Emma Burrows
eb379@cam.ac.uk

A Combinator Processor

Part II Computer Science Tripos

Peterhouse

2009

The cover image was generated by an ML* program running on my combinator processor.

Proforma

Name:	**Emma Burrows**
College:	**Peterhouse**
Project Title:	**A Combinator Processor**
Examination:	**Part II of the Computer Science Tripos, June 2009**
Word Count:	11,867
Project Originator:	Christian Steinrücken
Supervisors:	Arthur Norman
	Christian Steinrücken

Original Aims of the Project

The implementation of a processor on an FPGA using combinators as the instruction set, and a compiler to convert a subset of ML to the target instruction set. The project set out to build an architecture which would utilize both the on-chip and off-chip memory in a scalable way. The resulting system would be evaluated by running a series of programs written in the ML subset language and contrasting it to a comparable implementation.

Work Completed

All aims set out in the original proposal have been fulfilled. A combinator processor has been built on an Altera DE2 board and can scale to perform large computations. The compiler supports a powerful subset of ML and a type system has been built for the language. In addition to the original aims set out in the proposal, basic imperative features are supported. These have been used to build more powerful applications which test the scalability of the processor. Finally, the system has been compared to an existing research implementation and was found to perform favourably.

Special Difficulties

None.

Declaration

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

Signed

Date: May 14, 2009

Acknowledgements

I owe special thanks to:

- **Christian Steinrücken** and **Arthur Norman**, for supervising this project and for their encouragement.
- **Malte Schwarzkopf**, for his feedback and support.

Contents

1 **Introduction** **1**

2 **Preparation** **3**

2.1 LC & Combinators . . . 3
2.2 Combinator Graph Reduction . . . 5
2.3 Previous Work . . . 5
2.4 Combinator Conversion . . . 5
2.5 Processor Architecture . . . 7
2.6 Garbage Collection . . . 9
2.7 Software Engineering . . . 9
2.7.1 The Hardware Tools . . . 10
2.7.2 The Software Toolchain and Unit Testing . . . 10
2.7.3 The Iterative Development Model . . . 12
2.7.4 Simulation . . . 12
2.7.5 Redundancy . . . 13

3 **Implementation** **15**

3.1 The Compiler . . . 15
3.1.1 The Lexer . . . 16
3.1.2 The Parser . . . 16
3.1.3 The Type System . . . 18
3.1.4 The Combinator Converter . . . 21
3.1.5 The Memory Initialiser . . . 22
3.2 The Unit Tests . . . 23
3.2.1 Pretty Printer . . . 23
3.2.2 Abstract Syntax Tree Reducer . . . 23
3.2.3 Software Combinator Reduction . . . 24
3.3 ISA . . . 24
3.4 The Processor . . . 26
3.4.1 Overview . . . 26
3.4.2 Normal Execution . . . 27
3.4.3 Combinator Execution . . . 29
3.4.4 Arithmetic Execution . . . 31
3.4.5 Execution Termination . . . 31
3.4.6 On-chip Memory Model . . . 32
3.4.7 Off-chip Memory . . . 33
3.4.8 Garbage Collection . . . 34

3.5 Imperative Features 35

4 Evaluation **37**
4.1 Performance Analysis 37
4.1.1 Basic Performance 37
4.1.2 Garbage Collection 39
4.1.3 Comparison to Other Architectures 41
4.2 Applications 44
4.2.1 NQueens 44
4.2.2 Ponginator 46

5 Conclusions **49**

A Processor RTL View **51**

B Applications in ML* **52**
B.1 The `isPrime` Predicate 52
B.2 Combinator Pong, *Ponginator* 52
B.3 Library of Useful Functions 55

C Instruction Opcodes **56**
C.1 Arithmetic Instructions 56
C.2 Combinator Instructions 56

D Tables of Results **57**
D.1 Fibonacci 57
D.2 Garbage Collection and `isPrime` 57

E Verilog Sample Code **58**

F Initialisation Protocol **60**

G Original Proposal **62**

Chapter 1

Introduction

Functional programming languages are based on a model of computation that does not map naturally onto the mutable load-store architectures seen in conventional processors. In general, functional programming relies on function application and recursion rather than computation through the update of a global state.

In this dissertation, I aim to show a processor design that uses *combinators* as instructions. This may be regarded as a more natural form of processing for functional languages as it relates directly to the lambda calculus, the basis of many functional programming languages.

The concept of using combinators as a form of processing is not new. There have been several implementations of combinator processors in the past and currently, there is a significant research project in the area of graph reduction on FPGAs at the University of York. However, the issue of scalability has received relatively little attention. In particular, the question of how graph reduction, a fundamentally memory bound problem, can scale to utilise off-chip memory has not been tackled. This is a particular focus of my project and I hope to demonstrate how the utilisation of off-chip memory is crucial if this type of architecture is to do significant computation.

I designed a statically typed language called ML* that contains a subset of ML's functional and imperative features. Programs in ML* can be run in a designated interpreter or converted to combinator bytecode by a bespoke compiler. This bytecode then executes directly on a custom designed CPU, realised on a Altera DE2 Board.

This dissertation will describe each of these components in detail and show how they have been used to implement a scalable system, capable of running significant combinator programs.

Chapter 2

Preparation

This chapter outlines the work undertaken prior to implementation. It provides an introduction to lambda calculus and combinators. It also briefly describes previous work in this area and the influence that it had on the design. A description of the core architectural techniques used in the processor is provided, and software engineering techniques employed throughout the project are detailed.

2.1 Lambda Calculus and Combinators

The lambda calculus is the basis for most functional programming languages, including ML. It presents a formal way to notate and apply functions without the notion of state and can be viewed as a minimal Turing-complete programming language.

Lambda calculus expressions E have the following syntactic form:

$$\begin{array}{rcll} E & ::= & x & \text{Variable} \\ & \mid & \lambda x.\, E & \text{Abstraction} \\ & \mid & (E\ E) & \text{Application} \end{array}$$

where any occurrences of variables x have to be bound by an enclosing lambda abstraction.

Here are some examples of valid lambda expressions:

$$\lambda x.\, \lambda y.\, x \qquad \lambda p.\, (p\ (\lambda x.\, x)) \qquad ((\lambda a.\, \lambda b.\, (b\ a))\ (\lambda z.\, z))$$

Intuitively, an abstraction is a 1-argument function which describes what happens to that argument. For example, the term $(\lambda x.\, x)$ denotes the identity function that takes an argument and returns it unmodified. By contrast, the function $\lambda f.\, (f\ (\lambda x.\, x)\ )$ takes an argument f and returns the result of applying it to $(\lambda x.\, x)$.

Note that we consider the terms $(\lambda x.\, x)$ and $(\lambda y.\, y)$ equivalent by *α-conversion*, *i.e.* renaming bound variables has no effect on the meaning of the term.

When a function is applied to an argument, the term can be reduced by substituting the argument for each occurrence of the bound variable in the body of the abstraction, and removing the leading lambda binding. For example:

$$((\lambda a.\, \lambda b.\, (b\ a))\ (\lambda z.\, z)) \quad \rightarrow_\beta \quad \lambda b.\, (b\ (\lambda z.\, z))$$

Note how the $(\lambda z.\, z)$ term has been substituted for a. This step is called a β-reduction, and is denoted by the symbol $\rightarrow_\beta$. A full treatment of the lambda calculus will not be given here; see [6] for a detailed discussion.

Combinatory logic, introduced by Haskell B. Curry, is a variable-free relative of the lambda calculus. Instead of having abstractions, it comprises a set of predefined functions (the *combinators*) which can be combined by applications.

Combinator	**Lambda Encoding**	**Combinator Reduction**
`S`	$\lambda p,q,r.\ (p\ r)(q\ r)$	$\mathtt{S}\ P\ Q\ R \rightarrow (P\ R)(Q\ R)$
`K`	$\lambda p,q.\ p$	$\mathtt{K}\ P\ Q \rightarrow P$
`I`	$\lambda p.\ p$	$\mathtt{I}\ P \rightarrow P$
`B`	$\lambda p,q,r.\ p\ (q\ r)$	$\mathtt{B}\ P\ Q\ R \rightarrow P\ (Q\ R)$
`C`	$\lambda p,q,r.\ (p\ r)\ q$	$\mathtt{C}\ P\ Q\ R \rightarrow P\ R\ Q$
`Y`	$\lambda p.\ (\lambda x.\ p(x\ x))(\lambda x.\ p\ (x\ x))$	$\mathtt{Y}\ P \rightarrow P\ (\mathtt{Y}\ P)$

Table 2.1: Combinator reductions and representation as lambda expressions

Combinator reduction does not require substitution, but can instead be thought of as performing a rearrangement of its arguments, according to the reduction pattern of that combinator. It turns out that only two combinators, `S` and `K`, are necessary to encode any function that can be represented in lambda calculus, but it is convenient to have a few more. Table 2.1 shows some of the key combinators, their lambda calculus equivalent, and their reduction rules.

Lambda calculus expressions can be translated to combinators by a process called *variable elimination*. David Turner's *Advanced Translation Rules* [14] (shown in table 2.2) do this in an optimised way, using the five basic combinators: `S`, `K`, `I`, `B`, and `C`. The method is recursive, with translation starting at the innermost lambda abstraction and working outwards.

Rule	**Side Condition**
$\lambda^T x.\ x \equiv \mathtt{I}$	
$\lambda^T x.\ P \equiv \mathtt{K}\ P$	$(x \notin FV(P))$
$\lambda^T x.\ P\ x \equiv P$	$(x \notin FV(P))$
$\lambda^T x.\ P\ Q \equiv \mathtt{B}\ P\ (\lambda^T x.\ Q)$	$(x \notin FV(P)$ and $x \in FV(Q))$
$\lambda^T x.\ P\ Q \equiv \mathtt{C}\ (\lambda^T x.\ P)\ Q$	$(x \in FV(P)$ and $x \notin FV(Q))$
$\lambda^T x.\ P\ Q \equiv \mathtt{S}\ (\lambda^T x.\ P)\ (\lambda^T x.\ Q)$	$(x \in FV(P)$ and $x \in FV(Q))$

Table 2.2: Turner's Advanced Translation Rules
$FV(P)$ denotes the set of free variables in P.

Lambda calculus and combinators are powerful enough to encode data structures including integers and arithmetic using a *Church numeral* encoding. For efficiency reasons however, the hardware design has integers and basic arithmetic built-in as combinator primitives.

2.2 Combinator Graph Reduction

Combinator expressions can be represented as a binary tree, where the leaves of the tree are combinator primitives and nodes represent applications. Reduction can be viewed as a manipulation of these trees, which is implemented efficiently through the redirection (and potential sharing) of pointers. Figure 2.1 shows how combinators reduce in tree form.

2.3 Previous Work

A fundamental milestone in this area of research was work done on SKIM, the `S`, `K`, `I` reduction machine by Norman *et al.* [8, 13] at Cambridge in the 1970s and 1980s. It outlines a translation procedure from LISP to combinators and goes on to describe a combinator reduction mechanism in hardware. The core reduction principles of my processor are based on the stackless model for graph reduction presented by this work.

The University of York has also undertaken a significant research project in this area. In his PhD thesis [11], Naylor outlines the implementation of *The Reduceron*, an FPGA graph reduction machine. His initial work describes a technique for encoding Haskell with the combinators `S`, `K`, `I`, `B`, `B*`, `C` and `C'`. The reduction technique employed at York uses on-chip memory on an FPGA, a stack-based architecture and a copy garbage collector. The most recent implementation of their reduction machine, *The Reduceron 2*, has moved to a more complex implementation and uses supercombinators instead of combinators. Supercombinators were not a focus of this project, so no discussion of them is presented here.

A comparative analysis of the stackless implementation used in this project and the reduction model of *The Reduceron* can be found in section 4.1.3.

2.4 Converting ML to combinators

ML is a functional programming language based on lambda calculus, and a significant proportion of the language can be translated to combinators using Turner's translation rules.

There are some features of ML which are more difficult to encode as combinators because they transcend pure, stateless lambda calculus. These features include, but are not limited to, references, IO and exceptions and will not receive significant attention in this work.

Specific details of the conversion between ML and combinators is provided in section 3.1.4, so here I will present some of the more tricky aspects of the conversion which do not amount to a simple desugaring of the language.

Recursion in ML is implicitly introduced through function declarations. In the lambda calculus, recursion is implemented by the application of a *fixed-point* operator; one that is commonly seen in literature on the subject is the `Y` combinator, discovered by Haskell B. Curry:

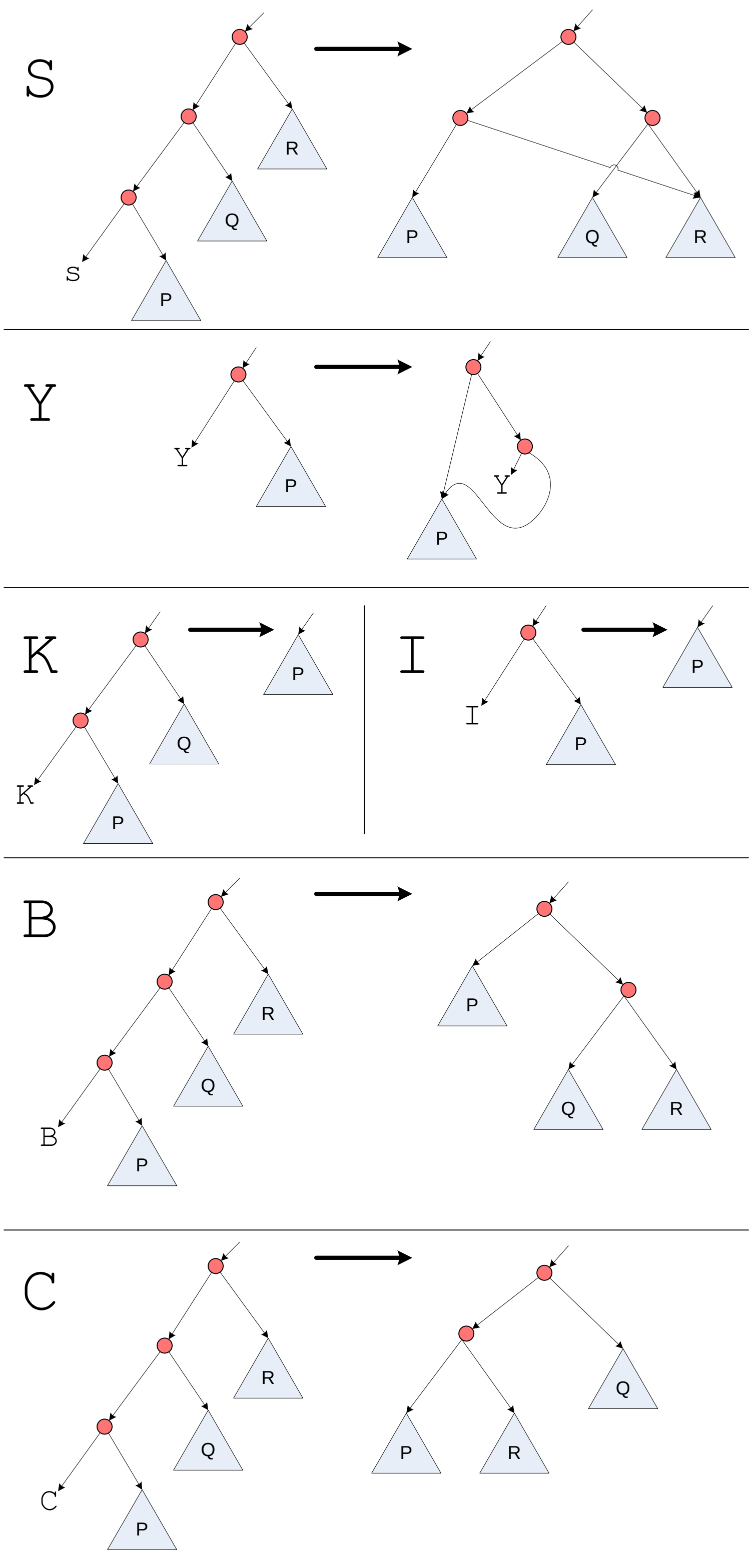


Figure 2.1: Combinator graph reduction

$$\mathtt{Y} \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x))$$

The conversion process makes use of the `Y` combinator accordingly and is further explained in section 3.1.4.

ML is a eager language, whereas combinators are traditionally implemented with lazy semantics. Since the combinator byte-code that runs on the processor is lazy, I chose to preserve these semantics in the high level language supported by the compiler. As such, this language can only be said to be ML-*like*, having clear differences from ML with regard to termination behavior. I will refer to this language as ML*.

Assuming a lazy model of execution raises the issue of arithmetic operations which are usually "eager" operations. For example, at the time of application, an addition operator expects its arguments to be reduced to a value. This does not work in the lazy model, as illustrated in figure 2.2.

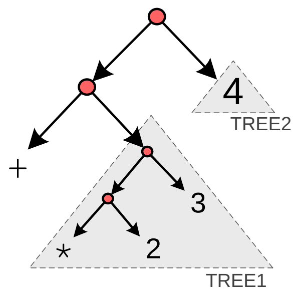


Figure 2.2: Lazy semantics and arithmetic reduction

Reduction starts at the deepest, leftmost part of the tree. Clearly `TREE1` in figure 2.2 will not be reduced in its current position. The "plus" combinator must therefore first perform a pointer manipulation on the graph to ensure its arguments are reduced before the actual arithmetic can be performed. A thorough treatment of this is given in section 3.4.4 since it is implementation specific.

2.5 Processor Architecture

As previously explained, combinator expressions can be represented as a graph, and reduction performed through manipulation of the graph's pointers. This abstract concept has to be transformed into an algorithm that can be implemented on an FPGA, and in a way that takes advantage of the available hardware resources.

Firstly, the combinator graph needs to represented in memory. The binary nature of the combinator graphs lends itself to using a bisected memory model. This also maps well onto the resources available on the FPGA, since the on-chip RAM memory of the Altera FPGA is divided into blocks which can be accessed simultaneously. The two memory partitions are referred to as `CAR` and `CDR`; the `CAR` memory array represents the child on the left-hand branch of an application and the `CDR` memory array represents the right-hand branch. Figure 2.3 (overleaf) shows a very simple combinator graph and how it is represented in memory.

