4</td>
<td>126.34</td>
<td>15.760</td>
</tr>
<tr>
<td>8</td>
<td>168.46</td>
<td>17.934</td>
</tr>
<tr>
<td>16</td>
<td>210.57</td>
<td>21.738</td>
</tr>
<tr>
<td>32</td>
<td>252.69</td>
<td>29.347</td>
</tr>
<tr>
<td>64</td>
<td>294.81</td>
<td>46.194</td>
</tr>
<tr>
<td>128</td>
<td>336.92</td>
<td>80.977</td>
</tr>
<tr>
<td>256</td>
<td>379.04</td>
<td>150.54</td>
</tr>
<tr>
<td>512</td>
<td>421.15</td>
<td>289.66</td>
</tr>
<tr>
<td>1024</td>
<td>463.27</td>
<td>576.62</td>
</tr>
<tr>
<td>2048</td>
<td>505.39</td>
<td>1127.7</td>
</tr>
<tr>
<td>4096</td>
<td>547.50</td>
<td>2240.7</td>
</tr>
<tr>
<td>8192</td>
<td>589.62</td>
<td>4466.8</td>
</tr>
<tr>
<td>16384</td>
<td>631.73</td>
<td>27606</td>
</tr>
<tr>
<td>32768</td>
<td>673.85</td>
<td>65112</td>
</tr>
</tbody>
</table>

Table 5.2 shows the results of evaluating properly balanced integer sums with the hwc compiler, and
with a workstation CPU. The data confirms the predictions, i.e., linear growth of the PC runtime and logarithmic growth for the Julia code. The data of Table 5.2 are plotted in Figure 5.1. The discontinuity in the CPU curve is due to cache misses that occur when the array of sum terms grows too large.

![Figure 5.1: Runtimes of balanced integer sum expressions.](image)

**Loops and Conditional Statements**

Next, we will investigate the runtime of loops and conditional statements (if and if/else statements). We will determine the actual values of the remaining compiler specific constants of the runtime model for the hwc compiler. Consider the following test program:

```c
int main( int x, int y )
{
    int total;
    total = 0;
    while( x > 0 )
    {
        total = total + y;
        x--;  
    }
    return total;
}
```

We have from Equation 5.2 and from the observations in Section 5.2.2 that the runtime cost (in clock cycles) of the loop will be composed as follows:

\[ T_l = T_{rs} + 4 + (N + 1)4 + NT_{ri} + \sum_{k=1}^{N} 4 \Rightarrow \]

\[ T_l = T_{rs} + (N + 1)8 + NT_{ri} \]
The compiler-dependent constant $T_{rs}$ is determined by supplying $x = 0$ as input, i.e., evaluating the loop $N = 0$ times. The observed runtime is 18 clock cycles, hence, considering the test harness startup overhead of 4 cycles, $T_{rs} = 6$ clock cycles for the hwc compiler. For the input $x = 1$, i.e., one iteration of the loop, the observed runtime is 34 clock cycles. Hence, $T_{ri} + 8 = 16$ gives rise to an overhead per iteration of $T_{ri} = 8$ clock cycles for the hwc compiler.

Similarly, the control overhead $T_{cu}$ of a conditional statement is determined by replacing the loop with an appropriate if statement. The observed runtime of the obtained program yields a value of $T_{cu} = 9$ clock cycles for the hwc compiler.

**Validation**

Now that we calibrated the runtime calculation model for hwc, we can validate the model by predicting and measuring the runtime of a simple, but nontrivial program. To this end, we study a Julia implementation of the Collatz verification routine of Section 3.1:

```c
int collatz(int n, int b) {
    int iterations;
    int result;
    result = 0;
    iterations = 0;
    while ( (n > 1) && (iterations < b) ) {
        if ( (n & 1) == 1 ) n = 3 * n + 1;
        else n = n >> 1;
        iterations++;
    }
    if ( n == 1 ) result = iterations;
    return result;
}
```

Consider the input $n = 5, b = 100$. This leads to the sequence $(5, 16, 8, 4, 2, 1)$, i.e., a return value of 5. The variable declarations and the initializations have zero execution time. The while loop is executed five times, the condition is evaluated six times. The runtime of the loop body is dominated by the if statement. The if statement outside the loop is executed after the loop, due to the data dependency it has on the loop.

The reader is invited to verify that the total execution time $T_i$ of the loop is given by

$$t_b(k) = T_{cu} + 3T_c + \begin{cases} 
2T_c, & r = 1 \\
T_c, & r > 1
\end{cases} \Rightarrow$$

$$T_l = T_{rs} + T_c + 6T_{ce} + 5T_{ri} + \sum_{k=1}^{5} t_b(k) \Rightarrow$$

$$T_l = 227 \quad (5.5)$$

The total execution time of the outer if statement is given by

$$T_{if} = T_{cu} + 3T_c \Rightarrow$$

$$T_{if} = 21 \quad (5.6)$$

48
which gives $T = T_i + T_{if} = 248$ clock cycles for the total execution time of the algorithm for the given input. The observed execution time was 252 clock cycles, which confirms the calculated execution time, considering the constant simulation startup overhead of four clock cycles, due to our test harness.

