A Message-Driven VLSI Architecture
For
Parallel Object-Oriented Systems

Carlo Emmanoel Tolla de Oliveira

a thesis submitted for the degree of
Doctor of Philosophy

of the
University of London

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY COLLEGE LONDON

SEPTEMBER 1991
ERRATA

• Page 21: where is written [Jona86] read [Jona88]
• Page 27: where is written direct read directly
• Page 32: where is written "machine has been" read "machines has been"
• Page 34: where is written "high level languages" read "high level language"
• Page 35: where is written "method driven model" read "message-driven model"
• Page 60: where is written "from its creation" read "from their creation"
• Page 62: where is written "customising the it to" read "customising it to"
• Page 77: where is written "accuses the message" read "indicates the message"
• Page 104: where is written "rigourously" read "rigorously"
to Josane
ABSTRACT

This Thesis investigates a "transputer-like" microprocessor architecture optimised for parallel object-oriented systems. This investigation comprises: specification of a virtual, parallel, object-oriented machine; design of a microprocessor architecture; simulation studies of the microprocessor; and implementation of a 2-micron CMOS microchip prototype.

Object-oriented software systems have now secured wide adoption within industry. However, few computers provide hardware structures optimised for object support or take advantage of object-based parallelism. The aim of this work is to address this problem, designing a parallel architecture based on a network of custom VLSI microprocessors, with specialised object-oriented hardware support.

A parallel virtual machine, derived from an abstract object-oriented computing model, was the basis to develop the whole architecture. Object-oriented languages and machines were studied to outline this abstract computing model and the underlying architecture.

A microprocessor-based system was designed to support the virtual machine. The resulting system, called BROOM, is a distributed architecture, supporting intrinsic object-based parallelism. Objects are embodied by the processing nodes and served in a non-preemptive multitasking schedule. Messages are the key activators for this scheduling scheme. They are processed as atomic instructions, decoded and executed by three pipelined units in the microprocessor node.

A simulator was written in C++ to investigate the microprocessor operation, assessing the feasibility of the proposed computing model. The simulator emphasis is on instructions and operations designed for object and message support, which were modelled under a message based timescale.

A microchip prototype of the BROOM node architecture was implemented in 2-micron CMOS. Message-driven computation is provided by three independent PLA based controllers, organised in a high level pipeline. Objects, messages and methods are supported by on-chip static memory caches. Methods are executed on a 32 bit stack-oriented datapath, controlled by single-cycle, eight bit instructions.

In summary, the prime contribution of this thesis is an object-oriented machine supporting high object-based parallelism and a special object-oriented instruction set. This is done through a distributed, multinode architecture and a high level pipeline organisation of the node processor.
ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. P. Treleaven for conceiving the idea for this project and advising me to the full development of it.

I would like to thank as well Dr. Apostolos Refenes for further counselling on details of the architecture and helping in revising and co-authoring papers generated by the project.

I am grateful to Aggeliki Balou for sharing the burden of designing the abstract machine and language carrying the idea to completion in her thesis. Also for helping revising and spotting problems in the architecture presented herein.

I am also grateful to Marco Pacheco and Marley Vellasco for sharing the task of designing the CMOS library as well for the helpful discussions about architectural and technical details.

I must show my gratitude to Dr P. Rounce and the staff in the Systems Group for trouble-shooting the software failures and keeping the CAD system running.

I want to express my gratitude also to Paulo Rocha and Simon Arridge for the careful revising of the text and useful comments and suggestions.

I would like to acknowledge as well the financial support given to me by CAPES (Coordenação de Aperfeiçoamento de Pessoal do Nível Superior - Brazil) and by the UFRJ (Universidade Federal do Rio de Janeiro).

Finally, I am deeply grateful to my wife Josane and my daughter Cibele for staying by my side all this time when I most needed them. As well to my parents Emmanoel and Tecla that cared for all that was left back in Brazil and always sent encouraging letters.
# CONTENTS

Chapter 1: **Introduction** ................................................................. 9  
1.1. Parallel Object-Oriented Computing ......................................... 9  
1.2. Motivations ............................................................................. 11  
1.3. Objectives ............................................................................... 12  
1.4. Research Contributions ......................................................... 14  
1.5. Thesis Organisation .............................................................. 16  

Chapter 2: **Object-Oriented Systems** .................................................. 18  
2.1. Object-Oriented Computing Model .......................................... 18  
  2.1.1. Procedural Model vs. Object Oriented Model .................. 18  
  2.1.2. Object-Oriented Model Variants .................................... 19  
2.2. Object-Oriented Languages and Environments ....................... 21  
  2.2.1. Sequential Languages .................................................... 21  
  2.2.2. The UC++ Language .................................................. 22  
  2.2.3. Orient84 ....................................................................... 23  
  2.2.4. POOL .......................................................................... 24  
  2.2.5. MS Windows .................................................................. 25  
  2.2.6. Conclusions ................................................................... 26  
2.3. Object-Oriented and Parallel Architectures .............................. 27  
  2.3.1. Sequential Architectures ............................................... 27  
  2.3.2. Inmos Transputer ....................................................... 28  
  2.3.3. ZOOM .......................................................................... 29  
  2.3.4. DOOM .......................................................................... 30  
  2.3.5. DICE ........................................................................... 31  
  2.3.6. Conclusions ................................................................... 32  