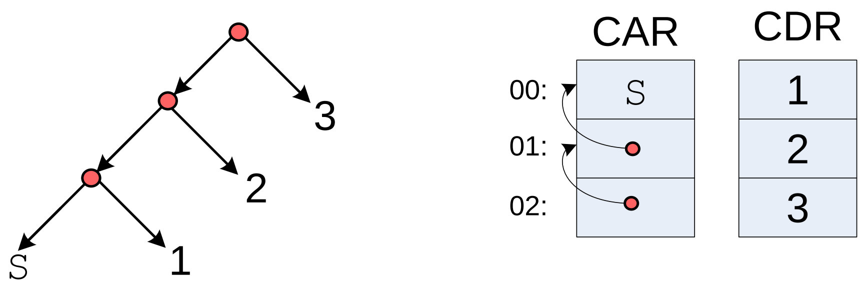


Figure 2.3: `CAR` and `CDR`: Representation of a simple combinator graph in memory

In order for a combinator reduction to be performed, the tree must be traversed to find the left-most leaf node, where reduction will start. It is also necessary to record the path taken so that it can be retraced at any time. Two pointers, **P** and **B**, are required to achieve this in a stackless manner. The **P** pointer points to the "current" node in the tree that triggers the next action of the processor, while the **B** pointer points to the last node that was traversed in the tree. As nodes are traversed, the pointer in the left part of the cell gets redirected to point to the parent node, from whence the flow of execution just came. Thus, there is always a way of retracing and restoring the tree. Figure 2.4 illustrates what a simple program looks like before any traversal has occurred and how it appears after the tree has been traversed and the `S` combinator found.

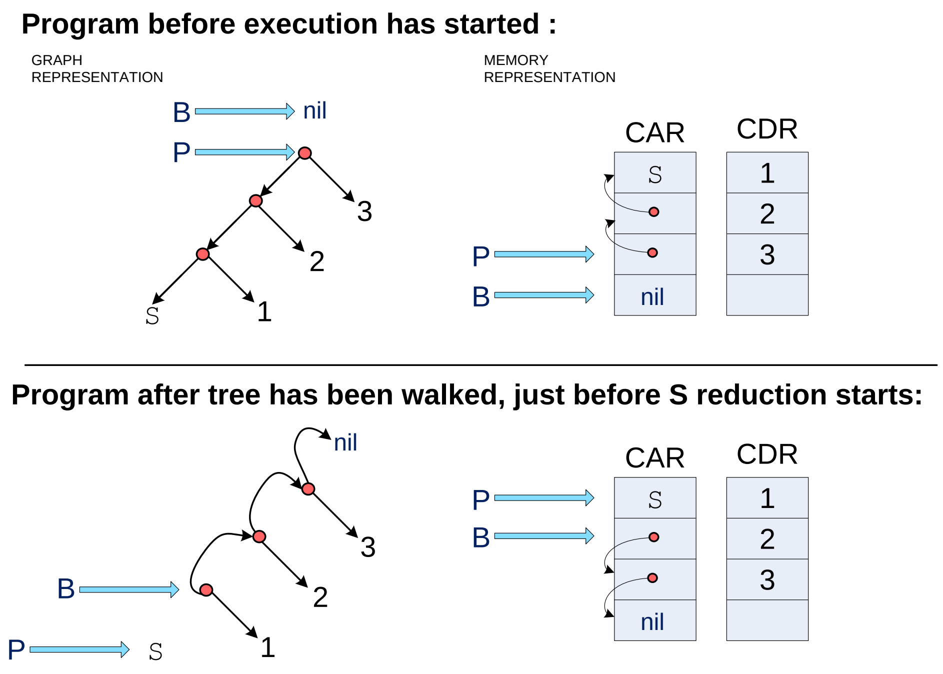

Program before execution has started :
GRAPH REPRESENTATION
MEMORY REPRESENTATION
B
nil
P
S
1
2
3
CAR
CDR
P
B
nil
Program after tree has been walked, just before S reduction starts:
nil
B
P
S
1
2
3
CAR
CDR
P
B
nil

Figure 2.4: Mechanism for graph traversal and restoration

The presence of shared pointers means that combinator reduction must leave the graph in a state such that only the node of the last argument is modified and the rest of the graph is left as it was prior to the reduction. If a combinator was able to destructively walk over all the memory cells that corresponded to its arguments, then a shared pointer to one of those arguments could be affected by the reduction. Thus if reduction requires more than one cell, it must be able to allocate a new one. To implement this, a *free memory chain* is used, which links together all the free memory cells. A combinator can then claim a cell from the free memory chain as it requires.

2.6 Garbage Collection

As combinators are able to consume free memory, it is clear that parts of the tree will eventually be abandoned and should be freed to allow the space to be reclaimed by future reductions. This motivates the need for a garbage collector. The method chosen was *copy garbage collection*. This can be implemented in a stackless manner which fits with the design principles of the rest of the processor, and is relatively simple.

A copy garbage collector works by copying all the cells referenced by pointers (*e.g.* the **B** and **P** pointers in the case of my processor) into a free memory partition or *copy space*. This typically requires suspending reduction until the process is complete.

2.7 Software Engineering Techniques

The project represented a significant body of work, so careful planning and management were essential for success. It also had a substantial hardware component and hence a detailed development plan and test bench were crucial to prevent the complexity from spiraling out of control.

The management strategy employed was:

- Ensure a thorough understanding of the hardware tools available and their limitations. Structure the hardware design to exploit existing resources within the tools.
- Develop a strong software toolchain to reduce the complexity of debugging in the hardware and allow unit testing of combinator programs.
- Employ an iterative development model to allow complexity within the hardware design to increase in a controlled manner.
- Parallelise development of the hardware design with a simulation model to allow for fine-grained debugging.
- Keep a consistent record of work and build in sufficient redundancy to make the project tolerant to failures on different levels.

2.7.1 The Hardware Tools

Hardware tools for FPGAs can be quite intricate. In this section, I will introduce some of the tools which played a significant role in this design and implementation.

I did an internship with the Computer Architecture group in the summer of 2008, during which time I experimented with Synplify [3], a vendor neutral synthesis tool and an alternative to Quartus [1], the Altera synthesis tool. I found that Synplify's ability to do RAM inference was far superior to Quartus', whose ability to do this still seems to be a work in progress. Later in the project, a significant issue was found with Synplify's ability to synthesize off-chip communication modules and so the design was ported to Quartus.

JTAG modules provide an interface for direct communication with the board while it is running. This is a useful basis for building test harnesses, as it allows a portion of the on-chip memory to serve as a buffer for messages in and out of the system. In particular, it can be used to load programs into the processor in a reconfigurable manner.

The SDRAM controller is an interface to the off-chip SDRAM memory. It controls the maintenance of state within the SDRAM and exports a simplified set of signals that can be used to read and write to the off-chip memory. A custom SDRAM controller can be generated using the SOPC builder in Quartus.

2.7.2 The Software Toolchain and Unit Testing

Component	**Role**
The Compiler	This is responsible for taking a program written in ML* and converting it to an abstract syntax tree which it subsequently converts to combinators.
The Type System	This indicates if the program is type safe according to the implemented type system.
The Memory Initialiser	This takes the combinator representation of the program and converts it to a form that could be loaded into the memory of the FPGA.

Table 2.3: Components of the Software Toolchain

Before implementation could commence, it was necessary to have a clear outline of the components involved in converting programs written in ML* to a combinator program running on the hardware. The components involved in this process constitute the *software toolchain* and are outlined in table 2.3.

The modularisation of the software toolchain into separate components allowed a comprehensive series of unit tests to be built up around each module. Figure 2.5 shows the unit testing that occurred at each level of the toolchain.

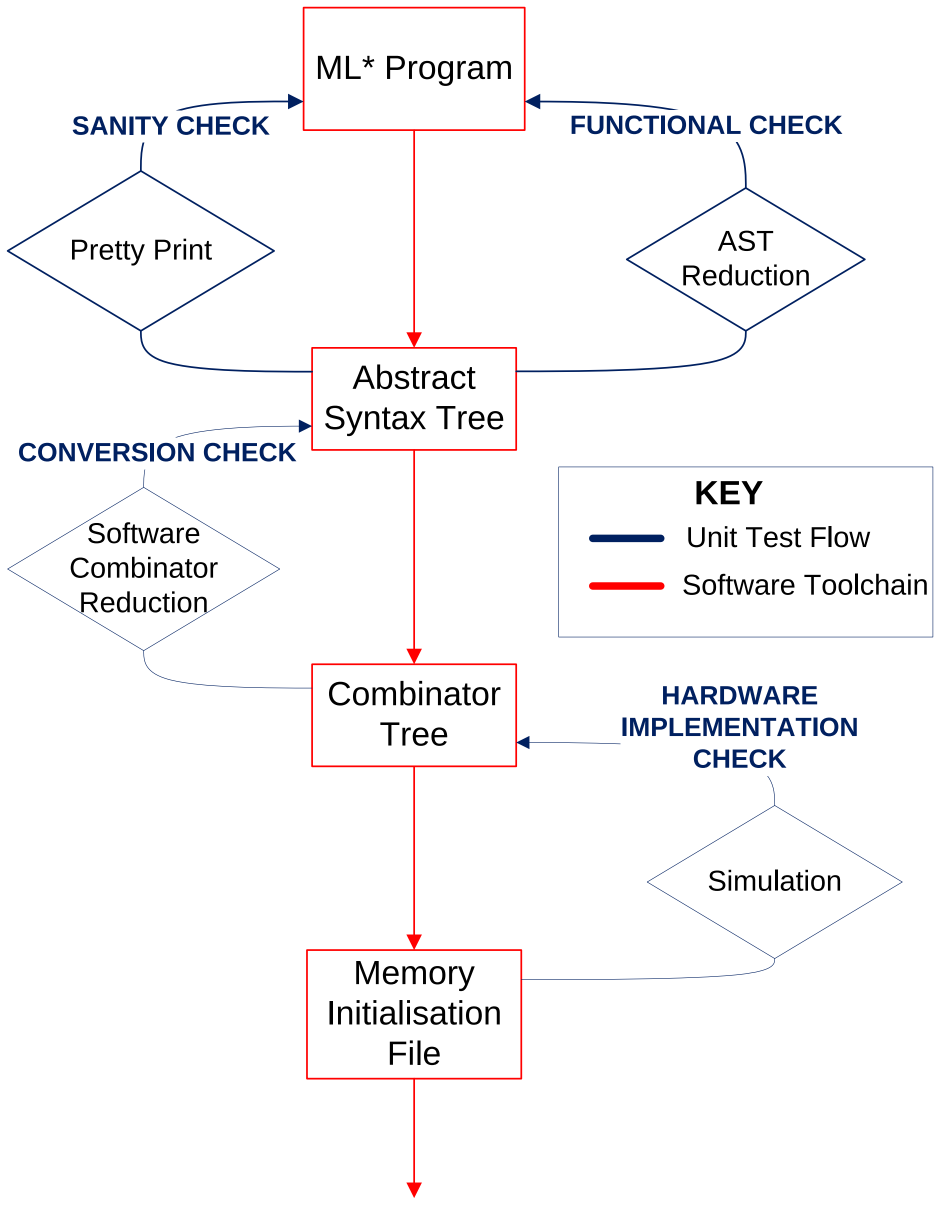


Figure 2.5: Software Toolchain and Unit Test Flow

The *Pretty Printer* and *AST (Abstract Syntax Tree) Reducer* both serve to check that the compiler produces the correct abstract syntax tree for the provided ML* program. The *Software Combinator Reducer* is used to reduce the translated combinator version of the program in software, checking that the output agrees with that of the AST reducer.

Finally, the combinator stream is converted to a memory initialisation file which is fed to a simulation model of the hardware, as described in section 2.7.4.

2.7.3 The Iterative Development Model

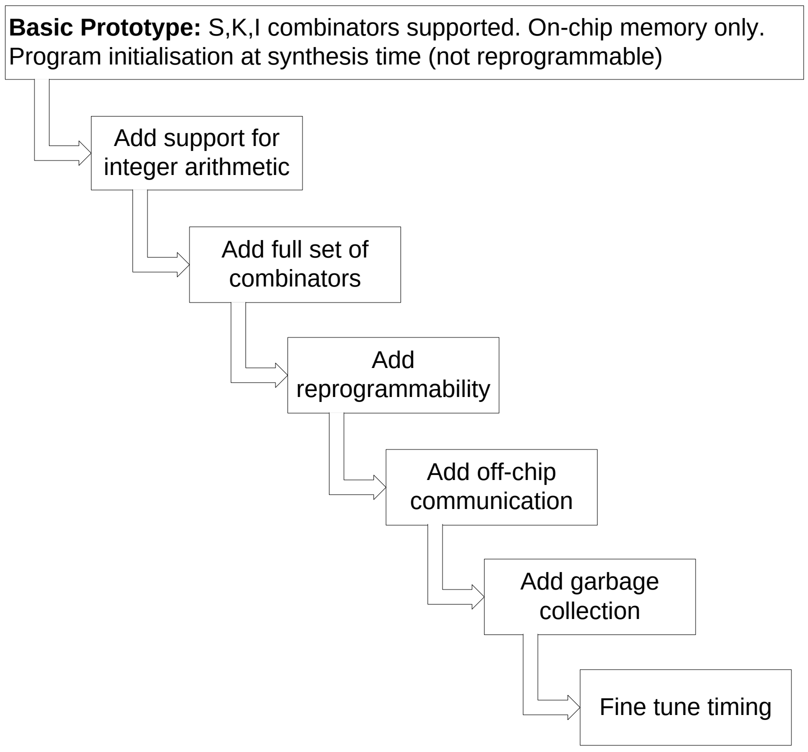

Basic Prototype: S,K,I combinators supported. On-chip memory only. Program initialisation at synthesis time (not reprogrammable)
Add support for integer arithmetic
Add full set of combinators
Add reprogrammability
Add off-chip communication
Add garbage collection
Fine tune timing

Figure 2.6: The Iterative Development Model

The design strategy employed for the hardware implementation was an iterative one. A basic prototype was implemented and tested and then increasing layers of the complexity were added to this model at each stage. Each iteration was fully tested before the next iteration was implemented hence providing a control on the complexity of the project and allowing modular debugging. The stages of development are shown in figure 2.6.

The stages in the development model were not linear in time or complexity. Some stages, such as the off-chip memory communication, required further subdivision and modularisation to allow for thorough testing.

2.7.4 Simulation

At each stage of development, a Verilog simulation model was implemented to facilitate debugging. The simulation aimed to use much of the same code as the hardware version to obtain a high level of reliability from the simulation results. Simulation was done using ModelSim [2], which supports the full Verilog language plus test harness features such as display statements. It also allows values to be "forced", simulating the toggling of switches and external stimuli.

As the processor evolved, more fine grained simulation was required, causing the complexity of the simulation model to grow in line with the complexity of the processor itself.

2.7.5 Redundancy

Redundancy of data is necessary to guard against failure on a physical level. For the project implementation, this was achieved by performing a daily backup to an external hard drive. Additional backup was made to a remote server after every major modification. Major updates and status reports were downloadable from a website, allowing my supervisors to keep track of my progress and serving as a useful diary throughout the project.

The dissertation was backed up with a Subversion repository on the Student Run Computing Facility (SRCF). This provided revision control and further redundancy.

Chapter 3

Implementation

The implementation of the compiler is presented. Important concepts are detailed with examples. The unit test suite is described. The design of the processor is explained and the components in its construction discussed.

3.1 The Compiler

The compiler takes a program written in ML* and converts it to memory initialisation files that can be loaded onto the processor.

ML* is a lazy subset of ML, represented by the abstract grammar shown below.

$\langle expr \rangle$	:=	`let` $\langle decl \rangle$ `in` $\langle expr \rangle$	Let expressions
	$\mid$	$\langle expr \rangle$ $\langle binop \rangle$ $\langle expr \rangle$	Binary operations
	$\mid$	$\langle expr \rangle$ $\langle expr \rangle$	Applications
	$\mid$	`fn` x `=>` $\langle expr \rangle$	Lambda abstractions
	$\mid$	x	Variables
	$\mid$	`if` $\langle expr \rangle$ `then` $\langle expr \rangle$ `else` $\langle expr \rangle$	Conditionals
	$\mid$	i	Integers
$\langle binop \rangle$	:=	$+ \mid - \mid \times \mid > \mid < \mid = \mid !=$	
$\langle decl \rangle$	:=	`fun` *fun_name args* `=` $\langle expr \rangle$	Function declarations
	$\mid$	`val` *val_name* `=` $\langle expr \rangle$	Value declarations

The compiler accepts as input an ML* program consisting of function declarations, value declarations and an expression to be evaluated. It should be noted that the declarations are lazy, meaning that they are only evaluated to a value when required. All components of the compiler were written in ML, and are explained in the rest of this section.

3.1.1 The Lexer

The principle role of the lexer is to tokenize the input. *Tokenizing* is the process of converting characters representing a program into a stream of tokens. The implementation of the lexer will not be presented in detail as it follows a fairly standard procedure similar to that presented in [10].

To implement the lexer, a ML datatype `token` was created which includes constructors such as `op_plus`, `num_int of int` and `semi_colon`. Lexing a list of characters then amounts to creating a function which maps characters onto the appropriate token constructions. The below example illustrates the lexing of a declaration:

`val a = 4;` $\rightarrow^{lex}$ `[decl_val, id 'a', op_equals, num_int 4, semicolon]`

3.1.2 The Parser

The parser implemented is a recursive descent *LL(0)* parser. The role of the parser is to take the token stream produced by the lexer and convert it to an abstract syntax tree (AST) for each declaration in the program. For ML*, a very simple abstract syntax can be used to represent an expression. This takes the form of a datatype and is shown below.

```
datatype AST = Var of string                         (* Variables *)
             | App of AST * AST                      (* Applications *)
             | Lambda of string * AST                (* Lambda terms *)
             | Int of int                            (* Integers *)
             | Let of ((string * bool * AST) * AST)  (* Let expressions *)
             | Inl of AST -> AST;                    (* Used in AST reducer *)
```

The constructors in the AST should have a fairly intuitive meaning with the exception of the `Inl` constructor, the role of which is explained in section 3.2.2, and the `Let` constructor, which is explained in more detail below. It should be noted that conditional operators, addition operators, etc. are not represented in the AST in their own right. Instead, they are represented within the `Var` expression. For example, the expression:

`if` b `then` e_1 `else` e_2

will have the AST showing in Figure 3.1 (next page).

