Improving the Performance of Architectures containing Random Access List Structured Memory

Jens-Uwe Dzikowski

March, 1995

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY COLLEGE LONDON
UNIVERSITY OF LONDON

This thesis is submitted to the University of London in partial fulfilment of the requirements for the degree of Doctor of Philosophy (Ph.D.)
ABSTRACT

The work presented in this thesis investigates how existing and future computer architectures can be enhanced to exploit the potential of Random Access List Structured Memory.

While most existing memory architectures can only offer a limited choice of possible data access mechanisms, Random Access List Structured Memory, as a new memory architecture, offers good support in both numerical and symbolic computing. A fast random access mechanism allows the manipulation of data elements stored in array structures, while the dimensions of these arrays can be dynamically modified. At the same time array elements can hold different data types and therefore allow the user to store complex data structures on the basic array structure. This way of expressing complex data structures as arrays can be used to avoid the use of conventional linked lists and their insufficient random access facilities.

The regular data structure of the arrays used in this memory concept offers new, elegant, ways to structure the data of common applications and should simplify many algorithms manipulating other large data structures. Any CPU designed to operate on a Random Access List Structured Memory architecture needs to achieve good results when manipulating individual data elements of these structures. However a general purpose processor can not be expected to achieve an optimal performance when manipulating large complex data structures. The requirement for high system performance, in a system architecture using complex list manipulations on a frequent basis, provided the motivation to investigate the potential of co-processors for these list manipulations.

A useful co-processor has to support the large variety of complex list manipulations, required for different fields of application. To meet this demand, a co-processor design using a micro-programmable instruction set is suggested. The large number of supported manipulations does not contradict with a specialised co-processor as long as the algorithms are closely related.
The development of such a co-processor is reported in this thesis. Starting with an explanation of the Random Access List Structured Memory concept, and the way it can be used in a computer architecture, the nature of complex list manipulations is explained. It is then indicated how a co-processor can be used to accelerate complex list manipulations as well as communication and memory management tasks. These general concepts are then applied to the tree shaped concept of Random Access List Structured Memory used in the experimental SPRINT architecture. Various hardware alternatives for the integration of such a co-processor, operating on Random Access List Structured Memory, are considered.

A design study for a basic co-processor architecture following these ideas is performed, applying a newly developed database method. This study includes the simulation of algorithms and co-processor hardware at various abstraction levels. The simulation results are then used to evaluate the performance advantages of the co-processor.
Acknowledgments

I would like to thank my supervisor Dr. Peter Rounce for his advice, the useful comments on my work and his availability whenever I needed help. Furthermore I want to credit his effort in reading various drafts. Also many thanks to my second supervisor Dr. Andy Betts for his opinions and the comments he made whenever I went to see him for advice.

My friends and former fellow-sufferers Dr. Jan Purchase and Dr. Wanderley Lobianco also deserve praise for reading drafts of this thesis and giving me useful remarks.

Many thanks to my girl-friend Kay Kanayama, for her encouragement, her patience and for being a permanent source of motivation. There are also credits for Angela Sasse and Julia Schnabel who, during the last three years, did their best to ensure I do not forget my native language. Thanks to John Pearce for several pleasant hours spent together on the golf course.

Last but not least, many thanks to my parents for their love and the financial support which has enabled me to carry out the research presented in this thesis.
## Contents

1 Introduction ................................................................. 19

1.1 Trends in Computer Science ........................................ 20

1.1.1 Hardware ............................................................ 21

1.1.2 Software ............................................................ 23

1.1.3 Memory Systems ................................................... 24

1.2 Why Random Access List Structured Memory ? ............ 26

1.2.1 Existing Memory Models ........................................ 27

1.2.2 Prerequisites for a List Structured Memory ............... 29

1.2.3 The Page-based, Tree-structured Model .................... 30

1.3 Background: SPAN and SPRINT ............................... 32

1.3.1 The SPAN project ............................................... 33

1.3.2 The SPRINT Processor ......................................... 35

1.3.3 Support for Modern Programming Paradigms ............ 36

1.4 Alternative Hardware Architectures ............................. 39

1.4.1 Access Rights and Protection ................................. 39

1.4.2 List Structured Memory as Extension ..................... 40

1.5 Pure List Structured Memory Architectures .................. 41

1.6 Structure of the Thesis .............................................. 43

1.7 Statement of Originality ............................................ 44
2 Manipulating List Structured Memory

2.1 Frequent Load-intensive Functions ................................................................. 48

2.2 Analysing the Functional Requirements ......................................................... 51
   2.2.1 List Traversal ......................................................................................... 51
   2.2.2 Memory Management ........................................................................... 56

2.3 Summary ........................................................................................................... 57

3 Hardware Alternatives

3.1 Enhancing Existing Architectures ................................................................. 60
   3.1.1 Structure ............................................................................................... 60
   3.1.2 List Memory Access .............................................................................. 62
   3.1.3 Complex List Operations ...................................................................... 64

3.2 New Architectures based on List Structured Memory ..................................... 65

3.3 Co-processor Design Considerations ............................................................. 67
   3.3.1 Instruction Set and Controller Structure ............................................... 67
   3.3.2 Activation .............................................................................................. 68
   3.3.3 Direct Communication Support ............................................................. 69
   3.3.4 Indirect Communication Support ............................................................ 73
   3.3.5 Enhanced Bus Systems ......................................................................... 74

3.4 Summary of possible Architectures ............................................................... 74

3.5 Conclusions ..................................................................................................... 77
## Contents

5.5 Data Manipulation - Arithmetic and Shifts ................................................. 110
5.6 Details of the Co-processor Datapath ........................................................... 113
   5.6.1 List Structure Registers .............................................................. 113
   5.6.2 Stack .............................................................................................. 115
   5.6.3 Buses ............................................................................................. 115
   5.6.4 Memory Management .................................................................... 117
   5.6.5 Processor Status Word .................................................................. 118
   5.6.6 Communication ............................................................................. 119
   5.6.7 Buffers ........................................................................................... 121
   5.6.8 SPRINT Interface ......................................................................... 121
5.7 Summary ................................................................................................. 124

6 Development of the Co-processor ................................................................. 127
   6.1 High-level Simulation ......................................................................... 128
      6.1.1 Simulated List Traversal ............................................................ 130
      6.1.2 Simulated Memory Management .............................................. 130
      6.1.3 Simulation Outcome .................................................................. 132
   6.2 Design Methodology ............................................................................ 133
      6.2.1 Microcode Generation ............................................................... 134
      6.2.2 Database Method ...................................................................... 137
## Contents

6.3 Behavioural Simulation ...................................................... 146

6.3.1 Extracting Database Information ........................................... 147

6.3.2 Joining the Components .................................................. 150

6.3.3 Scale of Simulation .......................................................... 151

6.3.4 Micro-program Optimisation ............................................. 153

6.4 Controller Design ............................................................... 155

6.4.1 Microcode Implementation ................................................ 155

6.4.2 Extraction Process ............................................................. 165

6.4.3 Gate-level Simulation ......................................................... 168

6.4.4 Micro-code Replacement ............................................... 170

6.4.5 Operation Mode of the Co-processor ................................. 171

6.5 Summary ................................................................. 172

7 Co-processor Evaluation .................................................. 175

7.1 Performance ................................................................. 175

7.1.1 Basic List Copy .............................................................. 176

7.1.2 List Slicing ................................................................. 181

7.1.3 List Deletion ............................................................... 182

7.1.4 List Flattening and List Re-construction .......................... 183

7.1.5 Other List Manipulations .............................................. 185

7.2 Practicality ................................................................. 186

7.2.1 General Connectivity of Co-processors ............................. 186

7.2.2 Co-processor Size ......................................................... 187

7.3 Summary ................................................................. 191
## 8 Conclusions

### 8.1 Review ................................................................. 193

### 8.2 Achievements ...................................................... 194

### 8.3 Outlook ............................................................... 197

## A Possible Co-processor Instructions .......................... 199

## B Description of the High Level Simulation Environment 205

### B.1 List Traversal .................................................... 205

#### B.1.1 Basic Definitions ........................................... 205

#### B.1.2 Register Manipulation .................................. 208

#### B.1.3 Stack Model .................................................. 210

### B.2 Simulation Environment ..................................... 211

#### B.2.1 Creating Lists .............................................. 212

#### B.2.2 Tasks ........................................................... 213

#### B.2.3 Manipulating Lists ....................................... 214

## C Protocol of a High-level Simulation ........................ 215

## D Description of the Micro-code Extraction ............... 219

## Glossary ....................................................................... 233

## Bibliography .............................................................. 237
# List of Figures

1.1 Example of a List Structured Memory ........................................................... 26  
1.2 Classification of Memory Architectures ......................................................... 28  
1.3 Example of a 3-level page structure ............................................................... 32  
1.4 Overview of the SPAN project ......................................................................... 34  
1.5 Structure of a SPRINT List Pointer ............................................................... 36  
1.6 One-dimensional Array .................................................................................... 37  
1.7 Two-dimensional Array .................................................................................... 37  
1.8 Example C-program Structure ....................................................................... 38  
2.1 Basic list traversal algorithm .......................................................................... 52  
2.2 Basic Traversal algorithm for 3-level page structure .................................... 55  
3.1 Basic structure for added List Memory ............................................................ 60  
3.2 Address structure for an added list memory .................................................. 61  
3.3 Advanced structure for added List Memory .................................................... 65  
3.4 Node with one CP for direct communication .................................................. 69  
3.5 Node with two CPs for direct communication ................................................ 70  
3.6 Node with CP and communication controller ............................................... 73  
3.7 Extended Node with dual-ported memory ..................................................... 75
List of Figures

4.1 Problem with a structured free page list ................................................. 82

5.1 Integer data format in SPRINT ............................................................... 100
5.2 Model of a paged stack ............................................................................. 103
5.3 Bottom-level stack-word in SPRINT ....................................................... 107
5.4 Second stack-word in SPRINT ............................................................... 107
5.5 Third stack-word in SPRINT ................................................................. 108
5.6 Shift register value and increment .......................................................... 111
5.7 Raise carry signal during increment ....................................................... 112
5.8 Add register value to identical aligned 1 ................................................. 112
5.9 Shifting by alternative interconnection ................................................... 113
5.10 Suggested stack implementation ............................................................ 116
5.11 Format of 21 bit Free Page Pointer ....................................................... 118
5.12 Format of 24 bit Free Page Pointer ....................................................... 118
5.13 Functional Registers in the Datapath ..................................................... 122
5.14 Parameter word format ........................................................................... 124

6.1 Data Model ............................................................................................ 139
6.2 Level 2 Flowchart .................................................................................. 143
6.3 Activations for component 121 .............................................................. 148
6.4 Components executed during step 526 ................................................. 149
6.5 Extract of function activating individual steps ......................................... 149
6.6 Extract of sequencing function .............................................................. 150
6.7 Basic structure of the Controller ............................................................ 161
List of Figures

6.8  Allocation of microcode words ................................................. 169

7.1  Runtime Comparison for List-Copy ........................................ 178

7.2  Speed-up for a List-Copy ...................................................... 178

7.3  Detailed speed-up for copying a 2-level List ........................... 179

7.4  Speed-up list slice 2-level ..................................................... 182

7.5  Speed-up list slice 3-level ..................................................... 182

7.6  Speed-up list deletion .......................................................... 184

B.1  C representation of a list-pointer ........................................... 206

B.2  C Data Frame for SPRINT words ......................................... 206

B.3  C representation of SPRINT words ....................................... 207

B.4  C registers to hold a list structure ........................................ 207

B.5  C procedures for registers manipulation ............................... 209

B.6  Skeleton for the list traversal algorithm ................................. 210

B.7  Preliminary C Definition of the Processor Status Word .......... 211

B.8  C stack definitions ............................................................. 212
List of Tables

1.1 Tag values in SPRINT ................................................................. 35

4.1 Comparison of page management methods .............................. 86

5.1 Stack demand for a SPRINT list ............................................... 107

5.2 Flags of the Processor Status Word ............................................ 120

6.1 Attributes of the STEPS relation ............................................... 144

6.2 Attributes of the FOLLOW relation ........................................... 145

6.3 Attributes of the CONSISTS relation ......................................... 145

6.4 Attributes of the COMPONENTS relation ................................. 145

6.5 Attributes of the LINES relation ............................................... 146

6.6 Attributes of the ACTIVATES relation ...................................... 146

6.7 Encoding of branches in the experimental co-processor .......... 162

6.8 Cycle Count for experimental Algorithms .............................. 164

7.1 Time demand for a list copy in SPRINT ................................. 177

7.2 Time demand for a list copy in the co-processor .................... 177

7.3 Time demand for a nested list copy ......................................... 180

7.4 Time demand for a list slice in SPRINT ................................. 181
7.5 Time demand for a list delete in SPRINT ..................... 183
7.6 Time demand for a list delete in the co-processor ............ 183
7.7 Time demand to flatten a list in the co-processor ............ 184
Chapter 1

Introduction

Computer Science is a fast developing subject. There is a continuous development of new fields of application for computers with a permanently growing variety of requirements. In the early days of computing the few existing machines were mainly used as calculators to automate large scale numeric processing. Since then the area of application has developed via the processing of business data towards general information and knowledge processing, covering, amongst others, subjects such as Computer Integrated Manufacturing, Image Processing and Artificial Intelligence.

To allow this expansion into such a wide field of applications, both software and hardware technologies had to undergo constant development. Together with the expansion of computing into new application areas, the complexity of computer programs has been growing constantly. Programming languages based on more and more sophisticated concepts have been developed and have replaced assembly language and early high level programming languages. Different areas of application demand a range of properties from programming languages to express the algorithms, mainly in the way data is stored and manipulated. Therefore a rich variety of programming languages aimed at different problem areas have been developed. As most of these new programming languages demanded a lot more support from the executing hardware than their predecessors did, there is a permanent effort to improve hardware architectures in order to offer this additional support.
Chapter 1. Introduction

List Structured Memory was suggested as one way of supporting modern programming languages using complex data structures to represent the information to be processed. This new memory model could be a way to overcome the limitations the standard 'von Neumann' linear memory structure imposes on the implementation of some programming languages.

This thesis is an investigation of how the performance of computer architectures based on Random Access List Structured Memory can be improved. It is neither intended to give a detailed justification of the Random Access List Structured Memory concept, nor to provide a comparison with other memory concepts. But both these areas can be considered in more detail once the architectural potentials of architectures with List Structured Memory are better understood.

This chapter will first indicate the place of this work in general trends of computer science. Afterwards the ideas of List Structured Memory are introduced together with an overview of SPRINT, an example architecture using this memory concept. An analysis of the prerequisites for List Structured Memory will lead to a first examination of alternative hardware architectures based on this concept. Finally an overview of the whole thesis is given.

1.1 Trends in Computer Science

As already stated, there is a rapid development in general computer science, but not all involved subjects progress at the same rate. One of the major influences on new developments in computer science during the last decades was, as for many other scientific subjects, the demand for increased performance\(^1\). Other important areas such as fault tolerance, software reliability and verifiability are frequently still considered as minor aspects. Better system performance can be achieved with both improved hardware and more sophisticated software.

\(^1\)In this context performance covers aspects such as: speed (response time) and data throughput as well as the complexity of the executed problems.
1.1. Trends in Computer Science

1.1.1 Hardware

The most common, traditional method of increasing hardware performance is based on exploiting the technological advances in electrical engineering. As the dimensions of transistors in integrated circuits get smaller and smaller, integration density rises and signal runtimes are reduced, enabling higher operation frequencies resulting in faster machines. As this process seems to reach its physical limits within an affordable technology, alternative ways of performance improvement become more important.

One of the main objectives in alternative hardware development is to exploit the potentials of parallel execution. This can be done at various levels from having multiple functional units in a processor to multicomputers [Hwa93]. There are strong arguments about design styles for individual processing units and what the instruction sets of these units should be like. While the RISC versus CISC conflict [HP90] is still discussed in academic environments, it seems that the markets have, at least for the moment, decided in favour of the RISC approach. The development of CISC processors, such as Intel 80486 and Pentium, and their use in Personal Computers (PCs) can be seen as an exception caused by compatibility requirements of existing architectures.

The reasons for this decision in favour of RISC processors might be:

1. Basic RISC processors tend to be smaller and have simpler designs, and therefore require less development time. This enables shorter product cycles, which again makes it easier to compete in the continuing competition for higher processing power. The shorter design time means, that the designer can exploit more recent and faster process technology, giving a speed advantage over other designs with longer development times.

2. Most commercial software is still written in third generation, imperative programming languages (C, Pascal, FORTRAN,...) which makes it easy to target RISC processors for the instruction set requirements of these programs.
3. The simple instruction set of RISC processors together with frequently used instruction pipelines allows higher clock frequencies and higher instruction execution rates. These high clock rates and MIPS\(^2\) numbers are popular marketing factors. A careful instruction set design can also improve pipeline performance, e.g. DEC ALPHA.

4. As a consequence of their general low level instruction set several common RISC processors have a wide applicability, even if not always the most efficient. This allows high production numbers combined with reasonable prices, which again promises the required profit to the manufacturers.

In addition to the commercial processor implementations, there is also a growing number of experimental RISC processors, such as Pegasus [SY90], SOAR [UBF+84] and SPUR [H+86], for modern programming languages. This trend towards building RISC processors is probably also a consequence of shorter development times in combination with lower hardware expenditure. Shorter development times, of course, also mean lower staffing costs for the developing team.

The above mentioned processors offer good performance for their specific programming language, whereas most other languages are supported unsatisfactorily. As a consequence, this type of language oriented RISC processor can either be used to implement single language machines, or as co-processor within a more general architecture.

Single language machines tend to be forced into a niche existence as most computer users do not want to be fixed to one programming language. A typical example are LISP machines [Ung89] which acquired a small market segment of users in the Artificial Intelligence area. Another major problem of single language machines is the fact that even the performance of sophisticated architectures built for a particular language can, after a short period, be easily matched by much faster hardware running a software implementation of the particular programming language. To

\(^2\) Million Instructions Per Second
avoid this, it would be necessary to continuously enhance the hardware of the single language architectures which is not very economic since the market share is limited.

The idea of using multiple specialised language co-processors within a more general architecture leads towards heterogeneous multiprocessor systems. The complexity of such a system can be expected to be high, while an efficient use of the individual co-processors is unlikely to happen. The major problem of an efficient utilisation of multiple language co-processors derives from the fact that this would require a balanced load for all co-processors. Further problems such as process scheduling would add to the complexity of a system using multiple specialised language co-processors.

1.1.2 Software

As a consequence of the steadily growing hardware performance, software developers can create more and more complex applications. This growing software complexity also creates new requirements for programming languages and software development tools.

As software development is very cost intensive, there is a strong interest to reduce the expense in this area. Object-oriented programming languages with their paradigm based on abstraction, encapsulation, modularity, hierarchy, typing and concurrency [Boo91] offer re-usability of program code which increases efficiency, together with a better support for maintenance and testing. Another important point is the fact that the concurrency of Object-oriented languages matches the natural concurrency of many applications and therefore frees the software developer from explicitly stating program sections to be executed in parallel.

In other areas of software development there is a growing demand for high reliability and program verification. The support for these needs offered by functional programming languages [Hud89] can be derived from the underlying paradigm of 'referential transparency and equational reasoning'. Strong static typing and data abstraction are basic concepts of functional languages.
In general there is a trend towards Object-oriented and functional languages and away from imperative (von Neumann) languages to meet the requirements of modern software development.

As applications differ in their requirements for programming paradigms, and software developers have their own preferences for programming languages, general purpose computer architectures need to support a broad spectrum of programming languages and underlying paradigms.

1.1.3 Memory Systems

The memory system is one of the points where the interests of hardware and software development intersect. It is necessary for hardware to implement a memory structure that allows fast and simple access to the data structures of the software being executed. On the other hand, effective software development depends on a memory structure that supports the requirements of the applied programming paradigm. Recent developments in the area of memory systems seem to concentrate on memory hierarchies and how to increase the bandwidth between CPU and memory.

Several language specific architectures [Ung89] implement individual memory concepts, as the standard, 'von Neumann', linear memory concept does not offer the best support. The 'von Neumann' architecture [BGvN73] was one of the first computer and memory architectures, developed almost 50 years ago when hardware was the major cost factor for computer users. Therefore a simple hardware concept offering just the basic functionality was appropriate at that time. Since then, hardware costs have dropped faster than software costs. The latter tend to be the major cost factor in modern systems. With this different cost situation and newer memory concepts available, it is questionable whether the 'von Neumann' memory architecture is the best design for many modern computers.

The development of individual memory concepts for language specific architectures suggests that a new memory concept, giving better assistance to the various
programming paradigms, might be desirable for general purpose computer architectures, especially when the new memory concept expands the possibilities of standard memory.

The 'Random Access List Structured Memory' approach\(^3\) seems to offer this kind of support to existing programming paradigms and therefore could be used as common basis for future architectures intended for general computing. Its memory structure is that of a group of lists, where each list can be dynamically extended. The individual elements of a list can be either of atomic data type or a pointer to another list. Using this concept, a memory structure can be a linear list of atomic data elements or a complex formation with tree or graph structure. Figure 1.1 gives an example for a tree shaped list structure. The whole data structure consists of a number of lists, combining atomic data types with pointers to sub-lists. Such a data structure is dynamic in two ways, firstly in that the length of each list can be modified according to demand, and secondly that complete sub-structures can be added or removed when required.

The length of individual lists and the depth of trees expressed in this way are logically unlimited. However, there will be physical limitations depending on the system implementation.

Dynamic re-sizable arrays in such a list structured memory with direct access offer an excellent extension to numeric applications with unknown array sizes. The possibility to grow and/or shrink arrays at runtime helps the software designer to avoid 'worst case' dimensions for arrays or even the use of linear linked lists. Similar advantages exist for large scale data processing, extensive collections of records can be kept and manipulated in arrays instead of complex multiple linked list structures. The way in which complex data structures are used will be implementation dependent. Within an object-oriented programming system, for example, a single list could be used to express an instance of an abstract data type, with individual sub-lists for code and data.

\(^3\)A explanation of this concept will be given in Section 1.2
Chapter 1. Introduction

The fact that the organisation of data in the list structured memory (Figure 1.1) matches the required memory model of many applications in symbolic computing (e.g. MIRANDA) suggests the suitability of list structured memory for symbolic processing.

The tree shaped organisation shown in Figure 1.1 is only one way of building a List Structured Memory, it is chosen for this example because it is used in this thesis. Other, more sophisticated memory organisations can be implemented to allow the construction of graph structures and cyclic memory references.

1.2 Why Random Access List Structured Memory?

So far it has only been stated that many modern programming languages face difficulties when implemented on a 'von Neumann' memory architecture. But what are the exact problems and how could a new memory concept improve on these shortcomings?

Numerous problems faced by recent software developments require the use of dynamic memory allocation as the amount and size of data to be processed can
1.2. Why Random Access List Structured Memory?

not be predicted before the program is executed. Dynamic memory is normally supplied, by the operating system, out of a special memory area called 'heap'. In a regular memory architecture the management of this heap can easily become a complex task, whereas it is a minor task in an architecture with Random Access List Structured Memory.

When user data of an initially specified size has to be expanded (e.g. growing an array beyond its current dimensions) regular memory architectures face immense difficulties. A trivial way to manipulate the size of memory objects, such as that provided by Random Access List Structured Memory, would greatly facilitate such expansions.

Another problem that becomes more important is the support of user data with complex relationships between various elements of completely different types (e.g. Hypergraphs and Hypernodes [LPB+93]).

Combining the random access attribute of array structures used in numeric processing with complex list structures used in symbolic computing promises a system that can produce good results in both application areas.

1.2.1 Existing Memory Models

In the regular linear memory model all stored data elements are fixed to a certain location within the address space and neighbouring memory locations are normally occupied by other data objects. Therefore this model can not offer general flexibility to expand the dimension of data elements in either direction. In most systems it is also difficult to move memory bound data objects to other sufficient sized locations as references to the object tend to use physical addresses. So when an object would be moved to another memory all references to this object would require updating. This updating of pointers can normally only be done by searching the entire memory space for such references. Even then, the unstructured 'von Neumann' memory does not uniquely identify these pointers as other data values can have the same binary pattern.
Chapter 1. Introduction

The concept of data-descriptors [Ung89] solves the problem of finding references to objects, as all referencing occurs indirectly via the descriptor and only the descriptor has to be updated with the new address when the object is moved. But the main problem of having to move an object each time it grows is still not solved.

List Structured Memory (e.g. LISP machines)

Random Access List Structured Memory

von Neumann Memory

Parallel Memories

Linear Memory

Descriptor Concept

Figure 1.2: Classification of Memory Architectures

In classic 'von Neumann' architectures any kind of data structuring disappears as soon as the data is stored in memory. Each memory value can be interpreted as any data type, as an address or even as code. The proper interpretation of a memory value has to be performed by the executed program. Consequently any dynamically allocated memory element can only be allowed to hold data of a particular data type. To enable a program to assign various data-types to one kind of memory element, software developers have to use complex data constructs (e.g. unions in C). Within these constructs one part of a known data-type tells the program how to interpret the remaining parts of the structure.

To overcome the limitations of unstructured memory the concept of 'tagged architectures' was developed in the late 1950s [Den81]. By giving each data element a few extra bits identifying its context, the processor can always find the right
1.2. Why Random Access List Structured Memory

method of data manipulation. Although the concept of tagged architectures offers several advantages [Feu73], it failed to have an impact on mainstream computer architecture and is mainly found in architectures for symbolic processing.

Traditional list-structured memory concepts, as used in LISP machines [PT87], only offer good performance for list processing with memory access limited to the list head. This originates in the memory structure where all possible data structures are built out of basic 'cons-cells', a pair of memory cells containing an atom or pointer to a sublist in the first element and a pointer to the rest of the data structure held in the second element. When access to an element inside the structure is required, these pointers build a chain that has to be evaluated, starting from the head of the list. Naturally this leads to bad performance for array style memory access as used in numeric applications. A significant improvement can be reached with a vector-coded [LLC90] representation of lists. But there is the limitation that as soon as vector-coded lists are used, these lists will be fixed to a certain size resulting in the same restrictions as arrays in regular memory models.

1.2.2 Prerequisites for a List Structured Memory

Most modern programming paradigms require fast dynamic generation of new data objects at run time. Heap storage management [Pra75] is the most common method of supporting this kind of dynamic memory allocation. When a fixed size is chosen for all newly generated memory elements the system will not produce external memory fragmentation and, consequently, a simple and fast storage management is possible. Since all dynamic allocated data objects have the same size it is easy to maintain a permanent flow of assigning and releasing these pages. To implement an efficient memory architecture using a memory allocation method based on a fixed page size, requires that the internal fragmentation is kept low. The advantage of limited internal fragmentation is the fact that it only causes a limited memory overhead, whereas external memory fragmentation requires complex and time intensive storage reorganisation.

A good way to keep internal fragmentation low is the choice of small fixed size
Chapter 1. Introduction

pages, as this automatically reduces the unused memory for compact contents. At first examination these small fixed size pages seem to be in conflict with the space requirements of large data structures, but these large structures can be split over several memory pages as long as this splitting is supported by the system. To maintain the direct access to elements in arrays or lists\(^4\), with these split up over multiple fixed size memory pages, it is necessary to determine the page and exact location of the data within the page at access time.

Another common aspect of programming languages used for symbolic processing is the frequent use of tags to overcome the previously mentioned problems of an unstructured memory. While static typed programming languages can live with compile-time and run-time checks, this is not possible for dynamically typed languages. As a result, any architecture intended for a machine supporting dynamic typed languages requires a tagged memory concept.

1.2.3 The Page-based, Tree-structured Model

The memory model introduced by Rounce [Rou93] was developed according to the requirements outlined in the previous section. The model takes the idea of the memory management mechanism used in multi-level paged memory schemes [FM87], and applies it to build lists out of individual fixed size pages.

List Access

All information held in a Random Access List Structured Memory is stored in lists, and efficient access to these list elements is essential. Therefore, any memory access (instruction fetch or data access) will involve the addressing of a list element. In order to address a list element, each reference will be kept in form of a 'List-pointer' and an element 'Selector'. The list-pointer indicates which list contains

\(^4\)As the mechanism introduced next treats arrays and lists in exactly the same way, there will be no more separation of the two, and the expression list will be used for both.
the required information, while the selector references the list element holding the actual information.

Whenever a program performs a memory access, the logical representation of 'list and element' has to be transformed into a physical memory address.

**Tree Structure**

As said before, a small page size allows the internal fragmentation to be kept low, but requires lists to be split over several pages, where the system has to cover this splitting. The tree structure introduced here demonstrates a method by which a list can be mapped on to a number of pages and provide equal access to each element in the list.

The actual data is kept in a selection of pages, where these pages are not constrained to any locality within the physical address space. On top of such a group of bottom level pages holding the data, a tree-shaped hierarchy of supporting pages keeps the order of pages at lower levels. Each of these supporting pages contains pointers to the pages of the next lower level. Figure 1.3 gives an example of such a tree structure including 3 levels.

The access to the tree structure representing a list is performed via a list-pointer to the root page of the tree. When addressing a data element the element selector is divided into a number of parts equal to the maximum number of levels in the tree structure. These parts, each of length $\log_2$ (page length), are then used to identify the pointers required for the data access. At each level of the tree, one of these pointers is required to access the appropriate pages at the next lower level and finally the data element at the bottom of the tree.

This page based approach to implement the memory model has the advantage of using standard linear-addressable physical memory. This is preferable, as this type of physical memory is much cheaper and available in higher densities than alternative solutions such as associative memory. Using a standard linear-addressable physical
memory also enables system designers to benefit from caching and virtual memory techniques already developed for existing memory models.

Further details about implementation details, such as page size effects, list length encoding and offset techniques are covered by Rounce [Rou93]. The offset technique of having a limited number of unused list-elements at the list head enables limited head-appending modifications to lists without any need of data movement.

1.3 Background: SPAN and SPRINT

So far the theoretic foundations of Random Access List Structured Memory have been outlined. The memory concept was developed as a part of the SPAN project. In this Section an overview of this project and the related development of an architecture based on Random Access List Structured Memory is presented.
1.3. Background: SPAN and SPRINT

1.3.1 The SPAN project

Within the ESPRIT research plan by the European Community, the SPAN (Symbolic Processing And Numeric) project was concerned with the integration of numeric and symbolic computing. The main part of the project was carried out during the period from 1987 to 1990. A good introduction into the project and particularly into the VLSI research is given in [RCMS89] an off-print of which is therefore included in the end of this thesis.

"The activities within SPAN range from hardware to application level software, with one of the principal activities being the design and implementation of a common virtual architecture, the Virtual Machine or Kernel System, which provides a goal for the high-level language and application software activities, and as a starting point for the hardware workpackages. An associated activity around the Kernel System and language work is concerned with providing support for multi-style programming via an object-oriented programming environment." [RCMS89]

The principle aim of the above mentioned object-oriented programming environment (see Figure 1.4) was to investigate and develop an object oriented framework for integrating heterogeneous hardware and software systems. Thus, the environment would provide for the production of programs which had components written in different languages, the language chosen being appropriate to the task at hand, and with the environment facilitating the exchange of data between the components.

The Kernel System model, although an abstract machine model, embodied a particular form of parallel architecture. This had a set of processors, each with local, but not private, memory. The local memories form the logically, global shared memory of the architecture. Any processor can communicate with any other processor via operations on the local memory of the remote processor, although the communications method was not specified.

The use of a list structured memory, as described in the earlier sections, was a major aspect of the Kernel System.
Chapter 1. Introduction

Figure 1.4: Overview of the SPAN project

The Kernel System (see Figure 1.4) which worked as an interface between software and the underlying hardware, was implemented at two levels: a high level Language called Parle, and a low level language called the Virtual Machine Code (VMC).

The VMC provided a full realisation of the Kernel System model at the level of a 'machine code' or assembler. It was intended to provide a model of the Kernel System that could be ported on to a variety of parallel architectures, and was designed to be a low-level language to reduce the effort to perform the port.

The aim of the VLSI workpackage, at the next lower stage, was to design a parallel computer system that efficiently supported the Kernel System. This was further constrained within the workpackage to support of the VMC. The design was not directed at a machine that directly modelled the VMC, but one that was functionally equivalent and to which VMC 'programs' could be directly mapped. Secondary objectives were to expand the design to make it as general purpose as possible:
supporting different programming styles, to provide an architectural environment suitable for a complex operating system (protection, flexible task scheduling, interrupt support, etc), to provide a robust environment to limit the effects of incorrect programs (on-the-fly bounds checks on list accesses, type checking of instruction operands). The result of this development is the SPRINT processor.

1.3.2 The SPRINT Processor

As the processor was designed to be functionally equivalent to the VMC, the complete memory structure of the parallel processor is list structured, using a tree shaped organisation as shown in Figure 1.1. In this implementation the list structured memory is realised with the following constraints.

The basic tree structure used to hold lists is limited to a maximum of three levels. Together with a given page size of 32 elements, this results in a maximal list length of 32K elements. A list offset of 5 bits allows up to 32 free elements at the list head, enabling list extensions in front of the first list element. The actual length of each list is stored in the list-pointer, and can be used to verify the element selectors used to access particular list elements. A 3 bit tag field offers support for the data-types shown in Table 1.1:

<table>
<thead>
<tr>
<th>Code</th>
<th>data-type</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>1-level list</td>
<td>list pointer to single level structure</td>
</tr>
<tr>
<td>001</td>
<td>2-level list</td>
<td>list pointer to 2-level structure</td>
</tr>
<tr>
<td>010</td>
<td>3-level list</td>
<td>list pointer to 3-level structure</td>
</tr>
<tr>
<td>011</td>
<td>special-list</td>
<td>general purpose</td>
</tr>
<tr>
<td>100</td>
<td>special-atom</td>
<td>general purpose</td>
</tr>
<tr>
<td>101</td>
<td>non-local</td>
<td>context data-type or reference to non-local memory</td>
</tr>
<tr>
<td>110</td>
<td>empty</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>integer</td>
<td>integer data or executable code</td>
</tr>
</tbody>
</table>

Table 1.1: Tag values in SPRINT
Most algorithms manipulating lists will have to distinguish the 3 basic list classification tags as well as the 'integer' and 'empty' tags. All five cases require different actions when encountered in a complex list structure. Furthermore the tags 'special-list' and 'special-atom' are not bound to any special function and can be treated to the needs of the user, while the 'non-local' tag is used for list addressing purposes which are not relevant for the evaluation of a potential co-processor.

SPRINT is based on a 40 bit word width, allowing the list-pointers to take the structure shown in Figure 1.5. The copy bit is a SPRINT internal aspect and not of interest for any list manipulation algorithms. The purpose of tag field, list length and offset have been outlined already.

<table>
<thead>
<tr>
<th>Copy(1)</th>
<th>Tag(3)</th>
<th>Length(15)</th>
<th>Page Number(16)</th>
<th>Offset(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>38</td>
<td>3635</td>
<td>2120</td>
<td>5 4 0</td>
</tr>
</tbody>
</table>

Figure 1.5: Structure of a SPRINT List Pointer

Since each page has a base address being a multiple of 32, the 16-bit Page Number can always be zero extended by 5 bits resulting in a basic 21-bit address space. The intermediate page pointers in two- and three-level lists are not bound to these limitations given by the format of a list-pointer; the SPRINT architecture has the option of using extended addresses for intermediate and bottom level pages. As the address-bus of SPRINT is 24 bits wide this is also the limitation for the extended address space.

Since SPRINT uses a basic tree shaped version of the list structured memory concept, there is no simple way of expressing complex graph and cyclic data structures. Therefore this architecture is not particularly suitable for programming languages such as LISP depending on this kind of data structuring.

1.3.3 Support for Modern Programming Paradigms

The 'Random Access List Structured Memory' approach, as developed in the SPAN project, can be interpreted as creating a separate memory space for each list, similar
1.3. Background: SPAN and SPRINT

To segments in memory architectures based on a linear memory organisation. The only way to address objects in memory is via the list-pointer and a selector. This behaviour matches the concept of encapsulation of the Object-oriented paradigm. The same correspondence between the memory model and Object-oriented paradigm exists for the hierarchical structure. This hierarchical structure enables the programmer to build complex objects consisting of sublists representing program code, status and local data. This enables complete information hiding, as the whole object is only accessible via its list-pointer.

When the list structured memory is used to implement a one-dimensional array (Figure 1.6), the access works as in a linear memory with the exception that the length of the array can be extended on demand.

![Figure 1.6: One-dimensional Array](image)

When the number of dimensions of an array is higher, the array can be expressed by an hierarchy of sub-lists, as shown in Figure 1.7 with a 2-dimensional array. With such an structured approach it is possible to change the number of elements in any dimension by adjusting the corresponding lists.

![Figure 1.7: Two-dimensional Array](image)

Standard imperative programming languages can organise code and data in the traditional way as with linear memory by mapping a program on to a single list, or
alternatively profit from the list structured memory and organise different components of a program in a structured way: an example is shown in Figure 1.8.

![Figure 1.8: Example C-program Structure](image)

In addition to the fact that the tagged memory concept enables the support of dynamic typed languages, it also simplifies the use of heterogeneous lists in other programming languages. The availability of basic types in the form of tags also may be used to implement the abstract data types used in modern programming paradigms.

As mentioned in the section about SPRINT, the basic tree shaped concept of list structured memory offers no simple way of building graph structures and cyclic references. In slightly more complex variations of list structured memory architectures these structures can be represented without complications. Since these, more complex, graph structured memory architectures will enable multiple references to lists, additional mechanisms are necessary to keep track of active lists, or to find inactive lists. In a tree shaped list structure any list can be deleted as soon as its only reference is deleted or overwritten, whereas in a system allowing multiple references it is always possible that another reference still requires the list. Any standard ‘Garbage Collection’ technique [Pra75] developed for existing symbolic computing environments should also be suitable for more complex list structures. Since some symbolic programming languages such as LISP require graph and cyclic structures to develop reasonable performance, a tree shaped memory structure as in SPRINT is not ideal.
1.4 Alternative Hardware Architectures

The tree-based memory concept introduced in the section 1.2 and used in the experimental SPRINT architecture is a basic concept that can have multiple variations. One variable is the amount of access privilege checking and the creation of separate memory sections for individual processes. The SPRINT architecture does not offer any data protection by means of limited read and/or write permission to individual lists. SPRINT also has no separation of data belonging to different processes. The 3 tag bits of SPRINT only allow a coarse grain differentiation of data types. One or two additional bits in the tag field could be used to support a wider number of basic data types (e.g. multiple numeric types and a special indicator for program code).

Of course there exist a number of other important factors influencing the whole architecture of systems containing Random Access List Structured Memory. For example, whether the complete system memory or only a part of it is designed as a Random Access List Structured Memory. Another factor is the support of virtual memory and the method of implementation, should one use page or list swapping.

1.4.1 Access Rights and Protection

The most elementary version of a Random Access List Structured Memory does not use any additional hardware support for list access control. This means that the whole memory, starting from the memory root pointer, can be seen as a pool of lists holding both code and data. When a list element refers to another list, the list-pointer is directly included in the element. This corresponds to a single linear address space (paged or not) in regular memory architectures. It represents a direct implementation of the tree structured memory hierarchy shown in Figure 1.1.

Better memory protection and information security, which are desirable in a multitasking environment, can be achieved with indirect list access. Instead of using pointers to list structures, memory elements now only hold the addresses of descriptors.
Within each descriptor information about access rights and ownership can be kept together with the basic list information. Whenever a reference to a new list is evaluated, it is necessary to verify the access privileges of the demanding task. Only after this verification succeeds can access to the requested list element be granted. This approach is comparable to the protection methods used in existing virtual memory systems [Fur89]. The information about lists held in the descriptor could also include a reference counter to support Garbage Collection methods relying on this information.

With respect to Figure 1.1, there is a level of indirection in moving from one list to a sub-list, and it is now possible to have multiple pointers to a list and thus to adopt a graph structured rather than tree structured memory.

Even if expressions such as pages and segments are used, the alternatives discussed here are purely for the management of the physical level of the list structured memory. On top of this physical memory management it is still possible to have an additional layer implementing a virtual memory system.

1.4.2 List Structured Memory as Extension

So far list structured memory has only been considered as a basic memory concept for new architectures. Adding Random Access List Structured Memory to existing architectures, based on a linear memory system, is a second possibility [RD93]. Architectures targeted for numeric applications are the first area of interest. In this type of architectures dynamic arrays could replace extensive linked lists. However this kind of support could also be helpful in the process of manipulating data where the amount and type of data are not known before run-time (e.g. buffering polymorphic message channels).

The main idea behind this concept is to attach the list structured memory to the bus system of the architecture, but to hide all the details of the list memory from the CPU of the host system. A controller will be used as interface between the existing architecture and the List structured memory. For basic read and write operations
the controller translates what seems to be a linear address into a list selector and an element selector, and then performs the requested read or write access inside the list structure. For more complex list manipulations there are two alternatives:

a) the CPU manipulates the list structure while the controller is disabled, or

b) a more complex controller directly executes the manipulations.

1.5 Pure List Structured Memory Architectures

When developing a new architecture completely based on Random Access List Structured Memory, the scalability of the performance of such an architecture should be an important consideration. The scalability of architectures is essential, because commercial products have to cover a range of computing power requirements, which will allow the user to stay with the architecture when the demand for performance grows. There are multiple factors of the hardware structure influencing the performance of a computer system. By weighing these factors carefully, a set of hardware configurations representing a range of machines with growing performance (and price) can be created. Some general scaling elements for the performance of computer architectures are:

- CPU and memory speed;
- special purpose accelerating hardware;
- degree of parallelism at node level;
- number of computing nodes;
- processor interconnection topology; and
- inter node communication bandwidth.
The basic speed of processors and memory is technology dependent and is beyond the scope of this work. Anyway there is the fact that developing a CPU according to RISC design philosophy will enable a shorter product cycle and, consequently, allow new versions using the latest technology to be introduced more often. Some of the ideas for the design of advanced processors [Hwa93] such as 'super-scalar processors', 'vector units' and in particular 'instruction pipelining' can also be applied to CPUs operating on List Structured Memory. One major element of all processors will be the support for tag analysis, and manipulation, in parallel with the regular data manipulations. Performing the checks and manipulations on the tag-field within regular instructions takes a significant amount of complexity out of programs for symbolic computing and gives additional software reliability which otherwise might not be included in code. In addition to the choices given in the design of CPUs, there are also several techniques in the fields of memory hierarchies and virtual memory which can be employed to vary the performance of a computer system and create a range of machines using a common concept of List Structured Memory.

Whenever the CPU spends a large amount of its time on performing a small group of tasks, it is worth checking if there is a more efficient way of performing these tasks. Normally purpose built hardware can accelerate these tasks. Depending on the area of application and frequency of demand, the designer can improve the performance of a machine by adding such hardware to handle these requests. Typical examples for acceleration hardware are: floating point units (FPU), direct memory access units (DMA), graphical co-processors and digital signal processors (DSP). For architectures based on Random Access List Structured Memory complex list manipulating functions are an additional area offering the potential for acceleration.

While adding co-processors and other accelerating hardware can improve the performance of a machine quite considerably, there is still the limitation of bus arbitration. Normally only one processor can have the right to access the bus system at any time. This limits the amount of possible parallelism of execution to one processor requiring memory access, while all other units operate on internal data or have to stay idle. The only way to overcome this limitation is to improve the topology of the node's internal bus system and the use of multi-ported instead of single-ported
memory. Such increased complexity provides a higher node performance, generated by allowing multiple memory transactions in parallel.