We have now validated the runtime calculation model for all constructs supported by the hwc compiler. While it is impractical to perform exact calculations for very large programs, we trust that the reader understands that the model can be leveraged to perform complexity analysis for such programs. An exact, compiler assisted analysis is possible for large programs that allow static branch prediction.

### 5.2.3 Summary

In Section 5.2.1, we presented a method for determining the runtime of a Julia algorithm through static analysis of the source code. The method is applicable if the number of iterations of each loop in the program can be determined statically. The calculation method yields an expression for the runtime, which depends on four compiler-dependent constants and one hardware dependent constant.

In Section 5.2.2, we validated the runtime calculation method using hwc, our proof-of-concept implementation of the Julia language. The compiler-dependent constants found for hwc are summarized in Table 5.3.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_c$</td>
<td>4</td>
</tr>
<tr>
<td>$T_{rs}$</td>
<td>6</td>
</tr>
<tr>
<td>$T_{ri}$</td>
<td>8</td>
</tr>
<tr>
<td>$T_{cu}$</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 5.3: Runtime constants for the hwc compiler.

We have shown that Julia exhibits the following properties:

- Julia exploits implicit statement level parallelism.
- Julia exploits fine-grained parallelism at the (sub)expression level.
- For sufficiently simple programs, the runtime of Julia programs is exactly predictable by the programmer.
- For larger programs, the Julia language allows the programmer to analyze the runtime complexity.

### 5.3 Hardware Usage

In this section, we present a model for the estimation of the circuit size of compiled Julia designs.
### 5.3.1 Estimating Hardware Costs

The hardware usage of a Julia program can be estimated by counting the number of state machines, registers, and combinational circuits that are inferred from each statement.

We define the following symbols for the hardware estimation model:

- $S_{\text{in}}$, a state machine that implements the input capture protocol for a single Execution Unit input point.
- $S_{\text{out}}$, a state machine that implements the data transmission protocol for all output points of an Execution Unit.
- $S_r$, a state machine that implements the control flow coordination mechanism of a Repetition Unit.
- $S_c$, a state machine that implements the control flow mechanism of a Conditional Unit.
- $R(k)$, a $k$-bit register.
- $A(k)$, a $k$-bit binary adder/subtractor.
- $M(k)$, a $k$-bit multiplier.

Due to extensive, and relatively unpredictable, optimizations of the generated VHDL code during final synthesis, it is neither possible nor meaningful to obtain exact sizes for these state machines and registers in terms of configurable logic blocks. Hence, our primary goal here is to establish an intuitive view on the growth and cost of Julia programs, rather than an exact quantification.

**Built-in Operators** For the built-in operators of the Julia language, the Julia compiler generates an Execution Unit consisting of the following components:

- An input capture state machine for each operand.
- An input buffer (register) for each operand.
- A combinational circuit (adder, etc.) or a multiplier.
- One output transmission state machine.

Hence, for an $n$-ary operator with $k$-bit operands, the cost of a single operator is given by

$$C = n(S_{\text{in}} + R(k)) + S_{\text{out}} + U(k)$$

(5.7)

where $U(k)$ is $A(k)$ or $M(k)$, for addition/subtraction or multiplication respectively.

**Conditional Statements** A compiled conditional statement consists of the following components:

- An input capture state machine for each unique storage locator that is used as an rvalue inside the condition expression, the `if`-block or the `else` block, except for storage locators that are declared inside either block.
• A buffer (register) for each externally declared storage locator that is used as an lvalue and/or an rvalue inside the condition or one of the code blocks.

• One output transmission state machine.

• One control flow coordination state machine.

• An implementation of the conditional expression.

• An implementation of the if code block.

• An implementation of the else code block.

Hence, the for the following (contrived) example

```c
if( x > 0 )
{
    int a;
    a = 3;
    y = a;
} else y = z;
```

the hardware cost equals \( S_c + S_{out} + 2S_{in} + 3R(32) \), in addition to the cost of the conditional expression and the statement blocks. The term \( 2S_{in} \) is due to the rvalues \( x \) and \( z \), and the term \( 3R(32) \) is due to the use of \( x \), \( y \), and \( z \).

**Sequential Loops** A compiled while loop consists of the following components:

• An input capture state machine for each unique storage locator that is used as an rvalue inside the condition expression or the loop body, except for storage locators that are declared inside the loop body itself.