An *LL(0)* parser parses its input from left to right and recurses from the constructs at the leaves of the tree according to a context free grammar. The first stage in the implementation of the parser was to construct an unambiguous context free grammar that described the language and avoided left recursion. A grammar is considered to be left recursive if there is a production of the form:

$$\langle expr \rangle := \langle expr \rangle \ldots$$

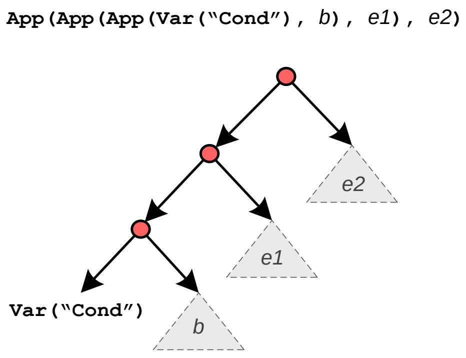


Figure 3.1: Abstract Syntax Tree for an IF-expression

Implementation of this sort of grammar would fall into non-termination due to the existence of ⟨*expr*⟩ at the left of the production. The context free grammar used for expressions within the language is presented in figure 3.2. The grammar avoids the issue of left recursion through the use of the ⟨*fact*⟩ symbol, and also expresses arithmetic operator precedence.

$$
\begin{array}{rcl}
\langle \mathit{decl} \rangle & := & \texttt{fun}\ \ \mathit{fname}\ \mathit{fargs}\ \ = \langle \mathit{expr} \rangle \\
 & | & \texttt{val}\ \ \mathit{vname}\ \texttt{=}\ \langle \mathit{expr} \rangle \\
 & & \\
\langle \mathit{expr} \rangle & := & \langle \mathit{fact} \rangle + \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{fact} \rangle - \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{fact} \rangle = \langle \mathit{fact} \rangle \\
 & | & \texttt{let}\ \langle \mathit{decl} \rangle\ \texttt{in}\ \langle \mathit{expr} \rangle \\
 & | & \texttt{if}\ \ \langle \mathit{expr} \rangle\ \texttt{then}\ \langle \mathit{expr} \rangle\ \texttt{else}\ \langle \mathit{expr} \rangle \\
 & | & \langle \mathit{fact} \rangle \\
 & & \\
\langle \mathit{fact} \rangle & := & \langle \mathit{atom} \rangle\ \texttt{*}\ \langle \mathit{fact} \rangle \\
 & | & \langle \mathit{atom} \rangle \\
 & & \\
\langle \mathit{atom} \rangle & := & (\ \langle \mathit{expr} \rangle\ ) \\
 & | & \texttt{true} \mid \texttt{false} \mid i \mid x \\
 & | & \lambda x.\langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle\ \langle \mathit{expr} \rangle
\end{array}
$$

*Figure 3.2: Backus-Naur form of the grammar for expressions in ML**

In accordance with the grammar presented, the implementation consisted of the mutually recursive functions `parseDECL`, `parseEXPR`, `parseFACTOR` and `parseATOM`.

When the end of a production is reached in the implementation, an AST representing that particular language construct, plus any leftover tokens, is returned. If the expression does not parse correctly, an exception is thrown. The exception includes an error message to give the user an indication of the cause of the problem.

A program in ML* consists of function declarations, value declarations and an expression to be evaluated. The `parseDECL` function converts a function declaration so that the arguments become lambda abstractions. For example, the function `sum`:

```
fun sum n = if (n = 0) then 0 else (sum (n-1));
```

is equivalent to the following declaration, where `val_rec` is a recursive value declaration:

```
val_rec sum = fn n => if (n = 0) then 0 else (sum (n-1));
```

The `parseDECL` function returns a value of type `(string * bool * AST)` where the string is the declaration identifier, the boolean indicates whether the binding is recursive (*i.e.* for functions), and the last part is the AST for the body of the declaration. As is the case in ML, declarations at the top level of the program are just a convenient syntax for nested `let` expressions. For example, consider the program:

```
fun isZero n = if (n = 0) then 0 else (n+5);
isZero 4;
```

This is equivalent to the expression:

```
let fun isZero n = if (n = 0) then 0 else n
in isZero 4
end;
```

After a declaration is parsed, it is wrapped around the rest of the program with a `Let` constructor. Hence, the AST for the example above becomes:

```
Let(
    (* Declaration definition *)
    ('isZero', true, Lambda('n',
                             App(App(App(Var('Cond'),
                                         App(App(Var('Eq'),
                                             Int 0),
                                     Var 'n')),
                                 Int 0),
                             Int 1))),
    (* Let body *)
    App(Var('isZero'), 4))
```

In this way, whole programs can be parsed and converted to a single abstract syntax tree.

3.1.3 The Type System

ML is a statically typed language. It uses the typing discipline presented by Hindley and Milner [9] to check if an expression is typeable and return the most general type. The implementation of the type checker for ML* is based on this discipline.

The algorithm for typeability proceeds by the implementation of a set of inductive rules and axioms over typing environments, ML* expressions and types. The type system for ML* is shown in figure 3.3.

The algorithm descends recursively on the structure of expressions, generating constraints as specified by the type rules. These constraints are enforced through type unification and substitution. When the inference algorithm is applied to an expression, it returns a pair, (*substitution*, *type*), where the *substitution* should be applied to the typing environment in any future inferences, and the *type* is the type of the expression.

$$(\text{var} \succ)\ \frac{}{\Gamma \vdash x : \tau}\ ^{a}$$

$$(\text{int})\ \frac{}{\Gamma \vdash \mathit{integer} : \texttt{Int}}$$

$$(\text{fn})\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \rightarrow \tau_2}\ ^{b}$$

$$(\text{app})\ \frac{\Gamma \vdash M_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

$$(\text{let})\ \frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall A.\tau_1 \vdash M_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = M_1\ \texttt{in}\ M_2 : \tau_2}\ ^{c}$$

[a] if $\Gamma(x) \succ \tau$
[b] if $x \notin dom(\Gamma)$
[c] if $x \notin dom(\Gamma)$ and $A = ftv(\tau_1) - ftv(\Gamma)$

*Figure 3.3: Typing System for ML**

Free type variables can be unified with another type once, but this is not sufficient to allow for the reuse of functions in a *polymorphic* sense. For example, consider the code snippet:

```
let fun ident a = a
in if (ident true) then (ident 4) else (ident 5)
end;
```

The initial type of the `ident` function is $\alpha \rightarrow \alpha$, but when unified in its first application, it will be refined to the type `bool` $\rightarrow$ `bool`. This will not then unify with the integer application in the second and third application of the `ident` function. For this reason, ML has polymorphic type variables. Polymorphic variables are bound by universal quantification indicating that they can be instantiated to free type variables more than once.

In the implementation, types have the following form:

```
datatype typ = TInt                    (* Integer primitive *)
             | TAlpha of int           (* Free type variable *)
             | TAll of int * typ       (* Polymorphic type scheme *)
             | TFun of typ * typ       (* Function type *)
```

`TAlpha` represents free type variables which are uniquely identified by an integer. `TAll` is used to implement polymorphism.

In order to infer types, a source of fresh type variables is needed. Consider, as an example, the (FN)-rule: At the time when the lambda expression is first seen, the type of x is not known so it is assigned the most general type possible, namely a free type variable. In my implementation, new free type variables are obtained using an incrementing integer reference.

The `TAll` constructor allows polymorphism to be implemented in the following way. The type $(\forall \alpha_1 \alpha_2.\ \alpha_1 \to \alpha_2)$ would be represented in the type implementation as:

```
TAll(1, TAll(2, TFun(TAlpha(1), TAlpha(2))))
```

When a `TAll` is unified with another type, the type variable bound by the `TAll` is instantiated with a new free type variable. So in the previous polymorphic example, the application of the `ident` function to a boolean and to an integer will not conflict because, in each application, the free type variables that unify with `TBool`[1] and `TInt` are different.

Polymorphic types are created through `let` expressions. So, as the (LET)-rule describes, when the type of M_1 is found, the implementation iterates over the type and finds all the free type variables, wrapping the whole type in a `TAll` constructor for each variable. The implementation ensures that a `TAll` can never occur inside a function type.

The function `pt` takes an abstract syntax tree representing an expression and a type environment, and returns the substitution used for unification and the type of the expression. It uses the auxiliary functions defined below to implement the inference algorithm:

Function name	**Function Role**
`lookupTYPE x env`	Find the type of the variable `x` in the environment.
`create_polys ty`	Find the free variables in the type and create an equivalent polymorphic representation.
`instantiate_polys ty`	Assign new free type variables to each of the polymorphic variables in the type and return the resulting type.
`apply_sub(type, sublist)`	Apply the substitutions in the `sublist` to the `type`.
`mgu(typeA, typeB)`	Find the most general unifier of `typeA` and `typeB` and return the substitution list representing the unification operation.
`assign_env(environment, substitution)`	Apply the substitution to the environment.

This allows the typing of abstract syntax trees. Since full programs are represented as a single AST, typing of expressions is sufficient to be able to type full programs.

[1]Strictly speaking, the language has no `TBool` type, as booleans are implemented in a functional way. Their use here is purely for the sake of explanation.

3.1.4 The Combinator Converter

In section 2.1, I presented the *Advanced Translation Rules* for converting lambda calculus to combinators. The implementation proceeds in two stages: firstly, the AST is converted to a form such that the translation rules can be applied, and secondly the actual translation is performed according to the rules. The datatype for combinators is as follows:

```
datatype combinator = I | K | S | B | C | Y
                    | ADD | MULT | EQ | NEQ | GT | LT | SUB
                    | INT of int
                    | APP of combinator * combinator;
```

This corresponds exactly to the instruction set of the processor itself. Integers are represented explicitly in the AST, as are primitives for arithmetic and comparison. The translation of these AST constructs can be implemented using a trivial mapping to the associated combinator, *e.g.*:

$$\lambda^T x.\, \texttt{Var('Add')} \longrightarrow \texttt{ADD}$$

Unlike integers, boolean values have not been given an explicit combinator representation, and are encoded in the standard functional way:

$$\begin{aligned} \texttt{true} &\equiv \lambda x.\, \lambda y.\, x \equiv \texttt{K} \\ \texttt{false} &\equiv \lambda x.\, \lambda y.\, x \equiv \texttt{K I} \end{aligned}$$

As a result of this, the equality combinator returns true or false as function values. This is nicely minimal in the sense that no additional primitives are required for boolean logic and conditionals, and so `if`-statements can be converted to simple applications, for example:

$$\texttt{if (}n\texttt{ = 4) then 1 else 2} \longrightarrow \texttt{(EQ }n\texttt{ 4 1 2)}$$

The Advanced Translation Rules for applications and lambda expressions (as can be found on page 4) do not need adjusting to support the AST for ML*, and can be implemented verbatim. Variables in ML* present some subtleties to the implementation as they come in four flavours, as explained in table 3.1 (overleaf).

Built-in variables are mapped directly to combinators (*e.g.* `ADD` above), and all other variable flavours can be converted to lambda-bound variables.

Declaration variables are transformed using *let-conversion*:

$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end} \quad \longrightarrow \quad (\lambda x.\, e_2)\, e_1$$

Recursive variables, as indicated by the *is_recursive* field in the declaration triple, can be transformed to lambda-bound variables using an application of the `Y` combinator:

```
("fibonacci", is_recursive, declaration_body) →
                 App(Y, Lambda("fibonacci", declaration_body))
```

Any recursive variables in the *declaration_body* have now been bound by the lambda abstraction.

Variable Class	**Description**
Lambda-bound Variables	These are any variables which are bound by a lambda abstraction. Function arguments also fall into this category since these are converted to lambda abstractions.
Built-in Variables	These are variables for expressing functionality such as addition, multiplication, equality, etc. *e.g.* `Var("Add")`
Declaration Variables	These are variables which reference previous declarations, for example, the underlined variable in the below: `let a = 4 in a end;`
Recursive Variables	These are variables that reference the function being declared, *e.g.*: `fun sum n = n + (sum (n-1));`

Table 3.1: The 4 variable flavours

The intermediate state of the translation at this stage looks very much like pure lambda calculus, and contains only lambda-bound variables, on which Turner's translation rules can operate. The conversion starts from the innermost lambda expression and then propagates out until the outermost lambda expression is found and the whole program has been converted.

The alternative version of the compiler was also implemented so that, of the basic combinators, only `S`, `K` and `I` were supported.

3.1.5 The Memory Initialiser

A special kind of RAM block called a JTAG block is used to allow program files to be loaded onto the board from the PC and transferred to the working memory. Three memory initialisation files are required, the `CAR` initialisation file, the `CDR` initialisation file and the `SPECIAL` initialisation file. The `CAR` and `CDR` initialisation files correspond to the left and right sides of each node in the tree, as explained in section 2.5. The `SPECIAL` file indicates where the program ends and the free memory chain begins. The free memory chain is a linked list of the unused cells that resides in the `CAR` partition of the memory.

Application is represented in the underlying structure of memory. If the value of the `CAR` memory in some cell is A and the value of the `CDR` memory in that same cell is B, then this has the implicit meaning *"apply A to B"*. Each combinator has a designated opcode, as explained in section 3.3. If an application consists of primitive combinators then it can be trivially mapped to memory through the insertion of the associated opcodes, as indicated by figure 3.4.

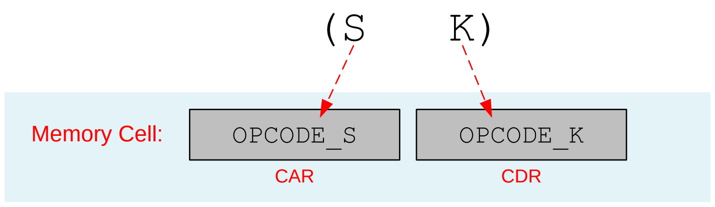


Figure 3.4: Mapping simple combinators into memory

In an expression of the form `APP(`A`,`B`)`, A and B may themselves contain further applications. In this case, we must first establish A and B into their positions in memory, and then create a pointer to them in the current `APP` cell.

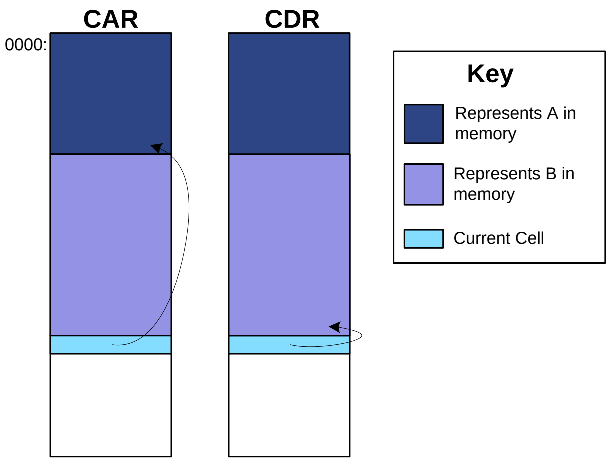


Figure 3.5: Mapping `APP(`A`,`B`)` *in memory*

Establishing combinator expressions into their appropriate positions in memory (*i.e.* in the memory initialisation files) is done using a function called `make_MIF`. It takes the input combinator expression and appends it to the MIF file (which represents a contiguous memory block), returning the address that can be used to reference that expression. So for `APP(`A`,`B`)`, the function first maps A into memory, then saves the address to reference A and does the same for B, finally instantiating the cell to point to A and B. Figure 3.5 illustrates this diagrammatically.

3.2 The Unit Tests

The unit tests were all written in ML, and comprise a test suite which can be used to check different aspects of the compiler.

3.2.1 Pretty Printer

The Pretty Printer takes an abstract syntax tree and turns it into a string that is semantically equivalent to the original program. This is a simple process of identifying mini-ASTs that correspond to particular programming constructs and filling in the concrete syntax. The pretty printer is useful for ensuring that the parser has interpreted the expression correctly and serves as a basic sanity check.

3.2.2 Abstract Syntax Tree Reducer

The AST reducer takes an abstract syntax tree produced by the parser and evaluates it. It does this by repeatedly performing single reduction steps until a value (integer or lambda term) has been obtained, which is then returned in the form of an AST.

Built-in operations, such as integer arithmetic, are looked up in the initial environment and use the reserved `Inl` constructor. For example, the addition operator looks as follows:

```
Inl(fn a => (case a of Int(i) => Inl(fn b => (case b of Int(i2) => Int(i+i2)))))
```

The AST reducer was particularly useful at the start of the project and during the initial verification of the parser. Later in the project, I found the software combinator reducer more useful as it was more closely related to the reduction mechanism of the processor.

3.2.3 Software Combinator Reduction

The Software Combinator Reducer takes a program in the form of a combinator stream from the combinator converter and reduces it to a value. Again it uses the single-step reduction method, and does this repeatedly until no more reductions can be performed. The algorithm is fairly straightforward as it simply applies the relevant combinator reduction rule. The snippet of code below shows the reduction of an S combinator in the reducer. The boolean in the returned pair indicates that a reduction occurred.

```
reduce_single(APP(APP(APP(S, P), Q), R), env) = (true, APP(APP(P,R),APP(Q,R)))
```

3.3 The Instruction Set Architecture (ISA)

Instructions for the processor are fixed length 32-bit instructions. There are 3 classes of instruction: *application instructions*, *combinator instructions* and *integer instructions*. The classes of instruction are distinguished by an opcode in the top two bits of the instruction. The format of instructions in each class is described below and in figure 3.6.

Application Pointer Instructions: These are pointers to other cells. The combinator `APP(`A`,`B`)` is represented by two instructions, one in the `CAR` portion of the cell pointing to A and one in the `CDR` portion of the cell pointing to B (assuming A and B are pointers and not simple combinators). Note that a cell's address refers to both the `CAR` and `CDR` partitions, so a cell always contains two instructions.

Combinator Instructions: These perform a manipulation of the graph in memory. These instructions represent the basic combinators `S`, `K`, `I`, `B`, `C` and `Y`, and the arithmetic operators.

A full list of the `OP_comb` codes for the basic combinators and the `OP_arith` codes for the mathematical ones can be found in appendix C.

Integer Instructions: These represent 28-bit integer primitives.

unused

opcode

b	Bit value indicating if memory location is off-chip
Addr	Address of memory location

APPLICATION INSTRUCTION

31 : 30	29 : 28	27	26 : 20	19:0
0 1		**b**		**Addr**

OP_comb	Combinator (S, K etc.) identifier
OP_arith	Arithmetic operator (+, -, etc.) identifier
args	Number of arguments reduced for arithmetic operator
d	Bit indicating if arithmetic operator takes one or two arguments

COMBINATOR INSTRUCTION

31 : 30	29 : 28
1 1	

27	26 : 8	7 : 0
0		**OP_comb**

27	26 : 12	11	10 : 8	7:0
1		**d**	**args**	**OP_arith**

---

int_val	28 bit integer value

INTEGER INSTRUCTION

31 : 30	29 : 28	27 : 0
1 0		**int_val**

Figure 3.6: ISA: Instruction format

3.4 The Processor

The processor was written in Verilog and consists of six logically separate components that interact in order to perform the required reduction functionality. These are:

- The core of the processor: the reduction engine
- The on-chip memory unit and the memory management unit
- The off-chip communication interface: the SDRAM arbiter
- The memory initialisation and reprogramming interface
- The results interface
- The garbage collector

For the processor to work, all of these units have to be able to pass control smoothly between them.