Increasing the number of processing nodes normally can be used to scale architectures. There are again problems in the areas of memory access organisation and inter processor communication. The experimental SPRINT architecture [RCMS89] demonstrates how a multi-processor system based on distributed List Structured Memory could be designed, whereby the existing tags are used to generate a mechanism providing a global addressing scheme and to handle process communication.

1.6 Structure of the Thesis

After the general background has been given, this thesis will systematically investigate individual aspects related to the enhancement of architectures with Random Access List Structured Memory. The individual aspects introduced in the following chapters are not necessarily presented in a temporal order, instead the work is structured according to topics. The topics of the chapters are as follows:

Chapter 2 will review the ways of manipulating data structures in Random Access List Structured Memory. Special interest will be given to load intensive functions performing complex list manipulations. The traversing of list structures and memory management are discussed as two important areas offering potential for hardware to accelerate operations on data structures.

Chapter 3 demonstrates the two basic concepts using Random Access List Structured Memory. First it is explained how architectures using linear memory can be enhanced by adding Random Access List Structured Memory. The functionality of a controller interfacing the system buses and the added memory is covered for basic read/write operations as well as for complex list manipulations.

The second option is to develop new architectures based completely on Random Access List Structured Memory. A number of architectural constraints which have
to be considered in such a design are presented. Then various integration concepts for a co-processor, performing complex list manipulations, are demonstrated.

Chapter 4 covers the development of algorithms for co-processors to perform complex list manipulations. A major aspect is memory management and in particular the administration of free pages. Garbage collection support, error handling and variations of general algorithms are discussed next. Finally these points are applied and a set of instructions for a SPRINT co-processor is introduced.

Chapter 5 discusses methods to improve the effectiveness of the datapath in list manipulating co-processors. The implementation of a stack mechanism and the arithmetic functionality are of special interest. These general techniques are then applied for the SPRINT co-processor.

Chapter 6 gives an overview of the various simulations performed during the development of the co-processor. A high level simulation is used to study the structure of list manipulating algorithms and to identify the datapath requirements. The further development of the algorithms then requires a formal description of the micro-program. Existing micro-code development techniques will be analysed and a new technique is introduced. A behavioural simulation of the micro-program is used to check the functionality of the algorithms and the datapath. Once the characteristics of the micro-program are known, it is possible to develop a structure for the processor control part. A controller structure with reduced area requirements is introduced and its functionality proven with a final gate-level simulation.

Chapter 7 compares the performance of the simulated co-processor to software solutions performing the same list manipulations. An examination of the co-processor practicality shows that the suggested co-processor could be produced using currently available technology.

1.7 Statement of Originality

The work presented in this thesis uses the ideas of Random Access List Structured Memory and in particular the architectural concepts of the SPRINT architecture.
On top of these foundations the following contributions to knowledge were achieved:

- A co-processor for list manipulation has been developed. With this co-processor it is possible to support the manipulation (copy, modify, delete, ...) of complex data structures in hardware, whereas in a conventional architecture the same tasks have to be performed in software, as corresponding DMA devices only support continuous linear address areas.

- A new methodology for the development of microcode has been presented. The integration of databases into the development of microcode using the flowcharting approach offers a significant amount of automation, while the flexibility of the design process is hardly limited. This new methodology is applicable to processor structures where current microcode design methods fail to offer sufficient support.

These findings and related aspects are presented as follows:

**Chapter 3** Presents different configurations of co-processor(s) connected to processor nodes and considers the advantages and disadvantages of these architectures. The SPRINT architecture is used to illustrate the individual concepts. A model to enhance existing computer architectures by adding Random Access List Structured Memory is introduced. The ideas for this approach were developed jointly with Dr. Peter Rounce.

**Chapter 4** Discusses different approaches to implement the memory management in an architecture with Random Access List Structured Memory. Different solutions for the administration of free memory pages and their allocation are presented. A set of possible functions for a SPRINT co-processor are explained, considering general aspects as well as special aspects related to the architecture.

**Chapter 5** Explains various design aspects of a datapath for a co-processor operating on Random Access List Structured Memory. The implementation of the stack mechanism and the arithmetic functionality are discussed in detail.
The general ideas for the datapath design are applied to the architectural guidelines of the SPRINT architecture and the datapath of the experimental co-processor is introduced.

Chapter 6 Demonstrates the use of the new microcode design methodology using databases to automate the flowcharting approach. The method was developed during this project and was published in the 'Microprocessors and Microsystems' journal [DR94].
Chapter 2

Manipulating List Structured Memory

The principal ideas behind the Random Access List Structured Memory concept were introduced in the previous chapter, and this section will concentrate on the manipulation of data stored in such a memory. The analysis of these memory manipulations is important, as the resulting knowledge will be the key for improvement.

The basic type of memory manipulations will of course be formed by elemental operations on single memory words. As in a regular memory architecture, there will be operations where the processor reads or writes a data value from/into a given memory location. During such a memory access either the active processor, or a special Memory Management Unit (MMU), is responsible for converting the address from a List-pointer and Selector representation into the corresponding physical memory address. Depending on the requirements of the architecture more complex atomic single word operations such as read-modify-write cycles can be supported in the same way as in a standard memory.

While these standard operations are just a basic necessity for any memory concept, a real advantage of a Random Access List Structured Memory lies in the area of complex list operations. A number of possible operations are introduced here. While some of these functions are essential, others might just be useful and could be derived from essential operations. The set of essential functions includes: list
construction, list length extraction and single element extraction. In addition there are the useful operations which extract a given number of elements out of a list, concatenate lists, delete lists and insert additional elements into an existing list.

Since Random Access List Structured Memory with its hierarchical arrangement of pages is a complex construct, it cannot be expected of a CPU to perform all of these memory manipulations efficiently. As a consequence it should be considered how architectures using this memory concept can be improved with additional hardware at the node level. Providing a co-processor for manipulations of larger, potentially nested, list structures is an obvious proposal. The advantage of such a device could be even greater if the included functions could be adapted to the requirements of individual situations.

2.1 Frequent Load-intensive Functions

Depending on the environment a computer system will be used in and the applications to be run, different sets of list manipulations will be required frequently and would therefore be good candidates for acceleration. The SPAN project did not provide adequate numeric material about the frequency of particular list manipulations for specific applications in a list structured memory architecture. Therefore the arguments in this section can only be based on assumptions.

Numerical Applications

For mainly numerical applications the most attractive aspect of Random Access List Structured Memory is the availability of dynamic arrays. The possibility of concatenating new elements to a list without the need to move each element of the array gives much more flexibility to various algorithms. Typical manipulations on complete arrays are functions such as: initialising arrays with a particular value, extracting parts and generating a copy of an array. More complex functions such as sorting can often be implemented using these basic functions.
2.1. Frequent Load-intensive Functions

Depending on the actual implementation of the list structured memory, certain data structures such as multi-dimensional arrays and sparse matrices can be expressed in novel ways. However the variety of different data structures and the related methods of manipulation can hardly be predicted. As a co-processor can only contain a limited number of algorithms, a fixed selection of instructions to be implemented in a co-processor is a waste of potential. A much better solution would be to supply the user with a library of specialised algorithms executable on a co-processor. Depending on his requirements, the user then can choose a number of algorithms to be implemented in the co-processor. A further advantage of such a collection of algorithms is the fact, that a user could add his own algorithms, meeting individual requirements. A typical scenario for this kind of user defined algorithms could be the support of the user defined tag fields of the SPRINT architecture (see section 1.3.2).

The usefulness of a co-processor will largely depend on the number of implemented functions and the adaptability to the needs of the user. The number of implemented functions depends mainly on the size of the individual algorithms and the storage capacity of the co-processor. Including the required algorithms in a co-processor can be performed at various levels:

- One basic method would be to allow the user to choose a number of algorithms and then modify a basic chip according to this choice. This could be done by storing the algorithms in a PROM\(^1\) integrated in the co-processor. Naturally, once he has chosen, the user can not modify the co-processor.

- The use of an on-chip EPROM\(^2\) increases the usefulness of a co-processor only slightly, as the process of replacing the algorithms is quite complex.

- When the algorithms are stored at system startup in a RAM\(^3\) integrated in the co-processor, a change of algorithms can be performed by the user. However

---

\(^1\)Programmable Read Only Memory
\(^2\)Erasable Programmable Read Only Memory
\(^3\)Random Access Memory
any change of algorithms would still require to shut down and re-start the
machine with all its consequences.

- A further improvement of this idea is to allow the user to change the algorithms
stored in the co-processor while the machine is active. The additional com­
plexity of the co-processor structure and the invoking routines enables the user
to change the functionality of the co-processor while his data is maintained in
memory.

The latter two solutions offer good support, as the flexible instruction set allows the
user to select a number of algorithms according to demand.

Assuming a co-processor with such a flexible instruction set, individual algo­
rithms manipulating complex list-based data structures could be executed entirely
by the co-processor. For other algorithms where the internal functionality of the co­
processor is not sufficient, the device still could be used to improve the data access
of the CPU.

Symbolic Applications

When looking at applications in the area of symbolic and functional processing,
the major advantages of the memory concept derive from the support for complex
memory structures and dynamic typing. Depending on the programming paradigm
of the different applications, frequently used functions might be:

- copying nested list structures;
- merging lists;
- deleting or initialising lists; and
- garbage collection.

All these functions can be applied to complete lists as well as specific parts of
lists. The way a co-processor could support the garbage collection process natu­
rally depends on the implementation of the list structured memory and the chosen
method.
2.2. Analysing the Functional Requirements

Again a flexible instruction set could enable the implementation of non standard functions inside the co-processor, as well as functions supporting the CPU.

Communication and I/O

Within multi-processor systems the transfer of complex lists structures between processor nodes is, of course, an additional issue that can be accelerated with purpose build hardware. Transferring these list structures to and from secondary storage media has similar requirements.

2.2 Analysing the Functional Requirements

Once the possible functionality of a co-processor has been determined, it is necessary to find out how the different functions can best be integrated. As there will always be a limitation on the functionality that can be included in a single device, these resources should be used in a way supporting common requirements of many significant functions.

A common element of most of the above mentioned functions is high memory access frequency. Any co-processor implementing a number of these functions can therefore be seen as a Direct Memory Access (DMA) processor for Random Access List Structured Memory. To get the best performance out of such a DMA device it is necessary to optimise the data throughput. This means that the algorithms should have a target of one memory read or write instruction per cycle. As a consequence of this aim, address generation and other processor internal data manipulations should be executed in parallel whenever this is possible.

2.2.1 List Traversal

The list structures of the targeted memory architectures and the possibility of nested lists will require the algorithms to traverse lists. This process of traversing a list
means to sequentially read all elements of a (sub-)list and manipulate them according to the encountered data-type and the executed function. A closer examination of this component and the basic algorithm shows the more important aspects.

Figure 2.1: Basic list traversal algorithm

The basic traversal algorithm shown in Figure 2.1 is efficient as long as the manipulated list does not contain any pointers to other lists. If such nesting of lists occurs, there are two principal methods to handle the traversal of sublists:

- **Depth First Traversal (DFT):** Record the actual status and start to manipulate the nested list. After the traversal of the sub-list is finished, restore the old status and continue traversing the original list.

- **Layer First Traversal (LFT):** Record where references to sub-lists occur and the information given for the nested lists. As soon as the traversal of the original list is finished, start handling the stored references.

Both methods require dynamically allocated memory to keep the information about the points to continue processing, as there is always the possibility of more
2.2. Analysing the Functional Requirements

than one reference to sublists. The way the stored references are handled is different:
DFT uses a 'Last-In-First-Out' (Stack) strategy, whereas LFT can operate on the
basis of either the 'LIFO' or the 'First-In-First-Out' (Queue) strategy. The process
of deciding which of these two methods best suits the requirements of a co-processor
is based on these facts:

- A Stack normally requires less control than a Queue. Therefore both methods,
  DFT and LFT, could be implemented with a stack control mechanism.

- The size of a single stack-entry for LFT is smaller, but in the case of a complex
  list the number of stack-entries for LFT can outgrow the one for DFT by
  many times. While the number of entries in the case of DFT is limited to
  the maximal tree-depth, in the case of LFT the worst case estimation is the
  number of elements in the current lists. Thus, judged by the stack size to be
  expected, DFT seems to be the better choice.

- Depending on the individual operation (copy,...) in the case of LFT the list
  element containing a reference to a nested list may have to be accessed a second
  time to write or update data (e.g. during a list-copy instruction a nested list
  is encountered, after the original list is duplicated, the sub-list will be copied.
  Depending on the architecture of the list structured memory and the details
  of the copying algorithm, the information about the list-pointer to the copied
  sub-list might only be known after this copying process is completed. As this
  information is required in the pointer included in the top-level list, this list
  element will require updating after the sub-list is copied). In the case of DFT
  the necessary data for such a modification of a list pointer is always available
  when the control returns from handling the nested list. As the second access to
  a list element would mean an extra overhead, DFT is the better choice again.

  Even if there are various possibilities of implementing a Random Access List
  Structured Memory, the information about the length of a list will always be avail-
  able. In the two implementations mentioned so far, the list length is held either in
  the list-pointer or, amongst other list information, in a descriptor table.
Hence, whenever a list-traversal occurs, it is possible to determine the last ele­
ment of a list by comparing the number of already traversed elements with the actual length of the list. The way that the first element of a list is accessed depends on the actual implementation and whether the system supports variable list structures (e.g. different number of levels for basic lists).

For a system such as SPRINT, with a maximum of three page levels, a maximum element offset of one page length and direct list-pointers, the process of calculating the address of the first element for each of the 3 possible structures could be described as follows:

1-level structure: Take the address of the list-pointer as the base-address and add the offset. This is the address of the first list element.

2-level structure: Use the address in the list-pointer as an intermediate page. The first word of this page contains the base-address of the first 2nd level page. Just add the offset to get the location of the first list element.

3-level structure: This time the list-pointer refers to the top page. Get the first word as address of the intermediate page and use the first word of this as base address. Now add the offset to this base address. The result is the address of the first list element.

The next element of the traversed list will normally be in the next memory word of the base page. As each page can only hold a number of elements equal to the page length 'n', this is not possible all the time. Instead of accessing the \((n+1)\)th element of the base page, it is necessary to use the first element of the next page containing data belonging to the current list. The base address of this next page can be determined by reading the value in the following word of the intermediate page of a multi-level list. Within lists of Level 3 or more, the same system is used for pages in the intermediate and top levels. As the method shows, it is necessary to keep track of the last element used at each level of a list. To speed up this process the actual addresses of top-, intermediate- and base-page(s) are kept in variables as well, so that their values only have to be regenerated when the end of the current page at
this level is reached. Depending on the manipulation performed on the list, it often necessary to keep track of two tree structures: one for the source and one for the target-list. The general algorithm to traverse a list, according to the assumptions made earlier on, is shown in Figure 2.2.

```c
base_counter := offset;
intermediate_counter := 0;
top_counter := 0;
steps := 0;
switch (list_level)
{
    case 1: base_page := listpointer.addr;
        break;
    case 2: intermediate_page := listpointer.addr;
        base_page := intermediate_page[intermediate_counter];
        break;
    case 3: top_page := listpointer.addr;
        intermediate_page := top_page[top_counter];
        base_page := intermediate_page[intermediate_counter];
        break;
}
while (steps != length)
{
    HANDLE_ELEMENT(base_page[base_counter]);
    base_counter++;
    if (base_counter = PAGE_LENGTH)
    {
        base_counter := 0;
        intermediate_counter++;
        if (intermediate_counter = PAGE_LENGTH)
        {
            intermediate_counter := 0;
            top_counter++;
            intermediate_page := top_page[top_counter];
        }
        base_page := intermediate_page[intermediate_counter];
    }
    steps++;
}
```

Figure 2.2: Basic Traversal algorithm for 3-level page structure

To access an indexed element of any page it is only necessary to concatenate the address of the page and the index of the element. This is possible by the normal conditions of paged systems: that every page has to start on a page boundary, and the page length is a power of 2. Thus the lowest \( \log_2(\text{PAGE LENGTH}) \) bits of each address are zero and the concatenated element counter will determine the required memory-word.
2.2.2 Memory Management

Several of the list manipulating algorithms mentioned earlier will either demand new pages (copy,...) or return pages (delete,...) to the free page pool.

Calling the operating system for each of these requests is not efficient, as this would require halting the co-processor activity, passing information to the CPU, getting the CPU to execute the necessary system routines, returning the result, and finally restarting the halted co-processor. Depending on the structure of the processing node, this procedure will normally involve sending interrupts to the CPU and the arbitration of the system buses, adding complexity to the co-processor control. In the CPU the execution of the operating system routines might require the switching of processes and costly operations to save register values. In case of a CPU busy-waiting for the co-processor to execute a complex list manipulation, the handling of co-processor interrupts during the busy wait phase can also add complexity to the CPU routines.

On the other hand, the handling of fixed size memory pages is not too complex. Therefore it is appropriate to include basic memory management functionality into possible co-processors, allowing the above mentioned algorithms to perform the required functions locally.

Once the hardware support for basic memory management is included in a co-processor, it is necessary to consider if the complete memory management is to be implemented in hardware, or how it is going to be split between operating system and co-processor.
2.3 Summary

The ideas presented in this chapter identify complex list manipulations which are expected to be load intensive functions and will be used by application software and system routines. Possible features for the co-processor, that were identified, are:

- A set of functions manipulating dynamic arrays to support numerical applications.
- A collection of list manipulating functions to support operations on complex list structures used in symbolic computing.
- A combination of functions required to flatten and reconstruct list structures to enable a transfer of lists between different processors and/or secondary storage media.
- A basic set of memory management functions, as these are required by most of the previous algorithms and operating system calls are not effective.

Due to the variety of data structures a Random Access List Structured Memory concept can support, there is a large number of possible algorithms. A co-processor can only hold a limited number of these algorithms. To enable a frequent use of such a device it might therefore be an advantage if the instruction set could be adapted to the need of the user.

The functional resources of a co-processor will be another limiting factor, consequently the functions to be implemented in a co-processor should be closely related and utilise the same resources. List traversal is a complex component of many list manipulations and suitable to be executed in a co-processor.

An analysis of the requirements of the list traversal indicates that a 'Depth First Traversal' produces better results than 'Layer First Traversal'. A stack mechanism will be required to store the data related to higher level lists whenever the traversal function recursively handles nested lists.
All these results were used to identify the functionality of the implemented co-processor as introduced starting from Chapter 4. Individual stages in this design are the investigation of memory management functions and the selection of a basic co-processor instruction set, both presented in Chapter 4, and the conversion of the list traversal algorithm into a purpose build datapath as presented in Chapter 5.
Chapter 3

Hardware Alternatives

The idea of Random Access List Structured Memory and the advantages it can offer to computer architectures have been introduced in the first chapter. Since Random Access List Structured Memory is a concept for memory organisation and not a complete architecture, it can be used in different architectural environments. The two basic possibilities mentioned so far were based on the ideas of: adding list structured memory to existing architectures, or building computer systems completely based on this memory concept. Within each of these basic configurations there are still many possible variations. Accelerating hardware for the processing of complex list manipulations, as discussed in Chapter 2, is just one example. Since both of the basic alternatives can profit from such an improvement of the node architecture, it is of special interest.

This chapter will analyse both basic approaches in more detail and derive individual variations offering different performance or satisfying individual requirements. When discussing the options for architectures purely based on Random Access List Structured Memory, the main focus is on the analysis of different methods of integrating a co-processor into the SPRINT node architecture.\(^1\)

---

\(^1\)As the SPRINT architecture is designed as a multiprocessor, it is necessary to differentiate between the architecture of a processing node, the node architecture, and the overall system architecture.
3.1 Enhancing Existing Architectures

As seen earlier, the Random Access List Structured Memory concept requires the use of tag fields within the individual memory words to distinguish between atomic data types and references to sub-lists. Since the majority of commercially available microprocessors do not support tagged memory words, it is not possible to build efficient architectures with these microprocessors directly connected to Random Access List Structured Memory. To enable standard architectures to profit from an added List Structured Memory, it is therefore necessary to integrate this memory into the linear address space. This means that the CPU has to operate on the Random Access List Structured Memory as if it would be a part of the regular memory.

3.1.1 Structure

To achieve this integration of List Structured Memory into the existing memory space, a special purpose controller has to be placed between the address bus and the List Structured Memory. The controller in such a configuration (Figure 3.1) will have a dual functionality. In a standard translation mode the controller will, during each read/write access, convert the pseudo linear address, used by the CPU to address a memory element inside a list structure, into a list-pointer and an element selector. In the second mode, the controller will allow the CPU to perform more complex manipulations of the list structure.

![Figure 3.1: Basic structure for added List Memory](image)

To enable a proper translation of pseudo linear addresses into a list-pointer and an element selector during a read/write access, the controller has to distinguish these
3.1. Enhancing Existing Architectures

accesses to the Random Access List Structured Memory from accesses to other memory areas or system components. When an element inside a list structure is accessed, the pseudo linear address also has to contain sufficient information to identify the required list and list element. This encoding of information inside the pseudo linear address is necessary as the microprocessor and other devices outside the controller only can interpret the list structured memory, and all the lists contained in it, as a group of memory pages.

To enable the controller to identify an access to the list structured memory and recognise the requested list and list element, a pseudo linear address used to access a list structure must contain at least three data fields:

- An indicator field to identify an access of the Random Access List Structured Memory: \( p \) bits.
- A list selector field to specify the required list: \( n \) bits.
- An element selector field to determine the required list element: \( m \) bits.

Thus the address of a list element would be structured as Figure 3.2:

```
<table>
<thead>
<tr>
<th>list memory select</th>
<th>list selector</th>
<th>list element selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p ) bits</td>
<td>( n ) bits</td>
<td>( m ) bits</td>
</tr>
</tbody>
</table>
```

Figure 3.2: Address structure for an added list memory

The list memory select field is used to distinguish addresses in linear memory from those in the list structured address space. In addition, it defines the maximum range for both address spaces. To avoid an abstrusely interlocked memory space, the list memory selector should be placed in the most significant bits of an address.

The list selector field is used to determine the list to be accessed out of a set of available lists. The size of this selector field defines the number of lists to be held in the list memory. The list element selector holds the information about the list element to be accessed.
Chapter 3. Hardware Alternatives

The latter two fields, list selector and list element selector, need to be large to maximise the number of accessible lists and the list length. Consequently the list memory selector should be as small as possible. The choice of its length $p$ determines the possible ratio of linear and list structured address space. Setting $p$ to the minimal value of one splits the memory space into two equal size sections for both address spaces. Larger values of $p$ would give more flexibility to model the ratio.

Once a value for $p$ has been decided, the remaining bits of the address width have to be split up between $n$ and $m$. This decision between the number of lists and their maximum length will be influenced by the area of application and the host architecture.

Within a multi-tasking environment a process switch can also be used to switch between different sets of accessible lists. Such a change between multiple list sets reduces the requirements in the length of the list selector $n$ and therefore allows slightly longer lists.

3.1.2 List Memory Access

When a memory access is intended for an element of the list memory, the corresponding value in the list memory selector field of the address will activate the list memory controller. The information in the list selector field is then used to identify the list specifier, which contains information about the list structure and the address of the top-level page. The controller always has to hold the specifiers for all list structures in the current set of accessible lists. Depending on the size of this set it will not always be possible to hold all these specifiers inside the controller. If the number of specifiers is too large to be held in a set of registers, the controller will have to maintain a table with all specifiers in memory and fetch the required specifier during each access. To accelerate this process, it is possible to hold a number of recently used specifiers in a fast memory inside the controller, similar to 'Translation Look-aside Buffers' (TLB) in virtual memory architectures [Hwa93].
3.1. Enhancing Existing Architectures

After the required specifier is obtained, the controller can extract the field containing the address of the top-level page and use the information given in the list element selector to traverse the tree structure of the list and access the location holding the required element. The traversal of the list structure normally requires one read cycle per level of the structure to analyse the pointer information held in intermediate list levels. Caching of page pointers used during recent accesses can accelerate the traversal by skipping the read accesses for intermediate levels.

In addition the controller will verify that the list selector is valid and that the selected list element is within the boundaries of the list. The latter test requires the knowledge of list length and possible offsets. In case one of these tests fails the access has to be terminated, since a program failure has occurred.

As the Random Access List Structured Memory concept uses tags in its internal data representation the memory controller has to manage these tags. A simple and common case is given when homogeneous lists are manipulated. To write data into such a list it is only necessary to tell the controller once which type of data is going to follow; the controller will then add the corresponding tag to each data value written. When reading data from the list memory a similar approach is possible. The controller is given an expected data type and returns the accessed memory contents as long as they match the given type. When a read value has a different type, the controller can raise an exception and the requesting program has to decide on the consequences of the type mismatch. A simpler, but less secure way of reading data from the list memory would be to instruct the controller to drop all the tag information and return the pure data values.

When the list structured memory is used to store complex data structures with varying data types the passing of type information will become more frequent, with a maximum of one type information per data access. This may seem to create a big overhead, but current general purpose architectures also have to split tag and data into two memory words when processing programs written in a symbolic, dynamically typed programming language.
3.1.3 Complex List Operations

So far only simple read and write operations on the list structured memory have been discussed. While these simple list accesses will be executing autonomously, more complex operations handling larger list sections will require different strategies.

In principle it would be possible for the processor to execute all complex list manipulations, in software, directly on the list memory while the controller is disabled. Any process requesting a non trivial manipulation of the list structures would initialise a system call which then would temporarily disable the list memory controller and perform the required changes. However, this technique would produce a considerable overhead to perform the list manipulations and ignores the potential of a controller executing the same manipulations in purpose built hardware.

Assuming that the controller is designed to execute some of these complex list manipulations, there are multiple ways to pass the information about the required instruction to the controller:

- A simple method would be to restrict the use of certain list selectors (e.g. list 0) and to use write operations on these lists to pass instructions to the controller.

- A more standard way would be to have a set of controller registers memory mapped into the linear address space. All communication between processor and list memory controller can then be handled similarly to other I/O devices in the system.

- If the host processor has specialised I/O instructions, it is also possible to connect the controller to a number of I/O-ports.

The details about the algorithms to be implemented are covered in Chapter 4, as there are only minor differences to the algorithms for an architecture with a purely list structured memory.
The response of the controller to an operation request while it is performing a complex operation must be considered. Obviously a further complex operation has to be refused, and delayed until the current manipulation has completed. However, a simple access request to a list element has the following possibilities:

- **Busy-wait** the access until the controller has completed the complex operation. As the manipulations of large list structures can be quite time consuming, this strategy can leave the processor idle for a considerable time.

- Terminate the access with a bus error exception, causing the process to be suspended. In general this strategy is an improvement to busy-waiting, but can cause excessive process switching.

- **Busy-wait** the access briefly until the currently executed complex operation reaches a point where suspension is possible.

- Split the controller into 2 components (Figure 3.3): an access controller and a complex operations controller, where both controllers contend for access to the List Structured Memory.

![Figure 3.3: Advanced structure for added List Memory](image)

The last two techniques both guarantee a reasonable system performance for the cost of additional complexity in the controller.

### 3.2 New Architectures based on List Structured Memory

While the extension of existing architectures with List Structured Memory can be seen more as temporary solution to enhance the facilities of current architectures,
the design of new architectures based solely on List Structured Memory seems to offer more potential.

When the basic concept of Random Access List Structured Memory was outlined in Chapter 1, it was already indicated that there is a wide spectrum of possible architectures related to the detailed structure of a list (tree-shaped, directed graphs, cyclic references), the list size, the way lists are accessed (direct, via descriptors) and the access protection to be implemented. The decision to develop a new computer architecture based on any combination of these factors automatically imposes a number of other architectural factors on the new design:

- To enable the differentiation of various types of list elements (atomic data, list-pointers, empty words, ...) any Random Access List Structured Memory architecture will require a tagged memory. As a result, any CPU used in such a system has to support the chosen memory word format. Most notably the CPU will have to perform efficient tag analysis and manipulation of data according to the tags.

- The processor architecture has to support efficient evaluation of list references, as addresses will frequently be given as a list of references to individual list elements starting from a reference point (e.g. \texttt{[root 9 4 2 4]} as in Figure 1.1). Normally this kind of support can only be achieved by specialised instructions for this purpose.

- As with the efficient list reference evaluation, the processor architecture has to offer fast access of individual list elements. Within larger lists using a hierarchical page structure these access times can normally be improved by maintaining the addresses of recently used intermediate pages in the register set or a data cache.

- Depending on the structure of the intended Random Access List Structured Memory, the new architecture might require a garbage collection mechanism. If such a mechanism is necessary there might as well be additional hardware requirements (e.g. additional flags and/or reference counters) to support the chosen garbage collection algorithms.
3.3. Co-processor Design Considerations

Even with these constraints a general analysis of architectures using the List Structured Memory concept is not practicable. It will therefore be necessary to reduce the analysis to a particular implementation of list structured memory, and afterwards relate the gained results to other implementations. The concept of the SPRINT architecture introduced in Section 1.3.2, offers a possible environment for such a study. The fact that an existing architecture is used to investigate particular architectural enhancements, will also be an advantage during the evaluation of new node architectures and list manipulation algorithms. Limiting the study to a particular version of list structured memory will, of course, influence the range of hardware alternatives and solutions to particular problems. However, it should be possible to transfer most of the ideas and concepts discussed, in combination with the extension of the SPRINT architecture, to other architectures.

3.3 Co-processor Design Considerations

Amongst the different possibilities of improving the node architecture of any system based on Random Access List Structured Memory, the integration of a co-processor speeding up complex list manipulations and memory management is most likely to be used in a wide area of applications. The internal structure of such a co-processor and the integration into the node architecture depend on the intended functionality and surrounding hardware. While investigating the expansion potential of the SPRINT node architecture, several constraints were found.

3.3.1 Instruction Set and Controller Structure

Any co-processor should support a wide range of applications and particularly the user-defined tags 'special-list' and 'special-atom'. A flexible instruction set, as discussed in Section 2.1, has the basic potential to offer this kind of support. Since a major objective of the co-processor development is the fast manipulation of lists, it is not feasible to perform instruction fetches from an external memory during execution. Thus a variety of algorithms has to be stored inside the device using modifiable
memory. The most common way to implement this is to build a microprogrammed controller with an on-chip Writable Control Store (WCS)\(^2\)\(^{[HP90]}\). The required set of instructions is normally down-loaded to this control store at system startup time, but it is also possible to perform a replacement of the available functions within an active system.

The sequencer of these microprogram instructions has to support multi-way branches to allow algorithms an efficient differentiation of cases when manipulating tagged elements. As the inner loop of most algorithms will be a fast sequence of the steps *Read Element − Analyse − Manipulate − Advance to next Element*, a high data dependency can be expected. The latter three stages also contain multiple branches on different criteria. A pipelining of multiple internal processing steps is therefore not practical, since high data dependency and frequent, often unpredictable, branches restrict the efficient use of such an instruction pipeline\(^{[Hwa93]}\).

### 3.3.2 Activation

Whenever an external process intends to use a co-processor function it has to make sure that this function is currently available. It is therefore not advisable to have compilers and interpreters producing direct code activating the co-processor. A better activation method is to make the calling process raise a software exception which will then call the required co-processor function. If the requested function is not currently available, the related exception handler can decide whether to load a different set of microprograms into the co-processor, or to execute the required list manipulation in software. Whenever there is a change of functions available in the co-processors writable control store, there also has to be a change in the exception handlers dealing with the co-processor activation.

\(^2\)A Writable Control Store is a memory space designed to hold the microprogram; depending on the intended functionality of the processor different microprograms can be loaded into this memory.
3.3.3 Direct Communication Support

Since the SPRINT architecture is designed as a multiprocessor system and requires the transfer of list structures between different processor nodes, it was examined how a list manipulating co-processor could support this kind of communication. A very efficient node architecture would have the co-processor directly communicating with the SPRINT RELAY communications switch [Rou90], as shown in Figure 3.4.

![Diagram of node with one CP for direct communication](image)

**Figure 3.4: Node with one CP for direct communication**

In such a configuration, the instruction set of the co-processor would have to include two instructions SEND-LIST and RECEIVE-LIST to handle the transfer of data between the communication system and the List-Structured Memory. When a process needs to transmit a list, the CPU activates the co-processor which would then traverse the list structure and write the individual elements on the communications port connected to the communication switch. On an incoming message the communication switch would signal the co-processor, which then would arbitrate the buses to write the list into memory. However, since the same device handles incoming and outgoing messages this can cause complications. When there is a fixed priority between the SEND and the RECEIVE instruction, there is a possibility of deadlocks if two nodes try to communicate with each other and both either want to transmit or receive.
Chapter 3. Hardware Alternatives

Even a more sophisticated system with a pseudo-parallelism of sending and receiving is problematic, as encountering nested lists would endanger the consistency of the stack. This can only be avoided by duplicating the stack for sending and receiving of messages. This duplication of hardware resources, in combination with a complex control strategy to allow a bi-directional communication, suggests the need for two separate co-processors with different responsibilities.

In a configuration such as the one depicted in Figure 3.5, each co-processor will have an individual set of functions stored in the on-chip memory.

![Diagram of node with two CPs for direct communication](image)

Figure 3.5: Node with two CPs for direct communication

Calling processes or, more precisely, the exception handlers will have to know which device is responsible for each individual function. Since there has to be a hierarchy of bus arbitration between multiple co-processors, this aspect should be used to decide the allocation of specific algorithms.

Incoming messages can not be predicted by the system. Therefore the receiving function should reside in the co-processor with the highest priority. In the event of an incoming message this device will then be able to suspend functions on a lower priority device or the CPU and arbitrate the system buses. This strategy of giving the highest priority to the reception of messages is mainly intended to avoid congestion on the communication network.
3.3. Co-processor Design Considerations

The function that receives messages and creates new list structures will have a high demand for new pages. Consequently the functions responsible for memory management should be located in the same device. The co-processor with the lower priority can host regular list-manipulating algorithms in addition to the list sending function.

While the receiving co-processor is idle, functions executed on a low priority device will have to call the higher priority device to handle page management functions. This technique works well as long as the packets of incoming messages are arriving in a fast and steady flow. In case of longer breaks during the reception of a message the receiving co-processor has to be timed out, returning the bus access to some lower priority processes. However, these activities with a lower priority will not be able to use any page management functions, as the sleeping receive function blocks the necessary resources.
There are a number of ways to overcome this problem of the page management limitations, each of the following solutions has its individual advantages and disadvantages:

**Split Memory Management:** Each co-processor handles an individual set of free pages.

- Fast
- No problems with calls from lower priority devices
- Migration of free pages, as using and returning of free pages happens in different co-processors
- Expensive reorganisation of free page lists needed to neutralise the migration of free pages

**External Page Management:** A special purpose device serves all co-processors.

- No problems with calls from lower priority devices
- Additional hardware required
- All co-processors will have to request external page management. The necessary bus arbitration will slow down the system
- Subsections of functions dealing with memory management have to be implemented as non-interruptible sequences to ensure memory consistency

**Page Management in Memory:** The free page information is stored in a special location in the list memory.

- Easy to implement
- Slow
- Subsections of functions dealing with memory management have to be implemented as non-interruptible sequences to ensure memory consistency

**Passing Page Management:** The data required to perform page management functions is passed between devices when the system buses are arbitrated.

- No problems with calls from lower priority devices
- Bus arbitration and activity switching will be slowed down slightly
- Subsections of functions dealing with memory management have to be implemented as non-interruptible sequences to ensure memory consistency
3.3. Co-processor Design Considerations

In general there is no trivial way of direct co-processor support for communication purposes in the SPRINT architecture. To achieve a reasonable performance additional hardware would be required. Consequently, other approaches with additional hardware requirements should be evaluated as well.

3.3.4 Indirect Communication Support

The main alternative to the idea of direct list structure transmission is to convert the list-structure to be transmitted into a linear sequence of words inside the computing node and then transmit this block of words. The receiver then converts the linear information back into its original tree shaped structure.

Unless incoming and outgoing messages are buffered completely before or after conversion, this approach faces the same problems with deadlocks as the direct communication approach. Two buffers and a controller need to be placed between the system buses and the communication switch as in Figure 3.6.

![Figure 3.6: Node with CP and communication controller](image)

When sending a message the system first has to make sure there is sufficient space in the buffer to store the whole message. The buffer itself can be organised similar to a video-RAM so that it can only be written from the system bus side, while the communication switch can only read the words to be transmitted. For
Chapter 3. Hardware Alternatives

an incoming message the communication switch writes the arriving words into the
buffer and as soon as the message has arrived completely, the co-processor can be
instructed to re-build the list structure.

The advantages and disadvantages of such an configuration, supporting indirect
communication, are as follows:

+ needs only one co-processor
+ ensures that all data is available before transmission is attempted
— duplicates memory inefficiently
— Restricts messages to a maximum size

3.3.5 Enhanced Bus Systems

Since the memory access frequency of the list manipulation algorithms to be im­
plemented on the co-processor is high, any implementation will require permanent
memory access while operating. In any of the configurations suggested so far this
would require arbitrating the bus system and forcing the CPU to stay idle until the
co-processor has finished the assigned task.

A more complex node architecture using dual-ported memory (Figure 3.7) would
allow parallel memory manipulations to be performed by CPU and co-processor.
However such a configuration would again lead to problems related to the sharing of
the page management. The most efficient way of memory management is to maintain
the free page information in memory, where the CPU and co-processor can access it
all the time. A synchronisation mechanism will be necessary to guarantee memory
management operations of one processor not to be interrupted by the other.

3.4 Summary of possible Architectures

As the concept of Random Access List Structured Memory can not only be used
stand-alone, but also to enhance architectures with standard CPU and linear mem­
ory, the first section of this chapter described how to integrate the two concepts.
The basic principle of this idea is to hide the presence of the new memory area from the existing architecture. Whenever a program accesses a list structure inside the new memory, the CPU performs a regular data access. The value of the linear address, put on the address bus, indicates an access to the Random Access List Structured Memory. A controller, interfacing the traditional system buses with the new memory, will then take the pseudo linear address and extract the details of the required list access. The list-pointer and element selector will then be used to generate a physical address inside the added memory area.

A further analysis of the functions to be performed by such a controller, during a single read and write operations, suggests a basic structure for such a device. Furthermore it is explained how the controller could support more complex manipulations of the list structures held in the added memory.

When Random Access List Structured Memory is used as the sole memory concept, the whole architecture will operate on logical addresses, expressed in list-pointers and selectors, which will be converted into physical addresses inside the active (co-)processor. To build an architecture on top of Random Access List Structured Memory will impose a number of additional design issues. Some general
implications of such a memory architecture were introduced before the analysis of detailed aspects of a co-processor design.

A major decision for the design of a co-processor, performing manipulations on complex list structures, is related to the support of the large number of different manipulations. The integration of a Writable Control Store (WCS) to hold a flexible instruction set was presented as a common solution. The sequencing of microprogrammes and the co-processor activation are two further points that have to be considered.

Depending on the architecture of the system in which a co-processor is going to be used, the requirements will vary considerably. Using the environment of the SPRINT architecture and of different processing node architectures, it was demonstrated how a co-processor can be integrated to match different requirements. In the case of the SPRINT architecture, used for this study, the handling of inter-processor communication is a major issue related to the design of a co-processor. The basic decision is whether the device will handle the communication requests connected directly to the communication network, or whether an additional communications controller is responsible for transmitting and receiving the prepared list structures. An analysis of the different alternatives considered performance as well as hardware and software complexity.

Overall an indirect handling of the inter-processor communication appears to offer the better compromise, as the additional hardware and memory required for the communication handling seems to be justified by the simpler co-processor structure and the reduced complexity of the memory management.

The analysis in Chapter 2 presented fields in which acceleration hardware could support list manipulations in a Random Access List Structured Memory. Together with the hardware concepts suggested in this chapter, this enables a closer investigation of the related algorithms, as it is done in the next chapter.
3.5 Conclusions

This chapter investigated architectural alternatives using Random Access List Structured Memory in combination with a co-processor for complex list manipulations. A first concept describes the enhancement of existing architectures based on linear memory. However, this concept is not ideal and therefore the alternative concept is based on a CPU designed to support Random Access List Structured Memory. Within this second approach different architectural alternatives of handling the communication between processor nodes were discussed. A processor node with indirect communication support was identified as the better choice and was consequently used during the design of the co-processor introduced in the following chapters.
The development of powerful algorithms to be executed in a co-processor plays an important role in the design of an improved computer architecture. While the general structure of a specific algorithm will normally stay the same, some details will vary depending on the details of the hardware used.

This chapter will first analyse the requirements for algorithms to perform efficient memory management and error handling. After that a number of possible variations to basic algorithms will be presented. The last part relates the general considerations to a co-processor integrated into the SPRINT architecture.

### 4.1 Memory Management

As mentioned in Section 2.2, it is not practical for a list manipulating co-processor to interact with the operating system for memory management purposes, since the frequent interaction with the CPU and the related bus arbitration would slow down the system too much. Therefore the basic functionality for the memory management has to be implemented in internal hardware.

In case of the page-based tree structure used to realise the Random Access List Structured Memory, memory management is reduced to a process of keeping track of the use of pages. This means that whenever an algorithm requires an additional
page, the hardware has to provide the address of the free page to be used next. When a page is freed by an algorithm it has to be returned into the set of available pages. An effective method of offering this functionality is to arrange all free pages in a Free Page List.

4.1.1 Aspects of Free Page Lists

There are various possible structures for lists intended to hold the set of available pages. A linear list, where each page contains a pointer to the next free page, is a simple and economic solution.

Whenever a request for a page occurs, the following procedure is used:

1. Buffer the current pointer to the head of the free page list.
2. Read the address of the next free page out of the first free page.
3. Replace the pointer to the list head with this new value.
4. Return the buffered page address.

The corresponding procedure to add a returned page to the free page list:

1. Buffer the current pointer to the head of the free page list.
2. Use the address of the returned page as pointer to the first free page.
3. Write the buffered address of the previous list head into the appropriate word of the new list head.

It may appear strange to develop a memory architecture with complex list structures and then to implement the free page management using basic linked lists.
The following listing gives a number of facts explaining why it is reasonable to implement the free page management using linked lists:

- The standard functions of allocating and returning free pages need access only to the first element of the free page list.

- Depending on the amount of available pages in the system, a single list has to be able to hold this number of elements. Thus the page size, in memory words (\( psize \)), the number of levels in the tree structure (\( lvl \)) and the system memory size (\( msize \)), also in words, would be required to fulfill the following condition: \( psize^{lvl} \geq \frac{msize}{psize} \).

- Holding track of a linked list requires less effort than keeping all current pointers (top- and intermediate levels) of a complete tree structure, this would be necessary to perform fast manipulations on a tree structured free page list.

- The pages used to hold the intermediate pages of a free page list, during phases when there are plenty of free pages available, have to be allocated to processes demanding memory when the amount of free pages shrinks. Otherwise \( 1 + \sum_{i=2}^{n} \frac{msize}{psize} \) with \( n = \lfloor \log_{psize} msize \rfloor \) \(^1\) pages are permanently useless holding the internal structure of the free page list. As the amount of unproductive memory can be estimated as \( \sim \frac{1}{psize} \), in a system such as SPRINT with \( psize = 32 \) about 0.1% of the total memory space would be wasted.