Chapter 3: **Virtual Machine** ............................................................... 34  
3.1. Message Driven Computing Model ......................................... 34  
3.2. Intermediate Language TOOL ............................................... 37  
  3.2.1. General Description ..................................................... 37  
  3.2.2. Syntax and Constructs .................................................. 38  
  3.2.3. TOOL Message Statements ........................................ 40  
  3.2.4. Conclusions ................................................................... 43  
3.3. Virtual Machine VOOM .......................................................... 44  
  3.3.1. System and Node Architectures .................................... 44  
  3.3.2. Object and Message Formats ....................................... 47  
  3.3.3. Instruction Set ............................................................ 49  
  3.3.4. Implementation .......................................................... 52  

Chapter 4: **BROOM System Architecture** ........................................... 54  
4.1. Machine Architecture ............................................................ 54
4.1.1. Introduction ........................................................................54
4.1.2. Network Topology ...........................................................56
4.1.3. System Modules .................................................................59
4.2. Node Architecture...........................................................................62
4.2.1. Introduction......................................................................62
4.2.2, Message and Instruction Format ......................................64
4.2.3. Message-Flow Architecture ..............................................66
4.2.4. Instruction-Flow Architecture ..........................................69

Chapter 5: BROOM Node Architecture ........................................................73
5.1. Communication Unit .......................................................................73
5.1.1. Organisation .......................................................................73
5.1.2. Protocol Arbiter ..................................................................74
5.1.3. Message Handling .............................................................75
5.2. Memory Management .....................................................................77
5.2.1. Organisation .......................................................................77
5.2.2. Message Decoder ..............................................................78
5.2.3. Message Binder ..................................................................79
5.2.4. Page Transfer Handler .....................................................81
5.3. Execution Unit .................................................................................83
5.3.1. Organisation .......................................................................83
5.3.2. Execution Architecture .....................................................84
5.3.3. Instruction Set ....................................................................86
5.3.4. Operation Control .............................................................90

Chapter 6: Architecture Simulation ................................................................92
6.1. Introduction ......................................................................................92
6.2. Program Structure ...........................................................................93
6.3. Message Driven Timing ..................................................................95
6.4. Simulated Architecture ...................................................................96
6.5. Simulation Procedures and Results ................................................99
6.5.1. User Interface and Simulation Procedure .........................99
6.5.2. Simulation Results .............................................................100

Chapter 7: VLSI Implementation ...................................................................103
7.1. VLSI Design Approach ...............................................................103
7.2. Datapath Cell Library .................................................................108
7.2.1. Design Philosophy ..............................................................108
7.2.2. Modular Register Building Blocks ....................................112
7.2.3. Functional Elements ...........................................................115
7.2.4. Interface and Switching Elements ....................................118
7.3. Chip Organization ............................................................................119
7.3.1. Operative Part Organization .............................................119
Chapter 1: Introduction

This chapter describes the status of object-oriented computing against the background of conventional procedural computing. It presents the motivations, requirements and advantages of designing a parallel architecture for object-oriented systems. A list of research contributions and the plan of the Thesis conclude this chapter.

1.1. Parallel Object-Oriented Computing

Object-oriented software systems have now secured wide adoption within industry. However, most of the current machines are tuned to support procedural languages, therefore object-oriented features derive little benefit from these machines. Few computers, like SOAR [Ungar84] and Rekursiv [Pounta88], provide hardware structures optimised for object support or take advantage of object-based parallelism. Nonetheless, given the current trend of software systems, parallel object-oriented computing is expected to have a major influence in hardware design.

Analysing the evolution of computing systems, shown in Figure 1.1, we can identify three main trends: (i) the evolution from procedure-based systems, through process-based, to object-based systems; (ii) the evolution from sequential to parallel systems; and (iii) the provision of specialist language support in hardware, as machines evolve conceptually [Trelea87]. These three trends lead to a natural pairing of languages and architectures, based on their programming paradigm, as happen with Occam in relation to the Transputer [Whitby90]. We will now examine each one of these trends and their interrelationship.

Evolution of programming systems from procedural to object-based systems forms a natural progression. A good example of this is the Pascal family of programming languages. Pascal was first introduced as a sequential, procedural
language. The process-based paradigm was then introduced to Pascal, resulting in its concurrent variation, Modula2. When the object-oriented languages began to appear, objects were introduced in Pascal as in version 5.1 of Turbo Pascal [Shamma90] or in the Object-Pascal extension. Other languages followed the same path, investing popular syntaxes with the features of emerging paradigms, so programmers would benefit from new advances with minimum learning effort.

Evolution from sequential to parallel systems is still largely on its way. Parallelism, we believe, is a more general programming paradigm, encompassing the sequential concept within itself. With this in mind, systems tend to enlarge their horizons by adopting parallelism as an extension. Examples are Modula2 [Ural87], a concurrent superset to Pascal; together with Concurrent Smalltalk [Yokote90] and Concurrent C [GehRoo89], which are extensions to Smalltalk and C respectively.

Finally, programming systems encourage the development of specialised hardware mechanisms to support their features. In Figure 1.1 we can observe that for each paradigm we can find machines that have specific built-in structures dedicated to language support. For example, for procedure-based languages like C
and Pascal, procedure call is the key feature. Hardware support is available in procedural machines in form of stack manipulation instructions in MC68000 [Bennet87] or register windows in the SPARC [Namjoo88]. Process-based languages like Occam are supported by process switching mechanism and communication channels in the Transputer [MayTay86]. Object-oriented languages like POOL find message passing facilities in DOOM [Bronne87], and Smalltalk gets method binding optimisations in SOAR [Ungar84].