3.4.1 Overview

The main purpose of the processor is to perform the combinator reduction in hardware. To do this, a facility is required to initialise the memory and read the results. Figure 3.7 shows the life cycle of the fully-fledged processor.

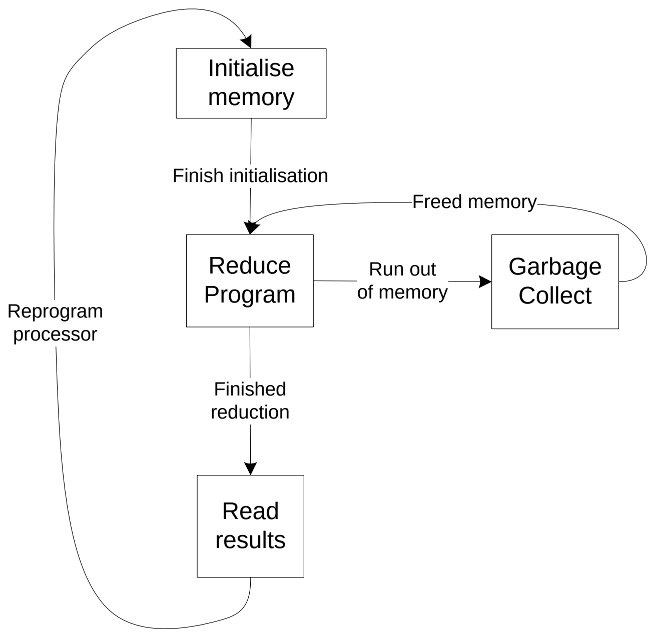


Figure 3.7: The life cycle of a fully-fledged processor

The processor first has to be initialised with a program for it to reduce. This is done using the JTAG modules. The initialisation protocol can be found in appendix F and describes how the switches can be used to load a program onto the board. Once reduction has started, the processor runs automatically until reduction is complete, periodically garbage collecting when it runs out of memory. The results of the reduction can be displayed on the HEX display or read back to the host PC using a protocol similar to that used during initialisation.

A high level overview of the processor is shown in figure 3.8. There are three routes of execution which I will refer to as *normal execution* (the top route in the diagram), *combinator execution* (the middle route) and *arithmetic execution* (the bottom route).

Verilog is a high-level language, based fundamentally on the parallelism of the hardware it is targeted at. The notion of time is imposed by global clocks, registers and pipelining. However, the reduction architecture implemented has a limit on the amount of parallelism it can exploit due to the memory accesses that must be performed. A RAM block on the FPGA can be instantiated to have two read ports and a single write port. Since the `CAR` and `CDR` memory partitions are mapped onto different RAM blocks, two reads and a single write to each partition can be performed in a single clock cycle.

The architecture must therefore have a control structure which maximises what can be done in a clock cycle, but yet obeys the limits imposed by the memory structure. The aim of figure 3.8 is to illustrate how control can be exerted by multiplexing paths of execution through the use of control registers.

3.4.2 Normal Execution

When the processor is in a *normal execution* mode, it walks the combinator tree to find the next combinator to execute. Once a combinator is found, the processor changes into a different mode depending on which class the instruction belongs to. If an integer value is found, execution terminates and the processor proceeds to tidy up the graph. When a combinator instruction is found, it sets up the control registers such that the next cycle will be in *combinator execution* mode. To avoid wasting a memory cycle, the processor also performs the first stage of that combinator's execution. This is called the *preprocessing* stage.

In order to establish which action to take, the instruction in the `CAR` partition of the memory cell indexed by the **P** pointer must be retrieved. The **P** pointer always points to the current instruction. The opcode in the top two bits of the instruction determines its class. If it is a combinator instruction, then its 27th bit can be used to distinguish between arithmetic operators and basic combinators.

If an arithmetic operator is found, the processor sets the control registers to force the transition to *arithmetic execution* mode: `do_arith` is a register indicating that the processor should perform arithmetic execution, `arith_op` is a register indicating what the arithmetic operator is, and `arith_stage` indicates what stage of execution the processor is currently on. A similar set of registers exist to force a transition to *combinator execution* mode. The *preprocessing stage* will be explained in the combinator and arithmetic section because it is easier to understand in that context.

If the instruction is an application pointer, the processor stays in normal execution mode and follows the pointer. As the pointer is followed, the tree is rearranged so that the processor

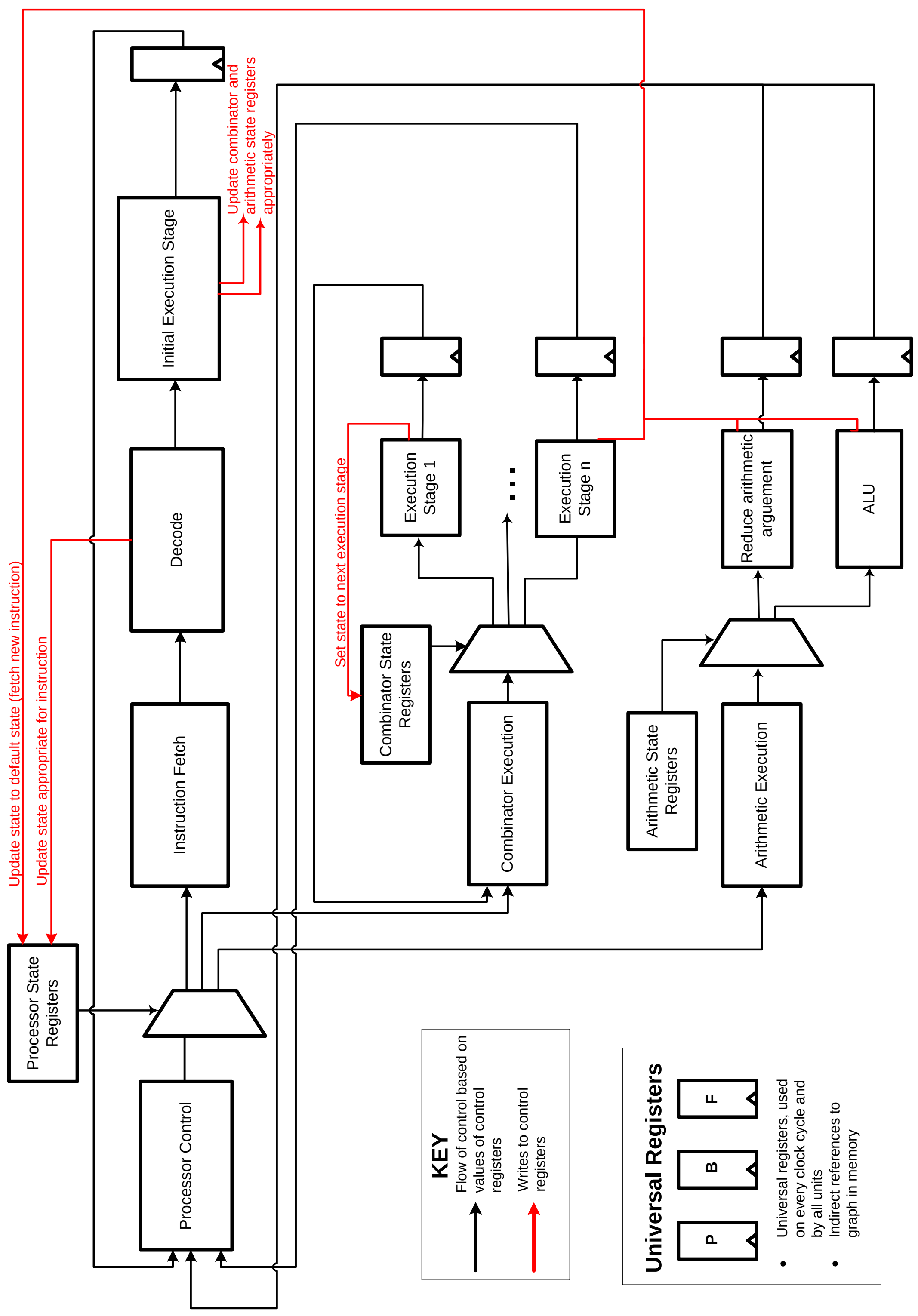

Processor State Registers
Processor Control
Instruction Fetch
Decode
Initial Execution Stage
Update state to default state (fetch new instruction)
Update state appropriate for instruction
Update combinator and arithmetic state registers appropriately
Set state to next execution stage
Combinator State Registers
Combinator Execution
Execution Stage 1
Execution Stage n
Arithmetic State Registers
Arithmetic Execution
Reduce arithmetic arguement
ALU
KEY
Flow of control based on values of control registers
Writes to control registers
Universal Registers
P
B
F
Universal registers, used on every clock cycle and by all units
Indirect references to graph in memory

Figure 3.8: Processor Architecture Overview

can retrace its steps. The process to do this was explained briefly in section 2.5, but we will consider it more specifically here.

Note that in the below I have used the Verilog style of non-blocking assignment – all of the above items occur simultaneously, so the values will all be updated for the next clock cycle. The process is as follows:

- Set the value held in `CAR[P]` to the back pointer, so we can retrace our steps: `CAR[P] <= B;`
- Update **P** to the value held in the current cell: `P <= CAR[P];`
- Update **B** to point to our current cell: `B <= P;`

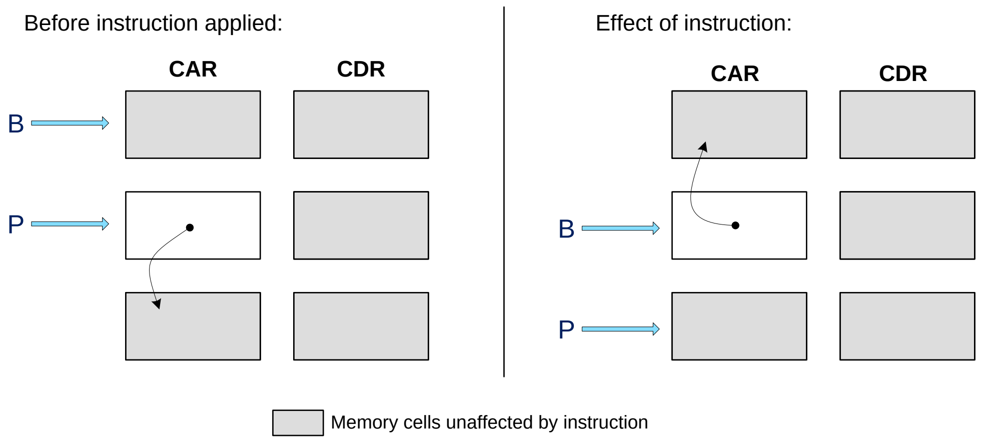

Before instruction applied:
CAR
CDR
B
P
Effect of instruction:
CAR
CDR
B
P
Memory cells unaffected by instruction

Figure 3.9: Effect of an application instruction

This process is also shown in figure 3.9.

So, in order to execute this instruction, a read from **B** and a read from **P** must be performed, as well as a write to the cell pointed to by **P**. This is exactly 2 reads and one write to the `CAR` partition and hence only requires a single clock cycle to execute.

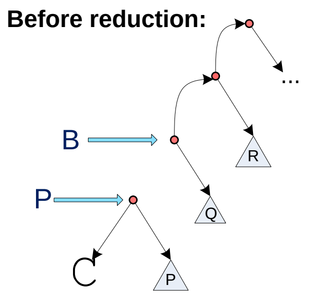
3.4.3 Combinator Execution

The combinator execution part of the processor was responsible for the reduction of the `S`,`K`,`I`,`B`,`C` and `Y` combinators. I will illustrate the implementation strategy used to perform one combinator reduction as an example. Consider the application of a `C` combinator as shown below:

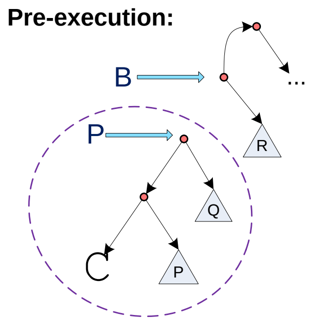
$$\mathtt{C}\ P\ Q\ R \rightarrow P\ R\ Q$$

This needs to be done in a way that will leave all the memory cells unchanged except for the cell at the root of the tree. To avoid having to reperform this reduction, the root of the subtree that corresponds to the combinator and its arguments, is overwritten with the reduced result. All other nodes in this subtree are preserved to allow for the presence of shared pointers.

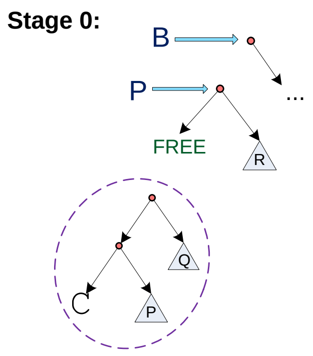
The stages of reduction are shown in the margin of this page. Initially, when we first find the combinator, the tree is in the state shown at the very top right of the page, and we need to transform it to the bottom stage. In this *Before Reduction* stage, there is no direct pointer to the *R* tree, so we need to retrace our steps a bit, restoring the tree at the same time. The process of retracing back up the tree is referred to as *unwinding*. In the *pre-execution stage* (the stage that happens in normal execution mode), the following occurs:



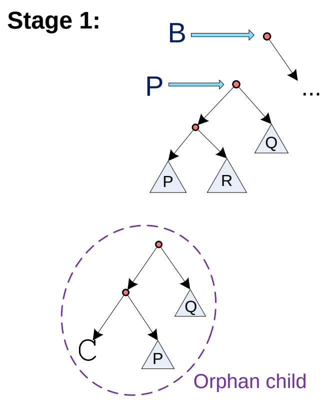
- *Unwind*: `B <= CAR[B]; P <= B;`
- Store the first argument in a register: `arg1 <=` *P*
- Restore the tree: `CAR[B] <= P;`



This creates the tree shown under *pre-execution stage*. The effect of restoring the tree is shown in the purple circle in the diagram; this tree now looks like an unreduced tree. After the pre-execution stage, the processor moves to combinator execution mode to perform the remaining stages of reduction. In this stage, the remainder of the arguments must be retrieved, a free memory cell must be obtained and we must *unwind* the tree a stage further. To get a free memory cell from the *free memory list*, the following must be performed:



1. Create a pointer to the cell indexed by register **F**, the free memory pointer. This cell is our new "free cell". Read from the cell indexed by **F**.
2. Set **F** to the cell indexed by **F**, effectively performing a "tail" operation on the linked list.



The above must be done in two cycles. In this stage we can create the pointer to the free memory cell by doing a write to `CAR[B]` with the value of **F**. To utilise the full memory bandwidth, the tree is also unwound a step in this stage. Finally, the third argument, *R*, is stored in a register and the value of the second argument, *Q*, is written to `CDR[B]`. The results of this stage are shown in the *Stage 0* tree. Note, in particular, that the tree that we restored has now been completely separated from our working tree.

All that remains is to instantiate the free cell with the relevant `CAR` and `CDR` values. Thus in *Stage 1*, we simply write to the free cell that was claimed, with the first and third arguments that were stored, *i.e.*:

`CAR[FREE] <=` *P*`; CDR[FREE] <=` *R*`;`

Reduction of the `C` combinator is now finished and the resulting tree is shown at the bottom of the diagram. The orphan tree has been restored and abandoned, allowing any existing pointers to the subtree to be unaffected by the combinator application. The control registers of the processor are reset so that control returns to *normal execution* mode.

Similar procedures apply to the other combinators. The execution time of different combinators is determined by the number of memory accesses they require to perform the appropriate graph manipulation.

3.4.4 Arithmetic Execution

In section 2.5, I mentioned an issue that arises with arithmetic execution due to the lazy nature of combinators, and I will now explain how this issue was handled in the implementation. The underlying principle of the solution is to divide each arithmetic operator into $k + 1$ operations, where k is the number of of arguments taken by that operator. The operator functionality then depends on how many arguments it has reduced.

We consider the case of the plus operator which has two arguments. If no arguments have been reduced, then the functionality of the operator is to rearrange the graph in such a way that its first argument will be reduced. The resulting integer does not cause termination of reduction as it is an integer *argument* rather than a *value*. Integer *arguments* and integer *values* are distinguished by probing what they are being applied to. If a partially executed arithmetic operator is found in the right-hand side of the cell, it has to be an integer argument and the processor rearranges the graph so that the next bit of reduction can be performed.

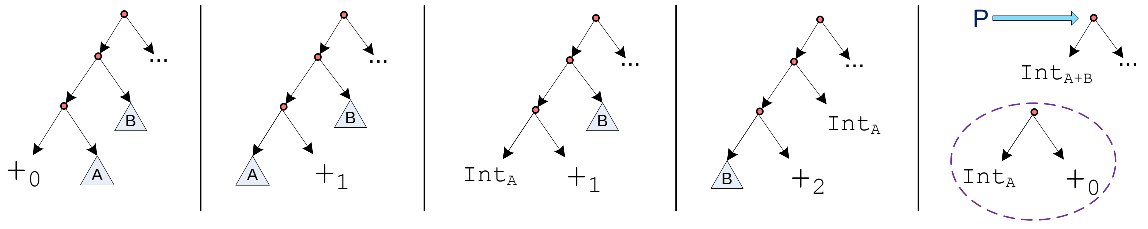


Figure 3.10: Arithmetic reduction implementation

Figure 3.10 shows the sequence of trees through which arithmetic reduction progresses. Note that the trees are not consecutive in processor cycles as multiple steps may be required between stages. In particular, the reduction of either of the arguments involves the reduction of potentially large trees to an integer before the next stage of the tree is reached. We can clearly see, however, that the problem of laziness within the setting of arithmetic has been solved, and it has been done in a stack-free way.

3.4.5 Execution Termination

Execution termination performs exactly the opposite of application instructions and walks back up the tree, restoring it at the same time. This continues until the root of the tree is found, where execution terminates.

3.4.6 On-chip Memory Model

In the explanation of the processor implementation, an *array access* model of memory has been assumed. That is, we have assumed that we can say `P <= CAR[B]`. This was the model of memory used in the first few iterations of the processor since it is the most intuitive and easiest to work with. This sort of memory model is supported by the synthesis tool, Synplify.