- Using a tree-structured list to keep track of the free pages would require complex algorithms, as a simple solution can cause system confusion when a request for a memory management function needs to call a related memory management function itself. (e.g. A list-manipulating algorithm freeing a page can cause the memory management to request a new page to expand its internal structure and include the freed page. This growth of the free page list simultaneously creates a vacant entry, causing problems when finally trying to return the free page. This problem is demonstrated in Figures 4.1 (a) to (c) )

\(^1\)\( \lfloor x \rfloor \) is the largest natural number equal or less than \( x \)
Figure 4.1: Problem with a structured free page list
Other possible structures would require a static assignment of pages to hold the set of free pages. Such arrangements unnecessarily waste memory capacity and therefore are not appropriate.

Regardless of which kind of dynamic list structure is going to be used, the first action after a restart of a machine with any memory management system will be to divide the whole system memory in pages and include these into the free page list. When using a linked list structure it is necessary to have a mechanism for marking the last available page.

4.1.2 Page Demand and Availability

One important decision in the design of the memory management system is the attitude towards page demand calculation. As the size of a nested list structure is normally unknown, some functions (e.g. *list-copy*) can not predict the amount of pages required for successful termination. Based on this uncertainty the algorithm designer has to decide which strategy of memory management he is going to use in his algorithms.

During this study, three possible strategies were evaluated. The first two concepts are a ‘Trial and Error’ strategy, or a policy where an additional list traversal is performed to carry out a ‘Demand Prediction’ for each function. Once the page demand for a function is calculated, the system has to know the number of available pages to decide if the function could be executed successfully. The third strategy can be seen as a combination of the first two methods. In principle the algorithms operate on a ‘Trial and Error’ strategy, however, the system also supports a function to predict the page demand and an additional ‘Scan’ function to test for the availability of the required pages. This ‘Scan’ function does not extract the total number of free pages, it just tests the beginning of the free page list to ascertain the existence of a number of free pages equal to the demand of the intended list manipulation.

While a list structured free page list would always know about the number of elements, a simple linked list does not contain this value inside the list structure.
Therefore the memory management functions in an environment using 'Demand Prediction' in combination with a known number of free pages have to maintain the number of available pages in an alternative way.

A simple solution would be to make each page, within the linked list, hold the number of free pages as well as the pointer to the next free page. During the initial construction of the free page list, a counter is incremented for each page attached to the list and its value stored in each page. At runtime the memory management functions do not require any additional activities when delivering pages on request. When a page is returned to the free page list, the function will increment the number of free pages held in the old list head by one and store this new number of pages in the appropriate word of the returned page.

This method requires a considerable number of additional memory accesses to maintain and check the number of free pages inside the free page list. To sustain a high data throughput it is therefore better to keep the number of available pages in a single counter dedicated to this task. Within an optimised architecture, such a counter can be updated in parallel with the other activities performed during an update of the free page list. The disadvantages of such a counter are the additional hardware complexity and the necessity to pass the counter information in addition to the pointer whenever the memory management is handed over to a different hardware component.

Accurate 'Demand Prediction' is not always easy to achieve. For each level of nesting within the list structure to be manipulated there will be a certain amount of administrative data to be stored. The actual amount depends on various factors (number of lists involved in the function, level of the nested list, traversal strategy) and consequently will make it more difficult to calculate the exact number of pages required to hold this administrative information.
4.1. Memory Management

When implementing algorithms to calculate the page demand, the designer has to choose from the following options:

- Complex functions to monitor the exact number of words needed to store the administrative data, allowing the calculation of the maximum number of pages required. Together with the number of pages required for the list structure to be generated, an exact page demand can be predicted.

- Performing a worst case estimation, assuming a whole page (or several pages, if the maximum amount of information does not fit into one page) for each level of nesting. This method can cause the rejection of executable functions.

The 'Trial and Error' strategy in contrast with 'Demand Prediction' does not require any information about the number of available pages. All algorithms manipulate the data structures as usual and just perform additional actions to enable error recovery when a page request fails. The code segments handling error situations will only affect the size of the algorithms, but not the execution rate. There will be a small overhead within the code of each page allocation, to test for the availability of a free page.

To enable algorithms using the 'Trial and Error' strategy to recognise that the system has run out of pages, it is necessary to mark the last available page. This marking can be done in the following ways:

- Using the tag field of the word containing the address of the next page to differentiate between the last and normal pages;

- Having an additional bit which is not used for tag or address to indicate the last available page; and

- Setting up pointers to impossible or indicator addresses.

To enable the algorithms to perform a fast decision about the availability of additional free pages, this information should be available in a flag of the processor status word. When an algorithm reaches a point where a new page has to be allocated in the next instruction, these flags can be used to branch to the appropriate
### Table 4.1: Comparison: 'Trial and Error' versus 'Demand Prediction' and Combined Method

<table>
<thead>
<tr>
<th></th>
<th>'Trial and Error'</th>
<th>'Demand Prediction'</th>
<th>'Combination + Memory Scan'</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Required Support</strong></td>
<td>no</td>
<td>Number of free pages has to be known.</td>
<td>no</td>
</tr>
<tr>
<td><strong>Internal Overhead</strong></td>
<td>Each page allocation requires a test for the availability of another free page. In case of failure, the location where allocation failed is marked.</td>
<td>no</td>
<td>See 'Trial and Error'</td>
</tr>
<tr>
<td><strong>External Overhead</strong></td>
<td>no</td>
<td>Additional list traversal to calculate page demand.</td>
<td>Additional list traversal to calculate page demand. Scanning of the Free Page List to test for availability of free pages.</td>
</tr>
</tbody>
</table>

code section or an error handling routine in case there is no free page available. Otherwise the algorithm would have to perform an additional read access or test a data register to retrieve this information. Depending on the way of marking, the processor status word can be updated in parallel with any change of the free page register.
4.1.3 Garbage Collection Support

Garbage Collection is a problem faced by many programming languages dealing with list structures and multiple references to individual data structures [ASS85] [FH88]. Depending on the implementation of the Free Page List and how the number of available pages is checked, there are different options to support garbage collection mechanisms.

In systems where the number of free pages is known all the time, a garbage collection can be triggered as soon as the number of free pages drops below certain predefined levels. The alternatives for checking ‘water-marks’ of this kind are:

- A regular timer event causes a comparison of page number and trigger value;
- After the execution of certain list manipulations, consuming memory pages, the number of free pages is checked; and
- Special functions (e.g. a memory scan function as in section 4.1.2) consult the memory management system and check the number of free pages.

However, when a system does not monitor the number of free pages the only real indicator for the need of garbage collection is the failure of a function demanding memory. As some common garbage collection algorithms [Pra75] [FH88] will require a certain amount of temporary memory, this can cause problems as there will be no free pages available immediately after a function failed because of memory shortage. In this case it is therefore necessary to delay a run of the garbage collection until the pages already allocated to the newly created list structure are recovered after the error occurred. The freeing of pages utilised in the uncompleted list structure should then allow the garbage collection algorithm to make use of some memory pages and the co-processor facilities.

Setting aside large memory areas only for garbage collection purposes, as required by some algorithms, solves the problem of lacking memory, however the cost of additional memory can be considerable.
4.2 Error Handling

The 'Trial and Error' approach for page allocation is not the only situation where algorithms can encounter problems requiring the attention of the calling program. Frequently this kind of problem even requires early termination of the algorithm. Whenever such an error, requiring early algorithm termination, occurs, the requesting program has to be notified about the fact, and the reasons for the disruption should be accessible.

To allow an easy identification of problems encountered during the execution of the last algorithm executed, each algorithm should not only hand back the required return values, but also information about its success. Even so, this information about the problem that caused the termination of the function is normally not enough to enable a complete error recovery. In most cases it is necessary to know the location within the manipulated list structure where the problem was encountered. There are different ways to mark these locations in a way that the recovery function can recognise and handle in an appropriate way. Some simple marking methods include the use of illegal values or pointers. During the error recovery the function has to know which encoding was used to mark the location of the problem. The problem of memory shortage can be encountered in two different variations: shortage of free pages during allocation to the manipulated list, and memory shortage for the stack. Additional problem sources can appear in the form of undefined values in tag fields, corrupted list structures, etc.

4.3 Possible Variations

While the arrangements for memory management and error handling define a basic list of requirements for the development of algorithms, there are further points the algorithm designer has to consider.
4.3.1 Homogeneous Arrays

In applications where it is certain that all elements of a list have the same (not necessarily scalar) data-type, it is possible to use special algorithms adapted to the particular circumstances. The knowledge of the unique data-type allows the creation of smaller specialised functions, by leaving out the cases for alternative types. A typical example would be functions to do fast manipulations on dynamic integer arrays.

The primary area of application for this kind of algorithm is the implementation of functions only required for one particular data-type. As the testing of data-types normally does not cause any overhead, there should be no advantage in creating different functions for a range of types.

4.3.2 Page Allocation Strategy

Whenever a nested list is encountered in a hierarchical list structure, the standard way of creating the tree for this list is to allocate additional pages on demand. This means that whenever the last word of a page is reached, the next write access would cause a new page to be allocated at this level of the list structure.

An alternative approach would be to allocate pages for the whole tree as soon as the traversal algorithm proceeds into a new list. The advantage of this method is seen mainly with 'Trial and Error' memory management, as it will realise a page shortage earlier.

In addition to the savings in processing time before facing the memory shortage, this method will also ease the error recovery. Error recovery normally involves the deletion of the list sections already created. The simplified error recovery is based on the fact that whenever the algorithm faces a memory shortage during the memory allocation phase for a particular list, no actual user data will be entered into the list. Consequently, the error recovery algorithm does not have to check the individual data words of the uncompleted list. In contrast the method allocating pages on
demand has to check the data words, as the already written elements can contain sublists which have to be deleted.

Depending on the way lists are defined in the architecture and represented in hardware, the improved page allocation method is not always applicable. In an architecture with offsets for the first list element, the number of pages required for a new list will no longer only depend on the list length, but on the sum of list offset and list length. As Chapter 5 will show, the datapath of the suggested SPRINT co-processor is not designed to calculate these sums.

Error recovery is not supposed to be a major factor of the load profile of any target architecture. Therefore a modification of the system hardware to support the improved page allocation strategy is not justified.

4.4 Algorithms for a SPRINT Co-processor

Based on the previous considerations about memory management and error handling, a basic set of list manipulating functions was developed to verify the expected improvements of a list manipulating co-processor. To match the architectural constraints given by the SPRINT architecture, a number of adaptations of the basic ideas had to be made.

4.4.1 Special Aspects

The major limitation in the SPRINT architecture is the dual address space. This split of the address space into a 21-bit and a 24-bit address areas requires the system to handle two separate free page lists. For efficiency reasons each list structure, stored in the system memory, should contain as many pages in the extended address space as possible. However, the top-level page of each list has to be a page within the 21-bit address space. By using pages in the extended address space where possible, the best possible utilisation of the memory space can be achieved.
In case of a system using 'Demand Prediction' these page utilisation constraints would require complex algorithms, creating new list structures, to calculate two separate page demand figures for the two address areas. In combination with the overhead required to maintain the number of available pages in each address space, an implementation with 'Demand Prediction' appears not to be appropriate.

As the SPRINT architecture allows the user to define the function of the tags 'special list' and 'special atom', there is a basic requirement for a flexible set of co-processor functions and the possibility to define the behaviour of functions when processing list elements of these types. In the current experimental version these types are dealt as uninterpreted 36-bit data. Alternatives would be to include special algorithms dealing with these data-types or to produce an error message and terminate processing of the list structure.

Traversing list structures with the 'depth first traversal' strategy will create a stack containing information about where to resume the processing at higher levels of the structure. When a function fails to allocate a new page or runs into another serious problem, it executes a code sequence to report the problem and terminates returning an error message. During the error recovery process that follows, it is necessary to free not only all the pages used within the newly created list structure, but also all the pages holding stack information.

4.4.2 Implemented Functions

When the system operates in a 24-bit mode the co-processor is designed to use pages in the extended 24-bit address space where possible. In cases when the system runs out of pages in this memory section, the algorithms will try to use pages within the 21-bit address space instead. There will be different error codes to inform the system about the shortage of pages in the extended address space or the complete failure if there was no page available in either address space.

Since SPRINT is primarily designed to implement the Kernel system of the SPAN project [ERT88] [RTM88] with its concept of not having multiple references to the same list, most algorithms were designed with this background.
List Copying

The general process of copying a list requires the co-processor to generate a new list structure and then perform a copy operation for each element of the original list. There are multiple possible variations of this basic concept. Some possible variations are related to the following questions:

- Should the offset of the newly generated list be identical to the original list or is another value desirable?
- Is the new list to be of the same level as the original or should it be smaller or larger?
- Are all list elements copied or only a part of the list?
- Is copying of nested lists required?

While the first three options may appear to require different algorithms, it is possible to have a parameterised function offering full support. The instruction to duplicate a list structure will always contain information about the required level of the new list and its offset. In case of an identical copy, the calling process simply passes the information of the original list. An additional part of the instruction tells the co-processor whether to start with the first element of the source list and copy a number of elements equal to the length of the original list, or if there is further information about the start element and the number of elements to copy.

Whether nested lists are copied or not is linked closely to the general architecture of the computer system. In the SPRINT environment this recursive aspect of the list duplication algorithm will be always required when copying lists. A function supporting both versions would normally suffer a speed loss from an additional branch in the element testing phase. Alternatively additional hardware would be required for a more complex sequencing, combining the list copy aspect with the branch done on basis of the elements tag field. Since a separate function not supporting the recursive copying of lists will be considerably smaller than its recursive
counterpart, it is better to include two different copying algorithms in architectures requiring both versions of sublist treatment.

**List Merging**

When merging lists, elements of one list are concatenated to either head or tail of a second list. While list level and offset are given by the target list, there are some other points of variation:

- Are all list elements or only a part of the source list merged with the target list?
- Will the original source list be destroyed or preserved?
- When preserving the source list, do nested lists require duplication?

For the same reasons encountered in list copying, the specification of the SPRINT architecture requires an algorithm to duplicate nested lists. However, when the function destroys the source list while merging the lists, all references can simply be copied and no duplication is required. The information about merging parts of a list can be handled in the same way as for partial list copying. Destroying a complete list while only moving a part is not supported and has to be executed as a number of consecutive operations.

**I/O and Communication**

The transfer of list structures between multiple processing nodes and between such nodes and secondary storage media requires the conversion of lists into a sequences of words and vice versa.

The following functions were designed to convert lists between the two formats:

---

2In the following context this list is called the source list, while the term target list refers to the extended list.
Check Length permits the calculation of the number of elements in a nested list structure. This size might be required to ascertain sufficient buffer capacities for the flattened list during the transfers.

Flatten List performs a traversal of list structures and writes the encountered list elements into a non-paged buffer.

Build List reconstructs list structures from the data read in a linear buffer.

Memory Management

While the demand of new pages for list creation and extension is handled inside the corresponding functions, there are also functions just designed to maintain the free page lists:

Init-Reset informs the co-processor whether the system is operating in 24-bit mode. At the same time the addresses of the first free pages in the 21-bit and 24-bit address space are passed. As this instruction resets the processor status, it can be used to initialise the processor, or alternatively to re-transmit the free page information after the lists have been manipulated in a different location.

Free Page allows external sources to return a page to the free page list. When the system is in 24-bit mode, the co-processor tests the page address to decide to which free page list the page is to be added.

Get Page-21 and Get Page-24 supports external processes with a possibility to get hold of the addresses of the first free page in each list. This can be used to get a single free page as well as to take over the memory management.

4.5 Summary

This chapter investigated important issues related to the development of a co-processor instruction set. The major aspects were:
4.5. Summary

Memory Management It was identified earlier that there is a need for a basic functionality inside the co-processor, performing the intended list manipulations, to avoid expensive calls of operating system routines. The handling of free memory pages was identified as the main aspect of memory management. A linear list of free pages was suggested as simple concept to administrate the available memory pages. As the available memory capacity is finite, a shortage of free pages can conflict with the generation of new data structures. However, the size of complex data structures (e.g. nested lists) is not immediately known, therefore the algorithms operating on this kind of structure can handle the page demand in different ways. Three concepts were analysed and the 'Trial and Error' approach was chosen for the experimental co-processor because of its simplicity.

Error Handling During the manipulation of complex list structures a co-processor can encounter various error situations related to corrupted list structures as well as memory allocation problems. A uniform concept of error management was presented to inform the calling process of the encountered problem and to allow error recovery.

Based on the general options for the algorithm design discussed so far, an example instruction set for the SPRINT architecture is introduced. For each of the basic application areas, a number of aspects influencing the functionality of the algorithms were highlighted.
Chapter 5

Datapath Architecture

The statement "The process of designing a large-scale integrated system is sufficiently complex that only by adopting some type of regular, structured design methodology can one have hope that the resulting system will function correctly and not require a large number of redesign iterations." of Mead and Conway [MC80] is in many ways even more valid for modern VLSI designs.

A common way of structuring the design of microprocessors is to partition the processor into functional units. Normally this means to separate units performing data manipulations from their control. Within both components, datapath and controller, there will be further partitioning to break the design into individual sections of manageable size. As a result of this partitioning, datapath and controller can be designed quite independently and will be combined at a late design stage.

Since the design of most controller sections requires detailed knowledge of the sub-structures included in the datapath, it is logical that a design process of a new processor has to start with the specification of the datapath. Once a basic structure of the datapath is defined, the controller can be developed according its own specification and to the constraints given by the datapath.

The implementation of the datapath\(^1\) offers a range of design factors enabling the developer to meet the special requirements given in the processor specification. The

\(^1\) Also called operative part
realisation of these design factors depends on the architecture of individual sections in the datapath. Anceau [Anc86] mentions four basic components of a datapath:

- **Storage Elements** (with different access properties)
- **Operators** (ALU elements) performing data transformations
- **Connection Elements** (buses) for information transfer
- **Input/Output Elements** for external connections

Depending on the priorities (cost, speed, size,...) given for the microprocessor, the designer can also vary factors such as the width of individual components in the datapath and the amount of duplication. The number and size of storage elements offering read and write access (registers) has a strong influence on the performance that can be achieved in the datapath.

General purpose microprocessors normally have to be able to execute any given algorithm and therefore have to have a very general datapath. In contrast the datapath of specialised processors with a given set of algorithms can be tailored to perform these algorithms more effectively. One method used to improve the performance of a datapath is to implement a set of purpose bound registers instead of a number of general purpose registers.

Within the SPRINT architecture the availability of various data-types expressed by different tags and the fact that different data-types have individual binary formats limits the effectiveness that could be achieved by a datapath built out of general purpose registers. To enable manipulations on list structures it is necessary to analyse list elements and divide them into separate components. These components then might be modified or combined with other values to form new memory elements. In an effective design multiple data manipulations need to be executed in parallel to enable fast list processing.

The following sections 'Fast List Access', 'Register Placement', 'Stack Implementation', 'Bus structure' and 'Data Manipulation' will introduce general aspects
of the datapath design. The final section will apply these ideas and discuss the datapath of the co-processor in more detail.

5.1 Fast List Access

One important factor of fast list processing is fast list traversal which can be accelerated when addresses of intermediate pages are known at access time. Since there is a strong locality during a list traversal it is not necessary to hold a huge number of page addresses, only the page hierarchy for those lists being manipulated currently. This means a register pair of page-number and offset for each level in the tree structure will be required. The size of these registers is determined by the address range for the register holding the page number, and by the page size for the offset register.

For the SPRINT architecture with its page size of 32 words this means that offset registers will be 5 bits wide. As the architecture allows lists with up to 3 levels, there will be 3 of these offset registers and the corresponding page registers. The length of these page registers will be 16 bits for the top-level register and 19 bits for the second and third level, to support the possible address space of 21 and 24 bits respectively. Lists only using 1 or 2 levels of the tree structure are always held in the lowest possible page and offset registers with the motivation of using a homogeneous method for traversing lists. No matter how many levels a list structure has, the data values are always stored in the elements addressed by the lowest level page register and the related element register.

All operations traversing a list and manipulating list elements need to identify when to terminate. Since lists do not have an element marking the end of the list, the manipulating device has to perform other checks to identify the end of the currently processed list. This identification of a final list element could be done by comparing the values of the element registers for the pages holding the list with a known value. A simpler method is to count the number of manipulated elements and compare this value with the known length of the list. The test is easier when a counter is
initialised with the list length and decremented for each element manipulated, since reaching a negative counter value can be used to terminate the list manipulation.

For some instructions, such as copying and merging lists, it is necessary to keep track of two lists at the same time. As it is not realistic to swap the register values all the time, it is essential to have two sets of registers, one for the source-list and one for the target-list. In such an algorithm the counter for the list length of the target list is best set to the initial length (i.e. 0 for a newly created list) and then incremented for each new element. After completion of the algorithm the length register then will contain the length of the newly generated list.

5.2 Register Placement

As mentioned before, most registers in the datapath of a co-processor to operate on Random Access List Structured Memory will be purpose bound and will not require the whole memory width. To provide a high performance analysis and manipulation of data words, a high level of parallelism is required. One way of achieving this is to place individual co-processor registers in datapath bus locations matching with the bit positions of their data values in the system wide format for data words.

When this concept is used for a co-processor in the SPRINT environment, the format for list-pointers (Figure 1.5), the address format and other data representations (Figure 5.1) suggest to place the individual registers at the following bus positions.

<table>
<thead>
<tr>
<th>Unused</th>
<th>Tag(3)</th>
<th>Unused</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
<td>38</td>
<td>3635</td>
<td>32 31</td>
</tr>
</tbody>
</table>

Figure 5.1: Integer data format in SPRINT

The Page Offset Registers will always form the lower 5 bits of an address, and possible input values can be passed in the same position of a list-pointer. Therefore all Page Offset Registers are placed on the lower 5 bits of the datapath.
5.3. *Stack Implementation*

The 16 bit page number used in a SPRINT list-pointer is held in bits 5 to 20, therefore the top-level Page Number Register will also be located in this location. With the extended address space for the intermediate- and bottom-level pages, the Page Number Registers for these levels will stretch from bit 5 to 23.

The register holding the list length will be placed between bit 21 and 35, as in the list-pointer format. Since the tag field of each data word indicates how the data word has to be treated, the tag information will be needed for branching the control flow to the appropriate instructions. Therefore the section of the processor status word dealing with the tag field is placed in bits 36 to 38 of the processor status word.

5.3 Stack Implementation

As described in the section about list traversal (Section 2.2), a stack is required to enable the handling of nested list structures. Since there is no upper limit for the depth of tree structures, there can be no limit for the stack-size, consequently a dynamic sizeable stack is required. The standard way of keeping a stack would be to store all the pushed values in a separate memory area and have a pointer to the current stack top. However, this method would require a linear memory area, conflicting with the paged memory organisation of the system, and is therefore unsatisfactory.

The only purpose of the stack is to maintain list information of higher level lists during a recursive traversal of sublists. The complete stacked information will be manipulated according to 'Last In First Out' guidelines, therefore access will only be to the top most element on the stack.

5.3.1 Paged Model

Implementing a stack based on top of the paged memory organisation would allow a large temporary stack without the need for reserving a permanent memory area. A
basic way of implementing such a page-based stack is to design the stack of multiple pages of which one page is manipulated at a time. Whenever the current page is filled, a new page will be allocated to extend the stack. The single pages of such a stack just have to be connected as a linear list. As soon as the stack shrinks and the current page becomes empty, the previous page will be re-activated. A practical way of connecting the list of saved stack pages is to store a pointer to the previous page in the first element of the new page whenever a new page is allocated.

On-chip Stack

In an advanced version of this concept, the current stack page can be held in a set of registers inside the coprocessor. Instead of writing stacked values immediately to memory, they are stored in one of the registers. Only when all words of the on-chip stack are filled, the current values of the registers are moved to a page in memory, allowing the on-chip registers to be re-used. When the on-chip page of the stack runs empty, the previous page is read back from memory.

The first advantage of such an on-chip stack page lies in the fact that the writing of registers requires fewer bus resources and therefore allows additional instruction level parallelism during a stack access. A second advantage is that the memory access requirement of the co-processor is reduced. Within a fast co-processor, where the clocking is not adjusted to the memory response time, this can allow the co-processor to perform instruction sequences, including stack accesses, in a reduced time.

This method of an on-chip stack with pages migrating into memory when necessary can in a wider sense be compared with the idea of register windows [Hwa93] [PS81] used in a number modern RISC processors. Since the process of traversing lists does not pass any parameters on the stack, an overlapping of individual 'windows' is not necessary. This allows an on-chip stack of only one page with only a limited performance reduction.
5.3. **Stack Implementation**

The implementation of this kind of stack in the SPRINT environment requires a block of 32 registers of word-size (40 bit) and an additional counter (5 bit) to monitor the fullness of the on-chip-stack.

### 5.3.2 Element Selection

With a whole page of the stack implemented on-chip, each access to the stack requires a single word to be declared as target or source register for the data transfer. The word to be targeted in the stack depends on the level of fullness of the stack.

The stack pointer in such an on-chip stack will just consist of a counter with \( \log_2(\text{PAGE LENGTH}) \) bits, capable of selecting words in the current stack page. The remaining information about the stack is contained in the information setting up the linked list of stack pages. The counter implementing the stack pointer can, at the same time, be used to monitor the number of elements occupied in the current page and trigger the movement of pages to and from memory.

![Model of a paged stack](image)

Figure 5.2: Model of a paged stack

The stack pointer will have to be incremented or decremented during each push or pop access performed. Depending on the exact timing of these counter manip-
ulations the performance of system can vary. The possible alternatives and there consequences are:

**Post-Increment:** As soon as a value is written to the stack the fullness monitor is incremented.

+ Immediate access for push operations.
+ Allows fast operation when the increment can be done in parallel with push.
  - Pre-Decrement necessary before pop operations.
  —— Can cause unnecessary swapping of stack pages. Immediately after the last word of the stack is written, the increment will trigger the page to be written into memory, even if the following operation would shrink the stack again, requiring the read back of the page just written.

**Pre-Increment:** Before a value can be written to the stack the fullness monitor has to be incremented.

+ Immediate access for pop operations.
+ Allows fast operation when the decrement can be done in parallel with pop.
  - Pre-Increment and checking of the stack fullness slows down push operations.
  —— Can cause unnecessary swapping of stack pages. Immediately after the last word of the stack is read, the decrement will trigger the previous page to be read from memory, even if the following operation is going to grow the stack, causing the page to be written into memory again.

Since the number of push and pop operations is identical for any complete list traversal, the basic overhead for both methods is the same. But the consequences of the unnecessary stack swapping can cause slight performance differences. For the traversal of fairly simple list structures, where a single page stack can handle all
5.3. Stack Implementation

the data stacked during recursive list traversals, the chosen method should have no
influence. Similarly the traversal of deeper nested lists should show almost identical
performance with both methods, as it is always possible to generate particular lists,
where one system performs better than the other. As it is difficult to come to a
decision based on the performance, another factor has to be found.

Although the number of memory pages in a system is large, it will always be a
limiting factor in any system. Therefore an attempt should be made to keep the page
consumption of an algorithm to a minimum. When this guideline is used for the
decision process about the strategy for the updating of the stack fullness monitor,
the pre-increment strategy is to be preferred, as this method never consumes more
memory than the post-increment strategy.

5.3.3 Page Transfer

No matter which method is chosen to increment the stack pointer, the process of
moving stack pages to and from memory is always the same.

When the system recognises that the on-chip stack page is full, the algorithm
allocates an empty memory page and sequentially writes all stack words to the
corresponding locations in memory. During this process a counter is required to
select the individual words of the on-chip stack page and to generate the offset
addressing the corresponding word in memory. Since the stack pointer value has to
be reset anyway, this register can also be used as counter while moving the on-chip
stack to memory.

After the whole stack page has been written to memory, the page number of the
page containing the old stack information has to be stored in the first word of the
on-chip stack to allow retrieval of the old data. During the page retrieval the stack
pointer again can be used to address memory and on-chip stack words. As soon
as the whole page is read in, the memory page that used to hold the data can be
returned to the free page list.
An empty stack can be used as indicator to the system, signalling that the traversal of a nested list has completed and the control flow is back to the top-level list. This method can be used to avoid the overhead of algorithms keeping additional information about the level of nesting of lists in the current state.

To distinguish between an empty stack and a stack where just the last word of a current page has been read, a special indicator has to be used to indicate the first word of the initial stack page. This indicator will prevent any attempts to retrieve further pages into the on-chip stack once the stack is empty.

### 5.3.4 Stacked Information

So far only the management of the stack has been covered, but the information to be stacked is also important. The recursive approach of list traversal will immediately replace the information held about the current list when a nested list is encountered. Therefore any algorithm intending to operate on a nested list will have to save the pairs of page- and offset-register of all involved levels together with the length register. In addition to these registers it will also be necessary to stack certain list specific information held in the processor status word (e.g. *overflow bits for individual offset registers and level of the nested list*).

As the process of moving stack pages to and from memory is relatively expensive the algorithms should attempt to save stack space by keeping the number of stack words per recursion level small. Most of the information to be saved will be held in special purpose registers and will not occupy complete memory words, allowing multiple components to be written into a single stack word.

In the SPRINT architecture used for this example the information shown in Table 5.1 will required to save the context of a list:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page-Register</td>
<td>Holds the page number of the current list.</td>
</tr>
<tr>
<td>Offset-Register</td>
<td>Holds the offset within the page.</td>
</tr>
<tr>
<td>Length Register</td>
<td>Holds the length of the current list.</td>
</tr>
</tbody>
</table>

With an word width of 40 bits this allows to compress the information into 3 stack words. This amount of memory is of course doubled when the algorithm handles a source as well as a target list.
5.3. Stack Implementation

<table>
<thead>
<tr>
<th>Information</th>
<th>Number of Registers</th>
<th>Size [bit]</th>
<th>Space [bit]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Offset Register</td>
<td>3</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Top-level Page Register</td>
<td>1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Lower-level Page Registers</td>
<td>2</td>
<td>19</td>
<td>38</td>
</tr>
<tr>
<td>Length Register</td>
<td>1</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Status Information</td>
<td>1</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>94</strong></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: Stack demand for a SPRINT list

The arrangement of the individual registers into the stack words uses the natural placement of registers in the datapath to minimise the shifts required for compression. The first stack-word (Figure 5.3) holds the bottom-level page-number and offset.

```
Length Register  Bottom-Level Page Number  Level-3 Offset
38               24 23                   5 4 0
```

Figure 5.3: Bottom-level stack-word in SPRINT

As these values only occupy the lower 24 bits of the stack word, it is possible to add the information of the length register with the cost of shifting the contents up by 3 bits from the original register position (bits 21 to 35) to a location 24 to 38 in the stack word.

In exactly the same way, the next word (Figure 5.4) holds the information about the intermediate list level in bits 0 to 23; the remaining 16 bits are used to store the page number of the top-level page, which has to be shifted by 19 bits from its original location (bits 5 to 20).

```
Top-Level Page Number  Intermediate-Level Page Number  Level-2 Offset
39               24 23                    5 4 0
```

Figure 5.4: Second stack-word in SPRINT

The third word (Figure 5.5) has to hold the top-level offset and the required information from the processor status word. This status information is split into
two sections, with the tag bits describing the level of the saved list and additional flags to give detailed information about the current status of the traversal.

<table>
<thead>
<tr>
<th>Tag field (shows list-level)</th>
<th>Status Information (overflow flags,...)</th>
<th>Level-1 Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
<td>36</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.5: Third stack-word in SPRINT

5.3.5 Optional Stack Size Reduction

When using the method described so far, each nesting of a list causes the list information about all possible list level registers to be stacked, independent of the number currently used. Since smaller list structures will only use a subset of these registers, the other registers will not be overwritten, and therefore do not require saving. If at a later stage a larger nested list is encountered, it is then still possible to save the old values of higher level registers. The stack mechanism will always guarantee the appropriate values to be rewritten by the time the traversal returns to an earlier list.

Using this method, it is enough to save only the registers that will be overwritten by values from the nested list. In practice this may not always be possible, as the algorithms are already compressing multiple register values into individual stack words. As a result of this compression it can happen that the register values which have to be saved are split over several stack words which then all require saving. But the worst case still only requires all compressed stack words to be saved, as for a maximum level list.

When such a system of different saving strategies for different sizes of nested lists is implemented, the algorithms handling the recursive list traversal will become more complex, to distinguish the different saving strategies for different levels of nested lists. Therefore this method offers the designer the option of a reduced stack demand for the cost of larger algorithms handling the recursion.

In case of the SPRINT co-processor such a strategy is implemented. Whenever a nested list uses only one page (level 1 list), the algorithm can avoid saving the
second stack-word. All three components of the bottom-level stack-word (Figure 5.3 will be required. In addition certain parts of the processor status word have to be saved, consequently the compressed third stack-word is also required.

5.4 Bus Structure

To enable a proper transfer of data words or their components between different registers or to the hardware outside the processor the datapath has to offer an appropriate interconnection system. The most common way of implementing such an interconnection network is to use an internal system of one or more buses. Each bus is made up from a number of pathways connecting the individual bit planes of a number of registers. During each processor cycle a maximum of one register per bus broadcasts its value over the bus, while any number of registers is allowed to listen to the bus and read the data value. The number of buses and which components are connected to a particular bus depends on the individual architecture. By having more than one bus the architecture enables additional parallelism as multiple values can be broadcast on separate buses.

In contrast to most regular computer architectures, data words in an architecture with List Structured memory will normally consist of multiple components (e.g. a list-pointer consists of offset, page address, list length and tags). When such a data word is going to be manipulated on a datapath with purpose bound registers, individual components of the data word will have to be transferred to different registers. To perform such an operation on a bus system the source register will broadcast its value on all bus lines, but the various target registers will only receive the part intended for them.

In a similar way multiple registers can send on a number of disjoint sections of the bus during one cycle. With this technique a receiving register can construct new data words from components out of various source registers. Another advantage of such a bus system is that it can be possible to temporarily split a bus into several sub-buses gaining additional parallelism by the use of these sub-buses to connect different groups of registers.
Technical Aspects

There are two basic ways of implementing buses at the electrical level [Anc86] Tri-state and Precharged. Tri-state buses allow fast operation frequencies, but they require large output drivers in all connected registers to drive the capacity of the bus lines. Consequently this kind of bus is normally used for short but fast buses.

Precharged buses on the other hand operate on a two cycle clock, and allow larger but slightly slower buses. During the first phase, the precharging phase, the bus lines are driven to logic 1 and then isolated from the precharging unit. In the second phase data is transmitted and the sending register has just to discharge the bus line to send a logic 0.

Since a basic target for the design of co-processor algorithms is to have a high data transfer rate, the possible clock frequency of any device will be adjusted to the timing of the external memory. As the speed of external memory will cause the system to operate on a moderate clock rate, the frequency restrictions of a precharged bus will not handicap the system performance. In addition the precharge phase can be overlapped with processor internal tasks such as branching and instruction decoding.

5.5 Data Manipulation - Arithmetic and Shifts

Arithmetic

Normally the Arithmetic Logic Unit (ALU) is one of the major components of the datapath in any kind of processor manipulating data words. When analysing the arithmetic requirements of the list traversing algorithms, it is obvious that the only numeric manipulations are the incrementation and decrementation of a small number of known variables. Because of the purpose bound register set these increment and decrement operations on certain data values are also limited to the specific registers containing these values. By moving the facilities for incrementing and decrementing
particular data words into the registers assigned for these values, the architecture can enable multiple registers to be updated during one processor cycle.

An additional advantage of registers with integrated incrementing facilities are the reduced interconnection needs and the faster timing potential. In an architecture using a general ALU to increment or decrement individual register values, two buses would be required just to transport information to and from the ALU within one processor cycle. Depending on the alignment of the registers containing the values to be modified, the data transfer can also require shifting of the data values to match the lowest bit of the value to be modified with the lowest bit of the ALU, as shown in Figure 5.6.

Alternatively the position of the carry in the ALU, executing the increment or decrement, must be raised to fit the position of the data (Figure 5.7).

In the third and most complex variant the ALU could perform an addition instead of an increment, therefore requiring a second operand (Figure 5.8). This operand has to have the same bit alignment as the register to be incremented and have a value of 1.

A general ALU also requires the timing of an architecture with any bus technology to allow the clock frequency to cover twice the transfer time of the buses in addition to the computation time of a maximal ALU operation. In contrast an
architecture with a precharged bus and register internal incremeners can use the precharge phase of the buses to perform register incrementations.

**Shifting**

A second major data manipulation unit in standard processors is the shifter, performing variable changes to the bit alignment of register values. Like an ALU it is usually connected to two buses and performs a maximum of one shift per processor cycle. Since there is only a very limited number of known shifts required in a co-processor for list manipulations, a general purpose shifter can be replaced by additional interconnection facilities. For each shift operation required by a particular
algorithm during a data transfer either the source or the target register requires an alternative connection to the differently aligned bus lines (Figure 5.9).

Figure 5.9: Shifting by alternative interconnection

In an architecture using this shifting technique, multiple shifts can be performed during one cycle. The limiting factor for this kind of parallelism will be the transfer capacity of the regular bus system, as each shifted value will finally occupy a regular bus segment.

5.6 Details of the Co-processor Datapath

Based on the general points worked out so far, this section will give a detailed description of the components included in the datapath design for the co-processor to be integrated into the experimental SPRINT architecture.

5.6.1 List Structure Registers

At the core of the datapath are two groups of registers each responsible for the manipulation of data related to one list. As mentioned before, the duplication of these
registers is necessary to hold source and target-list of complex list manipulations in parallel. Each of the two register blocks will contain the following registers:

**Offset Registers**

**Purpose**: Hold the element offsets for the current page, one for each of the maximal 3 list levels.

**Length**: 5 bit

**Number**: 3

**Bus Position**: Bits 0 to 4 to allow direct use in address generation.

**Top-level Page-number Register**

**Purpose**: For the top-level page-number of a 3 level list.

**Length**: 16 bit

**Number**: 1

**Bus Position**: Bits 5 to 20 to allow direct use in address generation in the limited 21 bit address space.

**Additional Page-number Register**

**Purpose**: Intermediate- and bottom-level pages in 3 level list, or all pages for smaller lists.

**Length**: 19 bit

**Number**: 2

**Bus Position**: Bits 5 to 23 to allow direct use in address generation in the full 24 bit address space.

**Length Register**

**Purpose**: Monitors number of list elements to be processed.

**Length**: 15 bit

**Number**: 1

**Bus Position**: Bits 21 to 35 to match list length in SPRINT list-pointers.
5.6. Details of the Co-processor Datapath

As the datapath is designed without a general purpose ALU, the page offset and the length registers have integrated increment/decrement units which can be operated in parallel. All 6 page offset registers report a counter overflow to dedicated bits in the processor status word. Since all algorithms will initialise the source length register with the length of the list to be manipulated and then decrement the register for each list element manipulated, an underflow of this register will indicate the completion of a (sub-)list traversal. Therefore the underflow of this source length register is also monitored and reported to a list completion bit of the processor status word.

5.6.2 Stack

The on-chip stack is realised by a block of 32 registers of 40 bits each. A 5 bit counter with increment/decrement facilities acts as stack fullness monitor and also provides its value to a 5:32 decoder selecting the current top of stack word. During the transfer of stack pages to and from memory, this counter will also be used to address the individual stack words involved in the transfer as discussed in the section about general stack mechanisms. The register block itself is best implemented as a small RAM area with one set of drivers attached to the bus system as shown in Figure 5.10.

5.6.3 Buses

The interconnection between the registers introduced so far and all other registers in the datapath is realised by two 40 bit wide precharged buses. Both buses can be separated into a number of small segments allowing parallel transfers with different source and target registers to be performed on one bus, as long as the bus segments are disjoint. Depending on their purpose and the required interactions with others, individual registers are connected to one or both buses. There are even registers with a different bus connectivity for read and write access.

The decision for the use of a two bus architecture is based on the aim of a high data throughput. Two processor internal buses allow parallel interactions with the
Chapter 5. Datapath Architecture

Figure 5.10: Suggested stack implementation

external data- and address buses, which enables memory accesses together with the address generation and first data analysis to be executed in one processor cycle. Any additional buses would of course increase the potential parallelism of data transactions during one cycle; but as most operational steps will be dealing with only address and the corresponding data value it is unlikely that most algorithms could use this potential. As the processor does not include an ALU executing binary operations there is no need for a three bus system in which two buses are used to supply the operands while the third bus returns the result to a register; for the same reason there is no result transfer phase during the co-processor basic cycle.

To interface with the external address and data buses, two latches are included in the datapath. The 24 bit wide address latch is only connected to the first internal bus and drives the external address bus with values concatenated from the components send by a number of registers. The bidirectional interface between the external data bus and the second internal bus is implemented through the 40 bit wide data latch.

There exists a common requirement of several algorithms to initialise certain registers with particular values or to use certain bit patterns in newly generated addresses or data values. To support these needs two abstract registers, the zero and the one register, are included in the datapath and provide logic 0's and 1's
on various sections of both buses. Due to the precharging technology used for the bus system, there is no need for a 1's register. The capacity of any bus line will supply a logic 1 as long as no other register output pulls down the bus line. The implementation of the zero register is made up by two groups of transistors allowing individual sections of the two buses to be pulled down.

The shift operations necessary to position the stack information into the given format (Figure 5.3 to 5.5) are implemented by additional shifting bus connections as described in the previous section. The shifting of the registers holding the length and the top-level page number is performed by additional bus connections of the registers, as this allows the use of a single bus to access the corresponding stack words. When the shifting facilities would be installed on the side of the stack block, the individual components of the manipulated stack word would have to be transferred on separate buses to avoid the overlapping. In addition this method would also mean a concentration of shifting facilities in one location, complicating the physical design.

5.6.4 Memory Management

In order to hold a pointer to a free page list and implement a management of free pages efficiently, it is necessary to include a dedicated register into the datapath. In case of a co-processor for the SPRINT architecture there even have to be two of these registers to hold the pointers for the first free page in both the 21 bit and the 24 bit address space.

Each register will contain the address of the first free page in the associated address space. Any element of the free page list will also indicate whether there is another free page and if so, hold a pointer to this page. As discussed in the section about general memory management, the information about the existence of a further free page will be transferred into the processor status word whenever a page is consumed. In the co-processor introduced here, the information is represented in an additional bit (represented by the 'F' in Figure 5.11 and 5.12) which is transferred into a corresponding flag of the processor status word.
Chapter 5. Datapath Architecture

The bus architecture of the co-processor allows writing the address of a free page to a register and at the same time to the address latch to read the first element of the free page list to determine the address of the next page whenever a free page is required. To enable this kind of operation the registers holding the free page pointers have to be designed in a way that allows writing of an old and reading in the new value in one cycle without problems. A two stage storage element such as a Master Slave Flip-Flop can be used to implement this functionality. During the regular bus transactions both stages can interact with the other registers, while the precharge phase is used to transfer the value from the first stage into the second storage element.

Alternatively the timing of the memory access during the operative bus cycle has to changed in a way that allows the address latch to store the memory address in an early stage of the cycle, and then ignore the changes on the bus when the new value is written into the free page register.

5.6.5 Processor Status Word

This common component of the datapath represents an important interface with the control part of a processor. During the execution of an algorithm individual instructions produce results which can cause modifications of individual flags of the Processor Status Word (PSW). These flags then can be used by the control part to decide on how to continue the execution of an algorithm. The standard way of these decisions is to have binary branches with a condition fixed to a particular flag of the PSW. In most standard processors the PSW is modified by results of arithmetic operations performed in the ALU. In addition to the basic flags (overflow, zero,
negative and carry) of a condition code register many CPUs have got a system area as second component of the PSW. The flags in such a system area are used for purposes such as interrupt masking and processor mode switching [HN84].