In summary, observing the natural pairing of programming systems to a corresponding architecture, it is expected that object-oriented machines should emerge to support the major object-oriented programming requirements. This observation motivates this work, joining the trends in proposing a parallel architecture, supporting the object-oriented paradigm, with specialised hardware derived from language needs.

1.2. Motivations

The evolution of computing systems strongly points to parallel object-oriented systems. This motivates in many ways the development of a parallel object-oriented architecture. Current architectures are not optimised for object-oriented language requirements, thus object-oriented languages currently run on top of emulated virtual machines that slow down their execution. The availability of specific hardware is necessary to enable languages to develop securely into the object-oriented paradigm, without incurring into performance penalties. Moreover, object-oriented structuring can have a positive impact on computer architecture. It is a natural way to express parallelism that can be adopted in the hardware implementation. The object-oriented paradigm presents an accentuated locality of access that can allow object-oriented machines to run faster than their procedural counterparts.
The motivation for this work encompasses the reasons above, and consists in catering for the need of a parallel object-oriented architecture. The architecture was developed from a virtual machine to the best fit of object-oriented requirements. It is designed to convey a message-driven model with automatic message routing to simplify the porting of other languages to it. It is based on a multi-node distributed system interconnected in a tree of busses network topology. Objects are distributed among the nodes and multitasked in each node, implementing the object-oriented parallelism.

Many specific features of object-oriented systems are requirements to an architecture that is proposed to support it consistently. Message passing is an important object-oriented feature and the proposed architecture handles it with specific instructions and dedicated hardware to transfer, store and receive messages. Security of object operations is guaranteed by an object-structured memory and embedded mechanisms to bind methods and check object types. Flexibility is provided with support for dynamic creation and transport of object across the network.

1.3. Objectives

The aim of this work is to investigate the object-oriented model requirements, then design a parallel architecture based on a network of custom VLSI microprocessors, with specialised object-oriented hardware that support messages, objects and methods. The work is carried through a three level methodology, starting at the design of a virtual machine and intermediate language. It is followed by the design of the computer architecture at system and node level and is concluded by a VLSI implementation of the node processor. A summary of the main objectives is given below:
Specification of an intermediate language and virtual machine corresponding to the object-oriented model. TOOL (Target Object-Oriented Language) and VOOM (Virtual Object-Oriented Machine) [Balou89] are the intermediate level language and virtual machine respectively. They incarnate the object-oriented model that serves as a base for the hardware implementation. The main point of this intermediate system is to specify a message-driven, object-oriented computational model. TOOL specifies it through a simple unix-shell-like syntax, stating programs as collection of objects. VOOM realizes TOOL specification through operations that execute TOOL statements.

Design of a hardware architecture to be the reflection of user parallelism embodied in TOOL and VOOM. This transcribes into a distributed machine with local private memory for each node, called BROOM (Basic Regular Object-Oriented Machine) [OliRef89a]. Message passing is the communication model and applies both to node and object communication. Messages travel on switched packets of fixed length, each packet being a whole message. The network is simplified to obtain the best throughput with least supporting hardware. The topology is a tree of busses, each bus serving a cluster of nodes and interconnecting with other clusters through a switching element.

Design of a microprocessor that supports messages and object operations by hardware. BROOM nodes routes and processes messages automatically. Messages are processed in the node through a pipeline of three independent units. The Communication Unit forwards and stores the messages, routing them into the bus network using a hardwired protocol. The Memory Manager decodes messages and prefetches the object and method associated with it. The Execution Unit performs the methods, using the parameters in the input messages and generating asynchronous output messages.
Implementation of a VLSI prototype that takes advantage of parallelism related to objects. The encapsulated nature of objects provides for a reduced address space, allowing caching of whole objects. This allows for a simplified instruction set that can execute all instructions in a single cycle. The pipeline is controlled by hardwired logic (PLA), executing the object-oriented supporting tasks in background. This enables the method execution to operate with short and simple instructions, processed through a stack-top based datapath. Although the design includes local fast access memory for objects, methods and messages, silicon area is used economically. The economy applied to the design aims at the integration of multiple nodes in a single chip with the use of advanced silicon processing.

1.4. Research Contributions

The research had followed the three level methodology, starting in the investigation of the computational model that led to the architecture design and concluding with the VLSI implementation of the prototype machine. The sequence is presented below:

Object-oriented computational model - Languages were studied and the main concepts were abstracted in a computational model description. The differences and similarities between procedural and object model were analysed, as well as object and method driven variants of the object-oriented model. The models presented are a generalisation for object-oriented computing and can be used to classify and characterise object-oriented languages.

Target abstract language - A language was specified to capture the abstract concepts of the computational model. A unified representation defines objects methods and messages alike, defining programs as collections of objects.
Objects and methods are defined as collections of message-like statements. This language can be used as a low level system programming language or as a target to high level compilers.

**Virtual Machine** - The virtual machine defines the architecture to match the target language, based on the intrinsic object parallelism. Three units are defined to express the semantics of TOOL, encompassing: object transformation; message transport; and object, message and method binding. This machine is a target for hardware implementation, following the advance of VLSI technology [Balou91].