Communication with the off-chip SDRAM occurs through the use of a SDRAM module which can be instantiated through Quartus' SOPC builder. Synplify's support for Quartus' auto-generated SDRAM controller files is temperamental. In particular, it shows irrational behaviour when using a 32-bit controller and fundamentally does not appear to meet the functional specification laid out by the Quartus documentation. As a result, I decided to port the processor to Quartus.

Quartus does not support the inference of RAM from arrays to the same degree as Synplify, and insufficiently for the processor. To use the on-chip RAM in Quartus, RAM modules have to be instantiated with explicit signals for reads and writes. This was quite a substantial change; the code below shows a memory access using the array style of memory and using the explicit RAM model:

Array Model
```
P <= CAR[P];
```

Explicit RAM model
```
if (start) begin
   car_rd_addr1 <= P;
   start <= 0;
end else begin
   P <= car_rd_data
end
```

From this example, we can see that the explicit model of RAM takes two cycles while the array model takes only one. This transformation is detrimental and in the worst case, could cause the processor performance to halve. So instead, the implementation aims to mimic the array access method to reduce performance loss. This was done by carefully clocking the RAM module relative to the processor clock. The crucial trick is to clock the RAM module twice as fast as that of the processor, skew it such that the values can be read from the RAM before the negative edge of the processor clock, and update the general purpose pointers **B**, **P** and **F** on the negative edge. Figure 3.11 illustrates the example with the new model:

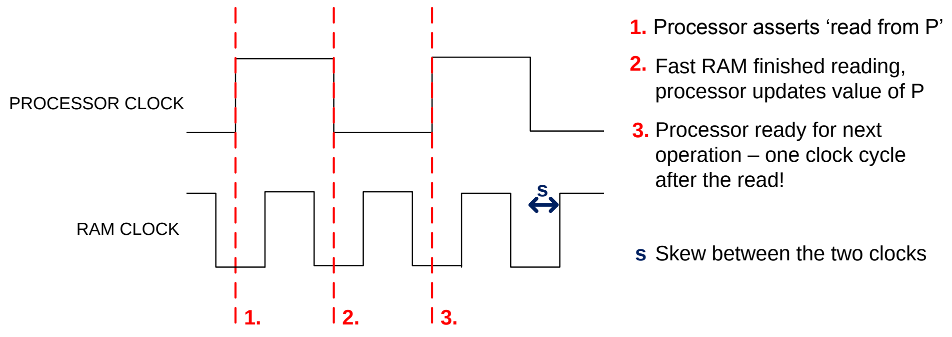

PROCESSOR CLOCK
RAM CLOCK
s
1.
2.
3.
1. Processor asserts 'read from P'
2. Fast RAM finished reading, processor updates value of P
3. Processor ready for next operation – one clock cycle after the read!
s Skew between the two clocks

Figure 3.11: On-chip memory model

Quartus' use of explicit signals to perform communication with the on-chip memory is very similar to the signals used to communicate with the SDRAM controller. This meant that integrating off-chip communication with the existing model was made much easier by the transition from Synplify to Quartus.

3.4.7 Off-chip Memory

The SDRAM on the FPGA board has 8MB of storage which is much larger than the 60KB available on-chip. Communication with the SDRAM has already been mentioned, so I will assume that we have a working SDRAM controller to perform the communication, and discuss the SDRAM arbiter that was built to interface between the controller and the processor.

In the on-chip model, a maximum of six memory operations can be performed per processor clock cycle, namely two reads and a write to the `CAR` partition, and the same to the `CDR` partition. Consequently, the SDRAM arbiter takes a maximum of six operations, executes those that are required and stalls the processor while they are performed. The reduction mechanism of the processor is abstracted away from the level of memory accesses through the use of logical addressing. This modularisation is useful for testing and simplifies operation significantly.

For every logical memory request, a part of the processor, the *MMU* (Memory Management Unit), is responsible for determining if the access is on-chip or off-chip and then propagates the appropriate signals to each section (either the two RAM partitions or the off-chip memory). This propagation of signals is all done combinatorially, so no extra delay is incurred. The arbiter selects between the memory requests so that read requests are performed before write requests. Figure 3.12 illustrates how the different parts of the design were connected to implement off-chip communication.

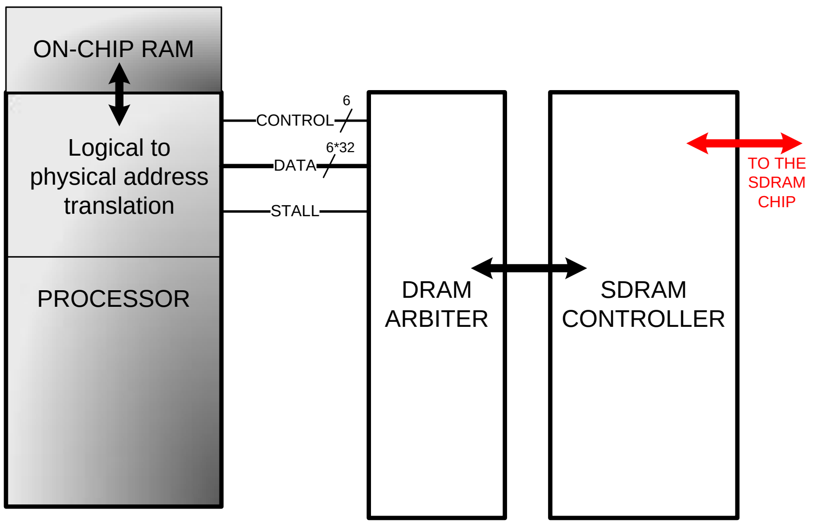


Figure 3.12: Off-chip memory model

3.4.8 Garbage Collection

The memory on the off-chip SDRAM is divided into two partitions, and on each garbage collection, the working memory switches between the spaces. The idea of the copy garbage collection algorithm is to copy the *working trees* into the new space. The working trees are defined as those trees indexed by the **B** or **P** pointers.

The method used is a stackless one. On each iteration of the algorithm, a single pointer is followed and the referenced cell copied. This continues until all reachable cells have been copied into the new space.

The algorithm for non-recursive garbage collection was first presented by Cheyney [7] in 1970. The implementation essentially involved the translation of his algorithm into Verilog while ensuring that memory accesses fit into the existing memory model. Figure 3.13 shows the effect of garbage collection on the graph in memory.

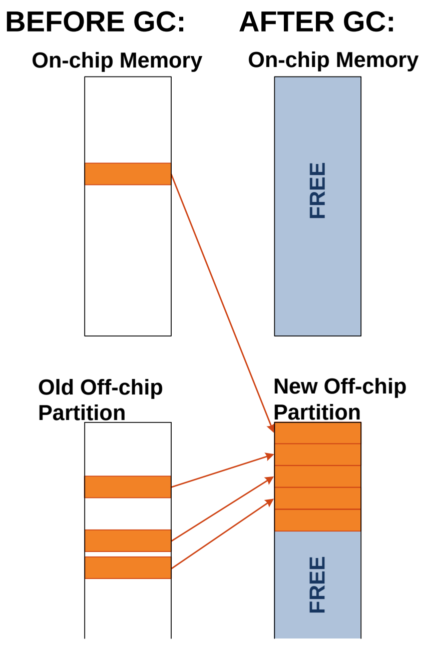


Figure 3.13: Garbage collection

Garbage collection is triggered automatically when the free memory pointer, `F`, exceeds the maximum address. Once the working set has been copied, the processor is set up so that reduction can begin again. The implementation does this with the following steps:

- Redirect **P** and **B** to point to the new off-chip partition where their working trees now reside.
- Reinitialise the free memory chain by iterating over the free memory cells, starting with the on-chip memory, which is now completely free, and then working through the off-chip memory from the point where the working tree ended.

- Redirect **F**, the free memory pointer, to the first cell of the on-chip memory. At this point, garbage collection ends and reduction resumes.

3.5 Drivers and Imperative Features

ML is a functional language that supports imperative features. As an extension to the project, I added some simple functionality to allow imperative features to be added to the processor. These are left sufficiently general to allow the programmer to control their use.

In conventional processing and operating systems, drivers are written to allow the programmer to access the functionality of external hardware. In my processor implementation, `IN` and `OUT` combinators are added to allow the user to access some functionality outside the scope of the processor itself. The `IN` combinator takes an integer and returns an integer and the `OUT` combinator takes an integer and a continuation, where the integer is the port to be accessed and the continuation is what to do after that access has taken place. There are several variants of the `OUT` combinator to allow the programmer to pass more arguments to the driver than just the port number. The functions `OUT` and `IN` are also added to the compiler to allow full support through the toolchain.

It is left up to the designer to decide how they will specifically wire in extra modules, or *drivers*, to the processor which use the `IN` and `OUT` combinators. In my implementation, I wrote drivers to allow the `IN` and `OUT` combinators to access the switches and VGA respectively. The port provided to the `IN` combinator dictated which switch will be accessed, returning an integer that is 1 or 0, and a variant on the `OUT` combinator takes 3 integer arguments which are used by the driver as the x-coordinate, y-coordinate and colour of the pixel being set on the screen. The VGA driver stores the pixel values in its own section of memory and updates the memory array according to the `OUT` instructions. A renderer then sets the pixels on the screen according to the values held in the memory array.

The particular drivers I implemented were needed for the applications in section 4.2.2, but different drivers could be written for other purposes. Standard ML supports references which allow state to be kept independent of function applications. It is plausible that references could be expressed using the `IN` and `OUT` instructions. If a driver were written to allow user programs to have their own "heap" which could be accessed via variants on `IN` and `OUT` instructions, then the compiler could effectively translate reference accesses to heap accesses through the `IN` and `OUT` instructions.

Chapter 4

Evaluation

Performance analyses of several aspects of the processor are presented. A comparison to a current research implementation is made. Finally, two example applications are presented, demonstrating the computational power of the processor and how the system as a whole can be used to run large ML* programs.

4.1 Performance Analysis

Incrementing counters were added to the processor design to obtain the statistics of various aspects of the computation. This allowed the benchmarking of programs on the CPU by tracking the number of execution cycles. Cycle count can be trivially converted to seconds by dividing by the clock rate of the processor (10 MHz). The tables for all results can be found in appendix D.

4.1.1 Basic Performance

The processor performance was analysed with a simple recursive program which naïvely computes the Fibonacci sequence. The implementation in ML* is shown below.

```
fun fib n =
        if (n = 0) then 0 else
                   if (n = 1) then 1 else ((fib (n-1)) + (fib (n-2)));
```

The graph in Figure 4.1 shows the number of cycles taken to calculate `fib` n. The black bars show the performance on a basic version of the processor, and the grey bars on an optimised version, using twice the amount of on-chip memory.

The results for the unoptimised version can be divided into four regions. I will discuss each region in turn before looking at the overall performance and the effect of increased on-chip memory.

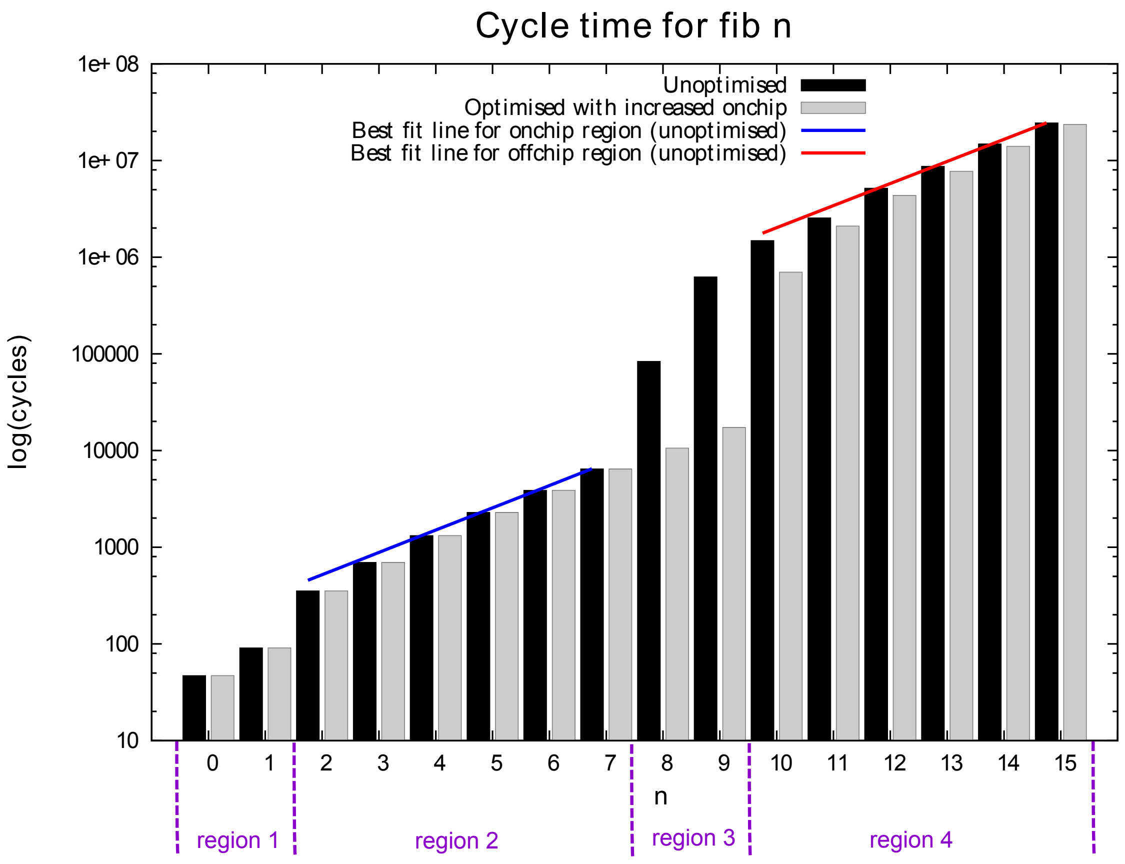


Figure 4.1: Fibonacci performance

Region 1

If we look at the code for the Fibonacci function, it is logical that `fib 0` and `fib 1` will have a shorter execution time than any other values since no recursive calls are required to perform their calculation. This is supported by the graph as the cycle times to calculate `fib 0` and `fib 1` are small.

Region 2

Region 2 of the results corresponds to values of `fib` for which only the on-chip portion of memory is utilised to calculate the result. The blue line shows the best fit function for the cycle values in this region. The equation of this line is $a \times (b^n)$, where a and b minimise the error of the best fit line, and were respectively found[1] to be 157 and 1.700.

Hence, the time complexity of calculations in this region is $\Theta(1.7^n)$. The tight bound for the Fibonacci algorithm can be found [5] as $\Theta(1.618^n)$. So, taking into account experimental

[1]Calculation for best fit performed using the *fit* facility in gnuplot

variations and a certain margin of error, the values of cycle time in region 2 fit the complexity of the processed function, and indicate that the processor is scaling to perform larger computations in a predictable way.

Region 3

Calculation of `fib` for $(n \geq 8)$ requires the utilisation of the off-chip memory. This is reflected in the graph as there is a substantial increase in cycle time for `fib 8` which does not fit the pattern displayed in region 2. Approximately thirteen times as many cycles are required to compute `fib 8` as are required to compute `fib 7`.

In section 3.4.6, I outlined how the on-chip memory model can perform six memory operations in parallel per cycle, while the off-chip model must perform them sequentially. We would therefore expect performance to decrease by approximately a factor of six when the off-chip is used. In addition, the computation itself should have approximately twice as many recursions, which means we would expect `fib 8` to take around twelve times as long as `fib 7`, which is indeed the trend observed in the data.

Region 4

In region 4, the computation is occurring primarily in off-chip memory. It has once again returned to the stable complexity pattern seen in region 2 and as such, the red best fit line has a very similar gradient to that of the best fit line in region 2.

Summary

As the above results demonstrate, the processor exhibits rational computational behaviour in the context of its implementation; the pattern of this behaviour is dictated by the program and its memory requirements. The impact of using twice the amount of on-chip memory is clearly reflected in the graph, as the "jump", due to the utilisation of off-chip memory, does not occur until `fib 10`.

Clearly, the code for `fib` is naïve so we would not expect it to perform efficiently. However, the example illustrates that the processor has the ability to perform computations with a significant recursive overhead.

The use of off-chip memory results in a noticeable degradation of performance. So far, however, the computation has not required garbage collection. We will now look at the effect of garbage collection, and how it can help to make performance scale more elegantly.

4.1.2 Garbage Collection

Garbage collection serves to compact the working set into a new partition of off-chip memory and means that the on-chip memory can be reused in future computation. Thus, it may be desirable to perform garbage collection frequently, if the overhead of performing garbage collection is outweighed by the gains from the more frequent reuse of on-chip memory. To see the effect of this, we consider the time to execute a benchmark program while varying the *garbage collection trigger point*, the threshold amount of off-chip memory that is used before garbage collection occurs.

I performed this analysis using the benchmark program `isPrime`, which takes an integer and naïvely computes whether that integer is prime. The program uses repeated subtraction to implement a remainder function, and iterates over all values up to the integer being tested to see if any of them are divisors. The ML* program can be found in the appendix B. We will look at running `isPrime 1477`, which requires one garbage collection in order to complete the computation when it is using all available off-chip memory. The program terminates returning `true`.