In the given co-processor due to a lack of an ALU the the PSW flags can not be derived from a central datapath component. Instead the various incrementing and decrementing units integrated into the various registers directly report important abnormalities to dedicated flags in the PSW.

The list structured memory structure also requires the differentiation of a multitude of special situations. Table 5.2 gives an overview of the flags implemented in the co-processor.

The differentiation of a 21 bit and a 24 bit mode is implemented to improve the support of systems only using a reduced address space with no more the 21 bits. The flag is only set once during the initialisation of the co-processor but will be used in all decisions related to memory management.

In contrast to all other flags, bit 39 is not bound to a special function and can be used for any branch not supported by the other flags. Unlike the purpose bound flags, its value is not permanently guaranteed and therefore has to be generated directly in advance of any branch.

### 5.6.6 Communication

The co-processor introduced in this chapter is designed to support inter node communication via a separate controller and control memory as shown in Figure 3.6. This support requires the existence of algorithms to convert a list structure into a flat representation which can be transmitted via the communication network. On the receiving side a related algorithm will be necessary to regenerate the list structure.

As the control memory will be limited, a security mechanism should guarantee sufficient buffer space before a list is flattened and written into the control memory. As the size of a nested list structure can not be predicted, a third algorithm will
be required to calculate the memory demand for a list structure. In principle the complete list structured memory can be occupied by a single list, therefore a 24 bit counter will be essential to calculate the number of elements in a list structure. During the process of flattening or re-building of the list structure, the same counter can be used to generate the addresses inside the control memory. This of course requires an additional signal to allow the distinction of addresses in the list structured and the communication control memory.
5.6.7 Buffers

In addition to the registers bound to a particular variable introduced so far, some algorithms also require some temporary buffers. In order to support these requirements two additional buffer registers are included in the datapath. A 40 bit wide register is intended as a general purpose buffer for data words. The bus connections of this register allow algorithms to extract the individual components of the SPRINT data formats. In the same way new data words can be concatenated in this register.

A second register with a width of 24 bit is supported to buffer addresses. This buffering of address is essential for some algorithms when internal memory management functions modify the free page lists but the old value will still be required for some other data manipulations.

Figure 5.13 gives an overview of the functional parts of the datapath explained so far.

5.6.8 SPRINT Interface

The intention to allow the co-processor to execute a variable set of algorithms out of a large set of possible instructions together with the amount of information to go with an instruction activation suggests to integrate the co-processor as memory-mapped device into the SPRINT architecture. Using this technology the SPRINT CPU activates the co-processor by writing the instruction and related parameters into specific memory locations. In reality these memory locations are registers inside the datapath of the co-processor. After the co-processor has completed an instruction and the control has been returned, the CPU will read the return values from special registers of the co-processors which again represent normal memory locations for the CPU.

Depending on the instruction a number of different parameters will be required by the co-processor. The following registers can be used to pass information to the co-processor algorithms:
**Instruction**: This register is used to inform the co-processor of the instruction it is supposed to execute.

It contains four separate sections, two of them are supplied to the controller and tell the op-code of the instruction and an indicator telling whether to traverse the whole list or only a certain part.

The other two sections normally contain the required offset (bits 0 to 4) and level (bits 36 to 38) of the list to be generated.

**Input_1**: This register is used for multiple purposes. The normal function is to pass the first list-pointer for all instructions including a list-traversal.

During the initial co-processor setup it will contain the address of the first free 21 bit page, together with the flag indicating if there is such a free page available, another bit indicates if the co-processor is going to operate 24 bit mode or not.

Within the memory management function used to add a page to the free-page list, this register will keep the address of the page to set free.
5.6. Details of the Co-processor Datapath

**Input.2**: Again this register can be used for a number of different purposes. Whenever an instruction requires a second list-pointer, it will be supplied in this register.

For instructions such as list deletion or list initialisation, this register will contain the word used to replace the current values. The address of the first free 24bit page and a bit indicating if such a page is available are also supplied in this register during the initial co-processor setup.

**Param.1**: Whenever a list is not to be traversed as a whole, this register holds the modified values for the offset registers at all three list levels as well as the modified length value.

**Param.2**: Has got similar contents as the previous register, which are used to set up the offset of a list structure to prepare a tail-append operation.

Certain instructions will require the co-processor to traverse only a part of a given list. As indicated in the description of the instruction register, the co-processor supports this by modifying the control flow of the algorithms with an additional flag.

The use of this technique will not be needed when an algorithm only requires the traversal of the opening elements of a list. In this case it will be enough to modify the list length encoded in the list-pointer, as this value will be decremented to identify the final element of a list traversal. Thus the list-pointer should contain the number of list elements to manipulate instead of the real list length.

When the required traversal needs to omit leading elements of the list, it will be necessary to modify the offset registers to start the list traversal with the requested element. As the datapath does not contain an ALU, it is not possible to calculate new values for the offset registers inside the co-processor. Consequently the new values for these registers will have to be calculated in the CPU and supplied to the co-processor as parameters as it is done in Param.1 and Param.2. In addition it is again necessary to supply the number of elements to be traversed in a modified value of the length register, which this time can be supplied inside the same parameter word.
The structure used by Param_1 and Param_2 to hold the modified register values is given in Figure 5.14.

<table>
<thead>
<tr>
<th>Length</th>
<th>Level_1</th>
<th>Level_2</th>
<th>Level_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>14</td>
<td>109</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 5.14: Parameter word format

Because of this structure two additional shifting connections will be required in the datapath, to move the first and second level offset into the registers aligned on bits 0 to 4. To enable the data transfer to be executed on a single bus, the shifting facilities will be located at the reading ports of the target registers.

Two further registers are used to return results and error status to the CPU. The Output register holds either the resulting list-pointer of any list-manipulating instruction or the address of a requested free page. Each co-processor instruction will report the success status to the CPU. The Error register is originally initialised with zero, whenever an error occurs during the execution of an instruction, the error handling part of the algorithm will replace this zero with a unique value identifying the kind of error.

5.7 Summary

This chapter explained the special requirements to the datapath faced when building a co-processor to manipulate complex list structures inside the SPRINT architecture. As a fast list traversal is essential for a fast manipulation of the list structures, it was explained how holding the addresses of the page hierarchy together with the page offsets can speed up the list access. It was also explained how decrementing an element counter can be used to determine the completion of a list traversal, and why, for some functions, it is essential to keep two sets of these registers, one each for source and target list. Due to the SPRINT memory architecture and the format of data words, the values required to hold the tree structures of lists, do not occupy the complete 40 bits of word width. By aligning registers inside the
5.7. Summary

datapath according to the bit positions of corresponding values in the SPRINT data words, a parallel data transfer of multiple components can be used to accelerate the transactions performed on the datapath. To enable the traversal of nested list structures, a stack mechanism will be necessary. It was explained how such a stack can be implemented, using the underlying paged memory. The idea is to split the stack into individual pages and connect these pages to a linked list. To reduce the number of memory accesses is to hold the current page of stack words on-chip. The co-processor algorithms will only access the element being on top of the current stack, therefore the stack pointer needs only to cover the number of elements in the current stack page. When this page is implemented on-chip, the counter used to realize the stack pointer can also be used as loop counter when the current stack page has to moved to or read from memory. The next aspect of the stack implementation was related to the contents of the stacked words. It was shown which registers have to be saved during the traversal of a nested list, and how multiple values can be compressed into a single stack word. By using this form of compression the number of stack words per recursion can be reduced, limiting the overheads related to the replacement of stack pages.

A consideration of the communication requirements inside the co-processor suggests to connect the individual registers by a precharged bus system. By allowing the buses to be split into sub-sections, parallel transfers between multiple registers can be performed on disjoint sub-sections. As there are no complex numeric calculations required by the list manipulation algorithms there is not particular need to integrate an ALU into the datapath. The only required calculations are incrementation and decrementation of various counter. By integrating appropriate functional units into the registers, it will be possible to execute multiple calculations in parallel, resulting in faster algorithms. The simplified manipulation of registers not aligned with the LSB\(^2\) of the datapath was demonstrated as another major advantage of this concept. The demand for shift operations included in the list manipulation algorithms will be limited to a few known shifts. Including a compete shifter unit into the datapath is therefore not appropriate. The few necessary shifts can be accomplished

\(^2\)Least Significant Bit
by additional connections of source or target register to the alternative bus lines. Instead of shifting the required register values, the data is written to bus lines with the required offset.

In the final section the general concepts and ideas introduced so far were applied to design a datapath for the SPRINT co-processor. The explanation of the registers holding the list structures and the on-chip stack was followed by details about the bus structure and the registers supporting the required memory management functions. The structure of the Processor Status Word (PSW) was explained next, to ease the understanding of the branching support for the list manipulation algorithms. Additional registers were required to support communication functions and to buffer temporary data values inside the co-processor. Finally it was explained how a group of memory mapped registers interface the co-processor with the CPU.
Chapter 6

Development of the Co-processor

This chapter intends to give an overview of the various steps performed during the development of the co-processor. It represents an overview of the practical work carried out during the whole period of the project, and therefore interleaves with the theoretic aspects presented in earlier chapters.

While a major aim of the project was to analyse the architectural aspects of general hardware devices supporting systems based on Random Access List Structured Memory, the practical aspect always focussed on the design of a co-processor to enhance the experimental SPRINT architecture.

At the start of this project a detailed specification of the co-processor design targets (such as instruction set, structure of the datapath, etc.) had not been done. The only limitations in this project were given by the existing architecture of the List Structured Memory and the functionality of the SPRINT CPU. With this background, a Top-Down design strategy with iterated theoretical and practical steps was chosen to develop the specification of the co-processor step by step.

Most of the more theoretic constraints presented in earlier chapters were developed during this iterated process of specification and implementation of the co-processor architecture.

After analysing the general design options at a particular stage of the design and deciding for a promising solution, it was possible to consider the consequences of this
choice on the next design step of this special co-processor. Simulating algorithms and hardware at the various abstraction levels helped to identify problems and possible complications requiring further examination.

The first step in the process of developing a specification for a general co-processor architecture was to identify the application areas in which additional hardware could offer major improvements to the performance of an architecture. As indicated in Chapter 2 the acceleration of complex list manipulations involving a list traversal is one of the more promising areas.

The first section of this chapter describes the principles and results of a high-level simulation, used to develop the basic form of the list traversal algorithm and to test a number of algorithms performing basic manipulations of nested list structures. The second section then introduces the basic concept of the database method used to manage the development of the microprogram implementing the co-processor instruction set. In the following two sections it is described how these algorithms were verified, what results the simulation of these algorithms produced at lower design levels, and how these results were used in the process of designing the controller structure of the list manipulating co-processor.

## 6.1 High-level Simulation

Defining complex list manipulations as the main area of application for the co-processor represented the first step in the development of a specification for such a device. Identifying the central role of the list traversal in these list manipulations can be seen as the next stage in setting up the processor specification. To continue this process, it was necessary to perform a further analysis of both list manipulation algorithms in general and the common list traversal in particular. The study of list manipulating algorithms had to include three major considerations.

- List traversal was known to be a common element of many proposed co-processor instructions. A first consideration was the identification of further
correlations between individual algorithms. This was particularly important as a close correlation between algorithms generally suggests a joint use of resources. Such a joint use of resources, furthermore, promises a better utilisation of the corresponding processor hardware.

- Understanding the structure of algorithms is essential to the creation of a powerful hardware implementation using minimal resources. In particular the patterns of the control flow and the potential parallelism of frequently used components, such as the list traversal, can be used to develop an optimised hardware structure. A second consideration therefore concentrated on analysing the internal structure of list manipulation algorithms.

- A third consideration focussed on analysing the use of variables in list manipulating algorithms. This was done in order to find a basic mapping of algorithms on to a register set for the co-processor.

Simulating the list manipulating algorithms at a high abstraction level offered a convenient way to investigate all three aspects and to expand the specification of the co-processor.

Starting with the basic concept for a list traversal, as shown in Figure 2.2, a systematic simulation environment was developed, using the programming language 'C' [KR88]. This simulation environment was then applied to gain a better understanding of effective list manipulation algorithms. The implementation of individual algorithm aspects in this simulation was based on results described in earlier sections.

The theoretical analysis of the list traversal, as presented in Chapter 2, had shown that a Depth First Traversal (DFT) promises better results than the Layer First Traversal (LFT). In the same way Section 5.3 suggested the implementation of a stack built of individual pages to enable a fast and simple stack mechanism handling nested list structures.

The following description of the high level simulation covers the most important aspects of list traversal and memory management.
6.1.1 Simulated List Traversal

In order to simulate the traversal of list structures on an architecture with linear memory, it is obviously necessary to generate multiple data structures representing the data formats of the target architecture. This modelling requirement originates from the fact that the traversal algorithm needs to analyse and generate data words encoded in the word format of the simulated architecture.

The data structures used to implement the high level simulator are described in Appendix B. Modelling the data words of the SPRINT architecture required the development of a complex hierarchy of data structures. Starting with basic types, such as the 5 bit page offset, more complex data structures were defined as records of individual basic data types. SPRINT, as a tagged architecture, uses its tags to differentiate between multiple interpretations of bit groups inside the data word format. The data structures used by the simulator had to make use of 'union' data types to achieve the same functionality.

Due to the variety of data formats in the target architecture, and the data types used to describe the manipulated values in the simulation, it was necessary to include typed variables in the simulation environment, where each type of variable holds data words of one particular format. Together with these typed variables the simulation environment also included a special set of procedures, designed to manipulate these variables according to certain rules. These manipulation rules express the available operations for the corresponding data values in the simulated architecture. Appendix B contains details about the types of variables used in the simulation environment, as well as the related manipulation routines.

6.1.2 Simulated Memory Management

Since the simulator normally runs on a hardware with a regular memory architecture the program has to simulate the memory architecture of the List Structured Memory. As the simulation environment already models a paged memory, basic memory
management can be easily implemented. Instead of having a real free page list, the simulator uses the dynamic memory allocation facilities of the host architecture. Whenever a simulated function requires an additional page, the simulator allocates a memory area of sufficient size in the heap memory area. The routines used to do this memory allocation just have to make sure that the addresses of new pages fulfill the conditions given by the description of the co-processor. These conditions are: addresses only use 21 or 24 bits and the lowest 5 bits of each page address are zeros.

The basic memory management functionality is implemented with three functions. The first function is responsible for the generation of new 21 bit pages, and represents the consumption of pages out of the free page list for this memory area. A second, almost identical, function is responsible for the generation of pages in the extended address space, using 24 address bits. The third function frees these memory areas, once the simulated system returns a page to the free page lists.

In addition the simulator supports individual data read and write accesses to the memory. Since the forty bit words of SPRINT are modelled in complex data structures, there is no easy way of assigning words held in a memory cell to registers in the datapath and vice versa. Two functions to read and write memory words accomplish this functionality by copying\(^1\) memory areas between source and target of a memory access. The size of the memory area to be copied equals the size occupied by the data structure holding one co-processor word.

**Efficiency and Limitations**

Due to the overhead related to the complex structure of the data representation in the simulator, the amount of memory required for one page will be much bigger than the original 1280 bits in SPRINT.

By implementing the List Structured Memory on top of the dynamic memory of the host hardware, it is also difficult to simulate the behaviour of algorithms when

---

\(^1\)Using the UNIX memcpy routine
facing the lack of additional pages. This is based on the fact that there is no easy way of limiting the amount of heap memory available for the generation of memory pages used by the simulator. If required, this problem is overcome by restricting the simulator to a fixed number of pages. In this case the simulator has to be equipped with a separate routine performing an internal administration of these pages.

6.1.3 Simulation Outcome

Implementing the basic concept of list manipulating algorithms in a high level programming-language and simulating the functionality of the main algorithms enables the designer to identify needs and potentials of a list manipulating co-processor. The use of various, specialised data formats and their manipulation, involved in the simulation of SPRINT algorithms, required the use of variables with specific formats and corresponding manipulation procedures in the simulator. The variables used in a high level simulation tend to relate closely to the registers of the datapath executing the same algorithms in hardware. This and further reasons mentioned below, made it appropriate to generate a datapath consisting of purpose-bound registers identified during the high level simulation of the list manipulation algorithms. Each variable required in an algorithm of the high level simulation represents the need for a register in the corresponding datapath. However, it is not necessary to include a separate register for each variable, as some variables of the same format can under certain circumstances share a register. In particular this method of sharing registers can be used for variables with short non-overlapping life times.

A second major finding is the fact that the list manipulation algorithms contain a high percentage of data assignments with a low sequential dependency. This results from the fact that the algorithms have to divide data words into separate components, and then combine these components with other values to form new data words, on a frequent basis. While the hardware hosting the simulator required the individual assignments to be performed sequentially, the low sequential dependency offered potential for parallel transactions. The first step in making use of this parallelism was to design the co-processor datapath in a way that allows parallel data
transfers, and with register ports offering the necessary flexibility to support data transfers on subsections of the bus system. A bus system as suggested in Section 5.4 where a single bus can be split up into a number of sections, connecting different registers, enables the architecture to make use of the parallel potential within the algorithms.

### 6.2 Design Methodology

The high level simulation described in the previous section helped to analyse and specify the basic structure of the datapath, covering issues as which registers should be included and their communication requirements. However, aspects such as the positioning of registers, that do not use the full width of the datapath, and their detailed connectivity were not covered. There is also no information about time aspects of individual data transfers included in the high level simulation.

While the basic ideas for the register placement were already discussed in Section 5.2, the detailed connectivity of the individual registers can best be determined by analysing a detailed description of the algorithms. To add more of this connectivity information to the specification, the next design step dealt with the generation of a more detailed formal description of the algorithms. In Section 3.3 it was indicated that a micro-programmable processor is the most common method of supporting a flexible instruction set as required by the architectural details of SPRINT. As the support of the user defined data types represents an important aspect of the expected functionality, a micro-programmable instruction set was included into the co-processor specification.

With this fundamental decision about the structure of the controller, the next step of the design was to generate a first formal description of the various algorithms. At the same time this description was to provide a further step in the process of defining the microcode and the specification of controller details.

The rest of this section will first analyse existing microcode generation methods and describe the difficulties faced, when these methods are applied on list manip-
ulation algorithms. This analysis is followed by the introduction of an alternative microcode design method, developed in this project.

6.2.1 Microcode Generation

The process of generating a controller for a particular microprocessor requires the designer to state all the algorithms in a special description language that is capable of expressing all the transactions performed in the datapath. For each processor cycle all the necessary data transfers and operations have to be described. It is also necessary to record the sequencing of the cycles and their I/O behaviour.

There exist a number of methods from manual flowcharting [Tre87] to microcode compilers, at various abstraction levels [Mil89], supporting these needs. However some factors such as the labour incurred by a method, the tool availability or the tool development overhead limit the practicality of these methods for specific developments.

In case of the algorithms for a SPRINT co-processor a consideration of common methods comes to the following conclusions:

Manual Flowcharting is a proven and flexible methodology, which can be used for all kinds of processor designs. However, maintaining consistent flowcharts for instructions with several hundred cycles over a number of changes can be very difficult, time consuming and error-prone. In addition there is still the task of testing the instructions and transforming them into a final hardware product.

Microcode Assemblers convert algorithms given in a low level abstract notation into binary patterns of a supplied controller structure. A major problem with such an approach is that assemblers translate a sequence of statements, consequently such a tool would be required to express all demanded parallelism in the input language. In a complex design with extensive parallelism at cycle level it is hard to justify the development of such a language, as it is likely
to be quite large. Creating a translation for each instruction of such a language into the corresponding binary pattern requires a considerable amount of time. In addition there is a severe overhead for each change of the target controller structure. Microcode assemblers [BB87] are a common tool to generate microcode for systems with a limited datapath complexity, such as bit-slice architectures.

**Microcode Compilers** allow the input of algorithms in a High Level Language (HLL), which is then transformed into a low level representation that can be translated by a microcode assembler. The use of a HLL to enter the algorithms normally eases this process by supporting complex control sequences (loops, if-then-else,...) and variables. The compiler will also perform data dependency analysis and optimise programs by re-scheduling certain instructions. In the case of the particular co-processor these features were not particularly important, as the algorithms have a high branching frequency in combination with a high data dependency, which does not allow much optimisation by re-scheduling. The fact that the processor is based on a set of purpose bound registers precludes optimisation with sophisticated register assignment methods.

In general the development of a processor specific microcode assembler and compiler is very time consuming. Even if the microcode compiler can be created by modifying an existing version to the new architecture, the development time is not appropriate for a small scale development, or where re-usability is not expected.

**Retargetable Code Generators** not only use a HLL to describe the algorithms, but also allow the user to describe the datapath in a Computer Hardware Description Languages (CHDL) [BMPS91] [Mer93]. Based on this hardware description the system will automatically generate a microcode assembler for the given architecture. While this approach works fine for 'standard' architectures with multiple general purpose registers and arithmetic data manipulation facilities, the embedded Computer Hardware Description Languages are not always capable of expressing the hardware used in a datapath. In addition to
this, the fact that the automatic creation of the microcode assembler results in a severe limitation of the controller structure made these tools inappropriate.

Implications of this Analysis

As none of the microcode development methods discussed so far appeared to be particularly suitable for this project, it was necessary to find an alternative methodology. During this process individual concepts used in the standard approaches were considered for their suitability in an alternative approach.

The design of microcode using the manual flowchart method involves the process of generating a formal Register Transfer Level (RTL) description of the algorithms. For each processor cycle one or multiple data transfers are allocated as functional description. The separation of an algorithm into individual cycles is done according to data dependencies and limitations of interconnection resources between registers and functional units.

Tredennick [Tre87] develops this basic notation into so called 'Level 2' flowcharts, containing additional information about external memory accesses, state sequencing and other processor specific attributes.

Experimentation with a prototype RTL-style notation on some of the major co-processor algorithms revealed a high repetition factor for single data transfers. At the same time individual processor cycles often show a high number of separate data transfers within them, where the description of the different transfers becomes quite complex. This complex description of data transfers originates from the fact that the segmented buses and the shifting facilities, inside the bus structures, require additional information to document the functionality of individual data transfers. The high number of data transfers during one cycle in combination with their complex nature resulted in a RTL description where individual cycles are more complex than in most standard processors.

Due to the complexity of most list manipulation algorithms, the micro-instruction sequences required to implement their functionality are quite long. Together with
6.2. Design Methodology

the complexity of individual cycles this results in a manual flowcharting process
which is not realistic for this kind of micro-instructions. However, an automated
approach, using the basic ideas of manual flowcharting, promised to offer a more
realistic method of microcode generation. The process of converting this idea into
a new microcode development method is presented in the next section.

6.2.2 Database Method

One major advantage of the manual flowcharting method is its flexibility to sup­
port the design of microcode for any kind of processor. Therefore any attempt to
automate this method has to try to maintain this. Only by analysing the ideas and
underlying concepts of a methodology is it possible to guarantee that the positive
aspects can be carried over to an enhanced method.

An analysis of manual flowcharting showed that the basic concept is to split the
given information into simple components and then re-use and combine these compo­
nents in building more complex structures. As stated before, the general information
expressed in flowcharts is normally made up from the following components:

- A description of the individual data transfers performed during one processor
cycle.

- Sequencing information, specifying which cycle will be executed next.

- Information about I/O-behaviour of the the cycle.

- Additional processor-specific facts influencing individual cycles.

- References to identical cycles and/or count of duplicates of the current cycle.

While the latter two points are not always required to describe the micro­
instructions, their use can help to optimise the generated microcode at later stages.

When the above mentioned components of a flowchart were examined in more
detail, it became obvious that a set of possible values existed for each component. In
some cases the set was given explicitly, while in other cases it could be constructed from sub-components following given rules. This regular structure of the individual components used in manual flowcharting suggested that a database would be an ideal tool to store the different values. The premise behind this work was that once the various components expressed in a flowchart are stored in individual relations, an automated design method, utilising the database, should offer the same flexibility as manual flowcharting. This concept resulted in the development of an Data Model where each set of values used in a component is stored in a separate relation. In cases where a component is constructed from sub-components, only these sub-components were represented as relations in the Data Model.

Normally any database concept should be able to support the capturing of a database describing the facts involved in a micro-program development. For the work presented here, a relational database was chosen.

Originally the database method was only intended to capture the micro-program data at a relatively high abstraction level, using a RTL notation to describe individual data transfers. The main focus was put on to points such as comfortable data entry, easy updating and good searching facilities. However, during the development of the micro-program, other ways of using the database environment were integrated. One of the advantages in using a database tools is that database queries allow a simple consistency checking of the stored information. All these advantages and the fast and simple tool development, on top of existing database related tools, inspired to extend the tool to lower abstraction levels and support of the micro-code generation.

Data Model

The development of the Data Model, consisting of independent relations describing all aspects of the list manipulation algorithms, was the first step towards an automated tool for the micro-program development. This section will describe the stages
involved in the development of such a Data Model\(^2\), using the list manipulation co-processor to demonstrate individual concepts.

A top down view of the target co-processor will take the functions available to external units as the highest abstraction level. In the Data Model, expressing the micro-program inside the database, this fact is expressed by macro-instructions offering this functionality. At the next lower abstraction level the algorithms for each macro-instruction are separated into individual cycles. Below this level, the cycles are divided into individual parallel data transfers, while the lowest abstraction level describes the control signals activated to perform each of these data transfers. The Data Model shown in Figure 6.1 captures all these abstraction levels for the co-processor.

![Figure 6.1: Data Model](image)

There are 3 'library' relations: 'relation' being a standard database term for a table of records holding a particular kind of information in a Data Model. The central relation COMPONENTS holds records on all available transfers on the datapath. A single record within COMPONENTS details one transfer, indicating source, target and bus section used, where bus section is either a group of bus signals or a whole bus. The LINES relation lists all control lines leading into the datapath to control its operations. The last 'library' relation STEPS details all co-processor 'cycles', where a single cycle is made up of a set of parallel-executed transfers on the datapath, details of the transfers are recorded in the COMPONENTS relation.

The connection between STEPS and COMPONENTS is recorded in the CONSISTS relation through the specification of all 'components' records used in each

\(^2\)More details can be found in [DR94]
Chapter 6. Development of the Co-processor

'Step' record. The ACTIVATES relation does the same for specifying all the 'lines' records used in each transfer. Essentially ACTIVATES details all control lines activated for a particular transfer while CONSISTS records all the parallel transfers performed during a particular co-processor cycle. The last relation of Figure 6.1 is FOLLOW which records the sequencing of the 'steps' records in a larger algorithm. Thus FOLLOW is used to specify all the large scale functions of the co-processor.

Applying the Data Model

The process of capturing the algorithms in the database, and therefore expressing the micro-program of any processor, has to follow a sequence. The major factor determining this sequence is the dependency of individual relations. It is obvious that 'library' relations of the Data Model have to be captured in order to relate the described activities to an action at another abstraction level.

When this concept is applied to the given Data Model, it is most sensible to start the capturing process with the definition of individual data transfers in the COMPONENTS relation.

Capturing Data Transfers

Before individual data transfers can be defined, it is necessary to know the exact structure of the datapath, as each description of a data transfer will require a detailed specification of source and target register as well as the bus used. If the structure of the datapath, as an important design factor, has not been decided before, it has to be specified at this point.

Up to this point, the designer did not have to consider the details of data transfers performed in the datapath. However, the specification of data transfers requires these details. While the definition of source and target register tends to be predetermined, a datapath with multiple buses offers the designer a certain flexibility in selecting a bus for individual data transfers. However, the choice of buses can have
an significant influence on the performance achieved by an algorithm, and a ran-
dom choice of the bus is not appropriate. At this stage of the specification process,
however, the final grouping of data transfers executed during individual processor
cycles can not be predicted. It is therefore suggested to start assigning the buses
such that components which have to be executed simultaneously are arranged and
defined first. Memory accesses, for example, constitute such simultaneous transfers,
and will normally involve several components. Once a number of these major trans-
fers are generated, additional transfers can be assigned according to the following
criteria:

1. If the same section of a source register is transmitted to several target registers,
   the same bus section can be used in a form of broadcast. Multiple transfers
   use the same source register and bus section with different target registers.

2. If another, already defined, component implements a related transfer with the
   same source and target registers and similar bit positions, the same bus should
   be used.

3. If the transfer can be executed in parallel with other transfers, and only one
   bus has left available transmission capacities on the required bus section, this
   bus should be used.

4. If either the source or the target register so far only requires connection to one
   bus, this limited connectivity should be maintained.

5. If a transfer could be performed on either bus, and other transfers which can
   be executed in parallel do not limit the allocation, the transfer should use a
   bus section adjacent to other components, trying to reduce the fragmentation
   of the buses.

Using this concept of assigning buses to individual data transfers, it is possible
to capture a significant number of important data transfers into the components
relation. In cases where it is not possible to decide on a bus, the specification of
the data transfer can be delayed until the data transfer will be required to describe
the activities during a cycle at the next higher abstraction level of the Data Model.
When such a delayed specification of individual cycles is used, the combination of unrelated data transfers executed in parallel can help to decide on the choice of the bus.

**Capturing the other Abstraction Levels**

After an initial set of data transfers, covering the functionality of the list traversal, are defined, a first attempt at arranging parallel transfers into cycles can be performed. Starting with the development of a basic set of components and cycles, an iterative process of defining individual new data transfers as components, and arranging these components into cycles will follow. This process has to follow certain rules to maintain the database consistency and ease a later expansion to lower abstraction levels. The first rule is that parallel components are not allowed to have overlapping signals on sub-sections of the two buses. A second rule says that all components have to be disjoint (*i.e.* there are no pairs of components having the same source and target register and overlapping bus areas).

Database queries can be used to verify that the contents of the database follow these and other rules. In case of components being not disjoint, it will be necessary to split them up, and create a number of new data transfers using smaller bus sections. Cycles with overlapping bus signals require a re-arrangement of transfers into other cycles, or a change of components to avoid multiple registers writing their information to the same bus line(s).

As soon as the transfers implementing a certain algorithm are grouped into cycles and there are no more obvious problems, the next step is to arrange the sequencing of these cycles. During this process the database can be used to verify that each cycle has a predecessor and a number of successors matching the branching conditions.

The expansion to lower abstraction levels can follow when the algorithms of the most important functions are entered into the database and the collection of data transfers in the COMPONENTS relation reaches a stable state. A systematic analysis of all data transfers has to extract the detailed connectivity of each register in the datapath. For each register a list of read and write ports connected to each
bus can be extracted from the database. The next step is then to identify which of these ports are used individually or only in groups, as it will be necessary to assign a separate control line for each individually used port. The process of generating new addresses or data words out of several components will involve several registers sending data to ports of one register. However in some cases, all the ports may always be used together as a group and therefore will require only one control line. In a similar way the extraction of data words into several components will frequently use ports only as a group.

Once all required control lines are known and entered into the database, the final step will be the definition of the control line activations for each data transfer. As soon as this process is completed, the contents of the database can be used to verify the algorithms as Section 6.3 will demonstrate.

Details of the Data Model

In the following section the individual relations of the Data Model are presented in more detail. With reference to the original flowchart method [Tre87] the STEPS relation represents the outer frame of individual state in level 2 flowcharts, as in Figure 6.2.

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Type</td>
</tr>
<tr>
<td>Duplicates</td>
</tr>
<tr>
<td>Register_Set</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TASKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steps_ID</td>
</tr>
<tr>
<td>Identicalstep</td>
</tr>
<tr>
<td>Next_State</td>
</tr>
</tbody>
</table>

Figure 6.2: Level 2 Flowchart
A single state of a level 2 flowchart contains fields to describe:

- the type of external access, i.e. memory accesses outside the co-processor
- how to determine the next cycle, i.e. no successor, direct successor or branch, with details of the last 2 cases in FOLLOWS.
- the number of duplicates of a cycle: a duplicate is another cycle within STEPS with the same transfer specification, but different in other aspects e.g. NEXT_STATE, REGISTER_SET, etc. This is only held in a cycle which holds the original specification of the common transfers.
- for a duplicate cycle a reference to the record holding the common specification of all data transfers.
- name of the instruction of which the cycle is a part.
- REGISTER_SET holds information, that allows parameterisation of some cycles.

The attributes of the STEPS relation are shown in Table 6.1.

<table>
<thead>
<tr>
<th>STEPS_ID</th>
<th>Key(numeric)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCESS_TYPE</td>
<td>no access(NA), data read(DR), data write(DW)</td>
</tr>
<tr>
<td>DUPLICATES</td>
<td># of duplicates (zero if copy)</td>
</tr>
<tr>
<td>NEXT_STATE</td>
<td>final cycle(NO), direct successor(DI), branch(BR)</td>
</tr>
<tr>
<td>IDENTICALSTEP</td>
<td>key of the original (zero if original)</td>
</tr>
<tr>
<td>REGISTER_SET</td>
<td>internal information</td>
</tr>
<tr>
<td>INSTRUCTION</td>
<td>Name of instruction(alphanumeric)</td>
</tr>
</tbody>
</table>

Table 6.1: Attributes of the STEPS relation

More detailed information of the next cycle is held in the FOLLOW relation, as the STEPS relation only contains information about how the successor(s) is determined. The details of this relation are given in Table 6.2.

The STEPS relation holds the transfer various specifications indirectly through the STEP_ID of the CONSISTS relation. The CONSISTS relation, as seen in Table 6.3, relates the STEP_ID with the transfer in COMPONENTS.
6.2. Design Methodology

<table>
<thead>
<tr>
<th>STEP_ID</th>
<th>key of the current cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOLLOW_ID</td>
<td>key of the next cycle</td>
</tr>
<tr>
<td>CONDITION</td>
<td>description of the branching condition</td>
</tr>
<tr>
<td>VALUE</td>
<td>binary pattern of the actual condition</td>
</tr>
</tbody>
</table>

Table 6.2: Attributes of the FOLLOW relation

<table>
<thead>
<tr>
<th>STEP_ID</th>
<th>key of the cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMP_ID</td>
<td>key of the transfer</td>
</tr>
<tr>
<td>INFO</td>
<td>additional information(alphanumeric) if needed</td>
</tr>
</tbody>
</table>

Table 6.3: Attributes of the CONSISTS relation

There will be several CONSISTS records containing the same STEP_ID, but different COMP_IDs, so that CONSISTS identifies a set of transfers from COMPONENTS performed in parallel within one cycle. Table 6.4 gives a description of the COMPONENTS relation.

<table>
<thead>
<tr>
<th>COMPS_ID</th>
<th>Key(numeric)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM_REG</td>
<td>Name of source register(alphanumeric)</td>
</tr>
<tr>
<td>FROM_START</td>
<td>LSB of area to be transferred(0..39)</td>
</tr>
<tr>
<td>FROM_END</td>
<td>MSB of area to be transferred(0..39)</td>
</tr>
<tr>
<td>FROM_COND</td>
<td>processor specific additional information</td>
</tr>
<tr>
<td>BUS</td>
<td>used bus (B1, B2, DI[direct to status-reg])</td>
</tr>
<tr>
<td>TO_REG</td>
<td>Name of target register(alphanumeric)</td>
</tr>
<tr>
<td>TO_START</td>
<td>LSB of target area(0..39)</td>
</tr>
<tr>
<td>TO_END</td>
<td>MSB of target area(0..39)</td>
</tr>
</tbody>
</table>

Table 6.4: Attributes of the COMPONENTS relation

As described in Section 5.5, transfers may not only perform a simple register-register transfer where each bit of the source goes to the same bit position in the target, there are some shifting transfers where the bit position changes from source to target. This is the reason for the TO_START and TO_END fields in COMPONENTS.
Chapter 6. Development of the Co-processor

In the LINES relation, see Table 6.5, the names of the various control lines are stored together with additional information.

<table>
<thead>
<tr>
<th>LINES_ID</th>
<th>Key (numeric)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Name of the Control Line</td>
</tr>
<tr>
<td>DESCRIPTION</td>
<td>Additional explanation (if needed)</td>
</tr>
</tbody>
</table>

Table 6.5: Attributes of the LINES relation

Finally the ACTIVATES relation, Table 6.6, just connects the transfers with the control lines. Obviously there can be many relations with duplicate COMP.IDs and many with duplicate LINE.IDs, providing many-to-many relations.

<table>
<thead>
<tr>
<th>COMP.ID</th>
<th>key of the transfer (component)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE_ID</td>
<td>key of the control line</td>
</tr>
</tbody>
</table>

Table 6.6: Attributes of the ACTIVATES relation

This basic Data Model can be easily adapted to match the needs of other processors just by adding new attributes and relations as required. Although the Data Model embeds a model of the controller as breaking algorithms into cycles, cycles into transfers, and transfers into control line activations, this is a fairly general model with the only embedded hardware description being at the control line level. The description of the datapath in COMPONENTS is very general since the level of specification is of transfers, transfer sources and targets, and transfer pathways, which is a fairly general description of any electronic circuit. The Data Model does not embed any model for the implementation of the control logic.

6.3 Behavioural Simulation

The process of specifying co-processor algorithms using the database method helped to gain additional knowledge about the structure of the datapath, such as the ports of individual registers and the control lines required for their control. With this new information about necessary control lines and the way individual registers are connected to the buses, an improved behavioural model of the datapath could be
written and used to simulate and verify the list manipulation algorithms captured in the database.

During the design of the example co-processor the VERILOG [Cad] package was used to simulate the datapath. For the simulation an abstract gate-level model of the registers was created. This model does not use any delays and includes an abstract clocking scheme to model the individual phases performed during one clock cycle. To support the use of multiple control lines activating ports for certain bus sub-sections, the individual registers are generated in a bit-slice manner. Individual one bit registers are piled to form larger registers, while the control lines are only shared where this is possible. In a hierarchical way the one bit registers are generated out of simple components such as incrementers, flip-flops and latches, using a small library of standard elements.

The simulation patterns to operate the datapath are extracted from the database containing the description of the micro-programs. An uncomplicated way to test the functionality of the algorithms captured in the database is to maintain the abstraction levels of the Data Model and produce a hierarchical simulation producing the required control signals. Using this approach eases the generation of the simulation patterns and helps to identify error sources.

6.3.1 Extracting Database Information

The information held in the database supports the conversion to simulation patterns for the datapath in a bottom-up strategy. The first step in this approach is to model the activation of control lines performed during a data transfer. The VERILOG ‘task’ mechanism offers an ideal way of implementing this concept. For each data transfer, held in a record of the COMPONENTS relation of the Data Model, a VERILOG task is generated. The only purpose of these tasks is to activate the required control lines by setting a corresponding variable to one. Figure 6.3 gives an example of such a task.

When, during a single cycle, a register value is transferred to multiple target registers, the cycle description contains multiple data transfers with the same source
task comp_121;
begin
  l_reg_d = 1;
  l_reg_i = 1;
  l_reg_sel_alt = 1;
  stk_read = 1;
end
endtask

Figure 6.3: Activations for component 121

description. As each of these transfers is represented by an individual task, it happens that multiple tasks, representing parallel data transfers, activate a single control line. This kind of multiple activation of control lines by several data transfers executed in parallel has no consequences and is treated as a single activation of each line. For a functioning of the simulation, all control lines have to be inactive at the beginning of each cycle.

At the next higher abstraction level, all data transfers performed during one cycle have to be executed in parallel. Using the data of the CONSISTS relation, tasks can be generated for all cycles holding the original specification of parallel transfers \((i.e. \ not \ for \ cycles \ that \ are \ duplicates)\). Figure 6.4 demonstrates an example of such a task.

As this group of tasks only handles data transfers for original cycles, the simulation has to provide an additional level of translation, helping each duplicate cycle to activate the data transfers of the corresponding original cycle. In addition this level of indirection can be used to incorporate general information, held in the cycle description, into the simulation. In the example of the co-processor, this feature was used to determine whether data transfers operate on the list structures representing the source or the target list, as described in Section 5.6. The whole translation is implemented as one task, within which the tasks implementing the functionality of the matching original cycles are initiated. A small section of this task is shown in Figure 6.5.

The final stage is to allow the simulation to sequence individual cycles and simulate complete algorithms. The branching information collected in the FOLLOW
6.3. Behavioural Simulation

Figure 6.4: Components executed during step 526

```plaintext
# fork
delete_all;
#1 comp_89;
#1 comp_90;
#1 comp_144;
#
#1 comp_151;
#1 comp_156;
#1 comp_157;
#1 comp_158;
#1 comp_159;
#1 comp_161;
#1 comp_162;
#1 comp_216;
#
#1 comp_220;
```

Figure 6.5: Extract of function activating individual steps

```plaintext
begin
    case(inst)
        526 : begin
            set_1_2n = 0;
            step_526 ;
        end
        527 : begin
            set_1_2n = 0;
            step_47 ;
        end
        default : $display("Unknown step !\n");
    endcase
end
```

Figure 6.6: Determining the next instruction according to the flags of the processor status word.

The first two groups of tasks as well as the latter two large tasks can be automa-
Chapter 6. Development of the Co-processor

Figure 6.6: Extract of sequencing function

...translation of the text...

Figure 6.6: Extract of sequencing function

...translation of the text...

6.3.2 Joining the Components

To complete the environment for a behavioural simulation of the micro-programs, generated using the database method, the tasks extracted from the database have to be integrated, with the datapath model, into a core simulation environment. The main aspect of this environment is to provide timing information and a model of the hardware surrounding the co-processor. However, the model of the surrounding
6.3. Behavioural Simulation

hardware can be reduced to a memory area holding the list structures, to be manipulated, and the required free page lists. With help of some additional tasks the contents of the memory can be read from files, enabling a straight-forward mechanism for modifying the simulation environment.

The execution of co-processor algorithms can be triggered with help of additional tasks. As a first step, these tasks will write the original list structures into reserved areas of the list memory. Afterwards the required parameters will be written into the registers responsible for the communication with the external hardware. Finally the simulator will be supplied with the an instruction, the algorithm to be executed. The simulation of the algorithms is started when an external flag puts the simulator into an auto execution mode, executing the data transfers and sequencing the individual cycles. Reaching a final instruction of an algorithm switches off the auto execution mode and the designer can examine the status of the simulated hardware at algorithm termination.

6.3.3 Scale of Simulation

The main purpose of the behavioural simulation is to test the functionality of the micro-program versions of the co-processor algorithms. At the same time the simulations helps to prove that a given datapath offers the required functionality.

The first short simulations should check the basic functionality of the primary algorithms. This is of special importance as other, more complex, algorithms are likely to use the same sequences of processor cycles as a basic model. The identification of complications in these primary algorithms also tends to be easier, as there will be fewer influencing factors. Any modifications of the micro-program of the basic functions will also be more general and therefore more generally applicable. The next step should expand the tests to more specialised cases and less frequently used functions. Once the regular functionality is approved, the tests have to be expanded to check the behaviour in error situations. Again standard error situations need to be tested before the tests are expanded towards rare error situations.
When this concept was applied to the SPRINT co-processor, the first tests ensured the proper functioning of a basic list traversal with scalar data values. Starting with the duplication of short lists, the number of elements was increased to test the effects of overflowing element counters, the page replacement and allocation of new pages. The next series of tests concentrated on slightly advanced aspects of the list copying algorithm, such as duplicating lists with offsets, modifying the list offset during duplication and generating target lists with a modified list level. After the functioning of the list copy for lists with scalar data types had been verified, similar tests were executed with nested lists. The main aspect of these tests was to confirm the micro-program sequences dealing with the stacking of register information in case of recursive function calls. Lists with several levels of nesting where used to test the code sections moving the on-chip stack to and from memory.