• A buffer (register) for each storage locator that is used as an lvalue and/or as an rvalue inside the loop or the condition.

• One output transmission state machine.

• One control flow coordination state machine.

• An implementation of the conditional expression.

• An implementation of the loop body.

I.e. the cost of a loop is composed of an overhead \( S_r + S_{out} \), the cost of capturing and buffering the variables (as calculated under **Conditional Statements**), the cost of the conditional expression, and the cost of the loop body.

**Other Constructs** All remaining costs are computed trivially using the results established so far. The cost of an expression is equal to the sum of the costs of the operators in the expression. The cost of an assignment statement is equal to the cost of the expression on its right hand side. The cost of a statement list is equal to the sum of the costs of the individual statements. The cost of a function or user-defined operator is equal to the cost of the statement list in the body of the definition. Finally, the cost of a parfor loop is the sum of the costs of the individual loop instances.
5.4 Julia Performance in Practice

In this section, we evaluate the performance of Julia using a number of practical examples. We will use our prototype implementation of the language to perform measurements on actual compiled examples, and we will provide some estimates of what we believe can be achieved with a more mature implementation.

5.4.1 Binary GCD

In this first experiment, we evaluate the runtime of a Julia implementation of the ‘Binary Greatest Common Divisor’ algorithm. The Julia implementation is given below. Note that it is equally valid C99 code.

```c
int GCD( int a, int b )
{
    int c;
    int e;
    c = a | b;
    e = 0;

    while ( (c & 1) == 0 )
    {
        c = c >> 1;
        e = e + 1;
    }

    int p;
    p = e;

    while( p > 0 )
    {
        a = a >> 1;
        b = b >> 1;
        p--;
    }

    while ( (a & 1) == 0 ) a = a >> 1;
    while ( (b & 1) == 0 ) b = b >> 1;

    while ( a != b )
    {
        if ( a > b )
        {
            int t;
            t = b;
            b = a;
            a = t;
        }

        b = b - a;
        while ( (b & 1) == 0 ) b = b >> 1;
    }

    p = e;
    while( p > 0 )
    {
        a = a << 1;
        p--;
    }

    return a;
}
```
The Binary GCD program is a sequential algorithm, hence we expect the asymptotic behavior of this code to differ from that of a conventional software implementation by no more than a constant factor. Due to the higher clock frequency of the CPU, we expect the software implementation to be faster.

The runtime of the GCD algorithm reaches its maximum when successive values of the Fibonacci sequence are used as input. Figure 5.2 shows the runtimes of the C99 and Julia implementations for these inputs, as a function of input size. The runtime of both implementations grows linearly. This is confirmed by Figure 5.3, which shows the ratio between the runtimes of both implementations.

5.4.2 Transitive Closure

In the second experiment, we consider three implementations of the Transitive Closure algorithm, as described in Section 3.2: a Julia implementation, a C implementation and a manual VHDL implementation, created using the method described in Chapter 3.

The Julia and VHDL implementations accept a fixed size incidence matrix, hence recompilation was necessary for different problem sizes, but not for different instances. Due to current limitations of the hwc compiler, we had to calculate the runtime of the Julia implementation using the calculation method of 5.2.1 for larger input graphs. We verified these calculations using hwc generated code for all graph sizes up to and including 16 vertices.

Table 5.4.2 shows the runtimes of the TC algorithm for input graphs of different sizes. Due to the fast parallel matrix multiplication circuit, the Julia implementation is significantly faster than the C implementation for larger graphs, despite the lower clock frequency of the FPGA compared to the workstation CPU. Note that the work complexity of the TC algorithm, described in Section 3.2, is
Figure 5.3: Ratio between C and Julia runtimes for the GCD algorithm.

Table 5.4: Performance results for the Transitive Closure algorithm

<table>
<thead>
<tr>
<th>Vertices</th>
<th>C (nsec)</th>
<th>Julia (nsec)</th>
<th>VHDL (nsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>85.87</td>
<td>610.68</td>
<td>189.52</td>
</tr>
<tr>
<td>4</td>
<td>480.97</td>
<td>947.60</td>
<td>284.28</td>
</tr>
<tr>
<td>8</td>
<td>3103.76</td>
<td>1368.76</td>
<td>379.04</td>
</tr>
<tr>
<td>16</td>
<td>25292.56</td>
<td>1874.15</td>
<td>473.80</td>
</tr>
<tr>
<td>32</td>
<td>191648.27</td>
<td>2463.78</td>
<td>568.56</td>
</tr>
<tr>
<td>64</td>
<td>1494959.04</td>
<td>3137.63</td>
<td>663.33</td>
</tr>
<tr>
<td>128</td>
<td>11832881.80</td>
<td>3895.72</td>
<td>758.09</td>
</tr>
<tr>
<td>256</td>
<td>94278639.39</td>
<td>4738.03</td>
<td>852.85</td>
</tr>
<tr>
<td>512</td>
<td>754827829.70</td>
<td>5664.58</td>
<td>947.61</td>
</tr>
<tr>
<td>1024</td>
<td>6023629252.72</td>
<td>6675.37</td>
<td>1042.37</td>
</tr>
</tbody>
</table>