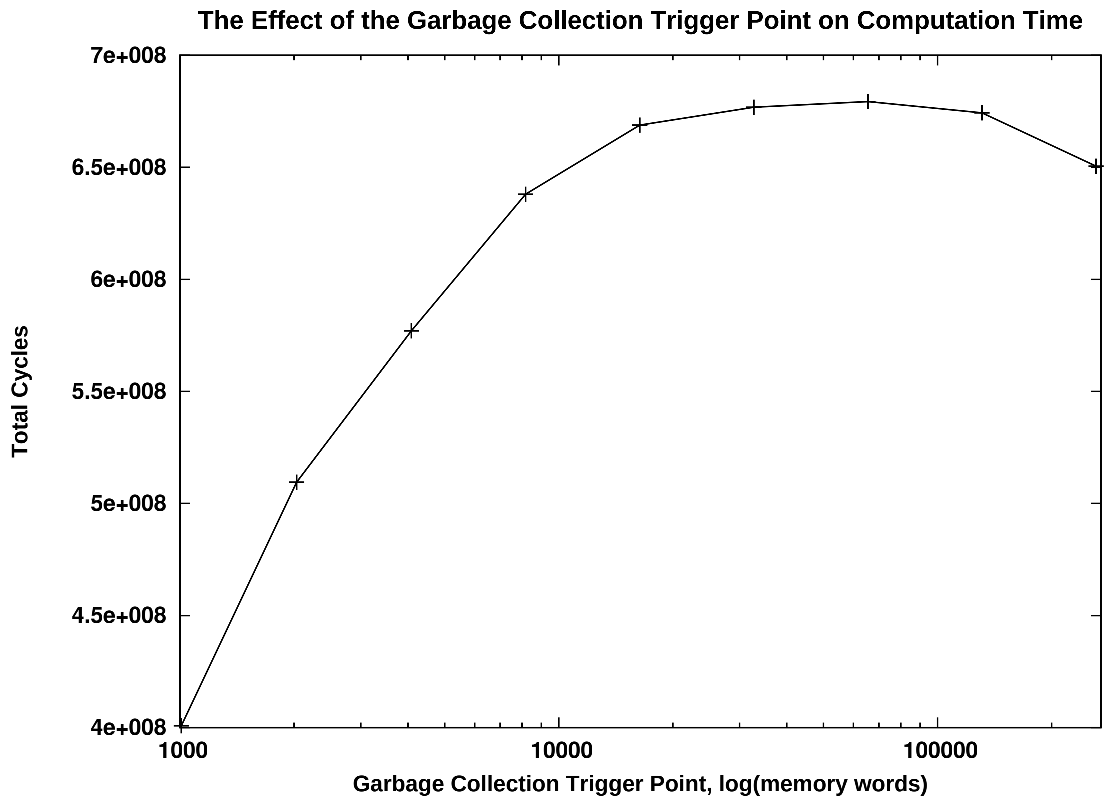


Figure 4.2: The effect of the garbage collection trigger point on the computation time

The graph in figure 4.2 shows the total number of cycles taken to calculate `isPrime 1477`. The *garbage collection trigger point* starts off at around 1,000, meaning that garbage collection will occur after 1,000 words of off-chip memory have been used. This ranges up to the full off-chip memory, increasing by a factor of two for each result taken.

The results are interesting and not immediately intuitive. When the trigger point is set to a small value, the computation takes the least time. Recall that part of the garbage collection process is the reinitialisation of the *free memory chain*, allowing reuse of the memory that was freed during GC. To do this, the processor must iterate over the working partition of memory. This working partition is proportional to the GC trigger point and hence a smaller trigger point value results in less of an overhead from GC.

Based on the analysis so far, it might be expected that the performance will degrade further with greater values for the GC trigger, however this is not the case. Looking particularly at the last two values in the series, we can see that the use of the full off-chip memory is quite significantly better than only using half of the off-chip memory and garbage collecting more frequently. In this case, we have the converse of what was happening for the low values of the trigger point. The garbage collection overhead is now much more significant due to larger trees residing in the sections of memory to be garbage collected, and due to the overhead of reinitialising the working partition with the free memory chain. In the penultimate result of the series, two garbage collections are performed, while in the last only one is performed. The last result therefore outperforms the penultimate one, as it does not have to incur the overhead of garbage collection twice for relatively small gains in the use of the on-chip memory.

The results discussed are specific to the combinator tree for `isPrime 1477`. The exact GC trigger point that will produce the fastest results varies per program, as it depends on the time taken to garbage collect the *working set*. For this reason, the garbage collection trigger point was made a parameter in one of the initialisation files.

As already discussed, one of the major overheads of garbage collection is the reinitialisation of the free memory chain. Since the garbage collection scheme implemented compacts the tree into a single portion of the off-chip memory, it is not in fact necessary to use the free memory chain at all – essentially, the free memory pointer can just be incremented to retrieve a new free cell.

Performing this optimisation was found to substantially improve results. The improvements found from testing `isPrime 1477`, with the GC trigger set to the maximum, found the below improvement in results. The time spent performing garbage collection has reduced by factor of 1,000 (this makes sense as the reinitialisation of the free memory chain requires hundreds of thousands of memory accesses).

	Unoptimised	**Optimised**
Cycles ($\times 10^3$) spent doing garbage collection	14,596	12
Cycles ($\times 10^3$) spent doing total calculation	650,510	622,776

This optimisation is specific to a compacting garbage collection scheme and cannot be applied in general. If, for example, a parallel garbage collector was implemented, then the free memory chain would have to be used.

4.1.3 Comparison to Other Architectures

It should be noted that the original proposal suggested a comparison to a C++ software version of the processor. However, it seemed more interesting to compare against an existing external implementation.

The Reduceron Series

Recently, a research project looking at the implementation of Haskell on graph reduction machines has been undertaken by Naylor *et al.* at the University of York. The project is on its third iteration of development [4]. The *Reduceron1* is a graph reduction machine that works on a similar set of basic combinators to my implementation. There is little published material available about this iteration of development as it served as a proof of concept for Naylor's PhD. It did not have a garbage collector and was only tested on a few examples. The second Reduceron is significantly more complex than the first prototype. The reduction mechanism [12] uses multiple stacks and a heap in order to perform reduction and has complex on-chip optimisations to improve efficiency. The *Reduceron2* no longer operates on simple combinators, but instead uses a method of supercombinators and template instantiation, which is beyond the scope of this dissertation. Neither the *Reduceron1* nor the *Reduceron2* utilise the off-chip memory of the FPGA. Table 4.1 summarises some of the comparable features of the *Reduceron1*, the *Reduceron2* and my implementation.

Feature	**Reduceron1**	**Reduceron2**	**My Processor**
FPGA synthesized on	Xilinx Spartan IIe	Xilinx Virtex II	Altera DE2 Teaching Board
Instruction Format	Combinators	Supercombinators	Combinators
Supports Garbage Collection	No	Yes	Yes
Maximum Clock Speed	25 MHz	91.5 MHz	10 MHz
Utilises Off-chip	No	No	Yes

Table 4.1: A comparison to the Reduceron project

Efficiency and Performance

The Reduceron project and my project approached the graph reduction problem on FPGAs from two different perspectives. The Reduceron project focuses primarily on performance and efficiency while the focus in this project was that of scalability. The architectures are quite different as a result of this decision.

The results provided by the research group at York suggest that my implementation outperforms the *Reduceron1* by approximately a factor of four in clock cycles while the calculations remains on-chip. The number of clock cycles to calculate `fib 3` using my implementation and the *Reduceron1*[2] is shown below. This factor of four performance difference would be expected to roughly remain the same for larger values of `fib` while the computation remained on-chip in both cases.

	The Reduceron1	**My Project Implementation**
`fib 3`	3328	694

The Reduceron implementation uses substantially more on-chip memory than my implementation - more specifically, a Xilinx Virtex board has approximately 126KB of memory

[2]Comparison made based on results published on the Reduceron website [4]

compared to the Altera DE2 board which has only 60KB. In my implementation, computation remains on-chip until `fib 7` (or `fib 9` if the amount of on-chip memory is doubled), while the Reduceron can compute `fib 10` on-chip.

The *Reduceron2* is a much more powerful implementation than the *Reduceron1*. In the calculations of `fib`, the *Reduceron2* outperforms my implementation by a factor of six in clock cycles while the computation remains on-chip. I suspect the primary reason for this difference is the greater amount of parallelism exploited by their design in accessing the on-chip memory.

Architectures and Scalability

As already explained, the *Reduceron2* is a stack-based reduction machine with multiple stacks and a heap. To improve the efficiency of the *Reduceron2* implementation, block RAMs on the FPGA are cascaded. The values read from the cascaded RAM blocks can then be rotated in order to retrieve the whole word required for execution. This is a similar principle to that used in this project where the memory is divided into `CAR` and `CDR` partitions in order to exploit the underlying parallelism of the RAM blocks on the FPGA. The research at York has pushed this parallelism to a much higher level than in my implementation. One of the advantages of the stack model in the York implementation is that it provides more natural parallelism in the memory accesses, since each stack and the heap can be accessed simultaneously.

The Reduceron only uses on-chip memory. This was motivated in part by the large on-chip resources that were available on the FPGA being used. Naylor acknowledges[3] that the utilisation of the off-chip memory using a Reduceron style of reduction would have to be done carefully to avoid large performance degradation due to the latency of the off-chip memory. My project has gone some way towards demonstrating how this can be done, with the on-chip resource being reused through garbage collection. Using a stack-based architecture makes the utilisation of the off-chip partition more difficult, as each of the stacks and the heap would need to be able to spill over cleanly into the off-chip memory.

The Reduceron is limited to supporting programs with 4K words on the program stack. In comparison, this implementation allows programs of 250K words to be supported. The stack-based architecture also puts a limit on the depth of recursion and computation that can be supported. In the *Reduceron2*, the maximum heap size is 32K words (in the Reduceron implementation, the heap is used to reduce the function currently being applied). My implementation does not have a limit on the computation and recursion depth it can perform, provided that the program fits into the off-chip partition.

Finally, we look at the language subsets of ML and Haskell accompanying this project and the *Reduceron2*. The Reduceron's subset of Haskell supports the declaration of datatypes, which is something that my ML* compiler does not support. Programs running on the *Reduceron2* must return an integer. This is not required in my implementation, though it is the most readable form of output. The user can, however, retrieve the tree returned by the reduction through the use of the JTAG functionality on the processor.

Finally, the Reduceron does not support imperative features. My project has gone some way towards this with the use of the `IN` and `OUT` combinators. It would be possible to use these

[3]Personal correspondence, 27 April 2009

features to write a program in ML* that displayed the output of a reduction tree on a screen connected to the FPGA.

Summary

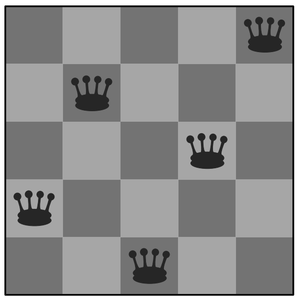
It is clear that the optimised *Reduceron2* outperforms my implementation. However, I hope I have demonstrated that this implementation and design scales to use the off-chip partition of memory more easily than that of the Reduceron. I have tried to demonstrate the way in which the focus of the two projects has caused their designs to differ, with the Reduceron project focusing on increasing the parallelism to improve efficiency, and this project has focused on using a relatively simple model and making it capable of using off-chip memory.

4.2 Applications

So far, I have concentrated on evaluating small benchmark programs on the processor. In this section, I present two larger programs written in ML* to illustrate the system as a whole. The two applications presented are *NQueens*, a list processing exercise, and *Ponginator*, the classic paddle-ball game.

4.2.1 NQueens

The NQueens problem aims to find the number of ways of positioning n queens on a chessboard which is $n \times n$ in size so that they cannot capture one another. The below diagram shows one solution to the problem when n equals 5.



To perform this calculation in ML, each solution is presented as a list – each column of the list represents a column of the board, and the value held at that list element is the row that the queen can safely reside at. For the example, presented above, its list representation would be `[4,2,5,3,1]`. The solution then uses recursion to solve the smaller $(n-1)$ problem down to the base case of a single element board. Once the $(n-1)$ solutions are found, the algorithm evaluates which row positions would be *safe* for a n^{th} queen to go in.

In ML*, lists are not primitives, so a lambda calculus representation of lists is used. There are multiple ways of representing lists in the lambda calculus, and the representation used in the ML* NQueens program is shown below:

```
fun cons h t = fn x => fn y => ((x h) t);
val nil = fn x => fn y => y;
fun head xs e = xs true e;
fun tail xs e = xs false e;
fun isNil xs = xs (fn h => fn t => false) true;
```

```
(* List possible row values a queen could be positioned in *)
fun fromAtoB p q =
   if (p > q) then nil else
      let val r = (fromAtoB (p + 1) q)
      in (cons p r)
      end;

(* Make a single element list *)
fun succeed n = cons n nil;

(* Detect if the list is a solution *)
fun isSolution s t = ((length s) = t);

fun abs w z = if (w > z) then (w − z) else (z − w);

(* Identify if a particular position is safe *)
fun is_safe bs bn =
   let fun nodiag bi blst =
      blst
         (fn h => fn t =>
            if ((abs bn h) != bi) then (nodiag (bi+1) b)
                                      else false)
         nil
   in
      if (not (member bn bs)) then (nodiag 1 bs) else false
   end;


(* Recursively compute the solution *)
fun nqueens en =
   let val enumN = fromAtoB 1 en
   in
      (let fun solutionsFrom es =
         if (isSolution es en) then (succeed es)
         else (concat
        (map solutionsFrom
           (map (fn eh => cons eh es) (filter (is_safe es) enumN))))
      in solutionsFrom nil
      end)
   end;

(* Find number of solutions to 5x5 problem *)
length (nqueens 5);
```

Figure 4.3: The NQueens Application

Part of the ML* version of the NQueens problem is in figure 4.3. The remainder of the code can be found in appendix B.3 which shows how various ML primitives are implemented using this list representation. We can see that it represents a non-trivial program. The conversion of this program into combinators by the compiler takes under ten seconds and produces multiple batches of output files that can be loaded onto the processor.

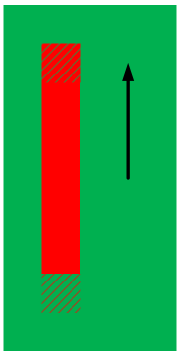
Running this program on the processor returns the value 10 indicating there are 10 possible ways to position 5 queens on a 5×5 board. The `head` and `tail` functions can be used to iterate through the solutions, for example to find the position of the queen in the first column and in the first solution, the last line of the code would be changed to:

```
head (head (nqueens 5));
```

4.2.2 Ponginator

Ponginator (combinator pong) is a one-player combinator version of the classic paddle and ball game. The user controls a paddle with a switch on the DE2 board and has to prevent the ball from going past the paddle and losing the game. The user aims to maximise their high score by staying alive for as long as possible. The game itself is very simple to play, but to run it on a combinator processor and write the program in ML* really requires a certain level of reliability in all areas of the system.

The program to run *Ponginator* can be found in appendix B. The style of programming is that of *differential programming*: Rather than programming every pixel on every cycle of the game, only the differences between the old game image and the new one are encoded. This simplifies the game logic and improves the performance of the game. For example, if the bat is moving up by one pixel, the pixel at the top end of the bat is coloured to the bat colour, and the pixel at the end of the bat is coloured to the background colour. This is illustrated by the diagram on the right.



Programming of the screen is performed using the `OUT` combinator with a VGA driver wired into the processor. 16-bit colour mode is supported, and the screen is divided into a 20×20 grid to simplify the game. The `IN 1` combinator is wired to switch 5 of the DE2 board to control the bat. When the player loses the game, the game stops and the screen indicates that they have lost. The user may then retrieve their high score on the HEX display. Figure 4.4 (overleaf) shows a screenshot of the game.

The game is actually surprisingly fun and quite difficult due to how fast it goes. The ball is made to vary its course around the screen by changing its offset slightly when it hits the bat.

For me, *Ponginator* is the highlight of this project. It represents the collective collaboration of all areas of my system to produce something tangible and fun.