All the simulations mentioned so far covered only the regular and successful manipulation of lists. A second set of tests was performed to analyse the algorithms in abnormal and error situations. The two major error situations are lack of memory and invalid data. As the free page lists could be modified by writing the contents of special files into the memory, it was no problem to produce simulation setups where the systems runs short of free pages at any required point of execution. With this method it was also possible to produce abnormal situations such as a shortage of pages in the extended memory space. In such a situation algorithms will be forced to allocate pages in the regular memory space for all required memory. Situations involving the use of invalid data can occur when an algorithm is not defined for certain data types identified by a particular value of the tag field. In the example co-processor the algorithms were not defined for the tag values 'special-list' and 'special-atom'. Any occurrence of these tag values therefore had to trigger an exception. The simulations of these error situations were performed to ascertain that error recovery can be performed after the co-processor terminates processing, facing one of these errors. To enable a full error recovery it is necessary to know what caused the error and at which point of the list traversal it occurred. Simulating the errors showed that the error marking concepts work and that appropriate error codes are written to the error register before the algorithms terminate.
6.3. Behavioural Simulation

After simulating the various aspects of the list copying algorithm, similar test series were executed for all other list manipulating instructions. The tests were used to confirm the proper functioning of algorithms in standard as well as error situations.

6.3.4 Micro-program Optimisation

With the successful completion of the behavioural simulations, the micro-program versions of the list manipulating algorithms reached a stable state. This situation offered a good point to consider further improvements of the micro-program. Improving the speed of execution and the size of the micro-program were the most important aspects.

At this stage, the speed of execution can best be enhanced by reducing the number of processor cycles required to perform a certain task. Without any changes of the datapath this requires the re-organisation of the individual data transfers so that they can be packed in fewer cycles. The process of re-arranging data transfers follows rules similar to the rules compilers use to optimise regular program code. The parallel data transfers also have to be analysed for data dependencies, before attempts are made to re-schedule the data-transfers to achieve a better utilisation of the existing bus capacities. Similar to 'building blocks' in regular code design, the micro-program can also be fractioned into small groups of cycles which then might be optimised. Like 'building blocks' these groups of cycles normally form a sequence, not containing any branches or branch destinations. The process of identifying possible groups, as well as the search for improved arrangements of data transfers, can be supported by database queries.

The second source of optimisation does not affect the speed of the final co-processor, but its size. When the micro-program can be compressed into fewer words, this can help to reduce the size of the controller and therefore of the whole chip. Depending on the actual structure of the controller used in the processor, different optimisations will have different effects. However, a reduction of cycles will always have positive effects, as it can enable the designer to include additional instructions,
Chapter 6. Development of the Co-processor

or to reduce the number of words in the memory holding the micro-code. One effective method to reduce the number of cycles checks the final instruction sequences of different algorithms. When the last cycles called by two instructions activate the same data transfers, this sequence is only required once, the second instruction sequence can end with a jump joining the code of the first instruction. In the micro-program of the example co-processor this method was used for example to share the error handling routines dealing with a memory shortage. In an even more extreme case the head-append instruction, adding a list to the head of a second list, consists only of a small start sequence before it joins the code of the regular list copy instruction to perform the actual work.

The structure of the co-processor algorithms and their frequent use of common components suggested to differentiate between original cycles and duplicate cycles. This separation of cycles into two categories is likely to influence the structure of possible controller designs. Reducing the number of original cycles and generating as many duplicates as possible, can be seen as an additional way of reducing the size of the controller.

It is obvious that a final version of the micro-program can be checked, with the help of a database query, to find cycles with identical data transfers that have not been found before. A more advanced analysis can search for cycles representing a functional sub-set of another original cycle. For each of these cycles, as well as its duplicates, it is then necessary to investigate if the replacement of the cycle by its super-set will influence the functionality of the algorithm. In cases where the additional data transfers do not influence the outcome of the list manipulation, it is then possible to simplify the micro-program and replace all occurrences of this cycle with duplicates of the super-set.

Performing all these checks on the micro-program can be repeated quite easily, as it only requires the execution of database queries stored in the file system of the test environment. A careful formulation of the individual test queries eases the result checking significantly, as the queries then only return errors or suggestions for possible optimisations.
6.4. Controller Design

All these optimisations of the micro-program should not change the functionality of the algorithms. If necessary the optimised micro-code can be simulated at behavioural level once more, otherwise the simulations at the next lower level should reveal any problems.

6.4 Controller Design

The behavioural simulation of the micro-program and the subsequent optimisation give a clearer picture of the required functionality of the controller. In this section it is demonstrated how these constraints were used to decide on a structure for the co-processor control part, using the database contents to support the decision process. Two major aspects in this process are the structure of micro-code store and the sequencing of co-processor instructions. In the later part of this section it is explained how the database information was extracted to match the structure of the controller and how this extracted micro-code was simulated at gate-level. The section ends with a discussion of micro-code replacement alternatives for the different implementations of the Writable Control Store and a description of the co-processor operation mode.

6.4.1 Microcode Implementation

The development of micro-coded processors over the last four decades has produced a wide range of possible controller structures [Hus70] [AR76]. While horizontal and vertical micro-code [BR82] represent the two extremes, most implementations apply more moderate technologies such as diagonal micro-code or two-level control stores [Sal76].

In order to find an appropriate controller structure for the list manipulation co-processor it was necessary to analyse the requirements of the intended micro-code. While the complexity of the list manipulating algorithms causes the micro-program to be lengthy, the actual number of different data transfers used during the
manipulations is relatively low. Even more important, the number of combinations of data transfers executed in parallel during the various cycles is also quite limited. The Data Model with its differentiation of original and duplicate cycles offers a method to identify the number of required combinations.

Logical Structure

The design of a controller will generally be influenced by several factors, where speed and size are amongst the most important ones. It has already been stated in Chapter 5 that the speed of the co-processor will be limited by the speed of external memory. Therefore it is not necessary to design an extremely fast controller, enabling concentration on size reduction. The specification of the co-processor does not urgently require the use of a Writable Control Store (WCS), however such an implementation is extremely valuable, as it enables the user to modify the functionality of the co-processor to individual requirements. Modifiable memory structures however require more space than permanent memories such as ROM. When the implementation of the controller uses a Writable Control Store, it is even more important to design the controller according to guidelines reducing the size of the modifiable memory.

A detailed examination of the micro-program stored in the database revealed that most original cycles were defined in the essential list manipulation algorithms. The few additional cycles used in the implementations of advanced algorithms could be replaced by a sequence of simpler cycles already used by basic algorithms. Due to the given functionality of the datapath, it is expected that the creation of further instruction sequences will show the same behaviour. Under these circumstances it will not be necessary to enable the user to create new combinations of data transfers, as a limited set of existing original cycles should enable the implementation of new algorithms. When a list manipulating co-processor is developed to product level, a large library of standard algorithms will define a moderate sized set of original cycles. When this collection of, more or less complex, data transfers is complemented with a small selection of elemental transfers, the resulting collection will enable the creation of additional micro-code for user-specific purposes.
6.4. Controller Design

With this approach, the concept of a two level control-store\(^3\) can be used to reduce the amount of modifiable memory required in the controller. The micro-word store is implemented in a technology allowing the user to change the instruction set, but consists only of sequencing information and activations of the required data transfers. Meanwhile the nano-word store, holding all the information about control lines activated during one cycle, can be implemented as ROM. The dimension of the nano-word store depends on two factors: the number of different data transfers and the number of control lines. A basic implementation of the store will contain one row for each group of parallel data transfers performed during a cycle (i.e. for each original cycle expressed in the Data Model). Additional rows can be added to include a number of simple data transfers, easing the development of additional algorithms by the user. The individual rows are frequently called words, so the nano-word store consists of a number of words \(W\) equal to the number of original cycles in the database, plus the number of additional data transfers.

Each of these words will, in the basic implementation of the nano-word store, contain a number of columns equal to the number of control lines leading into the datapath. However this number can be reduced with standard compression methods [And80], such as field encoding. Some registers will require a significant number of control lines to manipulate the bus activities, but only a limited number \(P\) of different activation patterns will be used. As all control lines of one register will be required in the same section of the datapath, an encoding of these lines will reduce the length of the nano-code words without causing major problems at the layout stage of the processor design. Field encoding generally reduces the ability to modify the nano-code, but for the given co-processor this is not a disadvantage, as the range of data transfers is well defined.

For each register in the datapath it will be necessary to evaluate the savings that can be achieved by encoding the associated control lines. By encoding \(O\) original control lines into \(E\) encoded lines for each of \(R\) registers, the size of the nano-code ROM can be reduced by \(W \times \sum_{i=1}^{R} O_i - E_i\) bits, where \(E_i = \lceil \log_2 P_i \rceil\). On the other

\(^3\)The controller of the Motorola 68000 processor [HN84] is a typical example for such a design.
hand there will be the additional cost for the decoders, restoring the values of the un-decoded control lines.

**Sequencing of Cycles**

Looking at the micro-code store, the format of a single word in this store depends on the format of its individual components, where the number of components and their format is closely linked to the method applied in generating the sequencing information. Again, a wide range of sequencing mechanisms has been developed in the past [Anc86] [BR82]. Common methods include the generation of micro-addresses within various PLA structures as well as bit replacement within a given micro-address.

The decision on the sequencing mechanism used in a micro-processor depends mainly on the structure of the algorithms that have to be executed on the processor. In case of a list manipulation co-processor, for the SPRINT architecture, the required algorithms are quite complex and contain frequent branches with several branching conditions. A closer analysis of the design database revealed the following sequencing patterns:

- Only about 57% of all cycles have a direct successor.
- Another 27% of all cycles use 11 different binary branches to identify the next cycle. Each of these branches is decided on one flag of the processor status word.
- About 6% of all cycles use two flags of the PSW to determine their successor by 3 different three-way branches.
- Nearly 7% of all cycles are related to the memory management and use three flags (22 to 24, see Table 5.2) to decide on a three-way branch.
- Finally most algorithms need the three bits of the tag field (PSW bits 36 to 38) to decide on an eight-way branch to treat the different data types during the traversal of a list.
The eight-way branches during the list traversal have complex sub-algorithms attached to the different cases. These have to be executed before the control flow joins again to handle the next list element. When an algorithm reaches one of these branching points, the sequencer has to generate eight different addresses to activate the next cycle. To handle this kind of branches, it appears to be best to temporarily forget about the following instructions and allocate all eight cases in neighbouring micro-code words.

The number of possible branch destinations will always correspond to the number of bits used in the branching condition. By aligning the possible branch destinations to appropriate locations in the micro-word store, it is possible to address the different cases by modifying the least significant bits of the micro-word address. The number of bits that have to be manipulated in this way will depend on the number of bits used in the branching condition.

In case of the eight-way branches this means that the first word has to be aligned on an micro-word address where the lowest three bits are zero. Then all eight possible branch addresses can be generated by modifying the lowest three bits of the follow address. As the most complex branches use three bits to distinguish the different cases, these will always be limited to a maximum of eight. Any required sequencing information can therefore be generated out of a basic follow address extended with three bits generated in the sequencer. To support branches connected to different flags of the processor status word, it will be necessary that the the sequencer is informed about which branching condition has to be used when determining the three bit extension.

The 'Modifier' Field

The concept of expanding a basic follow address with a numeric value between zero and seven works fine as long as all cycles use three bits to decide on branches with eight different branch destinations. When branches with fewer destinations are implemented in the same way, gaps appear in the microcode, as all branches
would be allocated to eight word blocks. As the algorithms use cycles with fewer branch destinations and even with direct successors, it is possible to fill holes in the microcode store with branches allocated to addresses with fewer trailing zeros or odd addresses for direct successors.

However, implementing branches with fewer successors requires the sequencer to produce address offsets not only bound to the branch condition, but also to additional placement information. For example, a branch operating on two flags requires additional information informing the sequencer whether to generate offsets in the range from zero to three, or between four and seven. In a similar way binary branches have to be accompanied by additional information telling the sequencer which of four possible offset pairs is required. Finally, to generate the offset for the address of a cycle being a direct successor, of the current cycle, the sequencer has to be supplied with a three bit value which is not modified at all.

Thus, a sequencer allowing the generation of 3 bit offsets requires a 3 bit input to allow the addressing of direct successors in all possible locations in the micro-word store. In Figure 6.7 this information is represented in the 'Modifier' field in the micro-code store.

The 'Condition' Field

The Condition field of the micro-code words contains information about which branching condition will be used to determine the next cycle. According to the list on page 158, the SPRINT co-processor uses 17 different branch modes. This normally would require \( \lceil \log_2 17 \rceil = 5 \) bits to binary encode the possible branching conditions. However, the method of assigning branch destinations to contiguous micro-code locations means, that the lowest order bits of the modifier value will be replaced by the offset required to address the wanted branch alternative. When performing a binary branch, this means that the sequencer will replace the lowest order bit of the modifier with a binary value, addressing the required branch alternative. Therefore the lowest bit of the modifier can also be used to hold information about
6.4. Controller Design

Figure 6.7: Basic structure of the Controller

the required branch. The more complex branches, addressing between three and eight different locations in the micro-code store, enable the designer to use two or even all three bits of the modifier to carry additional information about the required branch condition.

Using this compression technique, it was possible to combine the modifier and condition fields in a way that required only three additional bits to inform the sequencer about the required branching condition. Table 6.7 demonstrates how this reduction was accomplished.

The sequencer of the experimental co-processor produces three output signals out of 23 inputs. The functionality is achieved by a PLA structure, feeding the 23
Table 6.7: Encoding of branches in the experimental co-processor

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Condition</th>
<th>Branch-Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>- - -</td>
<td>0 0 0</td>
<td>direct successor</td>
</tr>
<tr>
<td>- - C</td>
<td>B A 1</td>
<td>8 - 1bit branches</td>
</tr>
<tr>
<td>- - 0</td>
<td>B 1 0</td>
<td>2 - 1bit branches</td>
</tr>
<tr>
<td>- - 1</td>
<td>0 1 0</td>
<td>1 - 1bit branch</td>
</tr>
<tr>
<td>- X 1</td>
<td>1 1 0</td>
<td>1 - 2bit branch</td>
</tr>
<tr>
<td>- D 0</td>
<td>1 0 0</td>
<td>2 - 2bit branches</td>
</tr>
<tr>
<td>X D 1</td>
<td>1 0 0</td>
<td>2 - 3bit branches</td>
</tr>
</tbody>
</table>

(‘-’ reflects a non-modified bit of the ‘Modifier’ field, X indicates a un-used bit of the ‘Modifier’, ABCD represent binary encodings for the different branch conditions, while 0 & 1 are fixed values to identify the branch type)

inputs into an AND array, producing 40 product terms which are then combined to the three outputs in the OR array. These three bit values can then be used to modify microcode-addresses and realize the required eight-way branches.

So far two components required in each micro-word have been identified. In addition to the ‘Modifier’ and ‘Condition’ field used in the sequencer there has to be a field containing the high order bits of the follow address, ‘Follow Base’ and, of course, a field activating the required data transfers for the current cycle.

The ‘Original Cycle’ Field

This activation of the data transfers is implemented by stimulating one line of the nano-code memory. The width of both, the ‘Original Cycle’ and the ‘Follow Base’ field depends on the algorithms included in the micro-code. Depending on the number of words contained in the nano-code store, the format of the micro-code word has to contain a field wide enough to hold the binary encoded number of the required activity.

With this knowledge, the process of adding simple data transfers to the nano-code, in order to ease the development of user specific algorithms, can be evaluated
more precisely. Depending on the number of required nano-words, adding a further group of parallel data transfers can mean a different amount of extra controller complexity. The addition of extra nano-code words, without an increase in the number of lines beyond the next power of two, does not require any change of the micro-word format. Therefore the only additional hardware costs will be the extra lines of ROM in the nano-code store. However, when the addition of an extra word to the nano-code causes the overall number of nano-words to hit another power of two, this requires an additional bit of modifiable memory in each word of the micro-code. This can cause a drastic increase in the size occupied by the micro-code store and the whole controller, this situation should therefore be avoided.

Another problem will be the compatibility with later versions, a typical problem of all devices using micro-programs. Adding extra lines to the nano-code might be possible with minimal hardware overhead in the initial version of a processor. When later versions, with improved facilities, are expected to be compatible with earlier models, all transfers introduced in one design have to be maintained. When now a new version requires extra nano-code lines, this can cause the number to pass a power of two, causing the micro-word format to grow. In this case, the supportive nano-code lines, originally implemented for low cost, can be the deciding factor about the size of the micro-word in later versions of a micro-programmed processor.

In the example co-processor the required number of nano-code lines are just below the 128 mark. This permits the use of a 7 bit field to encode the required nano-code lines. Adding a limited number of about 50 basic data transfers would support the development of user-defined algorithms significantly, and is therefore part of the initial design specification.

The 'Follow Base' Field

As the sequencing function determines the address of the next micro-word by concatenating the value of follow base field with the three bits generated in the sequencer, it is obvious that the length of the follow base depends on the number of
words held in the micro-code store. The number of micro-code words itself largely depends on the number of algorithms implemented in the co-processor. A compromise has to be made between controller complexity and processor functionality. While a larger selection of algorithms increases the range of application of the co-processor, the larger controller also means higher costs.

An example of instruction complexities is given in Table 6.8. As most algorithms share error sequences and other final sequences, the cycle counts for individual instructions are not given explicitly. Instead the complexity of all list traversing algorithms is given as an additional complexity on top of the basic list copy instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy List</td>
<td>427</td>
</tr>
<tr>
<td>Head Append List</td>
<td>+ 5</td>
</tr>
<tr>
<td>Head Append List(Merge)</td>
<td>+ 5</td>
</tr>
<tr>
<td>Tail Append List</td>
<td>+ 6</td>
</tr>
<tr>
<td>Tail Append List(Merge)</td>
<td>+ 62</td>
</tr>
<tr>
<td>Delete List</td>
<td>+ 193</td>
</tr>
<tr>
<td>Initialise List</td>
<td>+ 160</td>
</tr>
<tr>
<td>Flatten List</td>
<td>+ 160</td>
</tr>
<tr>
<td>Check List Length</td>
<td>+ 159</td>
</tr>
<tr>
<td>Re-Build List</td>
<td>+ 293</td>
</tr>
<tr>
<td>Initialise Processor</td>
<td>4</td>
</tr>
<tr>
<td>Get 21 bit free page</td>
<td>3</td>
</tr>
<tr>
<td>Get 24 bit free page</td>
<td>5</td>
</tr>
<tr>
<td>Return free page</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6.8: Cycle Count for experimental Algorithms

The numbers of this example suggest that the number of the micro-code words needs to be larger than $2^{10}$ in order to offer an acceptable functionality. This means that follow addresses require a minimum of 11 bits and consequently a follow base format of 8 bit.
The ‘Register Set’ Field

The final component of the co-processor micro-word, Figure 6.7, is intended to reduce the number of control lines. In Section 5.1 it was identified that several list manipulations need to keep track of two list structures. To support this need, the datapath was designed containing two independent register blocks holding the information about the source and target lists. The data transfers, used to implement the list manipulating algorithms, do not require facilities to move data directly between the two register blocks. It is therefore possible to use a generalised form of data transfers, and specify the manipulated register set separately. So a data access included in the Data Model will only contain the general function of a register related to a list structure. A separate field in the cycle description specifies whether the particular register of the source or target list structure is activated. As a consequence of the register set selector being allocated in the cycle description, it is not possible to have combinations of data transfers manipulating both list structures executed in parallel. During the micro-program design process it was confirmed that this kind of parallel transfer is rare, as normally the results of operations on one list structure lead to branches in the instruction flow before the second list structure can be manipulated. Situations where an access to both register blocks could be an advantage include a simultaneous initialisation of registers in both blocks. Unless these instructions are frequently repeated the runtime overhead for two separate cycles does not justify the additional width of the nano-code required to address both register blocks during one cycle.

6.4.2 Extraction Process

After defining the structure of micro-code and nano-code words, it is necessary to generate the binary patterns to simulate the controller functionality and to produce and operate the final processor. All the necessary information is held in the database representation of the micro-program. However, the information held in the database uses completely different concepts to structure and order the data. In the database the individual cycles are grouped according to the function they implement and to
the order of input, while the micro-words of the co-processor have to be arranged in a way that allows a compact micro-code store. The conversion process from the database representation to binary patterns for the micro-word and nano-word storage is performed by a program performing the following stages:

1. **Allocate branch destinations**
   It was already stated that the allocation of micro-code words is mainly guided by the fact that all destinations of a branch have to be in neighbouring micro-code words. The allocation of branches with multiple destinations can leave gaps which can be filled by branches with fewer possible successors. The allocation process therefore first assigns the most complex branches to appropriate locations. After all branches of the current complexity level are assigned to suitable micro words, the process continues with branches of the next lower complexity. During the whole allocation process an array structure is used to identify already reserved micro-code words and which cycles will be allocated to these locations.

2. **Extract original cycles**
   Due to the way the micro-program is stored in the database, the first occurrence of original cycles cannot be predicted. A special filter identifies all original cycles and generates a translation table, relating the database internal cycle number to the nano-code word implementing the functionality.

3. **Complete start cycles**
   To enable a simple activation of algorithms implemented in the co-processor, all initial cycles will be stored in a separate area located in the first words of the micro-code store. For each of the start cycles the location of the successive cycles and the activated nano-code word are identified using the data structures generated during the first two steps. The sequencing information is then used to calculate the follow base and the modifier. Together with the information about branching condition and the used register set the fields then can be combined to form the contents of the corresponding micro-code word.

4. **Complete intermediate cycles**
   For each cycle not representing a start or final cycle, the location of its suc-
cessive cycles and the activated nano-code word is determined as for start cycles. The calculation of the micro-word contents also follows the principles described for start cycles. In order to place the contents in the corresponding micro-words it will be necessary to scan the complete allocation list to find all occurrences of this cycle.

5. Complete final cycles
As final cycles will have no successors, the corresponding micro-words will be identified by a zero value for the follow address. This zero follow address then can be used to identify the end of an instruction sequence, returning the control flow to the calling processor. The identification of the other micro-word components and their allocation is identical with the process for intermediate cycles.

6. Print micro-code
After all micro-code words are allocated to particular locations and then filled with values for their individual components, it is finally possible to produce output in binary or hexadecimal format. This output can then be used in simulations as well as to load the micro-code into the memory of a processor.

7. Generate and print nano-code
Generating the information placed the nano-code words requires the filtering of the database for control line activations during individual original cycles. The translation from cycle identifiers used in the database to numbered nano-code words can be achieved with the table generated in step two.

The conversion program, as in Appendix D, is written in the 'C' programming language and includes the required database information in form of include files generated with special database tools. The main program only consists of basic data-type definitions and a few basic conversion routines. Meanwhile the include files, generated with a database report writing tool, represent the actual calculations of the program and can be quite long. The database extraction process is basically a

\footnote{As a cycle can be branch destination of several other cycles, there can be multiple copies of the same micro-word (see Figure 6.8).}
search process using several relations of the Data Model. The output is produced in 'C' syntax and represents a long sequence of repeated code segments manipulating the individual data components extracted from the database. The functions supplied in the main 'C' program are used inside this code to search the auxiliary data structures converting database structure into micro-code and nano-code words.

During the process of allocating branch destinations to locations in the micro-code store, the structure of the algorithms can lead to two different situations. In the first case multiple instructions share all successors of a branch. The placement function therefore first checks if the same combination of successors has already been used by an earlier branch, before allocating the branch destinations to a new location. While this case reduces the size of the resulting microcode, the next situation causes the microcode to be longer. As the branch frequency of co-processor cycles is quite high, and the structure of algorithms normally represents a complex graph, situations will occur where one cycle is successor of multiple, different branches. Normally the allocation algorithm requires a duplication of the cycle, as the neighbouring successors do not allow a jump to an existing location of the cycle (see Figure 6.8: the context of Inst 11 as one of four possible branch destination of Inst 2 does not match the neighbours of its appearance in Inst 1, therefore Inst 11 has to be allocated to two different locations in the micro-code representation).

However, this duplication of cycles will only happen for one cycle, as during the next allocation of successors the first case will guarantee that the two instruction streams are combined again. Generally this duplication aspect of the allocation method will cause a micro-code representation requiring more words than the number of cycles in the original micro-program.

6.4.3 Gate-level Simulation

The main purpose of this level of simulation, after a successful behavioural simulation confirmed the functionality of the datapath and the micro-program, is to verify the functionality of the controller. In particular this means the functioning of the
6.4. Controller Design

Instruction Sequence: Micro-Code: 
Branch Destination Allocation Cycle Contents Completion

<table>
<thead>
<tr>
<th>Addr Contents</th>
<th>Addr Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>??00 empty</td>
<td>??00 encoded Inst10</td>
</tr>
<tr>
<td>??01 empty</td>
<td>??01 encoded Inst11</td>
</tr>
<tr>
<td>??10 empty</td>
<td>??10 encoded Inst12</td>
</tr>
<tr>
<td>??11 empty</td>
<td>??11 encoded Inst13</td>
</tr>
</tbody>
</table>

Figure 6.8: Allocation of microcode words

sequencing algorithm and the correctness of the binary patterns for micro-code and nano-code extracted from the database.

Simulation Environment

While it is obviously necessary to create a new simulation model for the controller, the gate level simulation can re-use the datapath model and the description of the external hardware used in the behavioural simulation. Using the same list manipulations as in the behavioural simulation simplifies the comparison of simulation results and helps to verify that the simulated controller produces the required results. The tasks starting individual list manipulations require only small modifications to start the sequencing process instead of initialising auxiliary variables of the controlling tasks as in the behavioural simulation.

In the simulation model the regular structures of the micro-code and nano-code store and the sequencer PLA are implemented as memory areas of corresponding size. To implement the functionality, files are loaded into these memory areas. After
an initial verification of the controller structure this concept allows the testing of different micro-code configurations, as this requires different files to be read into the memory areas. To install a different set of instructions in the processor, it is only necessary to re-run the database extraction process, extracting only selected algorithms. As a result of this extraction process, two new files containing micro-code and nano-code are generated. The files describing the AND- and OR-planes of the sequencer PLA should not be modified, as this would require a new encoding of branching conditions and the generation of the follow address offset.

6.4.4 Micro-code Replacement

In Chapter 2 a range of alternatives hardware concepts for the implementation of the Writable Control Store (WCS) were discussed. The process of replacing the micro-coded instruction set of the co-processor varies in its complexity depending on the chosen implementation of the WCS. When the different concepts mentioned in Chapter 2 are reviewed under this aspect, the following alternatives can be found:

- Implementations using PROM technology to store the micro-code will only allow a change of functionality by replacing the co-processor with an equivalent device originally programmed with a different set of algorithms. This means that a separate device will be required for each possible set of instructions needed for a particular application.

- The use of EPROM technology will normally require the machine to be switched off during a micro-code change. While some EPROM technologies would only require additional hardware complexity to electrically erase and re-program the device, other, less sophisticated, alternatives would require the processor to be removed from the machine. However, with all concepts it will be possible to replace the micro-code stored on-chip, requiring only one chip to implement a variety of different instruction sets.

- When the micro-code store is implemented in RAM, it will be possible to change the instruction set without physical access to the co-processor. In the
simplest case, the required instruction set is down-loaded into the micro-code store while the architecture is booted. Whether it is possible to replace the micro-code while the computer is in use will depend on the exact structure of the controller and the way co-processor functions are called by processes requiring a certain function implemented in the co-processor.

The process of replacing the micro-code in a co-processor offering the necessary architectural support requires two steps. First the micro-program of the required algorithms has to be extracted from the database. During this process it is important to consider the fixed nano-code of the co-processor. During the second step the compacted micro-code is placed in the micro-code store of the processor. The EPROM technology requires to erase the old contents before the new code can be written using a specialised programming facilities. In a co-processor where the micro-code store is implemented in RAM, the down-loading will be performed by the either the CPU or the co-processor itself. When the CPU is responsible for installing the micro-code, the micro-code store of the co-processor has to be mapped to the linear memory space. Placing a new instruction set in the co-processor means just to write new values into a number of specialised memory locations. In the alternative scenario, where the co-processor itself places the micro-code, a special sequence of hardwired instructions loads the values from a particular memory location. This sequence can be triggered by a processor-reset or another specialised command.

As already mentioned in Section 3.3, the activation of co-processor functions should be performed via software exceptions. The replacement of the co-processor instruction set therefore always has to be accomplished by modifications to the related exception handlers, so that calling processes know about the availability of co-processor functions and how to activate them.

6.4.5 Operation Mode of the Co-processor

In order to support the flexible instruction set required from the co-processor, it is not possible to use fixed instruction codes for particular algorithms. This will prohibit a co-processor from snooping the instruction stream of the CPU and identifying
instructions intended to be executed by the co-processor. Instead it will be necessary for the active processes to know about the existence of the co-processor and its functionality. Whenever a function implemented by the co-processor is required, the running process has to activate a system call which then tells the co-processor what has to be done. In case the requested algorithm is currently not available, the operating system has to execute the function in software or to update the co-processor instruction set.

The process of informing the co-processor about the required function will normally include the provision of a number of parameters in specific registers. After all parameters are transferred, the final step of the calling process will be to supply the co-processor with the instruction code. When this instruction word contains a special indicator, changing the status of the co-processor, this represents an elegant method to activate the device. A single bit of the instruction register could be used as such an indicator, as a logic zero would represent a state where the co-processor waits for an instruction to arrive. As soon as the bit is set, the co-processor becomes active and arbitrates the bus system. The next step is then to execute the first step of the function required by the calling process. The micro-program will automatically use all the supplied parameters and complete the algorithm. When a final cycle is reached, the co-processor will cancel the activation flag, return bus control to the CPU and wait for the next instruction word to be placed by a calling process. The calling process can then retrieve the resulting value and the eventual error code form the appropriate registers inside the co-processor.

6.5 Summary

While Chapter 5 concentrated on the datapath required in a list manipulating co-processor, this chapter deals with the development of the whole device and in particular its controller. The different development stages of the example co-processor identified several problems discussed in the earlier chapters, and led to a steady process of refining the knowledge about the requirements and design alternatives for a list manipulating co-processor.
6.5. Summary

The initial simulation of co-processor algorithms using a high level programming language helped to identify the register requirements used in Chapter 5. The simulation also gave a good indication about the structure of the list manipulating algorithms and the possible parallelism of the individual register assignments involved in manipulating a list structure. This knowledge was then used in developing the concept of a segmented bus system.

The support of a large and open set of list manipulations was always a major point in the specification of the co-processor. This led to the concept of a Writable Control Store and a micro-programmable instruction set. An analysis of existing micro-code design techniques identified a number of problems which suggested a lack of suitability of the standard techniques used to develop micro-code. Based on this situation, a new design method was developed, using the ideas of databases and manual flowcharting. This new automated micro-code design method was used to capture the micro-programs of the essential list manipulations.

A behavioural simulation of the micro-programs was performed next, mainly to verify the functionality of the algorithms, but also to gain additional information about the requirements for a controller. For this behavioural simulation the database contents were extracted in a simple way, maintaining the data structuring of the used Data Model. This allowed a test of the full functionality of the algorithms without introducing additional simulation complexity for a controller model. Once the functionality of the basic micro-program was checked, it was optimised with help of the database tool, before a final simulation validated the optimisations.

After approving the functionality of the list manipulating algorithms, the next step involved a numerical analysis of the micro-program, to get more detailed information about the requirements for the control part of the co-processor. The minimisation of the controller size was given priority over the speed aspect. A two-level controller with a potentially modifiable micro-code store and a static nano-code seems to allow the realisation of a flexible instruction set, while the size of the controller is still not excessive. To identify the exact format of the micro-code words, it was necessary to define the sequencing technique. In order to support the eight
possible values of the SPRINT tag field, a efficient sequencing mechanism had to support eight-way branches. After analysing all required branch conditions, a sequencing mechanism fulfilling this criterion was developed. Each co-processor cycle determines the address of the next cycle with help of a basic follow address. This basic address is then concatenated with a three bit value generated by the sequencer. In addition to the components required for the sequencing, the micro-word format also includes a component identifying the nano-code instruction to be executed during each cycle.

After the controller format has been identified, the next step was to extract the micro-program information from the database and convert it into a format for the co-processor production process. To verify the extracted information a gate-level simulation of the co-processor was performed. Depending on the implementation of the Writable Control Store the same micro-code extraction process could be used to generate multiple micro-code versions implementing different instruction sets.

A final section explained the interfacing between CPU and co-processor and how the co-processor can be activated to perform the list manipulations required by the calling process.
Chapter 7

Co-processor Evaluation

The purpose of this chapter is to evaluate the concept of a list manipulating co-processor as suggested in this work. The two main aspects of the evaluation cover performance and practicality of such a design. A co-processor makes only sense if it can produce a significant acceleration of the architecture it is intended to enhance. At the same time the design aspects of a co-processor also have to be realistic. This means that a co-processor can only be integrated in an architecture if it can be produced in the suggested form.

7.1 Performance

In order to evaluate a co-processor it would be best to compare the performances of the basic architecture and an architecture including the device. However, building a co-processor just for evaluation purposes is an expensive approach when the same results can be gained by simulation.

As the SPRINT itself is an experimental architecture, there is a limited quantity of system software. In a commercial system using Random Access List Structured Memory, list manipulating functions would be called via the operating system. The current lack of an operating system for the SPRINT architecture makes it difficult to simulate list manipulations called from application level software.
Chapter 7. Co-processor Evaluation

Even so, the availability of an architectural simulator and a monitor for the SPRINT architecture allows the evaluation of list manipulation performance for the architecture. The monitor program [Gon93] allows the simulation of system calls for basic list manipulations (copy, slice, delete, ...), but offers only limited support for special variants (offset changes, level changes, ...). The complete monitor program is written in assembly code, and all functions are implemented as sub-routines, where more complex routines are realised on top of more elemental ones. The monitor, as a software solution, uses a slightly different approach to memory management, still using a linked list of pages, but counting the number of free pages. As a second difference, the allocation of memory for a new list structure is performed before the list manipulations are started. Despite all these restrictions, it is still possible to use the monitor as a basic model for a performance comparison.

The clock frequency of both the SPRINT CPU and the suggested co-processor will, during complex list manipulations, be limited by the access times of the Random Access List Structured Memory. Therefore it is possible to compare the performance of both alternatives by measuring execution times in processor cycles.

In the case of the SPRINT monitor, simulation results for function calls invoking list manipulating routines and the overhead for recursive calls used to manipulate nested list structures will only give an approximate measure for the amount of calculation in a proper operating system. It is therefore better to judge the performance advantage of the co-processor by comparing the average number of cycles required to manipulate a single list element. The additional overhead for calling the functionality and nesting lists is better considered separately.

7.1.1 Basic List Copy

The first function to evaluate the efficiency of the co-processor is the function implementing a basic duplication of a, possibly nested, list structure.

When the monitor program is used to copy non-nested lists, the time demand to copy a list depends on the level of the list structure. As the program does not
7.1. Performance

Consider offsets, it assumes each list to have the minimal possible level. The inner loop of the list copy algorithm used in the monitor makes efficient use of the SPRINT CPU resources, and gives a good impression of the speed that can be achieved when copying lists in software. Table 7.1 lists the number of cycles required to copy a single list element. In addition it shows the overhead for the allocation of additional pages depending on the number of list elements.

<table>
<thead>
<tr>
<th>List Level</th>
<th>Cycles / Element</th>
<th>Overheads for Page Allocation Bottom Level</th>
<th>Mid Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
<td>$\left\lceil \frac{\text{#Elements}}{32} \right\rceil \times 10$</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>$\left\lceil \frac{\text{#Elements}}{32} \right\rceil \times 12$</td>
<td>$\left\lceil \frac{\text{#Elements}}{1024} \right\rceil \times 8$</td>
</tr>
</tbody>
</table>

Table 7.1: Time demand for a list copy in SPRINT

The list copy algorithm designed for the co-processor supports the use of list offsets, and offers more sophisticated manipulations during the basic process of copying a list. In addition to the information for the software solution, Table 7.2 shows the overhead faced by the co-processor to recursively copy a nested list.

<table>
<thead>
<tr>
<th>List Level</th>
<th>Cycles for Setup+End Element</th>
<th>Overheads for Page Allocation and List Nesting Bottom Level</th>
<th>Mid Level</th>
<th>per nested List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7 + 2</td>
<td>5</td>
<td>-</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>9 + 2</td>
<td>$\left\lceil \frac{\text{#Elements}}{32} \right\rceil \times 6$</td>
<td>-</td>
<td>29</td>
</tr>
<tr>
<td>3</td>
<td>12 + 2</td>
<td>$\left\lceil \frac{\text{#Elements}}{32} \right\rceil \times 6$</td>
<td>$\left\lceil \frac{\text{#Elements}}{1024} \right\rceil \times 5$</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 7.2: Time demand for a list copy in the co-processor

With these cycle counts for copying a single list element, it is also possible to calculate the cost for copying a complete non nested list. Figure 7.1 contrasts software and hardware solutions for copying a flat list with a possible element count between one and the maximal number of 32k.

The obvious jumps in the time demand of the software solution originate in the fact that the number of cycles required to copy a single list element increases with the level of the source list. When the time demand of the software solution
Chapter 7. Co-processor Evaluation

Figure 7.1: Runtime Comparison for List-Copy simulated for the co-processor (lower graph) and a software solution on SPRINT is compared to the co-processor result, through calculation of the speed-up of the co-processor, these results are as shown in Figure 7.2.

Figure 7.2: Speed-up for a List-Copy performed by the co-processor

The small variation of the speed-up that can be seen in the second area, originates in the page breaks occurring in lists of two and three levels. The magnification in Figure 7.3 shows, that an increasing number of elements causes the variations to loose significance.

As mentioned before, it is difficult to estimate the complexity for function calls in a final software solution, as this will depend a lot on the implementation techniques.
7.1. Performance

In the monitor used to simulate the software solution of list copy, the overhead for calling the copy routine amounts to about 500 cycles, the same as for copying a nested list. Meanwhile the calling overhead for a list copy routine implemented in a co-processor can be estimated to be around 50 cycles. This big difference can be explained by the fact that any software solution requires a sub-routine call with all the register save and stack manipulation involved.

The amount of influence this can have on the general speed up achieved by a co-processor is demonstrated by copying a simple nested list structure. The list structures used for this example series, all consist of a top-level list with ten elements. In contrast to the earlier cases where all elements held scalar values, this time the ten elements contain pointers to further lists. Depending on the number of elements in each of the sublists, the speed-up of the list copy can vary considerably. Table 7.3 shows for a small range of list sizes, the number of cycles required to copy the nested list structure and the resulting speed-up achieved by the co-processor.

When the software implementation has to duplicate short sub-lists, the huge overhead, of about 500 cycles, involved in recursively calling the copying routine causes the performance to drop drastically. As the recursive overhead in the co-processor algorithms is relatively small, this improves the speed advantage significantly. When the length of the sub-lists grows, the general speed-up will asymptotically approach an average value for the speed-up of all sub-lists. In the last columns of Table 7.3
Chapter 7. Co-processor Evaluation

<table>
<thead>
<tr>
<th>Elements per Sublist</th>
<th>Cycles Software</th>
<th>Cycles Co-processor</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6296</td>
<td>509</td>
<td>12.37</td>
</tr>
<tr>
<td>10</td>
<td>6946</td>
<td>759</td>
<td>9.15</td>
</tr>
<tr>
<td>50</td>
<td>14449</td>
<td>2909</td>
<td>4.97</td>
</tr>
<tr>
<td>100</td>
<td>23149</td>
<td>5529</td>
<td>4.19</td>
</tr>
<tr>
<td>1000</td>
<td>179049</td>
<td>52209</td>
<td>3.43</td>
</tr>
</tbody>
</table>

Table 7.3: Time demand for a nested list copy

the speed-up gets close to the value of about 3.4, being the limit for a list copy of level-2 lists. For sublists with more than 1024 elements, the speed-up will suddenly grow once more and settle for values of about 3.8, the general copy speed-up for level-3 lists.

Absolute Performance Estimation

So far the performance of the suggested design has only been evaluated in comparison to a software implementation of the same algorithms. While the relative speed-up gained over a software solution is important, absolute performance measures are a second major criterion in judging the functioning of a co-processor. For the design proposed here, the copy rate of memory words is one of the most important performance measures. From the figures in Table 7.2 it can be calculated that the co-processor requires an average of 5.193 cycles to copy a single memory word of a three level list. The basic cycle time of the co-processor depends mainly on the memory cycle time of the List Structured Memory, as the memory access rate will limited the frequency of memory read and write accesses performed. This means that the co-processor with its average copy speed of $5.193 \frac{\text{Cycles}}{\text{Word}}$ or $0.193 \frac{\text{Words}}{\text{Cycle}}$ can achieve a maximum copy rate of $1.93 \times 10^6 \frac{\text{Words}}{\text{sec}}$ when used in an architecture where the main memory has a memory cycle time of 100ns. The achieved copy rate does not depend on the word format of the host architecture, and the same result can be achieved in architectures with wider word formats. Changes of the list structure however will influence the results.
7.1. Performance

7.1.2 List Slicing

As the extraction of sub-lists is considered as another major operation, the related list slicing function of the co-processor is compared with the software implementation. In the co-processor design the slicing functionality is already implemented as a special case of the copy operation: it is only necessary to supply the list traversal with different parameters for the first list element and the number of elements to copy. The software solution has a slightly more complex algorithm compared to the copy operation. The values in Table 7.4 indicate that the processing time for the software solution does not only depend on the number of elements to extract from the original list, but also on the size of this list. The reason for this lies in the fact that the SPRINT translate instruction [Rou90] requires an increasing number of cycles for additional list levels.

<table>
<thead>
<tr>
<th>List Level</th>
<th>Overheads for Page Allocation</th>
<th>Cycles / Element</th>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bottom Level</td>
<td>Mid Level</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>-</td>
<td>-</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>2 1</td>
<td>-</td>
<td>-</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>2 2</td>
<td>[#Elements] * 10</td>
<td>-</td>
<td>19</td>
<td>2</td>
</tr>
<tr>
<td>3 1</td>
<td>-</td>
<td>-</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>3 2</td>
<td>[#Elements] * 10</td>
<td>-</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3 3</td>
<td>[#Elements] / 1024 * 8</td>
<td>-</td>
<td>21</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 7.4: Time demand for a list slice in SPRINT

As the list copy-algorithm used by the co-processor to extract sub-lists is not influenced by the dimensions of the source list, the speed-up compared to a software solution will depend on the level of the source list. Figures 7.4 and 7.5 show the different speed-up values depending on the level of the source list and the number of elements to be extracted.

As for the list copy operation, the SPRINT software solution implementing the list slicing functionality has an initial setup overhead of about 500 cycles and another overhead of 500 cycles for copying nested list. This again will improve the speed-up of a co-processor during the manipulation of short and/or deeply nested lists.
A final comparison is performed to analyse the performance of the co-processor during a list deletion. The fact that this operation has no target list to traverse reduces the individual processing time for a single element, but causes little change to the characteristic time consumption of the software solution. The typical manipulation times for a single list element are shown in Table 7.5.