$$\Theta(k^3 \log k),$$ and its time complexity is $$\Theta(\log^2 k)$$. Hence, we expect the speedup of the parallel Julia implementation to be $$\Theta(k^3 \log k / \log^2 k) = \Theta(k^3 / \log k),$$ as compared to the sequential C implementation. Figure 5.4 shows the observed speedups plotted over a fitted cubic curve, confirming the expectations.

In Section 3.2, we described how the matrix multiplication circuit can be unrolled into a longer combinational circuit, i.e., without handshakes between the different stages of the multiplication process. The hwc compiler is not yet capable of performing this type of optimization, but it was implemented in the manually designed VHDL version. This accounts for the significant speed difference between
the VHDL version and the Julia version. We believe that it is possible to perform optimizations like this automatically in future versions of the Julia compiler.

It should be noted that, contrary to appearance, the performance of the VHDL implementation is worse than logarithmic with respect to the problem size. As circuits grow larger, the signal path length of the combinational matrix multiplier will grow, and it will not be possible to run the designs at the same high clock frequency as was done here (94.976 MHz). Due to the limited size of the FPGAs at our disposal, we were not able to observe this degradation ourselves.

5.4.3 Satisfiability

In this final experiment, we seek to demonstrate the potential of the Julia approach by studying the performance of the Boolean Satisfiability solver, described in Section 3.3. Compilation of the algorithm is beyond the current capabilities of the \texttt{hwc} compiler, but we estimate that basic compilation of the algorithm, i.e., without combinational grouping optimization (Section 4.9) can be achieved within three man-months of development work on the \texttt{hwc} compiler. Specifically, we need the following missing features, all of which are straightforward to implement:

- The \texttt{parfor} loop.
- Enumerated types.
- The \texttt{switch/case} statement (a generalization of the \texttt{if/else} statement).
- User defined operators.
- Function calls.
Here, we present the results of runtime experiments with a manual transcription of the algorithm, that features combinational circuits for formula simplification and branching. We estimate that the implementation of combinational grouping in the 	exttt{hwc} compiler is an effort of approximately six man-months. With such a compiler, it will be possible to generate a design that is near identical to our manual implementation. Again, due to the space limitations on the FPGAs at our disposal, we simulated the SAT solver in software for larger instances. Results were verified with instances up to approximately 60 variables and 250 clauses.

**Hardware vs. Software**

We have conducted two separate experiments with the solver. In the first experiment, we compared the runtime of a C implementation of the DPLL algorithm to that of the hardware solver for a set of uniformly random 3-SAT instances of increasing size, generated with 	exttt{mkcnf}, a commonly used SAT problem generator. The ratio of the number of variables to the number of clauses was set at 4.27. At this ratio, random 3-SAT instances are on average the most difficult to solve. The C implementation and the VHDL implementation performed identical searches, i.e., the search space was explored in exactly the same branching order.

![Average speedup vs. variables](image)

**Figure 5.5**: Speedup (hardware vs. software) of the DPLL SAT algorithm.

Figure 5.5 shows the observed speedups of the VHDL implementation, as compared to the C implementation. It can be seen that the VHDL implementation achieves considerable speedups (a factor 100 to 700, depending on problem size) with respect to the C implementation. This is due to the fact that the application of simplification rules (the inner loop of the algorithm) is performed in hardware in $O(1)$ time, as opposed to the $O(n)$ performance of the C implementation. Further, the speedup
grows as problem instances get bigger. The local peaks in the speedup graph (i.e., at 40 variables) are caused by unsatisfiable instances, which have a longer overall runtime.

### Hardware vs. Advanced Algorithms

In the second SAT experiment, we compared the performance of the hardware solver to two top of the line DPLL based SAT solvers: march [16] and zChaff [19]. As stated in Chapter 3, these solvers perform considerably better than the ‘classic’ DPLL algorithm, due to the use of better branching heuristics. Both our C solver and the hardware solver do not use such heuristics.

For this experiment, we solved two specially constructed SAT instances, based on the ‘Knights Tour’ problem: finding a sequence of knight moves on an $n$ by $m$ chess board, such that each square is visited exactly once. We used two unsatisfiable instances, for the 4 by 5 chessboard and the 5 by 5 chessboard. Both experiments were conducted with four different solvers in total:

- **csolve**: a naive C implementation of the classic DPLL algorithm.
- **ccmsolve**: our hardware solver.
- **march**: an advanced lookahead solver.
- **zChaff**: an advanced conflict driven solver.