**Message-driven architecture** - Objects can be represented by the self-contained computer metaphor, i.e., an object is a "computer" capable of handling its own needs. At this level, computation is defined by a distributed flow of messages originating and terminating on the objects themselves. The BROOM architecture is an implementation of this message flow paradigm. This is an architectural proposal for a class of machines that are totally based on the object-oriented paradigm.

**High level pipeline engine** - In the self-contained computer metaphor, objects decode and process messages as instructions. Due to the loose dependency and dynamic nature of message flow, messages can be pre-processed to a large extent. The BROOM node applies this concept to keep a high-hit rate small cache, maintained by pipelined processing of the message flow. This is an extrapolation of the pipeline instruction handling mechanism, applied to the level of messages, combined with a forward-looking caching strategy [Pachec89].

**VLSI node processor** - This is a 2-micron CMOS prototype of the node processor, implementing the concepts described above. It is a network building
block for the distributed object model with three pipelined internal units, matching
the virtual machine architecture. This machine is an application of the object-
oriented methodology to VLSI design. The units are designed as objects,
minimising the problems of alteration or replication in future implementations.

**Modular datapath CMOS library** - the VLSI processor datapath was
designed using a modular horizontal slice library. Registers can be built with
customised number of access and fitted with auxiliary logic to perform increment,
decrement, comparisons and shifts. A generic functional unit is provided, together
with bus interfaces and padding slices. This library is of general utility and has been
used for the implementation of three different projects [Pachec91], [Vellas91] and
proposed to others, including a silicon compiler.

1.5. *Thesis Organisation*

Chapter two investigates the object-oriented computing model. Object-
oriented systems, languages, environments and architectures are reviewed to
identify the paradigm characteristics and some possible implementations.

Chapter three presents the adopted architectural model at three levels: the
abstract message-driven object-oriented model, the target intermediate level
language (TOOL) and the parallel virtual machine (VOOM).

Chapter four introduces the Basic Regular Object-Oriented Message-
driven Machine (BROOM). It explains the message flow concept for a distributed
memory multi-node machine and discusses implementation issues on network
topology and system components.

Chapter five describes the design of the BROOM node architecture. It
details the function of the pipelined units responsible for transport, decoding and
execution of messages. It discusses the design choices for organisation and implementation of the node processor.

Chapter six describes the architecture simulator. It explains the program role in developing the features for the BROOM architecture. The simulation is dedicated to evaluate the high level message flow architecture, mapping directly the BROOM behaviour with a simplified, highly interactive implementation.

Chapter seven discusses the work involved in the VLSI implementation. It discusses the design approach, taking into consideration the tools available. It also describes the modular library developed for the project. Some low level design considerations are also discussed.

Chapter eight is an assessment on what was achieved with the work in three levels: the virtual level, architectural design and physical implementation.

Chapter nine draws some general conclusions and proposes ways to advance the research and developments initiated in this work.
Chapter 2: Object-Oriented Systems

In this chapter, a model for object-oriented computation is presented. Object-oriented languages and environments were studied to outline this abstract computing model and underlying architecture. In addition, current architectures were investigated in search of features that could be exploited in the development of an object-oriented machine.

2.1. Object-Oriented Computing Model

2.1.1. Procedural Model vs. Object Oriented Model

It is possible to identify a computational model examining the way code, data and state interact during computation, forming processes. Based on this we can draw a comparison between the procedural and object-oriented model. Figure 2.1 depicts the assembling of processes in both procedural and object-oriented computational models.

![Diagram showing the difference between procedural and object-oriented models](image)

Figure 2.1: Procedural and Object-Oriented Models
In the procedural model, code, data and state are maintained in three separated structures namely code segment (C), data segment (D) and process stack (S). During computation, these three entities are assembled to form a process, under the action of the activation record.

In contrast, as can be seen in figure 2.1, the object-oriented model keeps code and data together in a structure called object. The state (S) is kept in a context structure and is normally added to the object to form a process through an entity called message.

Another important difference between the two models is that processes in the procedural model normally have a coarse granularity, and therefore are much less related to each other. Objects have a much finer granularity, and are more interrelated, in a way that the message flow represents the whole process behaviour, i.e. messages can be considered as "instructions" in a program.

2.1.2. Object-Oriented Model Variants

The relation between the messages and objects in a computation can be used to characterise two variants in object-oriented models. When the arrival of messages determines activation of latent objects, the model is message-driven, while when the object is in charge of selecting the messages to receive, the model is object-driven.

In the object-driven model the objects are active from the moment they are created [Americ86]. They are provided with a special method, called body, that defines the behaviour of the object when it is created. Messages are always sent synchronously but can be answered asynchronously, i.e., the object can continue to execute its body after returning from a method. When an object is ready to receive
messages, it can search the system message pool to select the message that will execute next.

In the message-driven model, messages play the role of activating objects. An object is created in a latent state and will remain in this state until a message arrives to it. The message will then activate the object to execute a selected method. When the method terminates, the object will return to a latent state. Messages can be sent synchronously or asynchronously, i.e., after sending a message the current method can terminate or continue its activities and even send more messages. Messages are received in the order of arrival, but an object may ignore messages that do not match certain criteria.