The algorithm implementing the list deletion in the co-processor also profits
7.1. Performance

Table 7.5: Time demand for a list delete in SPRINT

<table>
<thead>
<tr>
<th>List Level</th>
<th>Cycles / Element</th>
<th>Overheads for Page Allocation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bottom Level</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>-</td>
</tr>
</tbody>
</table>
| 2          | 10               | \[
\frac{\# \text{Elements}_{32}}{32}
\] * 9 | -         |
| 3          | 11               | \[
\frac{\# \text{Elements}_{32}}{32}
\] * 9 | \[
\frac{\# \text{Elements}_{1024}}{1024}
\] * 12 |

Table 7.6: Time demand for a list delete in the co-processor

<table>
<thead>
<tr>
<th>List Level</th>
<th>Cycles for Element</th>
<th>Overheads for Page Allocation and List Nesting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Setup+End</td>
<td>Element</td>
</tr>
<tr>
<td>1</td>
<td>5 + 5</td>
<td>3</td>
</tr>
</tbody>
</table>
| 2          | 5 + 8     | 3       | \[
\frac{\# \text{Elements}_{32}}{32}
\] * 5 | -         | 22               |
| 3          | 6 + 10    | 3       | \[
\frac{\# \text{Elements}_{32}}{32}
\] * 5 | \[
\frac{\# \text{Elements}_{1024}}{1024}
\] * 5 | 25               |

When the time demand of software and hardware version are compared against each other, the speed-up shows again the typical dependency of the list level. The speed-up achieved by the co-processor when deleting a non-nested list is shown in Figure 7.6.

7.1.4 List Flattening and List Re-construction

In addition to the list manipulation algorithms evaluated so far, the performance of algorithms used to flatten and re-construct complex list structures is of major interest. While the functionality of the SPRINT monitor did not allow an effective comparison of software and hardware implementation, it is still possible to give absolute performance estimations. The structure of the inner loop of the list flattening
Chapter 7. Co-processor Evaluation

Figure 7.6: Speed-up for List-deletion performed by the co-processor

algorithm is closely related to the list copying algorithm. However, the algorithm stores the flattened list in a linear memory area and not a list structure. As the linear memory does not require the algorithm to check for overflowing pages and replace the related pointers, the list management overhead of the flattening algorithm is slightly lower than in the copy algorithm.

<table>
<thead>
<tr>
<th>List Level</th>
<th>Cycles for Setup+End</th>
<th>Element</th>
<th>Overheads for Page Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5 + 2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6 + 2</td>
<td>5</td>
<td>( \frac{#\text{Elements}}{32} ) * 2</td>
</tr>
<tr>
<td>3</td>
<td>8 + 2</td>
<td>5</td>
<td>( \frac{#\text{Elements}}{1024} ) * 2</td>
</tr>
</tbody>
</table>

Table 7.7: Time demand to flatten a list in the co-processor

It can be derived from the figures in Table 7.7 that the co-processor requires an average of 5.065 cycles to flatten a single word of a three level list. With the assumption of a memory cycle time of 100ns, as for the evaluation of the list copy algorithm, this corresponds to a maximum processing rate of \( 1.97 \times 10^6 \frac{\text{Words}}{\text{sec}} \). When the flattened list structures are transmitted over a communication system as suggested in Section 3.3.4, the communication device can be supplied with data at a maximum rate of \( 9.87 \frac{\text{Mbyte}}{\text{sec}} \).

Since the algorithm implementing the list re-construction is similar to the flat-
7.1. Performance

tenating algorithm, it is not astonishing that the performance achieved during list re-construction is almost identical.

7.1.5 Other List Manipulations

The process of manipulating list structures normally involves the traversal of the list structure. A datapath designed to optimise the list traversal will result in a systematic improvement of all functions using the facilities. The analysis of the previous functions has shown that the effectiveness of the list traversal, in the co-processor design, is capable of achieving a significant speed-up during the manipulation of single elements in non nested lists. Other algorithms implemented in the co-processor will use the same technique to traverse lists, and should therefore achieve similar speed-up values for the manipulation of single list elements.

It was mentioned before, that there is currently no operating system for the experimental SPRINT architecture. Due to this, it has been not possible to calculate exact values for the overhead involved for calling list manipulating functions both in software and in form of a co-processor function. However, the values obtained from simulations using the SPRINT monitor suggest that a co-processor can reduce the calling overhead by about an order of magnitude, increasing the speed advantage of the co-processor.

The manipulation of nested lists worked out to be another strong area of the co-processor. Compared to the software solution used in the monitor, the co-processor can handle the recursive call of the list manipulating function ten to twenty times faster. This again can produce a significant increase in the achieved speed-up, particularly for short sub-lists and deeply nested list structures.

Overall, the speed-up for complete list manipulations should be higher than the values obtained from the preceding simulations. The degree of improvement will depend on the techniques used to implement the procedures invoking the list manipulations. A second influencing factor will be the structure of the manipulated
lists. The co-processor will gain additional efficiency when manipulating short lists and those with a high amount of nesting.

The results gained for list structures with a maximum of three levels can also be used to predict the performance gain of architectures with different implementations of the Random Access List Structured Memory. As a maximum list length of 32k elements, as in SPRINT, is not satisfying for all applications, improved architectures can be assumed to implement list structures based on more than three list levels. The evaluation presented in this section showed that, with an increasing number of list levels, the speed-up a co-processor can achieve over a list traversal implemented in software improves. It is therefore forecast that this gain of speed-up will be also available in architecture with four, five or even more list levels.

7.2 Practicality

The evaluation of the feasibility of a chip design is based on two major aspects. The first point covers the connectivity of the co-processor with the surrounding architecture, while the second point deals with the amount of chip area required to implement the given functionality.

7.2.1 General Connectivity of Co-processors

The number of pins connecting a list manipulating co-processor with the surrounding architecture should never cause any complications. Depending on the architectural concept chosen, the number of connecting buses can vary, but should not cause the pin count to reach unrealistic values.

In architectures with indirect communication support (Section 3.3) a co-processor can be expected to have a similar number of pins as the CPU. The co-processor will

\footnote{The use of large pages is a second method to extend the capacity of lists, however the amount of internal fragmentation sets a limit to reasonable page sizes [Rou93]}
require the same connections to data- and address-bus and the other signals of the memory interface. As both devices have to use the same concept of bus arbitration and the central system clock, these connections will also match. The number of power and ground pins should also be similar.

Meanwhile architectures with direct communication support, as suggested in Section 3.3.3, will require the co-processor to have additional pins establishing the connection with the communication switch (Figure 3.4). The size of the additional bus will be the same as the data-bus, i.e. the width of memory words in the Random Access List Structured Memory.

The support of a Writable Control Store (WCS) does not require the designer to include any additional connectivity. The contents of the micro-code store will not be changed while the co-processor is active, therefore the CPU can use the standard buses to install new micro-code.

Connectivity of the co-processor design

As the co-processor, suggested in this thesis, uses indirect communication support, this means that it could be realised with a pin count of about 100. This estimate is based on 64 bits for the address- and data-bus, about 20 other signals for the memory interface, bus arbitration, etc. and finally another 20 pins to offer sufficient power and ground supplies.

7.2.2 Co-processor Size

Considering the area demand of a processor is not as simple as estimating the number of pins. There are multiple factors influencing the final size of a design. However, the only point of real importance at this stage is to show that the suggested co-processor could be produced using a currently available tools and technologies. At the current stage, where it is not certain that the experimental co-processor will be produced, it is not efficient to put significant effort into completing the design process, just to get
exact values about the space demand of the processor. Instead a rough estimation
of the chip area, based on the sizes of major components, is presented.

The following estimations are based on a 1.0 micron CMOS technology. A num-
ber of regular structures were generated as mega-cells using the SOLO 2030 [ES290]

Controller

The major components inside the controller will be the micro- and the nano-code
memory and the sequencing PLA. In addition to these larger components, there
will be a number of small units, such as a module to generate the internal clock,
the memory interface, a bus arbitration unit, reset logic and an unit addressing
memory mapped processor resources. Another unit will have to combine the control
line signals with the timing information and the register set selector encoded in the
micro-code word. The size of these smaller units will not be determined individually,
but considered in the factor multiplying the size of the major components to estimate
the wiring area and other signal logic.

The size estimation for the micro-code RAM is based on a word width of 23 bits,
with 2048 words of micro-code, using a maximum of 256 different nano-code combi-
nations. Due to the limitations of the mega-cell generator it has only been possible to
synthesize RAM blocks containing 512 micro-code words. The complete micro-code
store can be built out of four of these blocks, with a separate decoder selecting the
required block. The dimensions of the generated mega-cell are 2806.2\mu m \times 1933\mu m,
resulting in an overall size for the four RAM blocks of $\sim 21.7\text{mm}^2$.

The size limitations of the mega-cell package also prevented the synthesis of the
nano-code ROM as one block, as the maximal width of a ROM block is set to 128 bit,
whereas the number of control lines exceeded this limit. Again two separate blocks
of nano-code ROM were defined, both blocks will be activated simultaneously. The
generation of a ROM block requires the specification of the fill pattern, describing
all control line activations for the individual nano-code words. At the current stage
7.2. Practicality

no additional simple data transfers, as proposed in Section 6.4.1, have been defined. This meant that no nano-code could be generated for these transfers. Consequently it was not possible to create complete ROM blocks with a full set of nano-words activating a selection of 190 required control lines. However, the results obtained from generating the ROM blocks for the current nano-code, with 125 words and 190 control lines, allow a prediction of a ROM size of less than $5mm^2$ including about 50 words of nano-code adding basic data transfers. With a compression of control lines as suggested in Section 6.4, this size could be reduced further, but for the cost of additional complexity in the final control line generation.

The PLA implementing the sequencer was generated with the same tool. Its dimensions are $272.27\mu m \times 371.5\mu m$, equal to an area demand of $\sim 1.0mm^2$.

Datapath

The datapath section of the co-processor will mainly consist of the registers, the on-chip stack, the bus pre-charging unit and the connection to the outside buses. Additional area will be required to realise the interconnections of the internal bus system.

A central aspect in the design process concentrates on reducing the wiring area required to implement the bus structure, mainly to avoid dogleg shaped bus lines wherever possible. Using a standard cell height throughout the complete datapath is a typical approach to reduce these area intensive shapes. The basic register cell design and the increment decrement unit, used for this area estimation, were taken from the original SPRINT design. As the original cells were designed using a 2.0 micron technology, the cells did not represent current standards. The cells were therefore scaled down to a size conforming with the 1.0 micron design rules used for the development of the mega-cells. This scaling down resulted in cells with a basic height of $45\mu m$. The basic register cell has a width of $66\mu m$, while the same cell with additional incrementing and decrementing facilities is $174\mu m$ wide. During the automatic scaling process it was not possible to reduce all cell components to
minimal dimensions. It is therefore obvious that a complete re-design could produce smaller cells. However, the current cells are more than suitable to estimate area requirements of the major registers in the datapath.

Several registers need to connect sub-sections to corresponding bus segments. This fact requires that these sections are driven by individual control lines, which have to connect to the general control line of the register cell regulating the required input or output. It is therefore not possible to generate large registers by directly connecting the required number of elemental cells. Instead it will be necessary to have small separations between the individual cells of one register. When the height of this additional spacing is calculated with an average of $5\mu m$ per basic cell, this leads to a cell pitch of $50\mu m$ and the overall height for the datapath of 2mm.

To estimate the overall width of the register set, the datapath was divided into smaller sections which were considered separately. As connecting additional control lines can increase the width of registers, the following estimation was based on basic registers of $75\mu m$ and incrementing registers of $200\mu m$ width. The first unit to be considered is the group of registers monitoring a list structure. The block consists of three page-registers, three offset-registers and one length-register, of which the latter four include incrementing facilities. However, as the page- and offset registers operate on non-overlapping bus areas, it is possible to arrange them in one column. This results in an estimated width of 0.8mm per register block.

Two further registers with incrementing facilities are the 24 bit counter supporting list transmission functions and the counter implementing the stack pointer and addressing the individual words of the on-chip stack. As the stack pointer only occupies the lowest five bits of the data-word, it is again possible to combine this register with a second register occupying only higher bit positions, in this case the status register. Together the two counters and the status register can be considered as a block of 0.4mm width.

The registers intended to buffer addresses and general data-words do not require any incrementing facilities, neither do the two registers containing the addresses of
the next free pages. These four registers can be grouped into another block of 0.3mm width.

Finally there are the seven registers realising the interface to the CPU. As none of these registers require incrementing facilities, it is possible to place them in a block of just over 0.5mm width. According to this estimation, it should be possible to integrate all registers in a bus of less than 3mm length, equivalent to less than 6mm$^2$ chip area.

The on-chip stack itself is synthesised as a mega-cell RAM block of 32 words, each 40 bit wide. Together with address decoder the block requires an area of 0.81mm$^2$.

**Combined Estimation**

When the estimations for controller and datapath are combined, the area estimate for the larger blocks sums up to $\sim 35mm^2$ with about 60% being occupied by the micro-code RAM. Of course there will be a significant area demand for wiring and the minor components which were not individually estimated. If these components require twice or even three times the area of the estimated components, this means that the co-processor still could be produced with currently available technology.

**7.3 Summary**

The earlier chapters introduced the idea of a list manipulating co-processor and developed basic concepts for such a device. However, it was also necessary to give evidence that the suggested co-processor can achieve a system acceleration justifying its integration into a computer system.

An estimation of the possible speed-up was performed. The evaluation of co-processor and software algorithms was based on the memory cycle demand for the manipulation of individual list elements. The list manipulations used for the evaluation were important co-processor functions, such as the Copy, Slice and Delete
operations. The speed-up achieved for the manipulation of non-nested list structures is in the area of 3 to 4, depending on the number of list levels involved in the list structure. A co-processor achieves additional speed advantages out of the setup overhead involved in the initiation of list manipulations.

The second point to check was the area demand of the design. The regular memory structures used in the control part of the co-processor represent a significant percentage of the complete processor area. The mega-cell structures used to realise these memory structures were generated in a 1.0 micron technology. On the other hand, basic register cell designs, previously used in the SPRINT CPU, were reduced from the original $2\mu$ format and used to estimate the size of the datapath. The combination of these processor components requires about $35mm^2$, which strongly suggests that the proposed co-processor could be produced with currently available technology.
Chapter 8

Conclusions

8.1 Review

This thesis investigated ways of enhancing computer architectures to make use of the potential Random Access List Structured Memory can offer. As a first step this required a review of existing computer architectures, memory systems and the concept of Random Access List Structured Memory. The random access list, as the basic data structure in this new memory concept, offers good support for a wide field of computing applications, but complex manipulations on these data structures are a challenging task for a standard CPU. A repeated use of load-intensive list manipulations would reduce the performance of a computer system. The enhancement of computer architectures with a specialised co-processor handling these list manipulations has been proposed as a solution to this problem. An evaluation of different application areas was performed to identify list manipulations with a potential of frequent occurrence. Analysing these functions revealed the traversal of list structures as a common element, which also had potential to be accelerated in specialised hardware.

While it was important to identify and analyse the area where architectures can be enhanced, this process is only the first step of realising these improvements. The next stage was to examine the architectural foundations and identify ways of amending the existing architectures. A method of integrating Random Access List
Structured Memory into existing architectures with linear memory was introduced as a first alternative. It was described how such an design could make use of list data structures and how more complex list manipulations could be performed by a specialised controller. When integrating the list manipulating hardware into an architecture based on Random Access List Structured Memory, this will normally be in form of a co-processor. The structure and operation mode of such a device were discussed, leading to an investigation of different ways of integrating a co-processor into the architecture of a processing node. A memory mapped co-processor implementing indirect communication support was chosen for further investigation. Matching the architectural constraints of the SPRINT architecture, algorithms and a corresponding datapath for essential list manipulations were developed. The optimisation and analysis of the resulting micro-program provided data for the design of the control part.

8.2 Achievements

Recalling the developments and results presented throughout this thesis, it is felt that the major research contributions are:

- The development and evaluation of new architectures for processor nodes integrating list manipulating co-processor(s) into an architecture with List Structured Memory. During this process both direct and indirect support of inter processor communication were investigated. As the hardware complexity of a processor node with indirect communication support is significantly lower, this concept was proposed as basic construct for a list manipulation co-processor.

- The creation of a model which allows the addition of Random Access List Structured Memory to existing computer architectures with linear memory. The main component of this model is a controller translating pseudo-linear addresses into a representation used to address data words in the list structured memory. Depending on the complexity of this controller, hardware resources for complex list manipulations can be included in such a device.
8.2. Achievements

- An evaluation of different approaches for implementing the memory management in an architecture with Random Access List Structured Memory. The major aspects of memory management in this environment are the management of free memory pages and the memory demand of algorithms manipulating nested list structures. A simple linked list of free pages was identified as the most appropriate approach for the memory administration problem. Three different approaches were introduced to handle the uncertainty related to the memory demand during the manipulation of nested list structures. Depending on the expectations in the system, algorithms can be designed using a 'Trial and Error', 'Demand Prediction' or a combined approach.

- An analysis of various datapath design aspects aimed at a co-processor operating on Random Access List Structured Memory. The outcome of this analysis suggested the use of: purpose bound registers, segmented buses, register internal increment decrement units (instead of a general purpose ALU), shifting facilities integrated into register ports and an on-chip stack of page size.

- The creation of a new methodology for the development of microcode. The core concept of this new method is the use of a database to automate the flowcharting approach of microcode design. Different database tools are used to: model the algorithm description at multiple abstraction levels, capture the algorithms into this Data Model and check the consistency of algorithms. Furthermore database tools can be used for purposes such as: the optimisation of micro-program and the extraction of microcode for simulation and controller generation purposes.

Most of these general design principles were applied during the development of a list manipulation co-processor enhancing the SPRINT architecture. The following results were achieved:

- A collection of essential list manipulation algorithms were developed and simulated at a high abstraction level.

- Using the results of this high level simulation, a basic model of the required datapath was generated. The main features of this basic datapath model
were two blocks of purpose bound registers holding the variables required for an effective traversal of source and target list. The two processor internal buses were divided into subsections, which can be used individually, enabling an increased level of instruction level parallelism during individual processor cycles.

- A more detailed micro-program description of the list manipulation algorithms was developed using the database method. The information gained during this process allowed the refinement of the basic datapath and generate a final description including all connectivity details.

- A size estimation of the main datapath components, such as registers and the on-chip stack, indicated that these components could be realised with an area demand of approximately $7mm^2$ using a $1\mu m$ technology.

- A controller with a two level micro-code memory was designed, after analysing the characteristics of the micro-program structure. Splitting the information into two levels of micro-code and nano-code offered a possibility to reduce the overall size of the controller. While a dynamic replaceable micro-code allows the installation of various algorithms, its size can be kept limited. The nano-code on the other hand can be implemented in a permanent manner, reducing the space requirements.

- Macro-cells implementing the major components of the controller were generated, also using a $1\mu m$ technology. A micro-code RAM with a capacity of 2048 words and an overall size of $\sim 21.7mm^2$ constitutes by far the largest single unit within the controller. A 180 word nano-code memory, implemented as ROM, activating combinations of the 190 control lines will require less than $5mm^2$.

- With an estimated chip area of $35mm^2$ for the major components of controller and datapath, it can be foreseen that a realisation of the co-processor is possible with current technologies, such as the $1\mu m$ design rules used in the area estimation.
8.3. Outlook

• Comparing the list manipulation performance with similar algorithms performed in software, a speed-up factor of 3 to 4 was identified for the co-processor. The comparison is based on the fact that CPU as well as co-processor will be limited by the speed of the underlying memory system.

• An absolute performance estimation of the list copying algorithm produced a copy rate of $1.93 \times 10^6$ Words/sec, based on a memory system with a cycle time of 100ns. Similar calculations for the performance of the list flattening algorithm suggest that the co-processor could convert lists at a rate of $9.87 \frac{Mbyte}{sec}$.

8.3 Outlook

The work presented in this thesis demonstrated the usefulness of a list manipulating co-processor. At the same time it raises further questions and brings up research topics in a number of areas, such as:

• Additional development related to the designed list manipulating co-processor, analysing structural improvements as well as algorithm design and extended performance estimations.

• Investigation of hardware architectures performing list manipulations on Random Access List Structured Memory with other list structures, such as (cyclic) graphs.

• Studies related to the use of Random Access List Structured Memory as an extension of existing architectures, particularly the design of controllers for complex list manipulations as suggested in Section 3.1.

• Extension of the database tools used during the development of the suggested processor, to enable users to design personal micro-code for a processor. It will be necessary to refine the database tools in a way that does not require specific knowledge about the Data Model and the low level processor details.
Chapter 8. Conclusions

The design of the co-processor has been performed with an initial target of a full custom VLSI implementation. However, this kind of designs are quite complex and time consuming. Further studies should therefore investigate in how far simpler design techniques could be used for the design of similar devices. The availability of Field Programmable Gate Arrays (FPGA) with a steadily growing complexity might be one alternative, even if aspects such as the stable structure of the datapath do not draw maximum profit of the FPGA advantages.

The availability of an operating system for the SPRINT architecture would enable a better evaluation of complete list manipulations called from a user environment. Especially as this would allow the comparison of the influence of co-processor invocation overheads with the complexity of software solutions.

A final area would be the development of additional, more specialised, list manipulation algorithms to investigate the demand for extended datapath facilities and extra data transfers resulting in an extension of the nano-code.
Appendix A

Possible Co-processor Instructions

1. List-Duplicate:

   **Purpose:** Create an exact copy (Level, Offset and Length identical) of the referenced list.

   **Parameter:** List-pointer of original list.

   **Return Values:** List-pointer of new list.

   **Error Codes:**
   - 1: No 21-bit page available as root of new list.
   - 2: No page available for internal use in generated list.
   - 3: Overflow of top level page offset.
   - 4: No page available to expand stack.
   - 5: No 21-bit page available for new nested list.

2. List-Transpose:

   **Purpose:** Generate a copy with a different offset.

   **Parameter:** List-pointer and new Offset.

   **Return Values:** List-pointer of new list.

   **Error Codes:** *see List-Duplicate*
3. List-Change:

**Purpose:** Copy whole list, but with new level and offset.

**Parameter:** List-pointer, new Level and Offset.

**Return Values:** List-pointer of new list.

**Error Codes:** see List-Duplicate

**Remarks:** Calling procedure has to ensure that

\[ \text{Length} + \text{new Offset} \geq 32^{\text{new Level}}. \]

4. Part-Copy:

**Purpose:** Build new list from a part of the referenced list.

**Parameter:** List-pointer, Start-element, Number of elements, new Level and Offset.

**Return Values:** List-pointer of new list.

**Error Codes:** see List-Duplicate

**Remarks:** Calling procedure has to ensure that

\[ \text{Number of elements} + \text{Offset} \geq 32^{\text{new Level}}. \]

5. Head-Append-Full:

**Purpose:** Insert a copy of List1 as head of List2.

**Parameter:** List-pointer 1 and 2.

**Return Values:** List-pointer of expanded list.

**Error Codes:**

- 2 : No page available for internal use in generated list.
- 3 : Overflow of top level page offset.
- 4 : No page available to expand stack.
- 5 : No 21-bit page available for new nested list.

**Remarks:** 'Offset2 \geq \text{Length1}'.

**Alternative:** complete copy List1 tail append List2 as two steps.
6. Head-Append-Part:

**Purpose:** Insert a copy of a part of List1 as head of List2.

**Parameter:** List-pointer 1 and 2, Start-element 1, Number of elements 1.

**Return Values:** List-pointer of expanded list.

**Error Codes:** see Head-Append-Full

**Remarks:** 'Offset2 ≥ Number of elements1'.

**Alternative:** part-copy List1 tail append List2 as two steps.

7. Head-Merge-Append:

**Purpose:** Insert List1 as head of List2, deleting the top level and using the lower levels of the original list.

**Parameter:** List-pointer 1 and 2.

**Return Values:** List-pointer of expanded list.

**Error Codes:**

2 : No page available for internal use in generated list.

3 : Overflow of top level page offset.

**Remarks:** 'Offset2 ≥ Length1'.

**Alternative:** complete copy List1 tail append List2 as two steps.

8. Tail-Append-Full:

**Purpose:** Add a copy of List1 to the end of List2.

**Parameter:** List-pointer 1 and 2.

**Return Values:** List-pointer of expanded list.

**Error Codes:** see Head-Append-Full

**Remarks:** 'Offset2 + length2 + Length1 ≥ 32**Level2'.

9. Tail-Append-Part:

**Purpose:** Add a copy of a part of List1 to the end of List2.

**Parameter:** List-pointer 1 and 2, Start-element 1, Number of elements 1.

**Return Values:** List-pointer of expanded list.

**Error Codes:** see Head-Append-Full

**Remarks:** 'Offset2 + Length2 + Last-element1 - Start-element1 + 1 ≥ 32**Level2'.

10. Tail-Merge-Append:

**Purpose:** Add List1 to the end of List2, deleting the top level and using the lower levels of the original list.

**Parameter:** List-pointer1 and 2.

**Return Values:** List-pointer of expanded list.

**Error Codes:** see Head-Merge-Append

**Remarks:** Offset2 + length2 - length1 $\leq$ 32 ** Level2

11. Delete-List:

**Purpose:** Set free all pages of a list and overwrite all elements of these pages with a initialisation value.

**Parameter:** List-pointer and initialisation Value.

**Return Values:** none.

**Error Codes:**
   3 : Overflow of top level page offset.
   4 : No page available to expand stack.

12. Init-List:

**Purpose:** Set all 'leaves' of a tree of lists to a certain value.

**Parameter:** List-pointer and initialisation Value.

**Return Values:** List-pointer.

**Error Codes:** see Delete-List

13. Init-List-Part:

**Purpose:** Set all leaves within an area to a certain value.

**Parameter:** List-pointer, Start-element, Number of elements and initialisation Value.

**Return Values:** List-pointer.

**Error Codes:** see Delete-List
14. Page-Long:

Purpose: Supply 24bit page.

Parameter: none.

Return Values: Page Address.

Error Codes:

1 : WARNING: No 24-bit page available, returned 21-bit page.
2 : No page available.

15. Page-Short:

Purpose: Supply 21bit page.

Parameter: none.

Return Values: Page Address.

Error Codes:

1 : No 21-bit page available.

16. Free-Page:

Purpose: Add free page to the appropriate list.

Parameter: Page-address.

Return Values: none.

Error Codes: none.

Remarks: Check address-bits 21-23, to decide whether the page is a 21bit or a 24bit page.

17. Init-Pages-Reset:

Purpose: Reset co-processor, set 21bit or 24bit mode and write the addresses of the first 21bit (and 24bit) page(s) into the appropriate register(s).

Parameter: Mode-indicator,(24bit page-address) and 21bit page-address.

Return Values: none.

Error Codes: none.
Appendix A. Possible Co-processor Instructions

18. Check-Length:

**Purpose:** Check the number of elements in a multi level list.

**Parameter:** List-pointer.

**Return Values:** Number of elements.

**Error Codes:**
1. **3:** Overflow of top level page offset.
2. **4:** No page available to expand stack.

19. Check-Length-Part:

**Purpose:** Check the number of elements of the sublist.

**Parameter:** List-pointer, Start-element, Number of elements.

**Return Values:** Number of elements.

**Error Codes:** *see Check-Length*

20. Flatten-List:

**Purpose:** Convert a multi level list into a linear representation.

**Parameter:** List-pointer, Address of sending buffer.

**Return Values:** Number of elements.

**Error Codes:** *see Check-Length*

21. Flatten-List-Part:

**Purpose:** Convert a part of a multi level list into a linear representation.

**Parameter:** List-pointer, Address of sending buffer, Start-element and Number of elements.

**Return Values:** Number of elements.

**Error Codes:** *see Check-Length*

22. Build-List:

**Purpose:** Re-construct a multi level list from a linear representation.

**Parameter:** Address of receiving buffer, wanted Level, wanted Offset.

**Return Values:** List-pointer.

**Error Codes:** *see List-Duplicate*
Appendix B

Description of the High Level Simulation Environment

The purpose of this Appendix is to give a more detailed description of the high level simulation environment used to simulate the algorithms of a list manipulating co-processor for the SPRINT architecture.

B.1 List Traversal

It was already explained that multiple data structures had to be defined on the host architecture in order to simulate algorithms operating on a list structured memory. This section gives examples of definitions used to model SPRINT data words on a linear memory, followed by register definitions and manipulation routines.

B.1.1 Basic Definitions

As each of the forty bit wide SPRINT words consist of various components and the internal tag bits indicate how to interpret the lower 36 bits of the data word, this natural separation was also used for the modelling of the representation in C data structures. Within these lower 36 bits the main target was to build models for the
three known data formats (list-pointer, integer and empty) that also allow a wider utilisation to host a variety of user defined data formats.

The basic SPRINT list-pointers were modelled (Figure B.1) as records containing the three basic components, where each component had an individually defined data type.

```c
typedef short int value_5bit;
typedef char *addr_21bit_page;
typedef unsigned short length_list;

typedef struct {
    length_list length;
    addr_21bit_page listpointer;
    value_5bit offset;
} list_frame;
```

Figure B.1: C representation of a list-pointer

With similar definitions for the integer and empty values, the lower 36 bits of a SPRINT word can be expressed as a union of the basic representations (Figure B.2). The general definition of the data type intended to hold empty words would allow the user to simulate additional data types using this format to store any required data expressible within 36 bits. If necessary the components of the user defined data types have to be converted into two binary patterns of 32 and 4 bits length.

```c
typedef struct {
    short int empty;
    int int_val;
} int_frame;

typedef struct {
    short int bits32_35;
    int bits0_31;
} empty_frame;

typedef union {
    list_frame list;
    int_frame int_v;
    empty_frame empty;
} data_frame;
```

Figure B.2: C Data Frame for SPRINT words

Finally this 'data frame' is extended with variables for the tag field and the copy bit to form a structure representing a complete SPRINT data word (Fig-
B.1. List Traversal

The tag field will always indicate to the simulation program which format is used inside the data_frame, so that the components of any of the three basic representations can be manipulated correctly.

```c
typedef struct {
    short int copy_no_use;
    short int tag;
    data_frame data;
} word;
```

Figure B.3: C representation of SPRINT words

In a similar structured approach a purpose bound registers set, holding the variables of an individual list structure, was defined (Figure B.4). This allowed the provision of multiple sets of registers accessed in a uniform way.

```c
typedef short int count_5bit;
typedef unsigned short length_register;
typedef mem_40bit_word *reg_21bit_addr;
typedef mem_40bit_word *reg_24bit_addr;

typedef struct {
    length_register L_reg;
    count_5bit A_reg;
    count_5bit B_reg;
    count_5bit C_reg;
} counter_registers;

typedef struct {
    reg_21bit_addr lv11_page_addr;
    reg_24bit_addr lv12_page_addr;
    reg_24bit_addr lv13_page_addr;
} page_registers;
```

Figure B.4: C registers to hold a list structure

The basic data types of the C programming language do not offer sufficient support for data-types not matching the standard byte boundaries of most existing computers. Therefore another way of simulating the data manipulations of these registers had to be chosen.
Appendix B. Description of the High Level Simulation Environment

B.1.2 Register Manipulation

A special set of procedures was designed to make sure the register abstractions introduced so far were manipulated according to certain rules. These rules express the available manipulations for the individual registers. A second purpose of these procedures was to guarantee that the C data types always maintain values representing valid states of the corresponding registers. Figure B.5 shows a selection of these procedures.

Once these data structures for all data type of the co-processor had been defined together with their basic manipulation methods, the first step towards simulating a basic list traversal was performed. The next step was to prepare the simulation of a list traversal. A abstract algorithm for the inner loop of the list traversal is shown in Figure B.6.

As the skeleton of a traversal algorithm (Figure B.6) shows, there is an immediate need for a processor status word (Figure B.7) to monitor the overflow results whenever an offset register is incremented. This is necessary, as the overflow of an offset register indicates the end of the current page, and consequently has to trigger the replacement of the current page at this level of the list.

The model of the Processor Status Word furthermore contains a flag to indicate that the length register reached a zero value, which means that the traversal of the current list is completed. Other flags indicate whether the stack pointer has reached a 0 or 31 value. Finally the PSW includes the flags required to support the memory management functions included in the co-processor functionality.

The version of the traversal algorithm in Figure B.6 does not distinguish between the two different sets of registers for the source and the target list, but such a separation can be added without difficulties. The use of the second list structure will completely hidden in the tasks performing the required list manipulation for the individual algorithms.
int ass_5bit(reg, val)
    count_5bit *reg;
    value_5bit val;
    {
        if ((val >= 0) && (val < 32))
            {
                *reg = val;
                return(0);
            }
        else return(1);
    };

int ass_24bit(reg, val)
    reg_24bit_addr *reg;
    addr_24bit_page val;
    {
        int i_val;
        i_val = (int) val;
        if ((i_val % 32) == 0)
            {
                i_val = (i_val >> 5);
                if ((i_val >= 0) && (i_val < 524288))
                    {
                        *reg = (mem_40bit_word *) val;
                        return(0);
                    }
                else return(1);
            }
        else return(1);
    };

int inc_5bit(reg)
    count_5bit *reg;
    {
        (*reg)++;
        if (*reg > 31)
            {
                *reg = 32;
                return(1);
            }
        else return(0);
    };

Figure B.5: C procedures for registers manipulation

As soon as the abstract task manipulation, as used in the algorithm of Figure B.6, involves the traversal of a nested list, a stack mechanism had to be included into the simulation model.
Appendix B. Description of the High Level Simulation Environment

```c
int inner_trav_list()
{
    signal_line    ass_err;

    ass_flag(finished,dec_length(&L_reg));
    while (finished == 0)
    {
        ass_flag(finished,dec_length(&L_reg));
        if (lvl3_overflow)
        {
            ass_flag(lvl2_overflow,inc_5bit(&B_reg));
            if (lvl2_overflow)
            {
                ass_flag(lvl1_overflow,inc_5bit(&A_reg));
                if (lvl1_overflow)
                {
                    fprintf(stderr,"Error in memory structure: Level_1 overflow !\n");
                    return(1);
                }
                get_l_newpage(&(lvl2_page_addr),&(lvll_page_addr),A_reg);
            }
            get_l_newpage(&(lvl3_page_addr),&(lvl2_page_addr),B_reg);
        }
        tasks();
        ass_flag(lvl3_overflow,inc_5bit(&C_reg));
    }
    return(0);
}
```

Figure B.6: Skeleton for the list traversal algorithm

### B.1.3 Stack Model

To implement a stack model it is first necessary to define the data structures which are going to reside in the stacked data (Figure B.8).

Any stack can consist of four different kinds of elements. As mentioned earlier, the first word of each stack page will always contain a pointer to the preceding page. In addition a flag will indicate whether such a page exists, or whether the current page represents the stack base. The remaining words of each stack page will be made up from a mix of words used to store the information about higher order lists. Three different structures are used to store the compressed data formats as presented in Figure 5.3 to 5.5.

In the SPRINT co-processor environment any memory word will contain any
B.2. Simulation Environment

typedef short int flag;

typedef struct
{
  flag mode_24bit;
  flag free_24page;
  flag free_21page;
  flag finished;
  flag ll_lvl11_overflow;
  flag ll_lvl12_overflow;
  flag ll_lvl13_overflow;
  flag ll_lvl21_overflow;
  flag ll_lvl22_overflow;
  flag ll_lvl23_overflow;
  flag ll_lvl3_overflow;
  flag stack_full;
  flag stack_empty;
} status_word;

Figure B.7: Preliminary C Definition of the Processor Status Word

of three different formats: a regular SPRINT word, an address of a page (being an element of a top- or intermediate level page) or any of the co-processor defined stack words. This is represented in the definition of the forty bit wide memory word (mem_40bit_word) in the simulation model. Any memory page and the on-chip stack, as a special page, are defined as an array of 32 forty bit words.

To simulate the process manipulating the stack a further set of special functions were supported. The first function has the responsibility for the setting the base element of the on-chip stack. The more important functions are the three pairs of push and pop functions to manipulate the stack. Inside these push and pop operations additional functions provide the increment or decrement of the stack pointer register. Depending on the stack pointer value, the push and pop operations initiate additional save and restore functions moving the on-chip stack to and from memory.

B.2 Simulation Environment

Once the basic co-processor data-types and the corresponding registers and manipulation procedures had been defined, a full list traversal algorithm could be implemented. In order to really manipulate lists a number of elements had to be added
Appendix B. Description of the High Level Simulation Environment

typedef struct {
    addr_24bit_page old_page_addr;
    flag
} stack_base_word;

typedef struct {
    addr_24bit_page lvl3_page_addr;
    length_list length;
    value_5bit c_count;
} stack_1_word;

typedef struct {
    addr_24bit_page lvl2_page_addr;
    value_5bit b_count;
    addr_21bit_page lvl1_page_addr;
} stack_2_word;

typedef struct {
    value_5bit a_count;
    status_word status;
} stack_3_word;

typedef union {
    stack_base_word stack_base;
    stack_1_word stack_w1;
    stack_2_word stack_w2;
    stack_3_word stack_w3;
} co_word;

typedef union {
    word sprint_w;
    co_word co_w;
    addr_24bit_page n_lvl_pointer;
} mem_40bit_word;

typedef mem_40bit_word mem_page[32];

count_5bit stack_count;
mem_40bit_word stack[32];

Figure B.8: C stack definitions

to the simulator.

B.2.1 Creating Lists

In order to simulate the manipulation of list structures, the simulator had to have access to such a list structure. As the instructions of the co-processor are only intended to manipulate existing information and create new lists out of existing ones,
the simulator had to include a separate component to acquire lists. The simulator is therefore equipped with an option allowing the used to enter list structures. After entering the level and initial offset of the top level list and the number of elements, the user can chose the type of each list element and supply the related information when necessary. While there is no additional information required for empty data words, the simulator will ask for a numeric value for any integer data word. Global references and user defined data types are not supported in the simulation.

When the user generates a nested list structure by entering a list data type, the data entry part recursively enquires for list level, offset and length, generates the related list-pointer for the old list and then starts to ask for the elements of the sublist. After all elements of a sublist are entered, control will return to the information entry of the next higher list. An additional option allows the user to reduce the effort while generating long lists, and fills the remaining elements of the current list with increasing numeric values. At the same time this option represents an effective method of producing non-nested lists of any size. The only input required are list level, the initial offset and list length; then during the specification of the first list element the ‘auto-fill’ mode is triggered and all list elements are filled with numeric values. Appendix C gives an example protocol of a short simulation run using these facilities.

B.2.2 Tasks

Since various algorithms will perform different manipulations on the individual list elements encountered during a list traversal, the general simulation mechanism for the list traversal (Figure B.6) calls the ‘tasks’ function to implement the necessary functionality. Depending on the type of each list element and the instruction manipulating the elements, different procedures are activated to perform the required manipulation.

For example, an element containing a list-pointer will normally cause the current list information to be written onto the stack and replaced by the initial information
of the sublist. But as soon as the current instruction involves the duplication of list information, this requires the stacking of the information related to the target list.

B.2.3 Manipulating Lists

When manipulating list structures with help of the simulator, the user interface allows the user to select the required function. Depending on the chosen function the user will be asked for the associated parameters and input lists (Appendix C). With help of these values, the simulator then sets up the environment and manipulates the list structure(s). Some common parameters are related to the level and offset of target lists, others limit the traversal algorithm to particular sections of a list structure. A detailed description of the parameters required for individual functions is given in Appendix A, introducing an example instruction set for a co-processor.

Result Buffering and Analysis

After a list manipulation has been simulated the environment saves the resulting list in an array of buffers. With help of these buffers the user also can view the results of list manipulations and verify the manipulations performed by the simulator. A special list traversal is performed, displaying the offsets of each list element and the decoded value. It is possible to use the buffered lists as input for following list manipulations.

List Output

In order to support later simulations at lower levels of abstraction and to ease comparison of simulation results, the simulator is equipped with an option to write lists into files. The files produced with this option contain hexadecimal values of all words in the pages holding the list structure. Additional comments are generated to document the relationship of the pages in a more complex list.
Appendix C

Protocol of a High-level Simulation

The following protocol gives a short impression of the high-level simulator used to analyse the requirements of the basic list manipulation algorithms.

What do you want to do next?
(1) Run CP-Instruction
(2) Check Buffers
(3) Dump Buffer
(4) Quit

Please enter your choice: 1

Choose one of the following instructions!
(1) List-Duplicate
(2) List-Transpose
(3) List-Change
(4) Part-Copy
(5) Head-Append-Full
(6) Head-Append-Part
(7) Tail-Append-Full
(8) Tail-Append-Part
(9) Send-List
(10) Send-List-Part
(11) Receive-List
(12) Delete-List
(13) Init-List
(14) Init-List-Part
(15) Page-Long
(16) Page-Short
(17) Free-Long
(18) Free-Short

215
(19) Init-Pages-RESET

Please enter the code of the instruction to be executed: 3

Please specify the list you want to duplicate!

You can choose from these possibilities:
  (1) Enter a new list
  (2) Use the result of an earlier instruction

Please enter the code of your choice: 1

Enter the wanted level (1-3): 1

******* New List Level ***********************

How many elements?: 5

Which offset? (0-31): 22

Low page_buffer 104736

Element 1: Enter type: (1) - List (2) - Empty (3) - Integer: 1
  Enter the value: 3
Element 2: Enter type: (1) - List (2) - Empty (3) - Integer: 2
Element 3: Enter type: (1) - List (2) - Empty (3) - Integer: 1

Enter the wanted level (1-3): 1

******* New List Level ***********************

How many elements?: 15

Which offset? (0-31): 0

Low page_buffer 105664

Element 1: Enter type: (1) - List (2) - Empty (3) - Integer: 9

******* List Level Finished ***********************

Element 4: Enter type: (1) - List (2) - Empty (3) - Integer: 9

******* List Level Finished ***********************

Please enter the offset for the target list: 30

Please enter the level for the target list: 2

What do you want to do next?
  (1) Run CP-instruction
  (2) Check Buffers
  (3) Dump Buffer
  (4) Quit

Please enter your choice: 2
The buffers contain these lists:

( 0) Level : 2 Length : 5 Offset : 30
( 1) Level : 1 Length : 0 Offset : 0
( 2) Level : 1 Length : 0 Offset : 0
( 3) Level : 1 Length : 0 Offset : 0
( 4) Level : 1 Length : 0 Offset : 0
( 5) Level : 1 Length : 0 Offset : 0
( 6) Level : 1 Length : 0 Offset : 0
( 7) Level : 1 Length : 0 Offset : 0
( 8) Level : 1 Length : 0 Offset : 0
( 9) Level : 1 Length : 0 Offset : 0
(10) Level : 1 Length : 0 Offset : 0
(11) Level : 1 Length : 0 Offset : 0
(12) Level : 1 Length : 0 Offset : 0
(13) Level : 1 Length : 0 Offset : 0
(14) Level : 1 Length : 0 Offset : 0
(15) Level : 1 Length : 0 Offset : 0
(16) Level : 1 Length : 0 Offset : 0
(17) Level : 1 Length : 0 Offset : 0
(18) Level : 1 Length : 0 Offset : 0
(19) Level : 1 Length : 0 Offset : 0
(20) Level : 1 Length : 0 Offset : 0
(21) Level : 1 Length : 0 Offset : 0
(22) Level : 1 Length : 0 Offset : 0
(23) Level : 1 Length : 0 Offset : 0
(24) Level : 1 Length : 0 Offset : 0
(25) Level : 1 Length : 0 Offset : 0
(26) Level : 1 Length : 0 Offset : 0
(27) Level : 1 Length : 0 Offset : 0
(28) Level : 1 Length : 0 Offset : 0
(29) Level : 1 Length : 0 Offset : 0
(30) Level : 1 Length : 0 Offset : 0
(31) Level : 1 Length : 0 Offset : 0

If you want to see a specific list enter the buffer : 0

A : 0 B : 0 C : 30 Integer Value: 1
A : 0 B : 0 C : 31 Empty Element
A : 0 B : 1 C : 0 List of Level 1
A : 0 B : 0 C : 0 Integer Value: 1
A : 0 B : 0 C : 1 Integer Value: 2
A : 0 B : 0 C : 2 Integer Value: 3
A : 0 B : 0 C : 3 Integer Value: 4
A : 0 B : 0 C : 4 Integer Value: 5
A : 0 B : 0 C : 5 Integer Value: 6
A : 0 B : 0 C : 6 Integer Value: 7
A : 0 B : 0 C : 7 Integer Value: 8
A : 0 B : 0 C : 8 Integer Value: 9
A : 0 B : 0 C : 9 Integer Value: 10
A : 0 B : 0 C : 10 Integer Value: 11
A : 0 B : 0 C : 11 Integer Value: 12
A : 0 B : 0 C : 12 Integer Value: 13
A : 0 B : 0 C : 13 Integer Value: 14
Appendix C. Protocol of a High-level Simulation

A : 0  B : 0  C : 14  \quad \text{Integer Value: 15} \\
A : 0  B : 1  C : 1  \quad \text{Integer Value: 4} \\
A : 0  B : 1  C : 2  \quad \text{Integer Value: 5}

What do you want to do next?
(1) Run CP-Instruction
(2) Check Buffers
(3) Dump Buffer
(4) Quit

Please enter your choice : 4
Appendix D

Description of the Micro-code Extraction

The purpose of this section is to give a basic impression about the program used to extract the database information and generate the information describing the micro-code and nano-code.