The results of these experiments are shown in Table 5.5. The runtimes are given in seconds. Due to the long runtimes of the software solvers, the experiments were conducted only once.

<table>
<thead>
<tr>
<th>Instance</th>
<th>csolve</th>
<th>ccmsolve</th>
<th>march</th>
<th>zChaff</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 by 5</td>
<td>1765</td>
<td>0.0526</td>
<td>485</td>
<td>2.264</td>
</tr>
<tr>
<td>5 by 5</td>
<td>200406</td>
<td>2.5668</td>
<td>1664</td>
<td>243</td>
</tr>
</tbody>
</table>

Table 5.5: Performance results for the Knights Tour SAT problem. Times are in seconds.

Remarkably, Table 5.5 shows that the acceleration of the simplification routines in hardware can yield considerable gains, even in comparison to modern SAT algorithms. For the largest of the two instances, the difference between the fastest software solver (zChaff) and the hardware solver is a factor 94 in favor of the hardware solver. Equally remarkable is the difference between the hardware solver and the functionally identical C implementation: 2.6 seconds versus 55 hours.

### 5.5 Chapter Summary

In this chapter, we evaluated the performance of Julia programs using a combination of static analysis techniques and runtime experiments, performed with the prototype compiler hwc. Using a parallel implementation of the Transitive Closure algorithm, we have shown that Julia is capable of achieving the theoretically calculated speedup of a parallel algorithm in practice. Using a manual transcription of the DPLL algorithm for the Boolean Satisfiability problem, we demonstrated the potential of future
incarnations of the Julia language, by showing that speedups of several orders of magnitude can be achieved over conventional software implementations.
Chapter 6

Conclusions and Future Work

6.1 Summary and Conclusions

In Chapter 1 we posed that FPGA-based Custom Computing Machines have the potential to outperform conventional software implementations of certain algorithms by several orders of magnitude. We stated that there is no established systematic approach to designing such CCMs, given a conventional description of an algorithm, and that current efforts to generate such devices using a conventional programming language and compiler suffer from a number of problems.

In Chapter 2, we surveyed the broader field of Reconfigurable Computing, to which FPGA-based CCMs belong, and we highlighted some of the problems in current automated design efforts.

In the remainder of this thesis, we explored an alternative approach to the development of an automated CCM design system. Rather than finding ways to compile a set of language constructs to functionally equivalent circuits, we created and analyzed hand-crafted CCM designs, and ‘reverse engineered’ from them the pseudocode that best described these designs. In Chapter 3, we presented a method for manual CCM design, derived from this analysis process.

In Chapter 4, we presented ‘Julia’, an imperative programming language for FPGA-based CCM design, based on the manual design method developed in Chapter 3. In Chapter 5, we evaluated the performance of Julia programs. We did this analytically, using a cycle-exact calculation method, empirically, using our current prototype implementation of the language, and predictively, based on manual transcriptions of Julia programs and our expectations of future implementations of the language.

We now list our most important findings:

- We have shown that FPGA-based CCMs can outperform conventional (non-parallel) software implementations by several orders of magnitude. Based on our experience with the Satisfiability problem, we conclude that CCMs allow for the implementation of classes of fine-grained parallel algorithms that are economically or practically infeasible to implement on conventional parallel systems, due to the large number of concurrent processing units required, and/or the abundant communication requirements between different processing units.

- Based on our experience with manual CCM design, we conclude that it is possible to trans-
form algorithms into CCM designs in a systematical manner, whilst preserving the asymptotic performance characteristics of the algorithm during the transcription.

- Based on our experience with the development of the experimental programming language Julia, we conclude that it is possible to transform algorithms into CCM designs automatically, using an imperative programming language with explicit parallel constructs, whilst maintaining the performance characteristics of the algorithm during the transcription. More importantly, we conclude that this can be achieved without explicit hardware design constructs or overly restrictive syntax in the source language.

- In closing, we conclude that our ‘bottom up’ approach to language development, i.e. by analyzing hand-crafted CCM designs and assimilating frequently used design patterns into a formal syntax, is a fruitful alternative to top down language design. The main advantages of our approach are that we have managed to avoid common language design pitfalls (e.g. awkward syntax, poor abstractions), and that we have been able to stay closer to the performance of manually designed CCMs with our generated code.

### 6.2 Future Work

We conclude this thesis with suggestions for further research in the field of FPGA-based CCM design, and, in particular, further development of the Julia language.