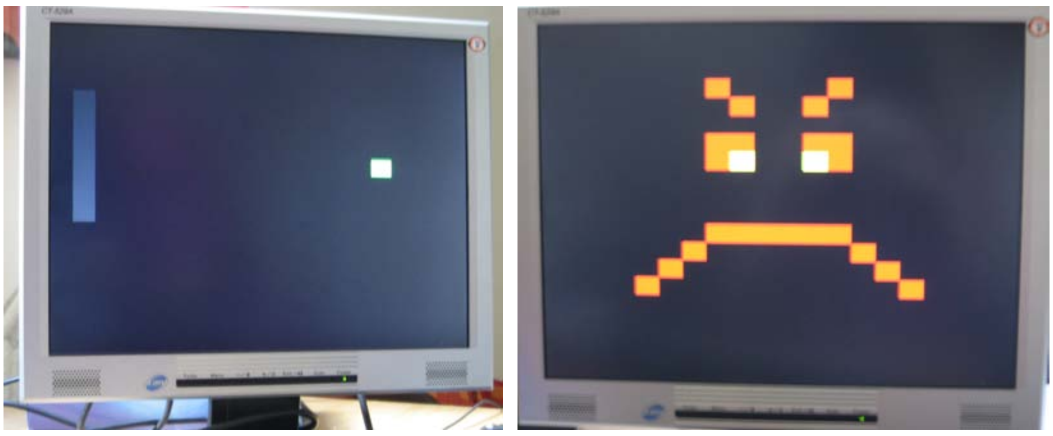

(a) Game Play Screen (b) The "Lose" Screen

Figure 4.4: Comb Pong

```
(((C ((C ((B C) ((C ((B C) ((B (B C)) ((C ((B C) ((B (B C)) ((B (B (B C))) ((C ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C))))
((C ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C)))) ((B (B (B (B (B C))))) ((C ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C))))
((B (B (B (B (B C))))) ((B (B (B (B (B (B C)))))) ((C ((B C) ((B (B C)) ((B (B (B B))) ((B (B (B C))) ((B (B (B (B C)))) ((B (B (B (B
(B B))))) ((B (B (B (B (B B))))) ((S ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C)))) ((B (B (B (B (B B))))) ((B (S ((B B) ((B B) ((B
C) ((B (B B)) ((B (B (B ((C I) 0)))) ((B (B (B Y))) ((B (C ((B C) ((B (B B)) ((B (B S)) ((C ((B B) ((B C) (C EQ)))) ((C ((C ((C ((C
((C I) 5)) 5)) K)) K)) 3))))))) ((C ((B B) ((B S) (C (OUT3 1))))) ((C B) ((C ADD) 1)))))))))))) ((B (B (B (B (B Y))))) ((S ((B B) ((B
C) ((B (B S)) ((B (B (B C))) ((B (B (B (B B)))) ((B (B (B (B C)))) ((B (B (B (B (B C))))) ((B (B (B (B (B (B C)))))) ((B (B (B (B (B (B
(B C))))))) ((B (B (B (B (B (B (B (B C)))))))) ((B (B (C ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C)))) ((B (B (B (B (B C))))) ((B
(B (B (B (B (B S)))))) ((B (B (B (B (B (B (B B))))))) ((C ((B C) ((B (B S)) ((B (B (B S))) ((B (B (B (B B)))) ((B (B (B (B S)))) ((B (B
(B (B (B C))))) ((B (B (B (B (B (B S)))))) ((B (B (S ((B B) ((B B) ((B B) ((B B) ((C EQ) 2)))))))) ((S ((B S) ((B (B S)) ((B (B (B S)))
((B (B (B (B B)))) ((B (B (B (B S)))) ((B (B (B (B (B B))))) ((B B) ((B S) ((B (B S)) ((B (C ((B B) n))) (C SUB))))))))))) ((C ((B C)
((B (B S)) ((B (B (B S))) ((B (B (B (B B)))) ((B (B (B (B C)))) ((B (B (B (B (B B))))) ((B B) ((B S) ((B (B S)) ((B (C ((B B) n))) (C
ADD)))))))))))) ((B (B C)) ((C ((B C) ((B (B C)) ((C ((B S) ((B (B C)) ((C ((B C) ((B (B C)) ((B C) (C I))))) 3)))) I)))) K)))))))))))))
((S ((B S) ((B (B B)) ((B (B S)) ((B (B (B S))) ((C ((B B) ((B B) ((B B) ((C EQ) 19))))) ((S ((B B) ((B S) ((C ((B B) ((C EQ) 19)))
((C ((C ((C ((C ((C ((C I) 19)) 19)) 18)) 18)) (K I))) K))))) ((S ((B B) ((B S) ((C ((B B) ((C EQ) 0))) ((C ((C ((C ((C ((C ((C I) 19))
0)) 18)) 1)) (K I))) (K I)))))) ((S ((B C) ((B (B S)) ((B (C B)) ((C ((B C) ((C ((B C) ((S ((B C) ((C ((B C) (C ((C I) 19)))) 18))) ((C
SUB) 1)))) (K I)))) K))))) ((C ((B C) ((C ((B C) ((S ((B C) ((C ((B C) (C ((C I) 19)))) 18))) ((C ADD) 1)))) (K I)))) (K I))))))))))
((S ((B S) ((B (B S)) ((B (B (B B))) ((B (B (B S))) ((B (C ((B B) ((B B) ((C EQ) 0)))))) ((S ((B C) ((B (B S)) ((B (C B)) ((C ((B C) ((C
((B C) ((C ((B C) ((S ((B C) ((C ((B C) (C I))) 0))) ((C ADD) 1)))) 1))) K))) (K I)))))) ((C ((B C) ((C ((B C) ((C ((B C) ((S ((B C)
((C ((B C) (C I))) 0))) ((C SUB) 1)))) 1))) (K I)))) (K I))))))))) ((S ((B S) ((B (B S)) ((B (B (B B))) ((B (B (B S))) ((B (C ((B B) ((B B)
((C EQ) 19))))) ((S ((B C) ((B (B S)) ((B (C B)) ((C ((B C) ((C ((B C) ((C ((B C) ((S ((B C) ((C ((B C) (C I))) 19))) ((C ADD) 1))))
18))) K))) K))))) ((C ((B C) ((C ((B C) ((C ((B C) ((S ((B C) ((C ((B C) (C I))) 19))) ((C SUB) 1)))) 18))) (K I)))) K)))))))) ((S ((B
S) ((B (B C)) ((B (B (B S))) ((B (B (B (B S)))) ((B (B (C ((B B) B)))) ((S ((B S) ((B (B C)) ((B (B (B S))) ((B (B (C B))) ((C ((B C)
((B (B C)) ((C ((B C) ((B (B C)) ((C ((B S) ((B (B C)) ((S ((B C) ((B (B C)) ((B C) (C I))))) ((C ADD) 1))))) ((C SUB) 1))))) K))))
K)))))) ((C ((B C) ((B (B C)) ((C ((B C) ((B (B C)) ((C ((B S) ((B (B C)) ((S ((B C) ((B (B C)) ((B C) (C I))))) ((C ADD) 1))))) ((C
ADD) 1))))) K)))) (K I))))))))) ((S ((B S) ((B (B C)) ((B (B (B S))) ((B (B (C B))) ((C ((B C) ((B (B C)) ((C ((B C) ((B (B C)) ((C ((B
S) ((B (B C)) ((S ((B C) ((B (B C)) ((B C) (C I))))) ((C SUB) 1))))) ((C SUB) 1))))) (K I))))) K)))))) ((C ((B C) ((B (B C)) ((C ((B C)
((B (B C)) ((C ((B S) ((B (B C)) ((S ((B C) ((B (B C)) ((B C) (C I))))) ((C SUB) 1))))) ((C ADD) 1))))) (K I))))) (K I)))))))))))))))))
((B (B (B (B Y)))) ((B (B (B (B K)))) ((C ((B B) ((B B) ((B B) ((B C) ((B (B C)) ((B (B (B B))) ((B (B (B B))) ((B (B (B B))) ((B (B (B
B))) (C ((B C) OUT3))))))))))) ((C ((B B) ((B B) ((B S) ((B (B S)) ((B (B (B B))) ((B (B (B B))) (C ((B C) OUT3)))))))) (C ((B C)
((B (B C)) ((B (B (B C))) (B (B (B C)))))))))))))))))))))))) ((B (B (B (B Y)))) ((B (B (B (B K)))) ((S ((B S) ((B (B S)) ((B (B (B S))) ((B
(B (B (B S)))) ((B (B (B (B (B S))))) ((B (B (B (B (B (B S)))))) ((B (B (B (B (B (B (B S))))))) ((B (B (B (B (B (B (B (B C)))))))) ((B (B
(S ((B S) ((B (B S)) ((B (B (B S))) ((B (B (B (B S)))) ((B (B (B (B (B S))))) ((B (B (B (B (B (B C)))))) ((B (B (B (B (B (B (B C)))))))
((S ((B S) ((B (B S)) ((B (B (B S))) ((B (B (B (B S)))) ((B (B (B (B (B S))))) ((B (B (B (B (B (B S)))))) ((B (B (B (B (B (B (B B)))))))
((B (B (B (B (B (B (B C))))))) ((B (B (B (B (B (B (B (B ((EQ (IN 1)) 1))))))))) ((C ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C))))
((B (B (B (B (B C))))) ((B (B (B (B (B (B S)))))) ((B (B (B (B (B (B (B B))))))) ((B B) ((B B) ((B B) ((B B) ((B S) ((C ((B C) ((B (B
EQ)) (C SUB)))) 0))))))))))))) ((S ((B C) (C I))) ((C SUB) 1))))))))))))) ((C ((B C) ((B (B C)) ((B (B (B C))) ((B (B (B (B C)))) ((B
(B (B (B (B C))))) ((B (B (B (B (B (B S)))))) ((B (B (B (B (B (B (B B))))))) ((B B) ((B B) ((B B) ((B B) ((B S) ((C ((B C) ((B (B EQ))
(C ADD)))) 19))))))))))))) ((S ((B C) (C I))) ((C ADD) 1))))))))))))) ((B (B (B (B (B (B (B (B Y)))))))) ((B (B (B (B (B (B (B (B
K)))))))) ((C ((B B) ((B S) ((B (B B)) ((B (B B)) ((B (B B)) ((B (B B)) ((B (B B)) ((B (B S)) ((B (B (B B))) (C ((B C) ((B (B (OUT3 1)))
(C ADD)))))))))))))) ((B (B B)) ((B (B B)) ((B (B B)) ((B (B B)) ((B (B C)) ((B (B (B B))) (C ((B C) ((B (B (OUT3 1))) ((B (C SUB))
((C ADD) 1)))))))))))))))))))))))) ((B (B (B (B (B (B (B (B Y)))))))) ((B (B (B (B (B (B (B (B K)))))))) ((C ((B B) ((B S) ((B (B B)) ((B
(B B)) ((B (B B)) ((B (B B)) ((B (B B)) ((B (B S)) ((B (B (B B))) (C ((B C) ((B (B (OUT3 1))) (C SUB))))))))))))) ((B (B B)) ((B (B
B)) ((B (B B)) ((B (B B)) ((B (B C)) ((B (B (B B))) (C ((B C) ((B (B (OUT3 1))) ((B (C ADD)) ((C ADD) 1))))))))))))))))))))))))))
((B (B Y)) ((B (B K)) ((S ((B B) ((B C) ((B (B C)) ((B (B (B B))) (C ((B C) OUT3))))))) ((S ((B B) ((B S) (C ((B (OUT3 1)) ((C SUB)
3)))))) ((S ((B B) ((B S) (C ((B (OUT3 1)) ((C SUB) 2)))))) ((S ((B B) ((B S) (C ((B (OUT3 1)) ((C SUB) 1)))))) ((S ((B B) ((B S)
(C (OUT3 1))))) ((S ((B B) ((B S) (C ((B (OUT3 1)) ((C ADD) 1)))))) ((S ((B B) ((B S) (C ((B (OUT3 1)) ((C ADD) 2)))))) ((B C)
(C ((B (OUT3 1)) ((C ADD) 3))))))))))))))))))))) ((S ((B B) ((B B) ((OUT3 7) 3)))) ((S ((B B) ((B B) ((OUT3 12) 3)))) ((S ((B B)
((B B) ((OUT3 8) 4)))) ((S ((B B) ((B B) ((OUT3 11) 4)))) ((S ((B B) ((B B) ((OUT3 7) 6)))) ((S ((B B) ((B B) ((OUT3 8) 6)))) ((S
((B B) ((B B) ((OUT3 11) 6)))) ((S ((B B) ((B B) ((OUT3 11) 6)))) ((S ((B B) ((B B) ((OUT3 12) 6)))) ((S ((B B) ((B B) ((OUT3 7)
7)))) ((B (S ((B B) ((OUT3 8) 7)))) ((B (C ((B B) ((OUT3 11) 7)))) ((S ((B B) ((OUT3 12) 7))) ((S ((B B) ((OUT3 7) 11))) ((S ((B
B) ((OUT3 8) 11))) ((S ((B B) ((OUT3 9) 11))) ((S ((B B) ((OUT3 10) 11))) ((S ((B B) ((OUT3 11) 11))) ((S ((B B) ((OUT3 12)
11))) ((S ((B B) ((OUT3 6) 12))) ((S ((B B) ((OUT3 13) 12))) ((S ((B B) ((OUT3 5) 13))) ((S ((B B) ((OUT3 14) 13))) ((S ((B B)
((OUT3 4) 14))) ((OUT3 15) 14)))))))))))))))))))))))))))))))))) 57005))))))) 7)))))) 7))))) 3)))) 4))) 1)) 5) 0)
```

Figure 4.5: Ponginator in combinator byte code

Chapter 5

Conclusions

All goals outlined in the original proposal have been met and in several respects they have been surpassed. The subset of ML supported throughout the system is larger than I ever expected it to be, and the addition of basic imperative features means that the system is flexible and extendable. Since a lot more material was available than could fit in this dissertation, I hope I have managed to convey an overall picture of how the system worked, as well as covering in more detail some of the more difficult and intricate parts of the project.

One of the fundamental aims that I wanted to achieve with this project was that of scalability and extensibility within the system. There are a number of ways in which I believe this has been successful. The project has demonstrated that combinator processors can be designed in such a way that both the on-chip and off-chip memory are utilised. To the best of my knowledge, this is the first time that a combinator based architecture has been extended like this. Substantial programs can be executed, allowing larger computations to be performed than have been possible on similar processors.

The processor is competitive with a research project in the same area, despite having had a much shorter development time and taking into account the fact that previous research projects do not utilise off-chip memory.

The scope of extension in this area is large. Garbage collection in this project requires a *stop-the-world* approach – it would be desirable for garbage collection to occur in parallel with reduction. One of the criticisms of graph reduction is that it is fundamentally memory bound. This does represent a problem, certainly in today's age when memory latency is one of the major bottlenecks in processor performance. Potential solutions may involve smart caching schemes of expression trees or parallelised execution.

Graph reduction may not be the most intuitive form of processing, and it is unlikely to catch up with conventional architectures in terms of speed or ubiquity. So, while combinatory world domination is a far speck on the horizon, the new resurgence of interest has shown some positive results, and I hope that I have demonstrated the principle of the mechanism and how it can be used to do some real computation.

Bibliography

[1] Altera home page. `http://www.altera.com/`.

[2] ModelSim home page. `http://www.model.com/`.

[3] Synplify home page. `http://www.synplicity.com/`.

[4] The Reduceron. `http://www.cs.york.ac.uk/fp/reduceron/`.

[5] H. Abelson, G.J. Sussman, J. Sussman, and A.J. Perlis. *Structure and interpretation of computer programs*. MIT press Cambridge, MA, 1996.

[6] H.P. Barendregt and E. Barendsen. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Goteborg*. Programming Methodology Group, University of Goteborg and Chalmers University of Technology, 1988.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of ACM*, 13(11):677–678, 1970.

[8] T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, and A.C. Norman. SKIM-The S, K, I reduction machine. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 128–135. ACM New York, NY, USA, 1980.

[9] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM New York, NY, USA, 1982.

[10] A. Mycroft. Compiler construction. Lecture notes, University of Cambridge, 2007.

[11] M. Naylor. *Hardware-Assisted and Target-Directed Evaluation of Functional Programs*. PhD thesis, University of York, UK, 2009. To be published.

[12] M. Naylor and C. Runciman. The Reduceron: Widening the von Neumann bottleneck for graph reduction using an FPGA. *Lecture Notes In Computer Science*, pages 129–146, 2008.

[13] W.R. Stoye, T.J.W. Clarke, and A.C. Norman. Some practical methods for rapid combinator reduction. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 159–166. ACM New York, NY, USA, 1984.

[14] D.A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1), 1979.

Appendix A

Processor RTL View

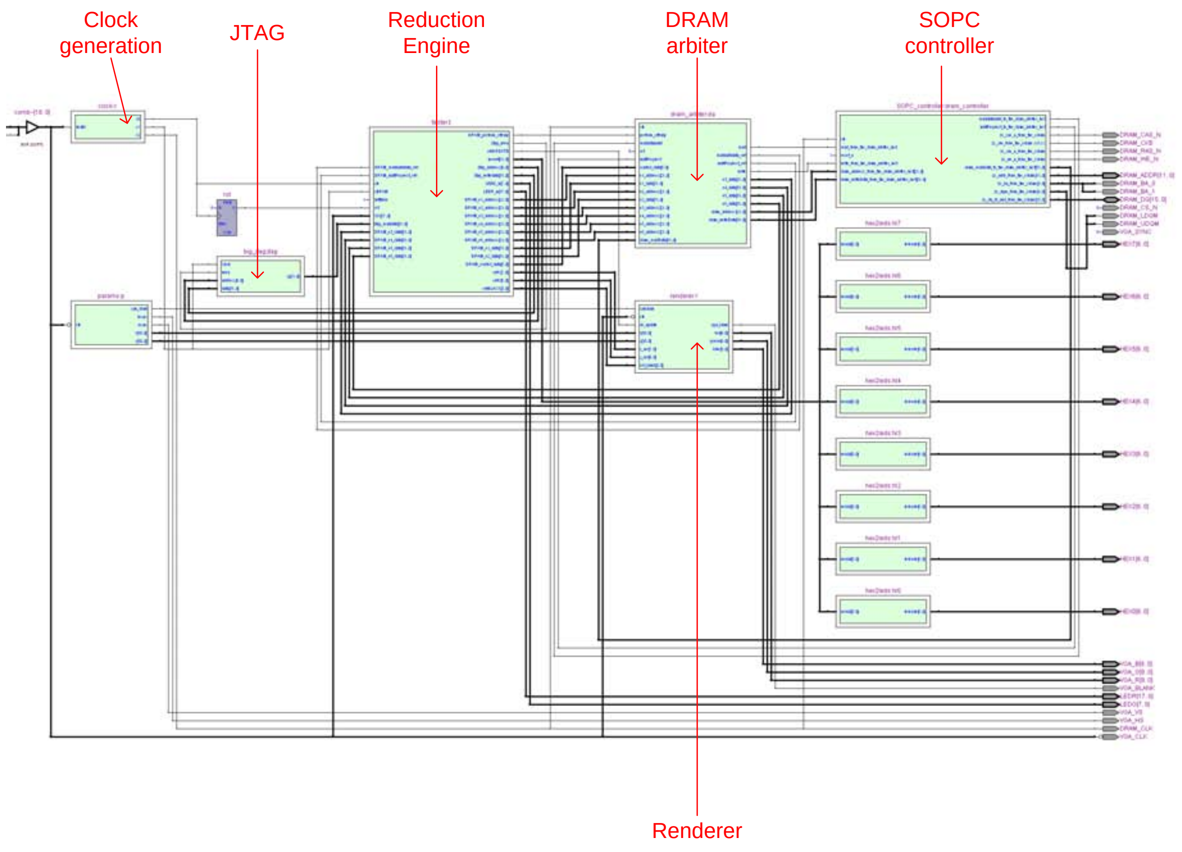


Figure A.1: Processor RTL View: Top level view of the processor register transfer level diagram produced by Quartus

Appendix B

Applications in ML*

B.1 The `isPrime` Predicate

```
(* Find d rem q using repeated subtraction *)
fun remainder d q =
  if (d < q) then d
  else (remainder (d-q) q);

(* Establish if v is a prime number or not *)
fun isPrime v =
  let fun isPrimeHelp e n =
  (* Iterates up to the value of prime being tested *)
    if (e = n) then true
    else
      if ((remainder e n) = 0) then false
      else (isPrimeHelp e (n+1))
  in (isPrimeHelp v 2)
  end;

(* Expression to test if 1447 is a prime *)
isPrime 1447;
```