Figure 2.2 represents the behaviour of the two object model variants. The object-driven model has a body that controls the scheduling of synchronously sent messages. The body is always active and is responsible for the execution of arriving messages. Messages arrive asynchronously and can activate a suspended body or wait in a queue for a body to select it.

Figure 2.2: Message and Object-Driven Models
In the message-driven model, an object is active only at the reception of a message. A method is then associated with an incoming message, bringing the object to life. The method is responsible to perform the required transformation on the object. Messages in this model, unlike the object-driven, can be issued asynchronously. In both object-oriented model variants, objects maintain their encapsulation, being accessible only through messages.

2.2. Object-Oriented Languages and Environments

2.2.1. Sequential Languages

Sequential languages like Simula [Dahl70], Smalltalk [KaePat86], Actor [Duff86], Eiffel [Meyer88] had the merit of pioneering into object-oriented scenario. Important concepts like encapsulation, message passing, data abstraction and inheritance were primarily developed in these languages. They are still the main source for exemplification of these object-oriented characteristics.

In the origins of the object-oriented paradigm we can find a similar approach with abstract data types [Ishika84]. The main difference between an abstract data type and objects is that the former is always passive, being activated by its class, while the latter is active, responding to messages itself.

Simula started the idea of generic classes structured in hierarchies as an extension to ALGOL 60 [Jona88]. This concept culminated with the advent of Smalltalk [Goldbe83]. This language consolidated the concept of active objects in substitution to abstract data types. Many other important language characteristics of the object-oriented paradigm were gathered in this language. Polymorphism is exploited to the maximum with the late binding mechanism that carries all method binding to the run time. Only during execution are the parameters' types fully specified and ready to determine the correct procedure to process them.
However, this new computing model was not supported by the available architectures, so that it had to run on the top of an interpreted virtual machine. The resulting slow implementation stimulated the appearance of other languages like Actor, with optimisations for the problem [Duff86]. Actor introduced the option for early binding, and modifications on the interpreter to accommodate object switching to a better proximity to the procedural model. Other languages like C++ [Strout86] and Eiffel [Meyer88] abandoned the interpreted solution and pressed for early binding in detriment of polymorphism.

2.2.2. The UC++ Language

This language accommodates the procedural model so well explored by mixed level languages like C. An active class definition adds the necessary power for parallel ambitions, following the object-driven model [Robert90].

As a language extension for C++, UC++ inherits all its characteristics. C++ is more an object structuring of the C language then an object-oriented language of its own. It introduces the concept of hierarchy of classes based on derivation, where the sub-classes add new members and methods to the available from its parents [Strout86].

UC++ introduces parallelism by means of the object-driven model. Classes can be declared active and then be entitled to a body method. At creation time all the bodies of the active objects are started. Message passing conforms to the receiver side parallelism protocol, i.e. methods can return and continue active, but messages are always sent synchronously [Way90].

\[1\] generic name for the contents of an object.
UC++ constitutes a classical example of adaptation of procedural languages both to the object-oriented and parallel paradigm. It is done in a way to minimise the impact of the conversion on the performance obtained in conventional machines. This introduces very little of object-oriented benefits, since the structuring of the program is rather an option than a programming style.

Encapsulation can be fooled with pointers and class friendship, and message passing is synchronous to avoid mishandling of the procedural stack. This results in the best of performance in conventional machines but do not contribute to a better definition of an object-oriented machine. On the contrary, object-oriented machines that propose to support it will have to relax encapsulation and accept pointers across object boundaries.

2.2.3. Orient84

Orient84 is an example of message driven parallelism. Messages are responsible for activating objects, that in turn can reject messages on the terms of an access list. One unique characteristic is the merging with a Prolog-like declarative language \cite{TokIsh86a}.

The language structure and syntax are derived both from Smalltalk and Prolog. This includes class hierarchies and inheritance. Messages are normally synchronous, blocking the object until they return. Parallelism has a very simple implementation, with an extension that permits non-blocking messages that do not return anything \cite{TokIsh84}.

Programs are descriptions of the object classes and methods. A class in Orient84 defines the inheritance list and the properties of the class (and of its instances). These properties include the variables, access list, method implementation and knowledge base. Inheritance is behaviour-based, i.e., only
methods are visible through the sub-class, unless for the knowledge part, where rules and facts are visible throughout the hierarchy lattice.

Orient84 exemplifies a combination of object and message-driven flavours. The main emphasis in the language is the security aspect, represented by the access lists. The access list introduces a visibility system that is tightly controlled by the object and can be redefined dynamically at run time. The access list defines the set of objects that can communicate with the current one and the subset of methods available to them [TokIsh86b].

This mechanism can emulate the object-driven behaviour by changing the access list at run time. Messages are reentrant and preemptive, controlled by the dynamical selectivity of the access list. Preemption and reentrancy are powerful language characteristics but impose expensive requirements for hardware support, adding to the complexity of managing the dynamics of message access selection.

2.2.4. POOL

POOL introduced the concept of object-driven computation and provides parallelism with active-from-creation objects. The message selection mechanism provides the language with a run time object-controlled variable method behaviour.

POOL has a very simple syntax but with complex semantics. As an object-driven language, objects are in control of the reception of messages and can even arbitrate what behaviour will correspond to any message received. Messages are always synchronous and parallelism is obtained by the creation of active objects or post-processing a message after it has been returned [Americ86].
A program is composed of object descriptions that are made in three different units. One of the units defines the root objects that are objects instantiated and activated when the program is loaded. Visibility can be tailored by use-lists that recursively define which objects are accessible to a certain object. It then follows that an object does not control its own access but instead, the objects that access it detain this prerogative.