Main Program

```c
#include <stdio.h>
#define Start_value 32 /* leaves space for */
#define maxcases 8 /* start instructions */
#define maxsteps 3000
#include "defines.lis" /* defines number of */
/* -used control lines */
/* -number of highest c.l */
/* -number of unique steps */

typedef struct
{ short unique_step; /* 9 bit */
  short reg_set; /* 1 bit */
  short follow_base; /* 9 bit */
  short follow_mod; /* 3 bit */
  short condition; /* 3 bit */
  long o_id;
} steps;
```
typedef struct
{ short used;               /* 1 bit */
  long oracle_follow_id;
} fol;

typedef struct
{ short used;               /* 1 bit */
  short oracle_step_id;
} uni;

fol test_array[maxsteps];
steps seq_array[maxsteps];

uni uni_array[(maxuniques + 1)];

int o_f_id[maxcases];
int line_conv[(high_linum + 1)];

int nano_matrix[(maxuniques + 1)][(used_lines + 1)];
int comp_matrix[(maxuniques + 1)][(complines + 1)];

int cases;
int count1;
int start_inst;
int search_inst;
int search_block;
int mod, cond, step;

main()
{
  int inc_value;
  int cont;

  for(count1 = 1; count1 <= high_linum; count1++)
  {
    line_conv[count1] = -1;
  }

#include "maplines.lis"
#include "body_gen.lis"
#include "uni_gen.lis"
#include "start_inst.lis"
#include "mid_inst.lis"
#include "end_inst.lis"
print_s_a_i();
#include "nano_fill.lis"
print_nano();
exit(l);
}

int print_s_a_i()
{
  int loop;
  long i_word;

221

```c
printf("%07X \n",0);
for(loop = 1; loop < maxsteps; loop++)
{
    if (seq_array[loop].o_id != 0)
    {
        i_word = (seq_array[loop].unique_step << 16);
        i_word = i_word | (seq_array[loop].reg_set << 15);
        i_word = i_word | (seq_array[loop].follow_base << 6);
        i_word = i_word | (seq_array[loop].follow_mod << 3);
        i_word = i_word | (seq_array[loop].condition);
        printf("%03X ",loop);
        printf("%07X \n",i_word);
    }
}

int print_nano()
{
    int loop_out,loop_in;

    for(loop_out = 2; loop_out <= used_lines; loop_out++)
    {
        for(loop_in = 1; loop_in <= maxuniques; loop_in++)
            printf("%4",nano_matrix[loop_in][loop_out]);
        printf("\n");
    }
}

int find_block()
{
    int count2,check_value;
    int diff;

    check_value = find_line(o_f_id[0],start_value);
    if (check_value == 0)
    {
        return(0);
    }
    else
    {
        diff = 1;
        do
        {
            if (diff == 0)
                check_value = find_line(o_f_id[0],check_value +1);
            diff = 1;
            for(count2 = 1; count2 < cases; count2++)
            {
                if(test_array[(check_value + count2)].oracle_follow_id != o_f_id[count2])
                    diff = 0;
            }
        }
        while ( (diff == 0) && (check_value != 0));
    }
```
Appendix D. Description of the Micro-code Extraction

```c
int insert_cases()
{
    int count3;
    for(count3 = 0; count3 < cases; count3++, count1++)
    {
        test_array[count1].used = 1;
        test_array[count1].oracle_follow_id = o_f_id[count3];
    }
}

int find_line(o_s_id, s_v)
int o_s_id;
int s_v;
{
    int count11;
    int found;
    found = 0;
    count11 = s_v;
    while ( (found == 0) && (count11 <= maxsteps))
    {
        if(test_array[count11].oracle_follow_id == o_s_id)
        {
            found = 1;
        }
        else
        count11++;
    }
    return(count11 * found);
}

int find_uni(o_u_id)
int o_u_id;
{
    int count21;
    int found;
    found = 0;
    count21 = 0;
    while ( (found == 0) && (count21 <= maxuniques + 1))
    {
        if(uni_array[count21].oracle_step_id == o_u_id)
        {
            found = 1;
        }
        else
        count21++;
    }
    return(count21 * found);
}
```

int f_line_id(o_l_id)
int o_l_id;
{
    int val;
    val = line_conv[o_l_id];
    if ((o_l_id > high_linnm) | (o_l_id < 1) | (val < 1))
        printf("Unrecognised control-line !\n");
    return(val);
}
Appendix D. Description of the Micro-code Extraction

Extract of 'body_gen.lis'

```c
inc_value = 8;
count1 = start_value;
cont = 1;
while ( (count1 <= maxsteps) & cont )
{
    if (test_array[count1].used == 1)
        count1 += inc_value;
    else
    {
        cont = 0;
        cases = 3;
        o_f_id[ 0 ] = 2429;
        o_f_id[ 1 ] = 2428;
        o_f_id[ 2 ] = 2431;
        if (find_block() == 0)
            insert_cases();
    }
}
count1 = start_value;
cont = 1;
while ( (count1 <= maxsteps) & cont )
{
    if (test_array[count1].used == 1)
        count1 += inc_value;
    else
    {
        cont = 0;
        cases = 7;
        o_f_id[ 0 ] = 2166;
        o_f_id[ 1 ] = 2167;
        o_f_id[ 2 ] = 2168;
        o_f_id[ 3 ] = 2169;
        o_f_id[ 4 ] = 2170;
        o_f_id[ 5 ] = 2171;
        o_f_id[ 6 ] = 2172;
        if (find_block() == 0)
            insert_cases();
    }
}
```

Extract of 'uni_gen.lis'

```c
count1 = 1;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 1 ;
count1++;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 2 ;
count1++;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 3 ;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 93 ;
count1++;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 123 ;
count1++;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 125 ;
uni_array[count1].used = 1;
uni_array[count1].oracle_step_id = 12440 ;
count1++;```
Appendix D. Description of the Micro-code Extraction

Extract of ‘start_inst.lis’

```c
start_inst = 1;
cases = 1;
cond = 0;
o_f_id[ 0 ] = 7;
search_block = find_block();
if (search_block == 0 )
{
    printf(" Problem, Block not found ! \n");
    exit(0);
}
seq_array[start_inst].reg_set = 0;
seq_array[start_inst].unique_step = find_uni( 6 );
seq_array[start_inst].follow_base = search_block / 8;
seq_array[start_inst].follow_mod = search_block % 8;
seq_array[start_inst].condition = cond;
seq_array[start_inst].o_id = 6;
test_array[start_inst].used = 1;
start_inst++;
cases = 2;
cond = 1;
o_f_id[ 0 ] = 1;
o_f_id[ 1 ] = 11;
search_block = find_block();
if (search_block == 0 )
{
    printf(" Problem, Block not found ! \n");
    exit(0);
}
seq_array[start_inst].reg_set = 0;
seq_array[start_inst].unique_step = find_uni( 10 );
seq_array[start_inst].follow_base = search_block / 8;
seq_array[start_inst].follow_mod = (search_block % 8) & 6;
seq_array[start_inst].condition = cond;
seq_array[start_inst].o_id = 10;
test_array[start_inst].used = 1;
start_inst++;
cases = 3;
cond = 4;
o_f_id[ 0 ] = 13;
o_f_id[ 1 ] = 2;
o_f_id[ 2 ] = 14;
search_block = find_block();
if (search_block == 0 )
{
    printf(" Problem, Block not found ! \n");
    exit(0);
}
seq_array[start_inst].reg_set = 0;
seq_array[start_inst].unique_step = find_uni( 12 );
seq_array[start_inst].follow_base = search_block / 8;
```
seq_array[start_inst].follow_mod = ((search_block % 8) & 4) | 1;
seq_array[start_inst].condition = cond;
seq_array[start_inst].o_id = 12;
test_array[start_inst].used = 1;
Appendix D. Description of the Micro-code Extraction

Extract of 'mid_inst.lis'

```c
cases = 1;
cond = 0;
o_f_id[ 0 ] = 8;
search_block = find_block();
if (search_block == 0 )
{
    printf(" Problem, Block not found ! \n");
    exit(0);
}
for (search_inst = 32; search_inst < maxsteps; search_inst++)
{
    if (test_array[search_inst].oracle_follow_id == 7 )
    {
        seq_array[search_inst].reg_set = 0;
        seq_array[search_inst].unique_step = find_uni( 7 );
        seq_array[search_inst].follow_base = search_block / 8;
        seq_array[search_inst].follow_mod = search_block % 8;
        seq_array[search_inst].condition = cond;
        seq_array[search_inst].o_id = 7;
    }
}
cases = 1;
cond = 0;
o_f_id[ 0 ] = 9;
search_block = find_block();
if (search_block == 0 )
{
    printf(" Problem, Block not found ! \n");
    exit(0);
}
for (search_inst = 32; search_inst < maxsteps; search_inst++)
{
    if (test_array[search_inst].oracle_follow_id == 8 )
    {
        seq_array[search_inst].reg_set = 0;
        seq_array[search_inst].unique_step = find_uni( 8 );
        seq_array[search_inst].follow_base = search_block / 8;
        seq_array[search_inst].follow_mod = search_block % 8;
        seq_array[search_inst].condition = cond;
        seq_array[search_inst].o_id = 8;
    }
}
cases = 3;
cond = 4;
o_f_id[ 0 ] = 26;
o_f_id[ 1 ] = 25;
o_f_id[ 2 ] = 28;
search_block = find_block();
if (search_block == 0 )
```

{ printf(" Problem, Block not found ! \n"); exit(0); }

for (search_inst = 32; search_inst < maxsteps; search_inst++) {
    if (test_array[search_inst].oracle_follow_id == 24 ) {
        seq_array[search_inst].reg_set = 0;
        seq_array[search_inst].unique_step = find_uni( 24 );
        seq_array[search_inst].follow_base = search_block / 8;
        seq_array[search_inst].follow_mod = ((search_block % 8) & 4) | 1;
        seq_array[search_inst].condition = cond;
        seq_array[search_inst].o_id = 24;
    }
}
Appendix D. Description of the Micro-code Extraction

Extract of 'end_inst.lis'

for (search_inst = 32; search_inst < maxsteps; search_inst++)
{
    if (test_array[search_inst].oracle_follow_id == 1 )
    {
        seq_array[search_inst].reg_set = 0 ;
        seq_array[search_inst].unique_step = find_uni( 1 );
        seq_array[search_inst].follow_base = 0 ;
        seq_array[search_inst].follow_mod = 0 ;
        seq_array[search_inst].condition = 0 ;
        seq_array[search_inst].o_id = 1 ;
    }
}

for (search_inst = 32; search_inst < maxsteps; search_inst++)
{
    if (test_array[search_inst].oracle_follow_id == 2 )
    {
        seq_array[search_inst].reg_set = 0 ;
        seq_array[search_inst].unique_step = find_uni( 2 );
        seq_array[search_inst].follow_base = 0 ;
        seq_array[search_inst].follow_mod = 0 ;
        seq_array[search_inst].condition = 0 ;
        seq_array[search_inst].o_id = 2 ;
    }
}
Extract of 'nano_fill.lis'

nano_matrix[find_uni(1)][f_line_id(3)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(4)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(5)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(6)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(7)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(8)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(9)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;
nano_matrix[find_uni(1)][f_line_id(10)] = 1;
nano_matrix[find_uni(1)][f_line_id(31)] = 1;

nano_matrix[find_uni(2)][f_line_id(2)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(4)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(5)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(6)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(7)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(8)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(9)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
nano_matrix[find_uni(2)][f_line_id(10)] = 1;
nano_matrix[find_uni(2)][f_line_id(31)] = 1;
Appendix D. Description of the Micro-code Extraction
Glossary

The following glossary is intended to explain, not define, terms and abbreviations used throughout the thesis.

**Data Model**: Abstract representation of all relations (→), and their correlations, used in a database.

**DFT**: Depth First Traversal is a general technique to systematically manipulate all elements of a tree-shaped data structure. The algorithm evaluates references to nodes at a lower levels before it advances to the next node at the current level.

![Example Tree](image)

*Figure Glossary.1: Example Tree*

When this method is applied to the example tree of Figure Glossary.1, the nodes will be manipulated in the order: 1, 2, 5, 6, 3, 7, 4, 8, 9, 10. It is suggested to apply this traversal technique in list manipulating co-processors, to reduce the amount of data on the stack.
Duplicate Cycle This expression is used in the micro-code development section to identify cycles, where all datapath transactions are identical to the transactions of a previously defined cycle (→ Original Cycle).

Dynamic Array Regular n-dimensional data structure, where each dimension can be modified at run-time.

Element Selector (→ Selector)

Free Page List Data structure used by the system to administer the unused memory pages in the Random Access List Structured Memory. A simple implementation uses a linked list where the first word of each free page contains a pointer to the next free page.

List-pointer Component used in combination with the selector (→) to access data values in Random Access List Structured Memory. The L. can be seen as a complex data-type containing the basic address of the list structure in combination with additional details. Depending on the system architecture the additional components can contain information about the length of a list and element offsets (→). In a tree shaped memory structure, as used throughout the thesis, the basic address held in the L. is the address of the top level page of the tree holding the list.

LFT Layer First Traversal is a second technique to systematically manipulate all elements of a tree-shaped data structure. In contrast to DFT (→), this algorithm will manipulate all elements of the current level, before evaluating the stored references to elements at lower levels. When this method is applied to the example tree of Figure Glossary.1, the nodes will be manipulated in the order: 1,2,3,4,5,6,7,8,9,10.

List Traversal The process of manipulating list elements in a sequential order. Depending on the requirements of the applications a traversal can be performed on partial or complete lists.

Micro-code Is the binary representation of the micro-program (→) stored inside a microprocessor.
Micro-program A M. describes the algorithms available in a microprocessor in a formal notation.

Modifiable Memory Includes all memory technologies where the memory contents can be changed several times. Random Access Memory (RAM) and EPROM variants are the most important classes.

Offset Optional component of a listpointer (→), which enables list structures to be stored in a way where the first list element is not required to be placed in the first location of the allocated memory structure. The main advantage of such a storage technique with offsets lies in the fact that list structures can be head appended by a any number of elements without moving the complete list structure.

Original Cycle This expression is used in the micro-code development to identify cycles, defining a combination of datapath transactions for the first time.

PLA (→ Programmable Logic Array)

Processor Status Word The P. is a special register containing the flags used as branching conditions by the micro-program.

Programmable Logic Array Regular hardware structure to implement combina­tional logic functions.

PSW (→ Processor Status Word)

Random Access List Structured Memory New memory concept supporting both numerical and symbolical computing. Main aspects are dynamic re-size­able array structures and lists with random access to individual elements.

Register Transfer Level Standard level of abstraction used in the description of processor structures.

Relation Standard database term for a table of records describing a particular type of contents stored in the database.

RTL (→ Register Transfer Level)
Selector  Component used in combination with the List-pointer (\rightarrow) to access data values in Random Access List Structured Memory. The S. can be split up into a number of fields equal to the maximum number of list levels in the given architecture. Each of these fields selects the current element in a page used to describe the list hierarchy.

Tag  The concept of tagging memory words was developed to overcome the problems of unstructured memory, where the interpretation of memory words depends completely on software. In a tagged architecture each memory word contains a number of bits describing the type of data held in this particular word.

Writable Control Store  A WCS is a memory space designed to hold the micro-program; depending on the intended functionality of the processor different micro-programs can be loaded into this memory.
Bibliography


Bibliography.


Butterworth-Heinemann are leading international publishers of a range of highly respected academic journals in Applied Science, Information Technology, Engineering, Materials Science, Policy Studies, Social Sciences, Biomedical Sciences and Medicine. Our journals are renowned for their high quality of both content and presentation. Papers are peer-reviewed and all our journals have distinguished editorial boards.

If you would like to receive subscription details, a sample copy or other information about the journal in which this offprint appeared, please complete and return the form provided below. Information about our other titles is available from our price list or catalogue (see form below).

**Notes for Authors**
To obtain details on how to submit papers to one of our journals, please contact: The Group Editor at the Oxford address below (please state the appropriate journal title).

**Reprints**
We can reprint any article appearing in our journals and tailor it to your needs with an advert or logo. For a quotation on your reprint requirements, please contact: The Reprints Department at the Oxford address below.

For further information about any other aspect of our journals, please contact us at:
Butterworth-Heinemann Ltd., Linacre House, Jordan Hill, Oxford, OX2 8DP, UK
Tel: +44 (0) 865 310366
Fax: +44 (0) 865 310898

**Return Form**

Area(s) of Interest: ........................................................................................................................................

Journal Title(s) .......................................................................................................................................

☐ Subscription Details Request
Please send me full subscription details and information about the above journal title(s)

☐ Sample Copy Request Form
Please send me a free sample copy of the above journal title(s)

☐ Journals Price List
Please send me a full price list for all your journals

☐ Catalogue
Please send me a catalogue of your journals

Name .........................................................................................................................................................
Organisation .............................................................................................................................................
Address .......................................................................................................................................................
Postcode / Zipcode ................................................................. Country ............................................................
Telephone .............................................................................................................................. E-mail ............................................................
Fax ............................................................................................................................... Signature ............................................................

Please return this form to:
Butterworth-Heinemann Ltd.
Turpin Distribution Services Ltd.
Blackhorse Road
Letchworth
Herts, SG6 1HN, UK
Tel: +44 (0) 462 672555
Fax: +44 (0) 462 480947

OR
Journal Fulfilment Department
Butterworth-Heinemann US
80 Montvale Avenue
Stoneham
MA 02180, USA
Tel: +1 617 438 8464
Fax: +1 617 438 1479

OFFPRCOVGEN94
Using databases to support the development of microcode

Jens-Uwe Dzikowski and Peter A Rounce

The design and verification of microcode for a processor with a complex instruction set can be a complicated task. In this paper it is shown how a standard relational database can be used at various stages to simplify such a design process, as automatic microcode generation is not always suitable for this process. Using the example of a coprocessor for list manipulation, it is demonstrated how the database can be used to enter the microprogram, check various aspects of consistency, generate simulation patterns for the datapath and finally to generate the optimized microcode.

Keywords: microcode development, databases

The procedure of generating a controller for a particular microprocessor requires the designer to state all the algorithms of the commands of the instruction set in a special description language that is capable of expressing all the transactions performed in the datapath. For each cycle all the necessary transfers and operations have to be described. It is necessary to record the sequencing of the cycles and their I/O behaviour.

In general, the non-automated flowchart method supports these needs and offers a convenient way of designing microcontrollers, but as soon as the instructions become more complex and lengthy, it becomes quite difficult to perform the entire design process by hand. An adaptable tool to support this process of generating a microcontroller is the obvious successor to manual flowcharting. The use of a database system forms the basis of such a tool. Alternative methods for automated microcode design are microcode compilers at various abstraction levels. Microcode compilers provide sophisticated facilities, but they are not generally available and the overhead for their development is not acceptable for many small projects. Even adapting a retargetable microcode compiler to match a particular architecture can be complex. We evaluate these methods more fully later, but neither was chosen for the application discussed here because of the relatively small size of our application (about 15 algorithms involving a total of 2000 cycles).

The database method developed and presented here is built on existing tools and had a short development time. We believe it provides a flexible tool, based on a proven methodology for developing microcode, that is particularly suitable for small and one-off developments. The method offers the designer several capabilities such as simple and comfortable data entry, easy updating and good searching as well as varied output facilities. An important feature is the fact that the data is accessible at all stages of the design and allows the designer to build in automated data consistency checking.

Databases allow the capture of the data relationships of a system within a data model. It is obviously necessary that, in order to be used for the development of microcode, the data model must be able to capture the data relationships of the target system. In the work reported, such a data model (Figure 1) has been applied to capture the functionality of the control part of a coprocessor at several abstraction levels.

The level of abstraction in Figure 1 decreases from left to right with the elements at the far left capturing the high level flow of control in the device, while the relations at the right end capture the lowest level of abstraction, the control signals activated for a given operation.

The database method has been developed for and used in the design of a coprocessor. The purpose of the coprocessor is to provide efficient high speed list management within a novel list-structured memory architecture (Figure 2) as implemented in the SPRINT processor.

The datapath in the coprocessor contains a set of problem oriented registers, but no ALU. Instead it provides data manipulation through selection and merging operations within parallel data transfers on various sections of two buses, and incrementation operations within registers. The design of the control section of the coprocessor requires a special description method that is powerful enough to specify such data transfers. Furthermore the control algorithms require branches with up to eight different successors due to the tagged memory architecture of the list-structured memory. It has proved possible to
The database approach allows the automated implementation of a proven microcode development methodology, which does not require a formal description of the algorithms to be implemented and without the need for large scale software development. A database package offers in readily available software:

- automatic management of data
- availability of a user interface
- powerful data manipulation facilities
- expandable data verification facilities.

All further software development is done at a high level. The design documented here was done with a standard Oracle database system, which provides the mechanism for data storage and the query language for consistency checking, and using SQLFORMS and SQL REPORT, the report-writing facility, for data extraction.

The organization of the rest of the paper is as follows. The next section shows how we used the database to capture the algorithms of the microprograms. The section ‘Support of the data entry process’ gives information about the user interface and the first verification of design data, while these are being recorded in the database. The facilities for analysing and verifying the microprogram are described under the heading ‘Consistency testing’. The connection of the database to a simulator is covered in the section ‘Creating simulation patterns’. In the section ‘Generating microcode’ we explain how to extract the information in a binary format suitable for production purposes. The final section ‘Evaluation’ compares this method with existing techniques and outlines the time needed for the development of such a toolkit.

### DATA MODEL

As the database approach should not limit the flexibility of the flowchart method, it is desirable to generate a data model that keeps all the necessary information in independent relations. This enables easy changes of a relation and its connected tools without influencing the other parts more than necessary. The rest of this section will describe the approach chosen for the example system, to demonstrate the underlying idea.

The example system is a coprocessor design divided as standard into a datapath and a controller. Operations within the datapath just consist of transfers between units and processing units: the example coprocessor has two buses in the datapath, but has only simple registers and registers with associated incrementers, although units such as an ALU could be supported in the model. The role of the controller is to sequence datapath operations to execute the functions of the coprocessor.

A top down view of the coprocessor is the large scale function performed by it, i.e. as seen by the adjoining units. At the highest abstraction level recorded in the data model are the algorithms that implement these large scale functions. The next lower abstraction level is the description of the individual cycles used to implement these algorithms. Below this is the description of the parallel transfers in the datapath that make up a single cycle, while the lowest abstraction level is the description of the control signals activated to produce a single transfer in the datapath.

The data model has to capture all these abstraction levels of the coprocessor model. The way this has been done is shown in Figure 1.

There are three ‘library’ relations; ‘relation’ being a standard database term for a table of records holding a particular kind of information in a data model. The central relation COMPONENTS holds records on all available transfers on the datapath. A single record within COMPONENTS details one transfer, indicating source, target and bus section used, where bus section is either a group of bus signals or a whole bus. The LINES relation lists all control lines leading into the datapath to control its operations. The last ‘library’ relation STEPS details all coprocessor ‘cycles’, where a single cycle is made up of a set of parallel-executed transfers on the datapath; details of the transfers are recorded in the COMPONENTS relation.

The connection between STEPS and COMPONENTS is
Databases to support the development of microcode: J-U Dzikowski and P A Rounce

recorded in the CONSISTS relation through the specification of all 'components' records used in each 'step' record. The ACTIVATES relation does the same for specifying all the 'lines' records used in each transfer. Essentially, ACTIVATES details all control lines activated for a particular transfer while CONSISTS records all the parallel transfers performed during a particular coprocessor cycle.

The last relation of Figure 1 is FOLLOW which records the sequencing of the 'steps' records in a larger algorithm. Thus FOLLOW is used to specify all the large scale functions of the coprocessor.

In the next section the individual relations of the data model are presented in detail. To illustrate the application in a single example a short start-sequence of a longer algorithm is presented at the various abstraction levels. With reference to the original flowchart method, the STEPS relation represents the outer frame of individual state in level 2 flowcharts, as in Figure 3.

It contains fields to describe:

- the type of external access (i.e. memory accesses outside the coprocessor)
- how to determine the next cycle (i.e. no successor, direct successor or branch) with details of the last two cases in FOLLOWs
- the number of duplicates of a cycle: a duplicate is another cycle within STEPS with the same transfer specification, but different in other aspects e.g. NEXT_STATE, REGISTER_SET etc. This is only held in a cycle which holds the original specification of the common transfers for a duplicate cycle a reference to the record holding the common transfer specification
- name of the instruction of which the cycle is a part
- register set holds information that allows parameterization of some cycles.

The attributes of the STEPS relation are shown in Table 1. The description of four steps forming the beginning of the 'Head-append' list operation is given in Example 1. Later examples present closely related information at the other abstraction levels.

More detailed information of the next cycle is held in the FOLLOW relation, as the STEPS relation only contains information about how the successor(s) is determined. The details of this relation are given in Table 2.

Example 2 illustrates the sequencing information for the previously shown steps held in the FOLLOW relation: the 'Condition' field indicates a register bit or bits to be tested for the contents shown in the 'Value' field.

The STEPS relation holds the transfer specification indirectly through the STEP_ID of the CONSISTS relation. The CONSISTS relation, as seen in Table 3, relates the STEP_ID with the transfer in COMPONENTS.

There will be several CONSISTS records containing the same STEP_ID, but different COMP_IDs, so that CONSISTS identifies a set of transfers from COMPONENTS performed in parallel within one cycle.

The identification of all the transfers performed during step 526 is shown in Example 3. There is no such information for steps 527 to 529, as these are copies of earlier

Table 1  Attributes of the STEPS relation

<table>
<thead>
<tr>
<th>Step_ID</th>
<th>ACCESS_TYPE</th>
<th>Duplicates</th>
<th>NEXT_STATE</th>
<th>IDENTICALSTEP</th>
<th>REGISTER_SET</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>526</td>
<td>NA</td>
<td>2</td>
<td>H_APPEND</td>
<td>1</td>
<td>BR</td>
<td>0</td>
</tr>
<tr>
<td>527</td>
<td>NA</td>
<td>2</td>
<td>H_APPEND</td>
<td>0</td>
<td>BR</td>
<td>47</td>
</tr>
<tr>
<td>528</td>
<td>DR</td>
<td>2</td>
<td>H_APPEND</td>
<td>0</td>
<td>BR</td>
<td>224</td>
</tr>
<tr>
<td>529</td>
<td>DR</td>
<td>2</td>
<td>H_APPEND</td>
<td>0</td>
<td>DI</td>
<td>328</td>
</tr>
</tbody>
</table>

Example 1  Description of four cycles

Table 2  Attributes of the FOLLOW relation

<table>
<thead>
<tr>
<th>STEP_ID</th>
<th>FOLLOW_ID</th>
<th>CONDITION</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>526</td>
<td>527</td>
<td>(Sta[38],Sta[37],Sta[36])</td>
<td>0</td>
</tr>
<tr>
<td>526</td>
<td>528</td>
<td>(Sta[38],Sta[37],Sta[36])</td>
<td>1</td>
</tr>
<tr>
<td>526</td>
<td>529</td>
<td>(Sta[38],Sta[37],Sta[36])</td>
<td>1</td>
</tr>
<tr>
<td>527</td>
<td>44</td>
<td>counter_change</td>
<td>0</td>
</tr>
<tr>
<td>527</td>
<td>45</td>
<td>counter_change</td>
<td>1</td>
</tr>
<tr>
<td>528</td>
<td>44</td>
<td>counter_change</td>
<td>0</td>
</tr>
<tr>
<td>528</td>
<td>45</td>
<td>counter_change</td>
<td>1</td>
</tr>
<tr>
<td>529</td>
<td>630</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Example 2  Related sequencing information

Table 3  Attributes of the CONSISTS relation

<table>
<thead>
<tr>
<th>STEP_ID</th>
<th>COMP_ID</th>
<th>INFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>526</td>
<td></td>
<td></td>
</tr>
<tr>
<td>527</td>
<td></td>
<td></td>
</tr>
<tr>
<td>528</td>
<td></td>
<td></td>
</tr>
<tr>
<td>529</td>
<td></td>
<td></td>
</tr>
<tr>
<td>530</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Microprocessors and Microsystems Volume 18 Number 9 November 1994  539
cycles, while step 526 is an original step representing the first step of the ‘Head-append’ instruction.

Table 4 gives a description of the COMPONENTS relation. For the coprocessor modelled, transfers may not only perform a simple register-register transfer where each bit of the source goes to the same bit position in the target; there are some transfers where the bit position changes from source to target. This is the reason for the TO_START and TO_END field in COMPONENTS. Example 4 demonstrates the way in which individual transfers are described in the COMPONENTS relation. The ‘Transfer condition’ field is blank for all shown transfers.

In the LINES relation, see Table 5, the names of the various control lines are stored together with additional information. Example 5 gives an impression of some control lines in the design.

Finally the ACTIVATES relation, Table 6, just connects the transfers with the control lines. Obviously there can be many relations with duplicate COMP.IDs and many with duplicate LINE.IDs, providing many-to-many relations.

A small section of the activation records is shown in Example 6: it can be seen that in the example component 217 activates a number of control lines with LINE.IDs 53, 54,118, 119 and 120.

This basic data model can be easily adapted to match the needs of other processors just by adding new attributes and relations as required. Although the data model embeds a model of the controller as breaking algorithms into cycles, while step 526 is an original step representing the first step of the ‘Head-append’ instruction.

Table 4 Attributes of the COMPONENTS relation

<table>
<thead>
<tr>
<th>Comp</th>
<th>Source Register low</th>
<th>high Condition</th>
<th>Used Target Bus Register low</th>
<th>high</th>
</tr>
</thead>
<tbody>
<tr>
<td>09 INSTR</td>
<td>0 4</td>
<td>B2 C.REG</td>
<td>0 4</td>
<td></td>
</tr>
<tr>
<td>90 INSTR</td>
<td>0 4</td>
<td>B2 OUTPUT</td>
<td>0 4</td>
<td></td>
</tr>
<tr>
<td>144 ZERO</td>
<td>0 0</td>
<td>B1 A.REG</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>145 ZERO</td>
<td>1 1</td>
<td>B1 A.REG</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>146 ZERO</td>
<td>2 2</td>
<td>B1 A.REG</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>147 ZERO</td>
<td>3 3</td>
<td>B1 A.REG</td>
<td>3 3</td>
<td></td>
</tr>
<tr>
<td>148 ZERO</td>
<td>0 0</td>
<td>B1 B.REG</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>149 ZERO</td>
<td>1 1</td>
<td>B1 B.REG</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>150 ZERO</td>
<td>2 2</td>
<td>B1 B.REG</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>151 ZERO</td>
<td>3 3</td>
<td>B1 B.REG</td>
<td>3 3</td>
<td></td>
</tr>
<tr>
<td>154 ZERO</td>
<td>0 0</td>
<td>B1 BUFFERWORD</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>157 ZERO</td>
<td>1 1</td>
<td>B1 BUFFERWORD</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>158 ZERO</td>
<td>2 2</td>
<td>B1 BUFFERWORD</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>159 ZERO</td>
<td>3 3</td>
<td>B1 BUFFERWORD</td>
<td>3 3</td>
<td></td>
</tr>
<tr>
<td>161 ZERO</td>
<td>21 23</td>
<td>B1 BUFFERWORD</td>
<td>21 23</td>
<td></td>
</tr>
<tr>
<td>162 ZERO</td>
<td>24 39</td>
<td>B1 BUFFERWORD</td>
<td>24 39</td>
<td></td>
</tr>
<tr>
<td>216 INPUT.2</td>
<td>5 20</td>
<td>B2 BUFFERWORD</td>
<td>5 20</td>
<td></td>
</tr>
<tr>
<td>217 INPUT.2</td>
<td>21 39</td>
<td>B2 L.REG</td>
<td>21 39</td>
<td></td>
</tr>
<tr>
<td>218 INPUT.2</td>
<td>36 38</td>
<td>B2 STATUS</td>
<td>36 38</td>
<td></td>
</tr>
<tr>
<td>219 INPUT.2</td>
<td>5 20</td>
<td>B2 OUTPUT</td>
<td>5 20</td>
<td></td>
</tr>
<tr>
<td>220 INPUT.2</td>
<td>36 38</td>
<td>B2 OUTPUT</td>
<td>36 38</td>
<td></td>
</tr>
<tr>
<td>281 ZERO</td>
<td>39 39</td>
<td>B2 OUTPUT</td>
<td>39 39</td>
<td></td>
</tr>
</tbody>
</table>

Example 5 Some of the activated control lines

SUPPORT OF THE DATA ENTRY PROCESS

Once the designer has an outline of how his algorithms will work and the form of the datapath, there comes the point where a first version of the design must be entered into the database. To support this process and to reduce the risk of errors it is useful to have a set of data entry forms performing first checks on the entered data and offering additional help. In the case of the data model of Figure 1, the order of data entry could be: COMPONENTS, STEPS, CONSISTS, FOLLOW, LINES and ACTIVATES. The first step will be to...
enter an initial set of data transfers as basic elements for the later construction of the cycles. A data entry form for the relation COMPONENTS is shown in Figure 4.

The data entry form for the COMPONENTS relation provides the following functions:

- Automated generation of a unique key (COMPS_ID) for each record.
- A check of whether the least significant bit (LSB) and the most significant bit (MSB) for both source and target register are within the allowed range and have the proper order.
- Checks the register names used in FROM_REG and TO_REG with the help of a hidden database relation CHOICES(TABLE,FIELD,VALUE) for valid values.
- Verify the name of the bus used.

All the tests and functions are implemented as SQL commands in the description of the data entry form. The entry of the cycle descriptions into the STEPS relation also has its own data entry form. Unique key generation for the STEP_ID is automatic and validity tests on the appropriate value, no successor, or only one successor for the particular cycle. An additional function allows a query on the existing FOLLOW entries for the current cycle.

The database system can be used to define certain more complicated functions, as for example a function to create a copy of a sequence of cycles already existing in a different instruction. These functions can automatically create new records for the cycles within the new instruction, the IDENTICALSTEP attribute and the DUPLICATES of the original cycle will automatically be updated, and the necessary records for the FOLLOW relation created (if both cycles are in the copied sequence).

Similar data entry forms can be created to enter the names and descriptions of the control lines into the LINES relation as well as the functionality of transfers into the ACTIVATES relation. With the SQL REPORT` tool it is quite easy to extract various formatted database listings during the data collection, and create final listings in the form of a 'Pretty printer', as in Example 7.

**CONSISTENCY TESTING**

As the complete information about the structure of the microprogram is kept in the database, it is possible to check all aspects of consistency at all stages of the design. This section will just mention a few interesting tests to show the advantages of the database usage. Tests can cover simple matters in a single relation as well as more complex facts where the information is split over several relations.

A simple example in the early stage of the data collection is testing that there are no pairs of transfers having the same source and target register and overlapping bus areas, i.e. all transfers are disjoint. This is necessary to enable the proper assignment of control lines in the later design.

During the process of specifying the cycles the designer can use the database system to search for cycles without transfers, cycles with overlapping bus signals or cycles where a register appears as source as well as target register. There is a large number of similar tests performed, and it is always for the designer to decide what are design errors or problems requiring special attention in the further design process.

After the information about the sequence of the individual cycles has been entered, the designer can check if all cycles have a predecessor and the right number of successors.

A further level of tests enables the designer to reduce the size of the microcode. This is done by searching for identical cycles that have not been discovered during the design process. It is also possible to identify cycles having transfers that are a subset of another cycle; it is then up to...
Execution continues with:

Register-Set 2

---

ZERO (00 - 00) ----> B1 ----> A_REG (00 - 00)
ZERO (01 - 01) ----> B1 ----> A_REG (01 - 01)
ZERO (02 - 02) ----> B1 ----> A_REG (02 - 02)
ZERO (03 - 04) ----> B1 ----> A_REG (03 - 04)
ZERO (00 - 00) ----> B1 ----> BUFFERWORD (00 - 00)
ZERO (01 - 01) ----> B1 ----> BUFFERWORD (01 - 01)
ZERO (02 - 02) ----> B1 ----> BUFFERWORD (02 - 02)
ZERO (03 - 04) ----> B1 ----> BUFFERWORD (03 - 04)
INPUT_2 (05 - 20) ----> B2 ----> BUFFERWORD (05 - 20)
ZERO (21 - 23) ----> B1 ----> BUFFERWORD (21 - 23)
ZERO (24 - 39) ----> B1 ----> BUFFERWORD (24 - 39)
ZERO (00 - 00) ----> B1 ----> B_REG (00 - 00)
ZERO (01 - 01) ----> B1 ----> B_REG (01 - 01)
ZERO (02 - 02) ----> B1 ----> B_REG (02 - 02)
ZERO (03 - 04) ----> B1 ----> B_REG (03 - 04)
INSTR (00 - 04) ----> B2 ----> C_REG (00 - 04)
INPUT_2 (21 - 35) ----> B2 ----> L_REG (21 - 35)
INSTR (00 - 04) ----> B2 ----> OUTPUT (00 - 04)
INPUT_2 (05 - 20) ----> B2 ----> OUTPUT (05 - 20)
INPUT_2 (36 - 38) ----> B2 ----> OUTPUT (36 - 38)
ZERO (39 - 39) ----> B2 ----> OUTPUT (39 - 39)
INPUT_2 (36 - 38) ----> B2 ----> STATUS (36 - 38)

Execution continues with:

case { (Sta[36], Sta[37], Sta[38]) }
    3'b000 next_step = 527 ;
    3'b001 next_step = 528 ;
    3'b010 next_step = 529 ;

527 Register-Set 2 (copy of step 47) 
-----

Execution continues with:

case { counter_change }
    0 next_step = 44 ;
    1 next_step = 45 ;

528 Register-Set 2 (copy of step 224)
-----

Execution continues with:

case { counter_change }
    0 next_step = 44 ;
    1 next_step = 45 ;

529 Register-Set 2 (copy of step 328)
-----

Execution continues with:

next_step = 530 ;

Example 7 Collected information of earlier examples

the designer to decide whether it will have any unwanted side effects, or if a replacement with the more powerful cycle is possible. Another useful test helps locate identical final sequences of instructions, because these only have to be included once.

CREATING SIMULATION PATTERNS

Once the designer has finished an initial version of his microprogram, it is essential to see whether the algorithms are working as expected. The best way to test the algorithms is to export the data describing the controller into simulation patterns for an existing description of the datapath and run a combined simulation.

There are multiple ways to generate these simulation patterns, the main difference being the level of abstraction and hierarchy maintained. The solution with the lowest level of abstraction would be to combine all the database information and flaten it into a set of binary testvectors representing the control line data, and then just activate these stimuli with an appropriate timing. The alternative with the highest abstraction level maintains the hierarchy of the data model and performs a sequence of interpretations at the various levels, mapping the individual cycles down to transfers and finally control line signals. Other solutions would be a combination of these two cases, flattening certain levels, but still mapping down at a different level. Which solution is chosen depends on the preferences of the designer and on the available simulator, but maintaining the hierarchy of the data model tends to reduce the amount of software development needed to create the stimuli patterns.

Whichever is chosen, the database report writing facility is the primary mechanism for extracting the test data. Maintaining the data model in the test data requires the minimum amount of work in creating the output specification since separate specifications only need to be created for each relation in the database. At the other extreme more complicated specifications, with nested information retrieval from other relations, are required. Another advantage of the maintained hierarchy is the fact that the algorithms can be easily analysed at each level, which normally eases debugging.

The simulation of the example coprocessor was written in VERILOG, using the approach that maintains the hierarchy of the data model. VERILOG 'tasks' offer an ideal way to implement the interpretation mechanism of information at various levels. The 'function' mechanism of VERILOG helps to implement the sequencing of individual cycles, as well as providing an efficient mechanism for mapping duplicated cycles onto the original cycle without code duplication.

SQL REPORT was used to output textfiles in VERILOG notation which are then automatically included into the model combining the controller with the datapath. The UNIX 'make' mechanism can be used to keep these files consistent with the database before the start of a new simulation run.

Assuming there exists a working VERILOG model of the datapath, a structured behavioural model of the controller can be driven with data extracted from the database. The steps to get such a controller are:

- Generate a set of VERILOG tasks activating the control lines, one task per transfer, as in Example 8.
- Create a second set of tasks (Example 9) for each unique cycle, stating all the tasks for the transfers during this cycle.
- Build a function (Example 10) activating the task that implements the cycle to be executed, or (for all cycles being duplicated) the task that implements the original cycle.
- Generate a second function (Example 11) to determine the next cycle according to the branching conditions and the processor status.