**A Complete Implementation of the Julia Language** Due to time limitations, our prototype compiler implements a subset of the Julia language only. While all essential features are present, a full implementation of the language would allow a broader class of elegant, practical test cases to be evaluated. We believe that the implementation of actual algorithms should drive the development of any language, hence we consider a full implementation to be an important aspect of a continued research effort.

**Analysis and Incorporation of More Example Programs** The current design of the Julia language is based on the study of a limited set of applications. It is inevitable that the study of additional algorithms will reveal shortcomings of its design, and its execution model. These experiences are indispensable for the development of a generally useful programming language. Hence, we recommend active further study of additional hand-crafted CCM designs.

**Combinational Grouping Optimization** In Section 4.9, we proposed a form of optimization where larger parts of the generated Execution Unit network are implemented in a single combinational circuit (‘combinational grouping’ optimization). We believe that it is feasible to implement this technique with a development effort of approximately up to six man-months, and that it will allow the Julia compiler to generate code that is approximately as fast as hand-crafted HDL designs.

**Analysis of the Clock Frequency vs. Combinational Grouping Tradeoff** In relation to combinational grouping optimization, it is useful to have a formal method of determining the point where the undesirable effects of this optimization (lower overall clock frequency of the CCM) outweigh the positive runtime effects, due to fewer handshakes in the signal path. If such a formal method can be found, a mature compiler could use this method to partition a design in asynchronous clock domains to achieve optimal performance.
An Execution Model with Global Memory  The Julia language and its runtime model do not, at present, permit dynamically indexed arrays. This is a considerable inconvenience to the programmer. We chose this model to avoid having to propagate complete arrays across Execution Unit boundaries at the cost of massive waste of FPGA real estate and unpredictable performance. We rejected an alternative implementation, where arrays are stored in separate, mutexed units that are connected to all call sites in the program, because it would inhibit the natural pipelining inherent in our execution model. In hindsight, we feel that the pipelined execution of statements is not very important to the overall execution time, because the existence of Repetition Units (loops) in a program result in poor utilization of the pipeline mechanism. Therefore, we feel that it is worthwhile to experiment with a compiler based on this approach. There are other advantages to abandoning the pipelined execution model, notably, the ability to implement real shared subroutines (as opposed to inlined subroutines) and global variables.

Extensions to the Julia Language and Compiler  There are several relatively small extensions that can be made to the Julia language to make it more useful. These include: built-in arbitrary precision integers, fixed point types, built-in support for multidimensional arrays, efficient array and matrix operations, built-in reduction operators, a print statement (for use during simulation of a design), the ability of the compiler to emit functionally equivalent C code for simulation, and a callback mechanism that allows data to be transmitted from the CCM while it is executing.

A SAT Solver with Lookahead Heuristics  In Chapter 3, we described a CCM implementation of the DPLL algorithm for the Boolean Satisfiability problem. While much work has already been done on SAT solvers in hardware, we believe that a so called ‘lookahead’ algorithm, or in fact any type of non-trivial branching heuristic, has never been implemented in hardware. Contrary to other hardware solvers, our architecture does not require static (compile time) ordering of branch variables. The implementation of a lookahead SAT solver in hardware would make for an interesting MSc. project.
Bibliography


Appendix A

The Julia Language Specification

A.1 Lexical Elements

This section describes the lexical elements of the Julia language. It describes the source alphabet and character encoding, the line terminators, comments, keywords, special symbol sequences, literals and identifiers used in the language.

A.1.1 Source Alphabet and Character Encoding

The source alphabet consists of the characters

\[
\begin{align*}
A & B & C & D & E & F & G & I & J & K & L & M \\
N & O & P & Q & R & S & T & V & W & X & Y & Z \\
a & b & c & d & e & f & g & h & i & j & k & l & m \\
n & o & p & q & r & s & t & u & v & w & x & y & z \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & ! & \% & & \\
( & ) & * & + & , & - & / & : & ; & < & = & > & [ \\
\} & \} & \hat{ } & \{ & \} & . & \\
\end{align*}
\]

in addition to the space character, the horizontal tab, the carriage return and the line feed character.

A letter is any of the listed characters from (uppercase) A to (lowercase) z. A digit is any of the listed characters from 0 to 9. Unless mentioned otherwise, all keywords and identifiers in the language are case sensitive.

A.1.2 Line Terminators

The following ASCII characters or character sequences are considered line terminators: The line feed character (LF), carriage return character (CR), and the carriage return/line feed sequence (CR LF). The matching is greedy, so the CR LF sequence is counted as one line terminator, rather than two.