POOL closely characterises the object-driven model, with its absolute control of message handling by the object. The object decides when to receive what message and can decide on the response to the message according to its state. Active from creation objects constitute a useful parallelism model, although restricted to the receiver side message asynchronism (the sender has to wait the completion of the message but the receiver can continue after returning). The select instruction exemplifies how complex can be hardware implementation, with message queues that can be searched and sorted by the object in command.

2.2.5. MS Windows

The message-driven, non-preemptive multitasking Microsoft (MS) Windows environment is a commercial example of object-oriented system emulation. Its device contexts and window objects explore the abstraction power of object-orientation. The fixed format messages and inter-application links anticipate what could be a fine granularity object system.

MS Windows is a Graphic User Interface (GUI) sitting on the top of a quite primitive operating system, MS-DOS. The great merit of this environment is to provide a standard object-oriented interface both for the user and the programmer. Although the programmer interface is not effectively written in an
object-oriented language it presents itself as a collection of predefined messages that can be sent to predefined objects [Micros87].

Windows applications can multitask in a non-preemptive style controlled by the application itself. The application can choose a moment where the contexts are minimal, reducing the switching overhead. This is performed by a receive&release message that indicates that the application is idle. An idle application is waiting for a message and can transfer the processor control to another application. Object switching can also be minimised through DLL's (Dynamically Linked Libraries) that are portions of the application loaded under request at run time, optimising the object working set [Petzol88].

Multi-tasking using messages is an interesting application of the message-driven model. Another relevant feature is the application level intercommunication through message passing. The practical implication of the mechanism is that granularity of applications can be very fine, with specific tasks being executed by object-sized programs. This is provoking a trend were large applications are in reality collections of smaller applications originated from distinct programmers. This demonstrates the usefulness of object-oriented approach for shared development of large programs.

2.2.6. Conclusions

Analysing these object-oriented languages leads to the conclusion that they represent a higher level of computation. The main indications of that is the data abstraction provided by object encapsulation and the computational flow originated by the message passing mechanism. In concurrent languages like Orient84 and POOL, the message flow controls the flow of parallel execution,
directly or indirectly. In the MS Windows environment messages not only control but are the source of concurrent execution.

The main concern of the object-oriented programmer is the management of message flow rather than the design of routines. Objects are service stations with elaborated computing capabilities. The mechanics of an object are abstracted from the programmer, who needs to know only about the object's interface.

Meanwhile, architectures are still oriented to crunch bytes of information as the maximum recognisable computation unit. The current RISC trend tends to reinforce this view even further [Patter85]. In fact, computation within methods is mainly procedural and can be supported by a RISC machine. However, the weak point concerning object support is the lack of an upper layer of processing that can account for message flow handling.

2.3. Object-Oriented and Parallel Architectures

2.3.1. Sequential Architectures

Architectures like Rekursiv [Harlan88] and SOAR [Ungar84] try to convey the semantics of sequential object-oriented programming. A revival of the microprogrammed architecture is the approach used in Rekursiv to reduce the semantic gap created by object-oriented languages. SOAR tries to minimise table look-ups with method reference caching strategies for method run time binding.

SOAR (Smalltalk on a RISC) is, as its name says, an architecture devoted to interpret Smalltalk within its hardware implementation. The main concern is with the optimisation of run time binding, which is claimed to consume a large percentage of the processor time. The strategy employed is to perform an optimised look-up when binding a method and go back to the caller to substitute
the symbolic reference by an absolute one. This method reference caching is interesting and can be applied to migrating object references.

Rekursiv is a machine that approaches more directly the object-oriented model. Objects are represented in an object-oriented memory, with boundary protection, permanent storage and automatic garbage collection of pages and references. Methods have the special support of microprogrammable instructions with a powerful microcode control, permitting recursive calls inside routines. The machine is implemented in three units: one for program flow control, one for numeric processing and one dedicated to object handling.

Architectures like SOAR and Rekursiv have pioneered in features for object-oriented support that can be used for the design of a node processor in a parallel system. The reference caching in SOAR and dedicated units in the Rekursiv for object structured memory management are good examples.

2.3.2. Inmos Transputer

The transputer is a building block for parallel architectures. It consists of a stack-based processor with built-in serial asynchronous links for communication. It has a scalable instruction format, independent from the word size, that produces very compact code. The computational model is process based, time sliced multitasked with fully synchronous message passing mechanisms. Processes are automatically scheduled by the machine from a list kept by two internal pointers [Homewo87].

Although being a distributed architecture, parallelism is restricted by the synchronous channel protocol. It is the most successful architecture for process-based parallelism, but there is no hardware support for message routing or binding. However, it has a compact, scalable instruction set, together with a neat, lean stack
top based internal organization [Whitby90]. These economic features can be proposed for fine granularity machines, where the silicon area can be used at its best.

2.3.3. ZOOM

ZOOM (Zen Object-Oriented Machine) is a complex abstract architecture supporting Orient84. The solutions for strong typing with high encapsulation and security are extensive and hard to implement. The class tables and access protected ports provide high flexibility and robustness at elevated implementation costs.