```verilog

Example 8 Activations for component 121

task comp_121;
    begin
        l_reg_d = 1;
        l_reg_i = 1;
        l_reg_sel_alt = 1;
        stk_read = 1;
    end
endtask
```

Example 7 Collected information of earlier examples
task step_526;
begin
  @(negedge clk2)
  fork
    delete_all;
    #1 comp_89;
    #1 comp_90;
    #1 comp_144;
    #1 comp_145;
    #1 comp_146;
    #1 comp_147;
    #1 comp_148;
    #1 comp_149;
    #1 comp_150;
    #1 comp_151;
    #1 comp_156;
    #1 comp_157;
    #1 comp_158;
    #1 comp_159;
    #1 comp_161;
    #1 comp_162;
    #1 comp_216;
    #1 comp_217;
    #1 comp_218;
    #1 comp_219;
    #1 comp_220;
  join
endtask

Example 9 Components executed during step 526

task instruction;
input inst;
integer inst;
begin
  case(inst)
  526 : begin
      set_l_2n = 0;
      step_526;
    end
  527 : begin
      set_l_2n = 0;
      step_47;
    end
  default : $display("Unknown step \n");
  endcase
endtask

Example 10 Extract of function activating individual steps

• Finally, include all the tasks and functions together with some coordinating code (for timing and resetting) into the VERILOG simulation.

To perform any changes in the microprogram it is only necessary to update the database, run the extraction of the modified file and restart the simulation, as the new version of the files will be automatically included in the VERILOG model.

GENERATING MICROCODE

As the data model does not require a particular structure of the controller the designer can decide this structure at any stage of the design process. There is a well known variety of more or less specialized controller models, and the designer will face the same decision as in other design methodologies. According to the chosen structure the contents of the database can then be used to generate microcode in the required format.

In the coprocessor for the SPRINT architecture chosen for this example, the controller is split into two sections, a microword storage and a nanocode block, as shown in Figure 5.

The microcode section is the major component responsible for the sequencing of the cycles of the individual algorithms, while the nanocode part holds the detailed description of the control signals activated. The decision to split the controller has been based on the number of control lines (~200) and the number of microcode words required to implement the algorithms (~2000 for a basic instruction set). The two-stage controller of the example
reduces the area required to roughly a quarter of the original size without any additional optimization.

Each word of the microcode contains four functional parts:

- **UNIQUE STEP** specifies the nanocode word to be executed during this cycle.
- **FOLLOW BASE** holds the most significant bits of the next microword to be executed; these bits will be concatenated with the bits generated by the sequencer.
- **CONDITION** is the basic field used to keep the information about the branching condition of the current cycle.
- **MODIFIER** is a field that keeps either additional information about the branching condition or an offset to allow the sequencer to generate any required microcode address for direct successors and branches with few successors.

In addition to these parts there is additional information used to parameterize the nanocode. For simplicity, this is not shown in Figure 5. The address generation is based on a k bit base address for the next cycle which is concatenated with a 3 bit field generated by the sequencer. These three bits enable branches for up to eight different locations, which have to be placed in subsequent words of the microcode. The **CONDITION** field of the microword contains information about the actual branching conditions of each step, and the **MODIFIER** field allows the sequencer to generate all possible combinations of 3 bit codes to enable direct successors in all possible microwords.

In the nanoword store the information from the **UNIQUE STEP** field of the microcode is decoded into the necessary signals on the control lines. An additional encoding of control lines to reduce the width of the nanocode can be included.

The process of generating the microcode and nanocode is a two-stage process. The first step is to extract the information held in the database; this is again done with the SQL REPORT tool. The output files in our application are written in a format that can be used as C-code. The second step is to order and compress the words of the microcode according to the requirements of the sequencer, and to combine the information kept in the nanocode. These manipulations cannot be executed inside the database, as the data model does not contain any information about the actual hardware structure of the controller. The process of arranging the microcode has been performed by a small C-program of about 100 lines of code, mainly type definitions and some functions, and, in particular, 'include' macros incorporating the files created in the first step. This method of including the database output into the source code of the C-program has the advantage that it requires almost no software development to obtain the information kept in the database, i.e., no specialized input routine to load the data. As the program is not frequently executed, run-time efficiency of this process is not a major factor. The output files generated can be directly included into a gate level VERILOG simulation.

The way the program proceeds is the following:

- Search for all branches with the largest number of successors, and reserve an appropriate section of the microcode storage for the successors. Always leave an empty section at the lowest addresses of the microcode storage to place the first instructions of the various processor functions.
- Repeat the process with a decreasing number of successors, always positioning a section of microcode where all successors can be reached by modifying the lowest bits of the base address. If possible fill gaps left by earlier allocations.
- Finally fill all the left gaps with allocations of steps having only one direct successor.
- Create an encoding for the unique cycles to compress the information about which nanocode word to activate.
- After all cycles are allocated in the microcode storage and the decoding for the unique cycles is decided, concatenate the information and generate the output in the form required from VERILOG.
- Extract the data about which unique cycle is going to activate which control lines and store in an appropriate sized matrix. Export the matrix in a ROM format for VERILOG.

After a successful simulation at gate-level the same output can be used in a macrocell generator to get a ROM block representing the nanodecoder.

**EVALUATION**

The idea and technique to use databases for the development of microcode evolved during an 18 month period of designing the coprocessor mentioned in the examples. The analysis of existing methods for this design revealed the following difficulties:

**Manual flowcharting** is a proven and flexible methodology, which can be used for all kinds of processor designs. However, maintaining consistent flowcharts for instructions with several hundred cycles over a number of changes would be very difficult, time consuming and error-prone. In addition there is still the task of testing the instructions and transforming them into a final hardware product.

**Microcode assemblers** convert algorithms given in a low level abstract notation into binary patterns of a supplied controller structure. A major problem with such an approach would be the fact that assemblers translate a sequence of statements, and consequently such a tool would be required to express all demanded parallelism in the input language. In a complex design with extensive parallelism at cycle level it is hard to justify the development of such a language, as it would be quite large. Creating a translation for each instruction of such a language into the corresponding binary pattern would require a considerable amount of time. In addition there would be a severe overhead for each change of the target controller structure.

**Microcode compilers** allow the input of algorithms in a high level language (HLL), which is then transformed into a low level representation that can be translated by a microcode assembler. The use of an HLL to enter the algorithms normally eases this process by supporting complex
control sequences (loops, if-then-else,...) and variables. The compiler will also perform a data dependency analysis and optimize programs by rescheduling certain instructions. In the case of this particular coprocessor these features were not particularly important, as the algorithms have a high branching frequency in combination with a high data dependency, which does not allow much optimization by rescheduling. The fact that the processor is based on a set of purpose bound registers precludes optimization with sophisticated register assignment methods.

In general the development of a processor-specific microcode assembler and compiler is very time consuming. Even if the microcode compiler can be created by modifying an existing version to the new architecture, the development time is not appropriate for a small scale development, or where re-usability is not expected.

Retargetable code generators not only use a HLL to describe the algorithms, but also allow the user to describe the datapath in a computer hardware description language (CHDL). Based on this hardware description the system will automatically generate a microcode assembler for the given architecture. While this approach works well for 'standard' architectures with multiple general-purpose registers and arithmetic data manipulation facilities, the embedded CHDLs are not always capable of expressing the hardware used in a datapath. In addition to this, the fact that the automatic creation of the microcode assembler results in a severe limitation of the controller structure made these tools inappropriate.

DATABASE DEVELOPMENT

The development requires a basic understanding of relational databases and the related database tools, but it is easy to achieve a fundamental knowledge for the first stages and then to develop more detailed knowledge as the requirements get more complex. A key feature of our database method is the ability to expand and refine the tool set as the application progresses. Based on an initial data model which can be developed within hours, the tools required for various stages of the design can be built and adapted on demand. Depending on the complexity of the data model, first data entry forms for the relations to be entered primarily can also be created in a few hours, as database packages offer good tool support. Adding the functionality to perform checks on the validity of entered data and to cross-query into the records of other relations can be done when needed and takes a few days to understand the tools and formulate the requirements. With a basic knowledge of the database query language, some simple queries to check the consistency of the database can be formulated instantly. Building a basic set of consistency tests will take some days, but again, additional more complex tests can be added at a later stage when the basic algorithms are going to be optimized. Extracting the database contents for a behavioural simulation and building a module in the simulator to coordinate the extracted data should take not more than a few weeks.

The effort of generating a simulation model of the datapath is of comparable complexity to the process of generating a description in a CHDL required for a retargetable code generator. The amount of time required for the final stage of extracting the database information and transforming it into binary patterns to be used in the final controller largely depends on the structure chosen for the controller. In our example it took about two weeks to model the sequencer, define the database output and write the program transforming it into micro- and nano-code.

While the last section mainly covered the time requirements for the tool development, there is also time involved in applying these tools. The amount of time required to enter the actual design data into the database of course mainly depends on the complexity of the later microprograms. The data entry itself can be an iterative process, adding new information at one abstraction level at the same time it is required at the next higher level. The runtime for some complex consistency checks can easily reach several hours, as it can grow exponentially with the number of database records.

The key differences between the database tool and the other methodologies are outlined below:

- The ability to make use of the power and reliability of existing software to reduce the development time which is particularly important for one-off implementations.
- The ability to expand and refine the tool set as the knowledge of the application develops, while other methods generally need a clear understanding of the target architecture before tool development can start.
- The lack of restriction to the structure of both the controller and the corresponding datapath.
- The ability to perform early consistency checks on various aspects of the design.

This work was developed over an 18 month period as part of a PhD programme by the primary author. Thus this period involved research into the different microcode development methods available as well as investigating architectural designs and the requirements of the coprocessor, implementing the database tool and the development and simulation of the microcode. In retrospect it is estimated that the database tool took up perhaps two to three months of this period. Overall the database method should enable the development of microcode for a given set of algorithms and a target architecture within a three to four month period. This estimation includes the time required to generate the tool set matching the architecture. It is assumed that a similar development would take at least six to 12 months using an alternative methodology, presuming the relevant expertise was available to develop the software.

CONCLUSION

As the use of a microcode compiler is not always appropriate, a method based on databases has been demonstrated that supports the design of microcode with flowcharts. It has been explained how such a system can be used to handle
Databases to support the development of microcode: J-U Dzikowski and P A Rounce

The complexity of larger designs and to keep the design data consistent. The method can be implemented without the need for a large amount of software development. A processor with an application-oriented register set that offers a considerable amount of instruction level parallelism was used to demonstrate the technique.

REFERENCES


Dr. Peter Rounce is a senior lecturer in computer science at University College London (UCL), where he teaches on microprocessor systems hardware and architecture. Following his first degree in physics from Sussex University, he first worked for ICL as a software engineer developing machine-level software. His subsequent career has been at UCL, where he took his PhD in physics, although much of this work involved the application of microcomputer technology to the control, data collection, data transmission and data analysis of an optical wind measuring experiment flown on a high-altitude balloon. Since joining the Computer Science Department, he has continued to work in the area of computer architecture and also has developed his interest and experience in VLSI technology and neural networks. His research activities are concerned with parallel computer architectures, neural networks and optical computing.
VLSI Architecture Research within the ESPRIT SPAN Project

P.A. Rounce, K. Chan and S. Mackay

Department of Computer Science, University College London, U.K.

and

K. Steptoe

Department of Electronic and Electrical Engineering, University College London, U.K.

This paper presents the research work in computer architecture currently being undertaken at UCL as part of ESPRIT Project 1588—the SPAN project. SPAN is concerned with the integration of numeric and symbolic processing on parallel architectures, and involves activities from application software to hardware. A major activity is the development of a common virtual architecture, the Kernel System, to provide a model parallel architecture for the project. The VLSI architecture work is concerned with investigating possible parallel VLSI architectures to support this Kernel System with the object of implementing one design in silicon. The work has been divided into the architecture of the processing element, the communications between processing elements, and the implementation of list structured memory; a key element of the Kernel System. Besides presenting this VLSI work, relevant material from the Kernel System is presented to provide the context within which the work is taking place.

Keywords: Parallel computer architecture, List-structured memory.

Submitted: May 1988
Revised: December 1988
Accepted: January 1989

1. Introduction

ESPRIT Project 1588—SPAN [4] is concerned with the integration of numeric and symbolic computing on parallel architectures. The partners in the SPAN project, besides the Department of Computer Science at University College London, are THORN EMI Central Research Labs, Thomson-CSF Cimsa Sintra Division, PCS of Germany, INESC of Portugal, the University of Athens, and CTI of Patras. The structure of the project can be seen in Fig. 1.

The activities within SPAN range from hardware to applications level software, with one of the principal activities being the design and implementation of a common virtual architecture, the Virtual Machine or Kernel System, which provides a goal for the high-level language and applications software activities, and as a starting point for the hardware workpackages. An associated activity around the Kernel System and language work is concerned with providing support for multi-style programming via an object-oriented programming environment.

The aim in the Kernel System is to provide an architecture that supports both numeric and symbolic computing with equal facility and that supports multiple-instruction, multiple data (MIMD) parallelism. The Kernel System consists of two language components: PARLE [8] which is a target language...
for the work on compilers, and the Virtual Machine Code (VMC) [5], which is a lower level counterpart to PARLE, and which provides a starting point for the hardware studies. Below the Kernel System there are a number of parallel computers architectures, in development within SPAN and in other ESPRIT projects on which the Kernel System is to be mapped. The purpose of this paper is to present the aims and objectives of the VLSI Studies workpackage and to discuss the issues central to its work.

The aim of the VLSI workpackage is to investigate possible MIMD architectures that will efficiently support the Kernel System, and to develop in silicon one such architecture. The starting point for this work is obviously the Kernel System, in particular the syntax and semantics of the VMC. Thus before discussing objectives, it is necessary to review the Kernel System and the VMC.

2. Kernel System

The principal objective of the SPAN project is to develop means to integrate numeric and symbolic computing at hardware, system software and applications level on parallel architectures. To meet this objective, a model architecture, the Kernel System, has been developed, which is the keystone of the project. It provides the mechanisms seen as essential to support efficiently both numeric and symbolic processing and to support parallelism. For the former it provides a list-structured memory, with lists, and list operators, as primitive elements. For parallel processing it supports multiple processors and multiple processes, with each processor having its own local memory and access to a shared, global memory space. It has procedural semantics with execution based on an explicit flow of control [14], constructs to allow sequential and parallel execution, and primitive operators to support process synchronisation.

In considering the Kernel System it is necessary to understand that a key philosophy in its development has been to provide only those mechanisms, which are essential to the project objectives, and to abstract these such that the Kernel System is not unnecessarily limited in its application. Related to this is the determination to limit the complexity of the Kernel System on the basis that system builders can achieve the complexity necessary to their objectives from more primitive elements, provided the appropriate primitive elements are available.

From the viewpoint of the VLSI studies the Kernel System can be partitioned into communications, list structured memory and processor: a partitioning that is reflected in the layout of this paper.

2.1 Communications in the Kernel System

One of the main issues in parallel architectures is the provision of communications between processing elements. In the Kernel System, a computer system is a network of computers (to be referred to as computing nodes). The address space of the computing nodes is considered to be a flat linear address space: it is not hierarchical. Each computing node has an address and is capable of communicating directly with all other computing nodes via their addresses. Thus the logical model of communications is point-to-point connectivity, which is seen as the optimal solution to the communications problem, but which can be mapped on to more practical, but less optimal, physical architectures. The ‘local’ memory of a node provides for rapid local code and data access, but this memory can be accessed directly by any other node via the communication network – a non-local access. Thus, the local memories are the constituent parts of a global, shared memory. The semantics of addressing non-local memory is identical to that of local memory, with the decision on whether an access is local or non-local being decided during execution from the address presented. Having a shared memory in this fashion means that code and data can be transferred between nodes simply by making non-local memory accesses, and that non-local accesses use the same instructions as local accesses, so that they do not have to be especially programmed as such. The direct initiation of processes on non-local processors is not supported, but has to be mediated by a process on the non-local processor.

The Kernel System provides support for communications, but does not prescribe the physical communications method, leaving systems' implementors free in this area.
2.2 List-structured Memory

A key feature of the Kernel System is its memory organisation. Memory elements are flexible objects, such that an element can hold an atomic item such as an integer or a non-atomic or structured item, a list. A list is an arbitrarily long sequence of memory elements. The memory elements within a list can themselves hold either atoms or lists. Thus the memory organisation is an hierarchical, tree-structured memory as shown in Fig. 2. A computing node can be considered to hold a ‘root’ pointer to the top-level list in its local memory. There is no limitation within the Kernel System as to the structure of the memory or as to the size of a list, except that list subscripts are integers and are limited by the maximum integer size of 32 bits. A further abstraction is that the computing node itself is very much like a memory element and it holds a list – the list that is the local memory of the node: this abstraction is used in the addressing of non-local memory.

The provision of a list-structured memory is seen as a key construct in providing efficient support for both numeric and symbolic processing; being designed to support the data structures and accessing mechanisms of both. The arrays of numeric processing are naturally supported by a list-structured memory using lists whose elements all contain the same type of object, whilst the records of PASCAL and the structures of C can be easily represented by lists with elements containing different object types. For symbolic processing the list-structure of the memory directly supports symbolic lists.

The Kernel System provides appropriate operators to manage and access lists: lists are first-class objects the same as atoms. For numeric programming, there is a random access to any list element to support array access, and this also simplifies access to records and structures. For symbolic processing, there are operators to construct lists and to extract sub-sections of lists. The immediate benefit of encapsulating an array or structure into a list is that bounds checks are implicit, since an access which is not within the list can be trapped.

2.3 Processor

The Kernel System is based on a decentralised parallel control flow model [14], and the approach of the designers has been to extend and generalise the traditional von Neumann computer model. Each computing node has a processor which executes instructions and manipulates data, and has access both to the local memory of the node and to the global shared memory. Parallelism is supported via processes and each processor maintains a process list: there are operators to place a process token on the process list of the local processor, and to suspend and delete processes. A processor always executes the process at the top of the process list. Within a process, there are procedural semantics with procedure call and branching, and with sequential execution of instructions; parallel execution is provided through the capability to spawn other processes, with sequential or parallel activation on to the local process list. Thus, the Kernel Systems supports parallel execution at the processor level and multi-processing within a processor.

To support interprocess communication and synchronisation, message passing primitives are provided that operate via the memory. This is implemented by allowing a memory element to have an empty state, such that the message-passing operators can place a message, and integer or a list, into a memory element, local or non-local, only if the element is empty and can only remove a message from an element if it is not-empty. If the condition is
not met, then the operation is blocked and the issuing process suspended. On a successful message transaction, the enabling condition is inverted: thus placing a message in an empty element makes it not-empty and removing a message from a not-empty element makes it empty.

Shared memory and message-passing are seen as the two fundamental forms of communications between processes by the Kernel System designers upon which system developers can build.


The purpose of the VMC is to provide a model of the Kernel System that can be ported on to a variety of parallel architectures. It has been designed to be a low-level language, a 'machine code', that reduces the effort to perform the port. The philosophy behind the design of the VMC has again been to provide the necessary components to support the Kernel System, but to limit the complexity. Thus, the VMC is a low level language with a RISC-style computational model, i.e.

- it has a small, simple instruction set with which more complex operations can be implemented. (The exception here being that list operators are members of the instruction set, as lists are primitive elements)
- there is only one addressing mode
- there is a load/store philosophy
- all instructions are assumed to take equal time to execute (of course, at the hardware level, this cannot be maintained for list operators and non-local accesses).

3.1. Communications

The VMC allows access to the memory of other computing nodes for both shared memory accesses, i.e. load and store operations, and message-passing, via the get and put operators. These 4 operators alone provide for communications between computing nodes. The semantics of specifying that an operation is non-local, i.e. must communicate with another computing node, is through an extension to the normal mechanism of addressing the local memory, as described shortly.

The VMC specifies nothing else about the communications, i.e. there is no specified communications hardware or protocol or message structure. The VMC model of communications is strictly one of accessing a memory element, and what is passed is the content of a memory element, atom or list, with no restriction as to the complexity of a list passed: it is possible to take a copy of the whole of a computing node's local memory by performing a load at a processor's address.

3.2. List-structured Memory

A typical one-dimensional list is represented syntactically as:

\[
[1 \ 2 \ 3 \ 4]
\]

while a more complex list structure is

\[
[1 [2 \emph{empty} \ 10] \ 4 \ 5 [6 \ 7 [123 \ \emph{empty}]]]
\]

Lists are qualitatively different from atoms, e.g. the atom 10 is different from the list of length 1 which contains the atom 10, i.e. \(10 \neq [10]\).

There are a number of list operators, allowing the creation and extension of lists, the merging of two lists into one, the extraction of a sub-list from a larger, and the creation of complex list structures. These operators allow direct manipulation of list structures, while the addressing mechanism allows direct access to an element within a list via 'selection' (i.e. applying a subscript to the list to locate an element), as is necessary for efficient numeric processing.

Because of the ability to store a list of any size in any memory element, the memory of a computing node does not have a fixed structure: it can be viewed as a dynamic tree structure (see Fig. 2). Access to memory locations within the memory structure can be made by selection from the memory structure's 'root' or global pointer, referred to in the VMC as \(gp\): each computing mode has its own \(gp\). Thus, \([gp \ 3]\) is the address of a memory element in the local memory of the processor operating on this address – the address of the third element in the root list of the processor. The sixth element of a list held in this memory element is referenced by \([gp \ 3 \ldots]\)
Lists are subscripted from 1 onwards and selection from an atom causes a fault. As can be seen, an address of a memory element is just a sequence of subscripts (selectors) to be applied to lists on route from a context (gp is one such context - there are others). Addresses are themselves lists and can be processed in the same way as other lists.

Communication with another computing node is initiated by specifying an address that references upwards out of the local memory into the address space of computing nodes. This is done by using a zero selector to the gp context, which is followed by the address of the destination computing node. Thus, the address [gp 0 8 3] signifies a superior access out of the processor to reference the third element in the root list of node 8. Thus non-local accesses have the same addressing mechanism - selection from the root of the computing node memory - as local memory accesses. All computing nodes exist at the top level, which provides a flat address space.

3.3. Processor

The processor has a reduced instruction set. It provides the usual integer, floating point and logical operators. It provides operators for handling lists - cons, cat, head, tail and length. The only operators for accessing memory are load, store and the message-passing primitives put and get: with all other operators finding their operands on top of a limited-depth, evaluation stack and placing any results there.

The memory operators access the contents of the addressed memory element whether it is an integer or a list (load and store also manipulate empty). The VMC implements copy semantics and when a load is executed a copy is considered to have been performed on the contents of the addressed element: the copy being placed on the execution stack, be the item an atom or a list. When a store occurs, the item overwritten in the addressed memory element is considered destroyed. Thus, there is no sharing of lists in the VMC, and there is no garbage collection required for lists, because they are immediately destroyed when overwritten or otherwise consumed. This mechanism simplifies the VMC, but has major implications for hardware supporting the VMC.

The VMC does not have the usual, numeric-style, unlimited-depth stack, and the return information from subroutine calls has to be explicitly removed from the evaluation stack into memory elements at the start of a subroutine. The purpose again here is to limit the complexity and to give flexibility to system implementors in the handling of subroutine calls. However, there is a need for a local data space for subroutine parameters, for local subroutine variables and for return addresses, and a need for a relative addressing scheme to access this: in particular to be able to support recursive subroutines. This is supported via a list called the dp-list and a pointer to this list, called the data pointer or dp, so that data accesses can be made relative to the current dp pointer via addresses, of the form [dp 2]. Generating a local data space for a subroutine is just a matter of creating a new list, and placing a pointer to this list in the dp register. The dp-list forms part of the local memory structure.

The VMC processor implements the same multiprocessing model as the Kernel System.

4. Objectives of the SPAN VLSI Architecture Workpackage

To fulfill the aim of the workpackage it is necessary to develop a computing node which can be interconnected in a network to produce a parallel computer which can support the Kernel System. In the VLSI workpackage, a simple, reduced-instruction set approach has been selected as the best approach to the node development, as it is felt that this offers the best route to the provision of high speeds of instruction execution and reduced time for design: making the computing node a feasible development by a small team. Thus the VMC is obviously the target language to be supported by the computing node, besides being the raison-d'etre of the project. At what level the node supports the VMC is a question for investigation and it is likely that there will need to be some translation from the VMC to the instruction set of the final node implementation.

It is readily apparent that the work of the project can be divided into the three areas of the list-structured memory, the communications network and the instruction processing unit. The objectives of
the workpackage are to investigate these systems with a view to the development of an integrated VLSI implementation. The rest of this paper examines each of the three areas in turn with a discussion of possible systems and the approach that is being taken within the workpackage, ending up with an examination of possible architectures for the computing node.

5. List-structured Memory

The implementation of the list memory is probably the key item in providing a computing node which allows efficient execution of both symbolic and numeric processing. The emphasis here is on 'both', since there are a number of approaches to list-structured memory, which while performing well for LISP-style programs where access is mainly to the head of the list, give poor performance for numeric-style array references which can be to any array element [10]. The approach of the workpackage has been to look for a list memory scheme that can be implemented on a standard, linearly-addressed, physical memory, since this is readily available both cheaply and in high densities.

The traditional LISP cell just holds a data object (an atom or a pointer to a sublist) and a pointer to the next LISP cell, and lists are produced by chaining cells together [1]. To access the nth element the pointers are followed from the head cell until the element is reached: giving very poor numeric-style array accessing. In this basic form, the best way to provide random access into lists is to force the cells to be linearly placed, i.e. as a linear array, in physical memory in which case the pointers to the next cell can be eliminated. To make this work all list elements must be the same size so that an element is located by simply adding its selector to the list start address. The fact that any element in the list can hold a sublist is not a problem since the sublist can be located elsewhere and a pointer held in the list element, provided the pointer can be held in a single memory element. This would be an optimal scheme. However, since lists are dynamic entities in symbolic processing (and also in numeric, if the traditional 'heap' is implemented by adding elements to a list), there is immediately the problem of management of variable length memory blocks, very similar to the problems encountered in segmented memory management systems.

Several other schemes have been investigated both practically and theoretically [10, 11], which improve upon the traditional LISP cell. Thus, delimited strings [11] are a means of representing complex list structures in a linear form, which although reducing the search time to locate elements in the list, still requires the list to be searched from the start in order to locate an element, since sublists are embedded. It has the same memory management problem as the linear array representation, but without the speed of random access, and its only advantage over the linear array lies in reducing the amount of memory required to hold a list. Content-addressable memory, on which it has been shown theoretically [11] to be possible to support random access to lists with a fixed arity, have been rejected by the VLSI workpackage, since these memories are not readily available in high densities, and because of the fixed list arity, which does not provide an easy mapping to the variable size lists of the VMC.

The approach of the workpackage has been directed at page-based schemes which, as in standard memory management systems [6, 13], offer the simplest means of managing free memory. A simple paged-based scheme for implementing a list structured memory chains the pages of a list together by having the last entry in a page hold a pointer to the next page [11]. This still requires a search from the head of the list to perform a random list access, albeit searching a page at a time. It can be viewed as the standard LISP cell with more than one data object held in each cell.

A different page-based mechanism has been developed within the workpackage. This is based on the multi-level page tables of paged memory management schemes [6, 13], and is shown in Fig. 3. In this figure, there are 3 levels of pages and the pages in the rightmost column, the third level, hold the list objects, which can be either 32-bit integers, the empty token or list pointers (pointers to similar list structures). The pages in the other 2 columns hold pagepointers that allow the structure to be traversed from left to right to locate a list element. The address of the leftmost or first level page of the list structure is held with other information in the list
Fig. 3. Access to an object within the list is from this list pointer using the subscript (position) of the element in the list. This subscript is divided into as many fields as there are levels in the structure, with all fields being the same size ($\log \text{page.size}$ bits): in the figure there are 3 such fields labelled A, B and C. The fields are used successively to traverse the structure.

Thus, field A is added to the physical address from the list pointer to give an address of an element in the first level page, from which a page pointer to a second level page is read. Field B is added to the physical address from this page-pointer to give an address into this second level page from which is read a pointer to a third level page. Lastly field C is added to the physical address from this last pointer to give the physical address in the third level page of the selected list element. The same number of operations is required to reach any element in the list. One of these multi-level structures is required for each list in the memory.

The larger the page size and the greater the number of levels in the structure, the larger the number of elements that can be held. With a 32-element page, a two-level scheme supports $1K$ ($32^2$) elements in a list, while three and four level schemes support $32K$ and $1M$ elements per list respectively. However, with larger page sizes there is increased wastage in unfilled pages at the ends of lists, but less support pages are required to map a list on to memory; while the larger the number of levels to traverse to reach an element, the greater is the access time. However, for a given number of levels, there is the same overhead on accessing a list element whether to the first or to the last element, so there is no penalty for random access. A reduction in the overheads of access time and memory usage can be made by making the number of levels variable dependent on the number of elements in the list, and holding the number of levels either in the list or in the list pointer. Then short lists will have fewer levels than large lists, e.g. a list with a length less than the page size would have all its elements in the top level page and provides a linear mapping of the list on to the physical memory.

Since few lists will have exactly the right number of elements to fill a structure, the valid elements of the list have to be marked in some way, and this can be done either by having a valid flag in each and every list element, or by holding the number of valid elements, the list length, in the list pointer or in the first level page.

This paging scheme is suitable for use with a cache memory which could speed up the structure traversal.

A number of variants on this scheme to reduce the access overheads where possible and to assist in the modification of lists are under investigation, and will be presented in a future paper.

6. Physical Memory Requirements

An important aspect of the memory is the size of a physical memory location. In order to support the page memory system as presented, all list elements must occupy the same amount of physical memory. This has to be some trade off between using the maximum that might ever be required and keeping it small enough that a large amount of the memory bits are not wasted. The basic integer size in the VMC is 32-bits and this is a convenient size for the
basic memory unit. However it is necessary to add some tag bits to this for identifying the different data types – integers, lists, and empty. Thus the memory element will be greater than 32-bits to support the tags and also to support the holding of pointers to lists.

The size of the memory space local to each computing node ultimately depends on the size and structure of each implementation of a complete parallel computer system, the program or programs being executed and the organisation of the programs across the computing nodes. If the final implementation is in terms of 1000s of computing nodes, the memory on each node might be quite small, 4K-64K words, while for system with a very small number of nodes, megaword memories are probably necessary. The maximum size is important at the design stage because it affects the size of the physical address that must be held in a list pointer and influences what else can be held there. Since there is no simple answer to the size question, and the factors affecting it are themselves matters for research, the physical pointer size is still under consideration, although somewhere around 20 bits would give some flexibility to the node memory space.

7. List Management

The usual problem of list management in symbolic systems arises since lists are dynamic objects, which must be deleted as well as created. The problem is worse in parallel systems where as in this case there is shared memory between processors and a list may be accessed from any processor.

The VMC applies copy semantics: a copy of a list is taken and placed on the evaluation stack whenever a load reads in a list, and when a store is made to a list element holding a list, the overwritten list is deleted. The VMC has no concept of a pointer to a list except for the logical list address, e.g. [gp 3], and its evaluation stack can hold any size of list in a single stack element.

At the hardware level, circumstances are different: list copying is not a simple matter, and overwriting list pointers leads to unreferenced lists on which garbage collection must be performed. Execution of a load that finds a list pointer will cause it to be fetched from memory and placed on the evaluation stack, and there is a design decision of when to copy the list that it references. In the case that a store is executed to take the list pointer from the evaluation stack and place it into a new location, a copy of the list is required, unless multiple references to a list are allowed. However, if the length of the list is taken instead, the list pointer is discarded from the stack once the length has been found, and no copy is necessary. Therefore, it is inefficient for the hardware to copy a list on load, and it should wait to do this, at least until the execution of whatever operator is to be applied to the list.

At this point, there is the question of whether to implement copy semantics or whether shared references to lists should be allowed, with a list only being copied when a write occurs into one of its elements. The latter choice leads to a requirement to perform garbage collection when a list is no longer referenced. However, copy semantics requires that a complete new copy of a list must be taken whenever a list pointer is located and then stored in a new location, and that garbage collection must done immediately whenever a list pointer is overwritten or when, after a get of a list pointer to the evaluation stack, the list pointer is discarded without storage.

It is a straightforward decision to implement copy semantics for non-local memory accesses, since having a list pointer reference a non-local memory makes the hardware for dereferencing the list much more complex, and since garbage collection across multiple processors is also more difficult [9, 10]. However, within the memory of a computing node the position is less clear. The advantages of a multiple reference scheme is that the amount of list copying should be reduced, since list copying is only needed when a shared list is modified. For example, LISP programs often build a list and then operate on the list without further modifications except for taking pointers to sub-sections of the list. This form of operation does not require a copy to create each sub-list, as can occur with copy semantics.

In order to support multiple references, a reference counting scheme [10] would appear the most feasible, since it is easier to spot when a list has to be garbage, and because some form of identification that a list is multiply referenced must be kept. The
need for this arises because a write into an element of a shared list must not occur as this would affect all references to the list, whereas the Kernel System assumes that the write is performed on only one list. Thus, a write into a shared list must force a copy of the list to be taken before the write occurs. This leads to the requirement for all writes to be addressed from the root node of the memory structure. If the address to be written starts at the root node of the memory, each list accessed during the dereferencing process can be checked to see if it is shared and a copy taken if it is. If the address does not start at the root node, e.g. it is a dp-relative reference, then it is necessary to backtrack up the memory structure to the root node in order to check that the dp context itself is not within a shared list. Backtracking up the tree requires that there is a backward pointer in each list to each parent node with as many backward pointers as there are pointers to a copied list. This is a problem that may not be resolvable with the result that to use reference counting all writes must be dereferenced from the root node, which is not compatible with the VMC.

Thus the initial position for the design is that copy semantics will be applied, unless some restrictions can be applied which in some way bypass the requirement to backtrack writes to shared lists or to cope with the multiple reverse pointers in a manner compatible with the rest of the architecture.

8. Communications

The requirement for the communications is to perform the 4 types of data transfers – load, store, put, get – between any two computing nodes using the address of the destination node to identify the route. There will need to be outgoing messages from a node stating the operation to be performed, and reply transfers. A particular problem for communications is that a data item to be transferred may be a list, so that the transfer of 'messages' between nodes must be supported at some level. Furthermore, because the list memory of a computing node is dynamic, it is in general not possible to specify the physical address of a memory element to be accessed in a non-local computing node. This means that logical list addresses must be used, which requires more information to be transferred and also requires some interaction with the memory management of the non-local node. Together, these could lead to the possibility of the memory management of a node becoming overloaded with concurrent requests for action from a number of non-local nodes. One solution to this, which reduces the complexity of the receiving logic and requires no higher level protocol to manage the transfers, is to transfer each message in an uninterruptable sequence. Without this restriction, a receiver can receive interleaved sections of messages from different nodes, forcing each message to be re-constructed on receipt, which requires either that the receiver is capable of reconstructing a number of interleaved messages coming in or that there is some higher level protocol that enables the receiver to accept one message from one transmitter at any time.

There are a number of possibilities for implementing the communications network [12] e.g. buses, switch networks, hypercubes, and a variety of factors to take into account in deciding which to adopt. Within this project, the main factors which have been selected as important in choosing the network architecture are:

- ability to support a large number of computing nodes, where large is 100s-1000s,
- minimal number of different logic units to implement a large network,
- limiting of the complexity of the logic to interface to the network,
- the ability to modify the network architecture to improve connectivity to computing nodes without having to modify the rest of the node architecture,
- an architecture that does not require software on intermediate nodes on a network route to facilitate the transmission, as is required in the INMOS Transputer®,

while a subsidiary factor is the possibility of building a single chip computing node with or without memory, but with the network logic on chip.

From these criteria, a single bus architecture has been immediately rejected because, although giving point-to-point communications, it is restricted to a small number of processors before the whole system becomes limited by the bandwidth of the bus, and
because it becomes impractical to directly connect up and drive a bus with 100s of processors attached. However, multiple bus systems are an attractive possibility. With these a small number of computing nodes could be clustered on a single bus to provide point-to-point connectivity for the local cluster, and then these basic bus-clusters could be interconnected in some fashion, e.g. via an inter-cluster bus system [3]. Provided the local cluster bus is not too heavily loaded with computing nodes, and that communications outside the local cluster is rare, this should be able to provide good connectivity. One drawback is that at least two interface components are required: one between a computing node and the local cluster bus, and the other from the local cluster bus to the inter-cluster bus. Furthermore, if messages are passed as an uninterruptable sequence, there is a potential deadlock problem when messages are travelling in opposite directions between two clusters, if messages can be spread out over all the intervening busses. This might be solved by storing messages at each bus-bus interface at the cost of providing a more complex interface.

At the other extreme is the hypercube, which has been much much discussed in recent years. Although not giving point-to-point connectivity, it certainly gives excellent connectivity between nodes of the network, with the number of network links to be traversed in the worst case routing in a system of \(2^n\)-nodes being \(n\). The problem with hypercubes is the complexity of the switch at each node, which requires \(n\) incoming and \(n\) outgoing links, and the large number of network links - \(n \times 2^n\). To support 1000 computing nodes would require a switch with 10 incoming and 10 outgoing links and 10,000 links in total. This problem is much reduced if serial-data links are implemented rather than parallel ones, but serial data transmission reduces the bandwidth drastically. The complexity of the switching arrangement and the number and width of links are two reasons for rejecting a hypercube.

A third possibility is some sort of switch network. A cross-bar network for even a small number of processors is prohibitive, so that this is not a contender for 100s let-alone 1000s of processors, but a message-switch network is a possibility [7]. A simple arrangement is shown in Fig. 4. Each computing node connects via single-direction, parallel-data links to two neighbouring switches. A switch routes an incoming message via the destination address at the head of the message. By using an asynchronous handshake for flow-control, there is no need to have very much storage at each switch: a message can be allowed to spread out over other nodes without deadlock occurring. The routing of messages between computing elements is via the switch units of intervening nodes, but this is facilitated purely by the switch hardware, and requires no software intervention from the processor node. Messages travelling in opposite directions are not a problem because there are separate data links in each direction. Each link in the network is functionally separate from other links, and only coupled with others when a message is spread over them. Thus sections of the network can operate concurrently on different messages. The linear network appears a good choice if communications are mostly over a small number of links from a processor, but a degradation must be expected when a large number of links must be traversed.

However, the linear switch can be expanded into multi-dimensional systems, e.g. a grid system as in Fig. 5, by the addition of extra links and a modified routing algorithm, and this improves the operation by drastically shortening the longest paths and improving the number of pathways through the network. A parallel-data, messaging network of this kind is under development within the workpackage.

The message network is seen as basically just a transport medium, which does not restrict the format of the messages except that the destination address must be the first data item in the message. It should also be noted that no implicit information is sent with the message, so that, for example, the source address must be included in the message if a reply is required.
9. The Instruction Processing Unit

To some extent that part of the computing node that executes instructions and provides register storage for operands and results, the traditional 'processor', is restricted by the VMC, which is a machine-code style of language based on the RISC model. Thus a RISC-style architecture is forced upon the workpackage, although this is a reasonable approach given the success of other RISC systems, and the small number of people working on the project. However there are still a number of design decisions to be taken, and areas in which the processor implementation can influence the speed of operation of the system. In particular, the Kernel System is a multiprocessor, multi-process environment, and fast process-switching and exception handling within a computing node is bound to enhance performance. This also extends to a fast function call mechanism, since both numeric and symbolic programs do much of this.

A major design decision is at what level to build the architecture. Should it implement the VMC as closely as possible or should it be at a much lower level with a more general architecture? However, the organisation and operation of the processor section cannot be considered in isolation from the operation and organisation of the other components of the computing node – the memory and the communications.

10. Node Architecture

The simplest architecture for the computing node is to build a traditional processor at a much lower level than the VMC and operating directly on a linear, physical memory, and to execute VMC programs purely through software after a compilation phase. In this architecture, the only hardware to be produced would be the processor and the network switch. The interface to the network could then be handled by interrupt, and all list reference and management functions by software. This approach does not necessitate the development of a new processor architecture, although it should enable a single-chip solution for processor and network switch to be produced. It is likely to provide a slow, somewhat inefficient implementation of the Kernel System and VMC.

A much more complex design can be imagined with a number of specialised 'processor' units to deal with the different functions, i.e. an instruction processor, a list manager, and a communications processor. These would operate semi-autonomously and have multi-ported access to the memory, each with the ability to translate between list addresses and physical addresses. The list manager would handle the major memory and list operator functions: copying lists, creating lists, garbage collection etc. The communications processor would handle message transfers, e.g. when sending a list it would retrieve the list elements from memory and format them on to the network, when receiving a list, it would build the list in the memory and, when the transfer is complete, organise the insertion of the list into the memory hierarchy. The instruction processor would be able to access elements of the list memory directly, so that instruction fetch and non-list operations would operate on the memory, but when a list operation or a non-local access is required, the appropriate co-processor would be activated. Having multi-ported memory would improve performance by allowing more concurrent activities, since all three processors have major interactions with the memory. This would be an elegant
solution to the computing node with an appropriate partitioning of the functionality, yielding a high-speed implementation. The major difficulty is the complexity of the system for a small team, although a conceivable project with state-of-the-art VLSI design software. It is also unlikely to yield a single chip solution in the short term, although a design at the 1 micron level might.

There are any number of solutions between these extremes of design. The approach currently taken within the workpackage is to support the simplest and the most used VMC operations, in particular address evaluation and memory access operations, directly on the processor, and to provide support for the more complex and less frequent VMC operations in software via a hardware exception mechanism. Thus the hardware is targeted to support:

- integer arithmetic and logical operations
- evaluation of list addresses, e.g. \( [gp \, l \, 2 \, 8] \)
- load, store, get, put
- instruction sequencing operations, i.e. branching and subroutine call.

All memory management and process management will be performed in software via hardware exception, as will transmitting and receiving messages over the communications network. Hardware support for these operations will be developed after the processor has been built and will be implemented as special purpose co-processors. The structure of the computing node in development, named SPRINT, is shown in Fig. 6 and shows one such co-processor for list processing, e.g. list copy and deletion. To support a communications co-processor, the interface to the network has also been placed off-chip, so that it is accessible from such a device; this also allows for different network architectures to be implemented.

### 11. Implementation

It is the intention of the workpackage to take the design of the computing node, including the communications switch and interface, to layout and manufacture with a view to building a small-scale parallel computer system. Facilities are available to the workpackage that allow for full-custom VLSI layout in 2 micron CMOS.

For the processor, a datapath-control organisation [2] is planned with registers and functional units placed on the datapath, where the regular layout allows for complexity to be more easily managed, while the control section will be built from PLA-style finite-state machines and combinational logic blocks. The major VLSI layout task will be the datapath which will have to be hand-crafted to produce the required units – registers, ALU, shifter etc.: the task will be made easier by the regularity of these components. The use of PLA-structures [15] in the control logic will allow for automatic generation of these structures starting from a high-level description file, specifying the functions to be performed. This will significantly reduce the complexity of the task to design, simulate and layout the control section of the processor. The datapath and control logic will be combined to provide a single-chip processor.

The communications switch and interface should be a lesser task and is likely to be laid-out from predefined, leaf cells: once specification and simulation have been completed. This will reduce the complexity of the layout task to one of placement and routing only.
12. Summary

The VLSI workpackage has the aim of producing an architecture for the Kernel System of the SPAN project that is efficient at executing the Virtual Machine Code and at running a Kernel System environment. Three inter-related areas for investigation have been identified—interprocessor communication, list structured memory management and the instruction processor. In each of these there are a variety of possible implementations, but the initial approach of the workpackage in each area has been outlined:

- a linear, or 2-dimensional, message-switching network based on a single-chip switching element,
- a multi-level, page-based list structure to provide simple memory management and efficient random access,
- a reduced instruction set processor with specialised hardware to speed address translation and memory operations.

It is clear that hardware systems can provide significant gains in performance over a software emulation of the Virtual Machine, although at a cost in silicon complexity and development time. Besides investigating possible architectures, it is the stated aim of the workpackage to implement a computing node in 2 micron CMOS using full-custom VLSI.

Acknowledgements

The SPAN project is supported by ESPRIT as PROJECT 1588. The involvement of UCL within SPAN is due to Professor P.C. Treleaven, whom we wish to thank for our place in the project. The designers of the Kernel System are Dr. P. Refenes of UCL (VMC) and Dr. S. McCabe of THORN-EMI Central Research Laboratories (PARLE), in collaboration with Professor Treleaven, and with contributions from the other workpackages within SPAN. We would like to thank Paul Refenes for technical discussions on the VMC.

References

Kevin Steptoe obtained his BSc (Hons) in Electronic Engineering from the University of Essex in 1983. In 1983 he joined University College London, Department of Electronic and Electrical Engineering as a Research Assistant working in the Integrated Circuit Design Centre. He was appointed a lectureship in 1986. His research interests include VLSI design and all aspects of CAD for VLSI. He is an Associate Member of the Institution of Electrical and Electronic Engineers.

Mr. Ken Chan received a BSc in Electronic Engineering from the University of Surrey in 1984. He worked for Mullard, Southampton as a VLSI design engineer, before moving to University College London as a Research Associate again working in VLSI design. His experience has covered both DSP and general purpose processor design.

Mr. Stuart Mackay has a BSc from Robert Gordon Institute of Technology, in Aberdeen, Scotland, and is now working for INMOS.