A.1.3 White Space

The following characters or character sequences are considered white space: The space character (Space), the horizontal tab character (TAB) and the line terminators, as defined in Section A.1.2.
A.1.4 Comments

There are two types of comments:

```plaintext
/* The block comment. */
// The line terminated comment.
```

*Block comments* may span multiple lines. All input after the leading slash and asterisk is ignored up to and including the first occurrence of an asterisk immediately followed by slash. *Line comments* are started with two leading slashes, and are terminated by one of the line terminators described in Section A.1.2.

A.1.5 Keywords

The following character sequences are *keywords*. They may not be used as identifiers:

```
bool enum operator switch
case for parfor to
do if return type
else int struct while
```

Not all listed keywords are used in the current incarnation of the language. Unused keywords are reserved for future use.

A.1.6 Special Symbols

The following characters and character sequences are *operator symbols*:

```
+ - * / ! &
&& || > < == !=
>= <= ˆ % ˜ |
```

The following characters and character sequences are *infix assignment symbols*:

```
+= -= /=
*= += -=
```

The following characters and character sequences are *postfix assignment symbols*:

```
++ --
```

Note that character sequences like `++`, `+=`, etc. are not operators, but special forms of the assignment statement. Their use is discussed in Section A.5.1.

A.1.7 Literals

A *literal* is any sequence that can be produced from the following production rules:

```
literal :=
    boolean_literal |
    integer_literal

boolean_literal :=
    'true' | 'false'

integer_literal :=
    nonzero_digit digit*
```
A.1.8 Separators

The following characters are separators (punctuators):

( ) { } [ ] , ;

A.1.9 Identifiers

An identifier is any finite sequence that can be produced from the production rule

\[
\text{identifier} := \text{letter} ( \text{letter} | \text{digit} )^* 
\]

that meets the following restrictions:

- the sequence is not a keyword.
- the sequence is not a valid literal.

A.2 Types, Constants and Storage Locators

A.2.1 Types

There are two primitive types:

- integer A 32 bit signed integer type, identified by the keyword int.
- Boolean A two valued logic type, identified by the keyword bool.

Note that the type system is currently very limited. The objective is to support enumerated types, record types and integers with user defined bit lengths in future versions of the language.

A.2.2 Constants

A constant is any sequence produced by the following production rule:

\[
\text{constant} := \text{literal} \text{range_index} 
\]

\[
\text{range_index} := \text{identifier} 
\]

Range indices are ‘pseudo variables’, used to distinguish between the loop instances of the parfor statement. This construct is discussed in Section A.5.3. The values of range indices are evaluated for each loop instance at compile time, hence they are considered constants when used in expressions. Range variables are of type int. Literals are of the type defined in Section A.1.7.
A.2.3 Storage Locators

A storage locator is any sequence produced by the following production rules:

\[
storage\_locator := \\
| variable\_locator |
| array\_item\_locator |
\]

\[
variable\_locator := \\
| identifier |
\]

\[
array\_item\_locator := \\
| identifier '[ constant ']
\]

Storage locators identify virtual storage containers for computed values. Unlike variables in languages for conventional architectures, a storage locator is usually not associated with a single physical memory location.

A.3 Operators and Expressions

A.3.1 Built-in Operators

An operator is uniquely defined by an operator symbol and the types of its operands. Julia operators are pure functions, i.e. they cannot have side effects. Tables A.3.1 and A.3.1 respectively show the standard unary and standard binary operators of the language.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operand Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>int</td>
<td>int</td>
<td>Unary minus</td>
</tr>
<tr>
<td>!</td>
<td>bool</td>
<td>bool</td>
<td>Negation</td>
</tr>
</tbody>
</table>

Table A.1: The standard unary operators of the Julia Language.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operand Type</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int</td>
<td>int</td>
<td>Integer addition</td>
</tr>
<tr>
<td>-</td>
<td>int</td>
<td>int</td>
<td>Integer subtraction</td>
</tr>
<tr>
<td>*</td>
<td>int</td>
<td>int</td>
<td>Integer multiplication</td>
</tr>
</tbody>
</table>

Table A.2: The standard binary operators of the Julia Language.

Note that the assignment symbol (‘=’), the postfix increment symbol (‘++’), etc. are not considered operators, due to the fact that both have side effects and neither has a return value. These and other special assignment symbols are discussed in Section A.5.1.

A.3.2 Expressions

Expressions are all sequences that can be created from the following production rules:

\[
expression := \\
| closed\_expression |
| unary\_operator\_expression |
| binary\_operator\_expression |
\]
Expressions may only appear on the right hand side of an assignment statement or in the condition of a conditional statement. Expressions are not short circuited. The evaluation of an expression cannot have side effects.