As a message driven architecture, its functionality can be understood by following the message path. The virtual communication topology is based on point-to-point connections between objects. A message to be sent must open a port in the sender object that will be connected to a port in the receiver. When a message is sent it is bound through a global/local dictionary in the destination processor and assigned to an object. The message sender is checked first against the object access table and then against the destination port access table. If the message passes the double check, the object class is retrieved from the class template table and the method indicated in the message is started. Messages can be sent synchronously or asynchronously. Messages blocked by the port access mechanism may remain in the port buffers until a receive instruction opens the access to it [Ishika84].

Objects can be primitive or user defined and user defined objects can be classes or instances. Objects are represented by a name and a contents list. The name includes an identification field, the object access list (scope) and the object type (property). In the contents part there is the class name, the private variable name list, the execution status slot and the port access lists. Class representation
has the same structure, except for the class field that is invariant. Primitive objects
contain an in-line table of method names and object state.

The machine representation is extensive and complex, yielding to a
complex implementation. The flexibility provided in for operating modes results in
an extensive instruction set with thirteen send/receive instructions and ten binding
instructions. The implementation would require 30 registers of 50 bits each
[Ishika84].

2.3.4. DOOM

DOOM (Distributed Object-Oriented Machine) is the abstract machine
underlying POOL. The notable queue mechanism is the embodying of the message
passing paradigm, but all operated by software. The hardware implementation is
more dedicated to communication routing and message assembling in a hypercube
network topology.

DOOM allows for more then one object to share a processor node. The
system architecture is a network of nodes with distributed data memory and shared
code memory. The node architecture is composed of a communication routing unit,
a virtual, software controlled queue and memory manager, and a stack based
execution unit.

DOOM is based on the object-driven model, with the object bodies being
in charge of the message flow. Objects are associated with stacks in the execution
unit, one stack for each object. The stacks contain the evaluation stack and object
data. Messages are collected in many queues, one for each method of each active
object. These queues are organised in a FIFO manner with the heads available for
selection by the current active object [Bronne87].
Object bodies are time-sliced to provide multi-tasking and are able to select the method that will examine its queue. There is also a special port for the direct reception of return messages. Since message passing is synchronous, return messages by-pass the queues to free blocked objects.

DOOM and its upgraded implementation POOMA [Americ90] represent the closer link that object-driven system presents with procedural languages. The node architecture is implemented with off the shelf components, using the transputer (DOOM) to implement the communication unit and the Motorola 68020 for the execution and queue/memory manager units. The main focus of the communication unit is to provide automatic routing in packet switched protocol. The routing is based on an efficient starvation/deadlock free algorithm for optimised chordal ring topology. POOMA is a good implementation of a massively parallel object system, but the object-driven model makes it difficult to exploit lower levels of parallelism available in the message passing mechanism.

2.3.5. DICE

DICE [DelRou90] is a simple implementation of a message-driven system, designed to support a parallel version of Smalltalk. Procedural threads are supported through a chain of linked messages. A very realistic approach towards communication and network topology is a strong point in the architecture.

DICE architecture exemplifies user-parallelism [Delgad90] in a distributed system. The machine is designed to allocate clusters of processors to each user, instead of dedicating all of them to a single program. Following this principle, the system architecture is arranged in clusters of processors interconnected through a tree of busses network. The interconnection through busses stimulates locality, providing a very large bandwidth, low latency local communication [Delgad87].
The computing model is message-driven, with both sender and receiver asynchronous message transmission. Synchronous messages link with each other forming process chains, where only the last message in the chain activates a method. Since methods can be reentrant, allowing for asynchronous returns, the integrity of object data must be maintained by locking semaphores called futures, that block invalid consumers and overwriting producers.

DICE is designed for automatic scalability, providing support to automatic garbage collection and dynamic load balancing through the use of symbolic references. Its design supports important object-oriented features, but little of that is reflected for the benefit of the architecture. The lack of architectural enhancement is mainly due to the implementation being based on conventional off-the-shelf processors, with all the special object-oriented support being provided through interrupt handling routines. This emphasises the importance of carrying the design down to the organisation of a VLSI node processor.

2.3.6. Conclusions

Elements to support object-oriented languages can be found in a variety of current architectures. However, few machines exploit the full potential parallelism of the paradigm. Furthermore, none of the machines has been designed to benefit from object-oriented features, enhancing node performance by taking advantage of encapsulation, message passing or fine object granularity.

Several points in the examined architectures are important contributions that can be used for the design of an innovative object-oriented architecture:

- Rekursiv introduced the idea of specialised units for permanent object management. This can be used to design the mass storage module suggested in section 4.1.3.
• Rekursiv revived the idea of microprogrammed instructions, but the implementation with off-chip memory cannot compete with on-chip RISC speed. However, some specific high level operations can be supported by hardware and microprogramming can take different approaches as described in section 4.2.3.

• The Transputer has an excellent instruction set for compacted implementation and introduced the concept of a parallel building block [Nicoud89]. The Transputer instruction set originated the set used in the BROOM architecture as mentioned in 4.2.4.