B.2 Combinator Pong, *Ponginator*

```
(* Define colours to be used *)
val black = 0;
val grey = 5;
val turq = 1;
val red = 4;
val yellow = 7;

(* Define pong parameters *)
val batWidth = 3;
val batWidthTotal = 7;

(* Hex value of 0xDEAD *)
val DEAD = 57005;
```

```
(* Display the angry face *)
val diescreen = OUT3 7 3 red (OUT3 12 3 red (OUT3 8 4 red (OUT3 11 4 red
  (OUT3 7 6 red (OUT3 8 6 red (OUT3 11 6 red (OUT3 11 6 red (OUT3 12 6 red
    (OUT3 7 7 red (OUT3 8 7 yellow (OUT3 11 7 yellow (OUT3 12 7 red
      (OUT3 7 11 red (OUT3 8 11 red (OUT3 9 11 red (OUT3 10 11 red
        (OUT3 11 11 red (OUT3 12 11 red (OUT3 6 12 red (OUT3 13 12 red
          (OUT3 5 13 red (OUT3 14 13 red (OUT3 4 14 red (OUT3 15 14 red DEAD)
          ))))))))))))))))))))))));

(* Draw over the bat and ball, and call diescreen *)
fun loser ballPx ballPy batPy =
  (OUT3 ballPx ballPy black (OUT3 1 (batPy - 3) black
    (OUT3 1 (batPy - 2) black
      (OUT3 1 (batPy - 1) black
        (OUT3 1 batPy black
          (OUT3 1 (batPy + 1) black
            (OUT3 1 (batPy + 2) black
              (OUT3 1 (batPy + 3) black diescreen))))))));

(* Play the game *)
fun drawGame ballPosX ballPosY ballDirX ballDirY batPosY =
  (* Draw over the old bat and draw the bat in its new pos *)
  let fun drawBat ballPosX1 ballPosY1 ballDirX1 ballDirY1 batYpos1 =
    (let fun outputBatDown oldpos pos =
      (OUT3 1 (oldpos-batWidth) black
        (OUT3 1 (oldpos + (batWidth + 1)) grey
          (drawGame ballPosX1 ballPosY1 ballDirX1 ballDirY1 pos)))
    in
      (* Move the bat up *)
      (let fun outputBatUp oldposB posB =
        (OUT3 1 (oldposB+batWidth) black
          (OUT3 1 (oldposB - (batWidth+1)) grey
            (drawGame ballPosX1 ballPosY1 ballDirX1 ballDirY1 posB)))
      in
        (* Retrieve input from the switch *)
        (if ((IN 1) = 1) then
          (if ((batYpos1-batWidth) = 0)
          then (drawGame ballPosX1 ballPosY1 ballDirX1 ballDirY1 batYpos1)
          else (outputBatUp batYpos1 (batYpos1 - 1)))
        else
          (* Detect whether bat is at the edge of the screen *)
          (if ((batYpos1+batWidth) = 19)
          then (drawGame ballPosX1 ballPosY1 ballDirX1 ballDirY1 batYpos1)
          else (outputBatDown batYpos1 (batYpos1 + 1))))
      end)
    end)
  in
    (* Draw the ball into its new position *)
    (let fun drawBallUpdate cBallPX cBallPY nBallPX nBallPY nBallXdir newBallYdir =
      (OUT3 cBallPX cBallPY black
        (OUT3 nBallPX nBallPY turq
          (drawBat nBallPX nBallPY nBallXdir newBallYdir batPosY)))
    in
      (* Check if ball intercepts with the bat *)
      (if (ballPosX = 2) then
        (if (ballPosY < (batPosY - batWidth)) then (loser ballPosX ballPosY batPosY)
        else
```

```
          (if (ballPosY > (batPosY + batWidth)) then (loser ballPosX ballPosY batPosY)
          else (drawBallUpdate ballPosX ballPosY 3 ballPosY true ballDirY)))
      else
        (* Check if ball at the edge of the screen *)
        (if (ballPosX = 19) then
          (*consider other two corner cases *)
          (if (ballPosY = 19) then
            (drawBallUpdate 19 19 18 18 false true)
          else
            (if (ballPosY = 0) then
              (drawBallUpdate 19 0 18 1 false false)
            else
              (if (ballDirY) then
                (drawBallUpdate 19 ballPosY 18 (ballPosY - 1) false true)
              else
                (drawBallUpdate 19 ballPosY 18 (ballPosY + 1) false false))))
        else
          (if (ballPosY = 0) then
            (if (ballDirX) then
              (drawBallUpdate ballPosX 0 (ballPosX+1) 1 true false)
            else (drawBallUpdate ballPosX 0 (ballPosX-1) 1 false false))
          else
            (if (ballPosY = 19) then
              (if (ballDirX) then
                (drawBallUpdate ballPosX 19 (ballPosX+1) 18 true true)
              else (drawBallUpdate ballPosX 19 (ballPosX-1) 18 false true))
            else
              (if (ballDirX) then
                (if (ballDirY)
                then
                 (drawBallUpdate ballPosX ballPosY (ballPosX+1) (ballPosY-1) true true)
                else
                 (drawBallUpdate ballPosX ballPosY (ballPosX+1) (ballPosY+1) true false))
              else
                (if (ballDirY) then
                  (drawBallUpdate
                      ballPosX ballPosY (ballPosX-1) (ballPosY-1) false true)
                else (drawBallUpdate
                        ballPosX ballPosY (ballPosX-1) (ballPosY+1) false false)))))))
    end)
  end;

(* Set up the game screen *)
fun drawBatStart k =
  if (k = batWidthTotal) then (drawGame 5 5 true true 3)
  else (OUT3 1 k grey (drawBatStart (k+1)));

(* PLAY PONG! *)
drawBatStart 0;
```

B.3 Library of Useful Functions

```
(* Useful functions *)
fun andalso a b = if a then b else false;

fun orelse a b = if a then true else (if b then true else false);

(* Pair representation *)
fun pair a b = (fn p => p a b);

fun fst p = p (fn x => fn y => x);

fun snd p = p (fn x => fn y => y);


(* List representation *)
fun cons h t = fn x => fn y => ((x h) t);

val nil = fn x => fn y => y;

fun head xs e = xs true e;

fun tail xs e = xs false e;

fun isNil xs = xs (fn h => fn t => false) true;

(* ML Primitive List operators *)
fun length xs =
  xs (fn h => fn t => 1+(length t)) 0;

fun append xs ys =
  xs (fn h => fn t => cons h (append t ys)) ys;

fun concat dlist =
  dlist
    (fn h => fn t =>
      if (isNil t)
      then h
      else (append h (concat t)))
    nil;

fun occurs p xs =
  xs (fn h => fn t => if (p h) then true else (occurs p t)) false;

fun member n xs =
  (* Check if n is a member of the list *)
  (* Requires equality type *)
  occurs (fn m => m = n) xs;

fun filter b xs =
  (* Filter out all elements of the lists that do not make b true *)
  xs (fn h => fn t => if (b h) then (cons h (filter b t)) else (filter b t)) nil;

fun map f xs =
  xs (fn h => fn t => cons (f h) (map f t)) nil;
```

Appendix C

Instruction Opcodes

The Instruction Set Architecture of the processor was outlined in section 3.3. In this section, the opcodes for arithmetic and combinator instructions are presented.

C.1 Arithmetic Instructions

The `OP_arith` codes for arithmetic instructions are described below:

Arithmetic Operation	`OP_arith`
Addition	0x01
Multiplication	0x02
Equality	0x03
Subtraction	0x04
Less Than	0x05
Greater Than	0x06
Not Equals	0x07

Table C.1: Arithmetic instruction opcodes

C.2 Combinator Instructions

The `OP_comb` codes for combinator instructions are described below:

Combinator	`OP_comb`
`S` combinator	0x01
`K` combinator	0x02
`I` combinator	0x04
`Y` combinator	0x08
`B` combinator	0x10
`C` combinator	0x20
`OUT` combinator	0x40
`IN` combinator	0x80

Table C.2: Combinator instruction opcodes

Appendix D

Tables of Results

The below tables show the data used in the performance analysis part of the evaluation chapter, section 4.1.

D.1 Fibonacci

Table D.1 shows the results from running Fibonacci for different values of n.

n	0	1	2	3	4	5	6	7	8	9
Unoptimised	47	91	353	694	1,319	2,285	3,876	6,433	83,468	623,110
Optimised	47	91	353	694	1,319	2,285	3,876	6,433	10,581	17,286

n	10	11	12	13	14	15
Unoptimised	1,486,504	2,554,240	5,160,175	8,719,030	14,795,392	24,410,608
Optimised	698,721	2,100,452	4,352,577	7,733,784	14,010,100	23,628,280

Table D.1: Fibonacci Results: Cycles taken to calculate `fib` n

D.2 Garbage Collection and `isPrime`

Table D.2 shows the result of running `isPrime 1477` while varying the Garbage Collection trigger point.

GC trigger (memory words $\times 10^3$)	262	131	65	32	16	8	4	2	1
Cycles ($\times 10^6$)	650	674	679	676	668	638	577	509	400
Number of GCs	1	2	3	8	15	27	54	91	147

Table D.2: Cycle times and number of garbage collections when varying the GC trigger point

Appendix E

Verilog Sample Code

The below code sample is part of the processor reduction engine and shows the Verilog implementation of the C combinator (stages 0 and 1), as presented in section 3.4.3.

```
op_C:
case (ALU_stage)
  0:
  if (B == nil) begin
    // Insufficient arguments so terminate execution
    unwind <= 1;
  end else if (cdr_rdaddr_b == B[27:0] && car_rdaddr_b == B[27:0]) begin
    // Unwind the tree a bit
    B_case <= 4;
    B_val <= {{4{1'b0}}, car_qb[27:0]};
    P_case <= 3;
    P_val <= B;
    F_case <= 0;

    // Collect the third argument
    arg3 <= cdr_qb;

    // Get a cell from the free memory chain
    cdr_wren <= 1;
    cdr_write_addr <= B[27:0];
    cdr_write_data <= arg2;

    car_wren <= 1;
    car_write_addr <= B[27:0];
    car_write_data <= {op_APP, 2'b00, F[27:0]};

    car_rdaddr_b <= F[27:0];

    // Go to the next stage
    ALU_stage <= 1;
  end else begin
    // Prepare the data that needs to be read
    cdr_rdaddr_b <= B[27:0];
    car_rdaddr_b <= B[27:0];
    B_case <= 0;
    F_case <= 0;
```

```
      P_case <= 0;

      cdr_wren <= 0;
      car_wren <= 0;
    end
    1: begin
      // Perform a 'tail' on the free memory pointer
      F_case <= 5;
      F_val <= car_qb;


      // Write arguments to the free memory cells claimed
      car_wren <= 1;
      car_write_addr <= F;
      car_write_data <= arg1;

      cdr_wren <= 1;
      cdr_write_addr <= F;
      cdr_write_data <= arg3;

      P_case <= 0;
      B_case <= 0;

      car_rdaddr_b <= B;
      cdr_rdaddr_b <= B;

      stall_ALU <= 0;
    end
  endcase
```

Appendix F

Initialisation Protocol

The initialisation protocol for smaller programs consisting of a single set of output files is described first. In this case there are 3 output files from the compiler, `car0.mif`, `cdr0.mif` and `special.mif`. The protocol to run a small program is described below. The protocol starts from the board being preprogrammed with the processor and connected to a host PC running Quartus.

A memory initialisation file can be loaded onto the board via the JTAG unit on the processor. To do this, the *in-system memory content editor* in Quartus can be used. From here, the relevant file can be imported and the save to memory button pressed to send the file out to the board.

1. Set switch `SW17` high
2. Load `car0.mif` onto the board
3. Set swtich `SW0` high. `LEDG0` will light up to indicate that the car partition has been initialised. Switch `SW0` can then be lowered.
4. Load `cdr0.mif` onto the board
5. Set switch `SW1` high. `LEDG1` will light up and the switch can then be lowered.
6. Load `special.mif` onto the board
7. Set switch `SW2` high. At this point all lights will turn off indicating that the processor is ready to reduce the program.
8. Set all switches low. The processor will start reducing the program.
9. `LEDR17` will light up to indicate that the processor has finished reduction. `SW2` can be used to retrieve the value held in the P pointer on the HEX display. If the program reduced to an integer or a combinator, the value will be shown on the hex display, and otherwise the pointer to the reduced tree will be shown. A similar protocol to that described above can be used to retrieve the full tree from memory.

The protocol described above can be extended to support arbitrary length programs in the following way. For an arbitrary length program, the output files will be of the form: `car0.mif`, `car1.mif`, ..., `car`*n*`.mif`, `cdr0.mif`, `cdr1.mif`, ..., `cdr`*n*`.mif`, `special.mif`. Between steps 5 and 6 in the above protocol, the following steps must be followed to load the rest of the tree into memory. The protocol describes how to load the n^{th} set of car and cdr files onto the board, and it must be repeated for each value of n output by the processor.

1. Set `SW16` high
2. Load `car`*n*`.mif` onto the board.
3. Set `SW4` high. Set the values of `SW[15:6]` to be equal to the value of n.
4. Lower `SW16`.
5. Set `SW16` high
6. Load `cdr`*n*`.mif` onto the board.
7. Set `SW5` high. Set the values of `SW[15:6]` to be equal to the value of n.
8. Lower `SW16`.

Appendix G

Original Proposal

Emma Burrows
Peterhouse
eb379

Part II Computer Science Project Proposal

A Combinator-Reducing Processor

17 October, 2008

Project Originator: Christian Steinrücken

Resources Required: See attached Project Resource Form

Project Supervisor: Jointly supervised by Dr A. Norman and C. Steinrücken

Signatures:

Director of Studies: Dr A. Norman

Signature:

Overseers: Prof J. Crowcroft and Dr P Sewell

Signatures:

Introduction

Many functional programming languages can be seen as an extension of the Lambda Calculus. In general they express computation without side effects or consideration of a mutable state. Modern processors work on memory as a mutable state by loading and storing values and working on a small subset to perform the necessary computation. This model raises various challenges for high performance support of functional programming languages.

Lambda Calculus can be translated directly to Combinator form using translation rules. A processor which uses Combinators as its instruction set rather than the traditional load / store architecture lends itself particularly naturally to executing programs in functional languages. A previous CST project by Martin Lester showed basic feasibility of a soft core that could do combinator reduction, but did not produce a complete system that could run significant tests. A group at the University of York have been publishing reports of a machine that they call the "Reduceron" – their work in its present state seems to have both interesting ideas and some significant limitations. My proposed project is to design hardware that can be implemented using an FPGA and that, by building on this previous work, can both run more realistic tests than was previously possible and can explore the crucial issue of how to exploit the small amount of fast on-chip memory that a VLSI processor can have.

ML (or a subset of ML treated using normal order reduction) serves as an example of a functional programming language and the Altera DE2 Development boards provide a platform on which to build the processor. The project will involve building a compiler to turn ML into combinator (or super-combinator) form and then building a processor in System Verilog, initially in simulation and then on the board itself. Analysis and comparison of the processor using different refinements with a variety of test programs will then be undertaken – to do this a model of the CPU implemented in software and converted to MIPS will be used a baseline and the performance of my customized processor can then be compared to it.

Starting Point

The following resources which I will build on existed at the start of the project:

- **C code to model a Combinator Reduction Engine –** A skeleton simulation of the processor written in C by Dr. Norman to be used as a baseline to be compared against the performance of my eventual design.
- **Altera Internship –** Over the summer of 2008, I worked with the Computer Architecture group of the Computer Lab which gave me familiarity with the hardware tools and basic code which I wrote to communicate with the JTAG module, a useful resource to read values out of the processor
- At the end of September, I started feasibility analysis of building some of the software tools that I will need in ML

Project Description

The overall aim of the project is to build a processor with Combinators as the instruction set which can run ML programs.

The initial subset of ML that will be supported will simply use ML as a familiar syntax to encode the lambda calculus. However, further work will aim to support a larger amount of the language. Further language features will be supported if they can be encoded in such a way that the architecture of the reduction engine will not need to be changed. For example, language features like pattern matching and datatypes can feasibly be transformed to the lambda calculus, so they may potentially be supported. However, language features like references and exceptions which map better to an architecture which is load-store and has linear code, will not be supported.

The compiler to convert a subset of ML to combinators will be written in ML. This will consist of a lexer and a parser to convert the input to an abstract syntax tree. The syntax tree will then be converted to combinators by an intermediate step to lambda calculus, followed by a direct translation to combinators using specified translation rules. The precise translation rules used will be a specifiable parameter in order to allow for efficiency comparison of the processor with a variety of combinators. For example the most basic translation, while still being Turing complete is:

$\lambda^* x.x \equiv SKK$
$\lambda^* x.P \equiv KP$ where $x \notin FV(P)$
$\lambda^* x.PQ \equiv S(\lambda^*.P)(\lambda^*.Q)$

The processor itself will be written in Verilog / System Verilog. The basic processing model is to perform combinator reduction by tree traversal and manipulation - which amounts to operations on memory using pointer manipulation. Rather than using a stack to keep track of the reduction being performed, a pointer will be contained with each part of the graph indicating the 'next place to look', thus when going down the tree, the pointers can be used to point back up the tree and then restored once a combinator is found.

In order to make the processor perform any useful operation, certain operations will have to be provided as primitives. However, combinator reduction is implicitly lazy, while operations like arithmetic operations are strict. To solve this problem, integer primitives will be represented with a continuation passing style, so for example, the equation $p + q$ would be represented as $q_c(p_c+)$

The work presented by York University on the Reduceron uses several parallel memories in order to try and speed up execution. However, these memories are local and on-chip, and this model of execution makes it hard to take advantage of any of the off chip memory without coming into direct conflict with the bottleneck between the on-chip and off-chip memory and thus seeing the parallelism disintegrate. The restriction of only using local memory substantially limits the size of the programs that can be run on their processor. In order to run larger programs, I plan to take advantage of the resources available on the board - and use both the on-chip and off-chip memory.

Finally, the reduction engine will be analyzed and tested using various benchmark programs. The results will be compared against benchmarks run with the C simulation of the engine run on a MIPS soft core.

Success Criteria

The primary criteria for the project, which must be completed to consider the project successful are as follows:

1. Sufficient tools to allow a subset of ML to be turned into combinators and loaded unto the board
2. A combinator reduction engine must have been implemented on the FPGA board
3. An analysis of the performance of this engine in comparison with the performance of the simulation on the traditional soft core must have been undertaken

Extensions for the project, which are not essential to consider the project complete, but nonetheless would increase the success of the project are follows:

1. A careful comparison of the performance and design of my combinator reducing engine as opposed to York's Reduceron
2. A parameterized system to allow the set of combinators which a program is expressed in to be specified, and tests performed comparing the performance of the engine as a function of this parameter

Plan of Work

This outlines the milestones and deadlines that will be used during the year in order to ensure that the project stays on schedule and can be successfully completed. The project timetable starts on the 24th of October, the date when the proposals are submitted.

- **October 24th – November 3rd :** Complete basic compiler in ML to convert subset of ML to combinators
- **November 3rd – November 17th :**
  - Run the simulation of the processor on a MIPS soft core.
  - Analyze performance of the simulation processor to use as a baseline against hardware implementation
- **November 17th – December 3rd :**
  - Implement a stand-alone S/K reduction engine in Verilog but only in simulation
  - Increase the number of combinators supported by the reduction engine
- **December 3rd – January 13*th* :**
  - Ensure that all parts up to December 3rd have been successfully completed
  - Plan dissertation

- **January 13th – January 22nd :** Get the reduction engine working on the board and add bus to communicate with the off-chip RAM
- **January 22nd – February 1st :** Complete progress report and prepare presentation
- **February 1st – February 7th :**
  - Add support for built-in primitives
  - Continue work on dissertation
- **February 7th – February 16th :**
  - Debug and catch up time: Catch up on any areas not met between January and February.
  - Continue work on dissertation
- **February 16th – March 2nd :** Prepare and benchmark larger tests and do full performance analysis
- **March 2nd – March 11th, plus part of Easter Vacation :** Complete bulk of dissertation
- **April 21st – May 12th :**
  - Compare processor with other work
  - Finish loose ends
  - Complete dissertation
- **May 15th :** Submit dissertation