A.4 Type Specifiers and Declarations

A.4.1 Type Specifiers

The following production rules generate type specifiers:

base_type :=
    'int' | 'bool' | identifier

type_specifier :=
    base_type ( ']' integer_constant ']' )*

A.4.2 Storage Locator Declarations

The following production rule generates storage locator declarations:

storage_locator_declaration :=
    type_specifier storage_locator

A.4.3 Function Declarations

The following production rules generate function declarations:

formal_parameter_list :=
    | /* Empty */
    | storage_locator_declaration ( ',' storage_locator_declaration )*

function_body :=
    '(' ( statement )* 'return' expression ';' ')' '}

function_declaration :=
    type_specifier '(' formal_parameter_list ')' function_body

Note that the grammar forces the return statement to be the last statement in the body of a function declaration, and that the return statement is not permissible anywhere else in the body of a function.
A.4.4 Operator Declarations

User-defined operator declarations are created using the following production rules:

\[
\text{unary_or_binary_opdef :=}\]
\[
\text{unary_operator} \left(\text{storage_locator_declaration}\right) \mid \text{binary_operator} \left(\text{storage_locator_declaration}, \text{storage_locator_declaration}\right)\]

\[
\text{operator_declaration := }\]
\[
\text{typename} \text{ unary_or_binary_opdef function_body}\]

A.5 Statements and Statement Blocks

This section describes statements and statement blocks. The following production rule enumerates the different types of statements:

\[
\text{single_statement :=}\]
\[
\text{storage_location_declaration} \mid \text{assignment_statement}\]

\[
\text{statement :=}\]
\[
\text{single_statement} ; \mid \text{conditional_statement} \mid \text{loop_statement} \mid \text{parfor_statement} \mid \text{return_statement}\]

\[
\text{statement_block :=}\]
\[
\{\text{statement* }\}\]

\[
\text{statement_or_block :=}\]
\[
\text{statement} \mid \text{statement_block}\]

The details of the different statement types are discussed in the following sections.

A.5.1 Assignment Statements

Assignment statements are sequences that can be created using the following production rules:

\[
\text{assignment_statement :=}\]
\[
\text{normal_assignment_statement} \mid \text{assignment_and_update_statement}\]

\[
\text{normal_assignment_statement :=}\]
\[
\text{storage_locator} \text{ infix_assignment_symbol} \text{ expression}\]

\[
\text{assignment_and_update_statement :=}\]
\[
\text{storage_locator postfix_assignment_symbol}\]

The normal assignment statements has the following meaning: the expression on the right hand side is evaluated, and its value is associated with the storage locator on the left hand side, until the locator is overwritten in a new assignment statement. It is a semantic error if the type of the right hand side expression does not match the type of the storage locator. The semantics of the remaining assignment statements are defined in terms of the normal assignment, according to the equivalence relations described in Table A.5.1.

Note that the assignment symbols can never be used inside expressions, i.e. the following statements are all incorrect:
Table A.3: Equivalence table of the special assignment symbols.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A += B;</code></td>
<td><code>A = A + B;</code></td>
</tr>
<tr>
<td><code>A -= B;</code></td>
<td><code>A = A - B;</code></td>
</tr>
<tr>
<td><code>A *= B;</code></td>
<td><code>A = A * B;</code></td>
</tr>
<tr>
<td><code>A++;</code></td>
<td><code>A = A + 1;</code></td>
</tr>
<tr>
<td><code>A--;</code></td>
<td><code>A = A - 1;</code></td>
</tr>
</tbody>
</table>

/* Illegal statements: */
a = b = 3; // = is not an operator.
a = b++; // ++ is not an operator.
2++; // 2 is not a storage locator.

A.5.2 Conditional Statements

Conditional statements are those sequences that can be created by the following rules:

```
conditional_statement :=
   'if' '(' expression ')' statement_or_block ( 'else' statement_or_block )?
```

The type of the conditional expression is required to be `bool`.

A.5.3 Loop Statements

Loop statements are those sequences that can be created by the following production rules:

```
loop_statement :=
   do_while_statement
   while_statement
   for_statement
   parfor_statement

do_while_statement :=
   'do' statement_or_block 'while' '(' expression ')' ';'
while_statement :=
   'while' '(' expression ')' statement_or_block
for_statement :=
   'for' '(' normal_assignment_statement ';' expression ';' assignment_statement ')' statement_or_block
parfor_statement :=
   'parfor' '(' identifier 'in' integer_literal 'to' integer_literal )
   statement_or_block
```

For each of the three sequential loop types, it is required that the loop expression is of type `bool`. The `parfor` loop is expanded as a macro. The loop variable (the identifier) is replaced by an integer literal during the expansion.

A.6 Programs

The following production rules describe the structure of Julia source files:
In Julia, there is no top-level program or ‘main’ routine. Each function or operator definition is compiled into a separate VHDL entity. The top-level construct of a Julia source file is a declaration list that contains function declarations, operator declarations, and/or type declarations. Variable declarations are not allowed at the top level.