• DOOM presents an interesting mixture of shared and distributed architectures but bottlenecks on the shared half must be avoided. DOOM communication architecture inspired the virtual machine and influenced on the design of object-driven model support (see section 3.3.3)

• DICE is a straightforward machine for the message-passing model with an elegant solution for communication, that take into account the compromise of minimum silicon area for the maximum exploitation of pin count bounded bandwidth. DICE network topology was chosen for BROOM's system architecture (section 4.1.2)

The proposed architecture of VOOM and BROOM not only congregates a collection of support features, but coordinate them into a coherent architectural style. BROOM is an effective representation of the object-oriented computational model, not only an adaptation of it into conventional architecture.
ADDENDUM

After the conclusion of the thesis, I was presented to an important work I did not came across when surveying the area. It is the MDP, a parallel message-driven processor developed in the MIT [Dally87]. The MDP is an original message-driven architecture dedicated to fine granularity, object-oriented computation.

The machine provides two level of computational flow, the higher being a message flow, the lower a conventional instruction flow. The machine is controlled by two distinct engines: the Message Unit (MU) and the Instruction Unit (IU). The Message Unit works independently of the Instruction unit, decoding and queueing the messages in background. Process are activated by the arrival of messages, that can preempt process with lower priority.

Special hardware support is given to message passing, object switching and process binding. The overall architecture takes advantage of fine granularity, providing on-chip memory and reduced method address space. This results in a small size silicon implementation. Message passing is managed by a hardwired controller and a on-chip communication unit. Object switching is supported by dual register sets. Process binding relies upon a translation mechanism implemented with associative memories. Although translation is done in a single cycle, binding is controlled by software and can take several execution cycles.

This architecture is the closest to BROOM and has important contributions to the message-driven computational paradigm. It supports many message level built-in instructions, including access to private state from other objects. The translation mechanism supports truly symbolic references. On the other hand, node parallelism is not fully exploited. Message binding uses execution time and various traps, such as page faults, interfere with method execution.
Symbolic addressing with associative memory and more extensive message level instruction support, present on MDP, are only suggested in VOOM and BROOM. However, a better performance for fine grain processing can be achieved in BROOM, with the high level pipeline that remove all binding and cache management from the execution time. Separate memories for object, method and messages also gives the potential for faster clock timings and more elaborated instruction pipelining. The MIT-MDP is an excellent example of message-driven architecture, but BROOM points to a more extensive exploitation of advantages of the object-oriented model.
Chapter 3: Virtual Machine

This chapter describes the virtual object-oriented machine used to design the architecture presented in this thesis. This virtual machine, VOOM, together with TOOL, the Target Language, form a system meant to convey the high level language requirements towards the design of a hardware architecture suitable for object-oriented computation.

3.1. Message Driven Computing Model

The message driven object-oriented model was adopted for the design of the intermediate level virtual machine VOOM. In this model, messages are responsible for the computational flow, activating all execution tasks. Analogous to instructions in a von Neumann machine, messages are the computation unit in the message-driven machine. This model was chosen because, we believe, it allows modelling of both the intermediate language and architecture in a more intuitive manner than other flavours of object-oriented computing model.

The message-driven model characterises a computational model where the control flow is unidirectional, i.e., originating always from the message to the object. In the object-driven model, control originates predominantly from the object, but also sometimes from the message. This results in a complex environment management that must be kept for all objects.

The message-driven model reduces the troubles with this management and maintains a useful analogy with the von Neumann machine, making use of established know-how. Although the object-driven model is closer to the programmer point of view and offers control advantages over the message-driven, in terms of hardware the latter can offer much better prospects.
In the message driven computing model, an object is latent until a message arrives to activate it. The message carries the execution state that indicates the passage of control to the selected method in the destination object. For the design of VOOM and TOOL we have decided on an atomic execution of messages, i.e., the message is executed to completion and the processor is only released under the method command. The atomic execution precludes the interruption of methods, avoiding context switching overheads and keeping the consistency of object state [Balou88].

An object in the message-driven model can be considered as a frozen process. This frozen process contains the code and data necessary for computation but is prevented from proceeding due to the lack of execution state. Messages carry the execution state and the binding of a message with an object instantiates an active process. Messages, in this sense, control the computational flow, passing the execution from object to object. In a message driven system, parallelism can be easily obtained, since more than one message can activate different objects at the same time.

In the adopted variation of the model, messages are executed atomically, i.e. during the execution of a method activated by a message, another message cannot be accepted for processing. A message can only be accepted by a busy object if it is a return message and the object execution is suspended, waiting for that particular message.

The message passing protocol for the message-driven model is illustrated in figure 3.1. The message flow is represented by a chain of linked messages (M₁, M₂, M₃). The execution states (S₁, S₂) are shown attached to the respective messages. When a message Mᵢ is created, an execution state Sᵢ is created and
attached to it. In the case of return messages, the execution state attached to it is the existing one derived from the suspended requesting object.

In figure 3.1, messages point to their destination objects and a faint arrow denotes the sender. Message $M_1$ activated object $O_1$ that sent the synchronous message $M_2$ to object $O_2$. Object $O_1$ has been suspended and object $O_2$ is sending the return message $M_3$, that is associated with execution state $S_1$ created for message $M_1$. Any other message sent to $O_1$ would be suspended until $M_1$ has been discarded, allowing the association of a new execution state to $O_1$.

This strict message driven model has been relaxed in TOOL and VOOM to allow the introduction of elements of object-driven computation. The behaviour of return messages is better defined in the object-driven model. The semantics of the model implemented in VOOM normalises these singular messages without penalising the smoothness of operation.
Object-driven computation is supported by sending a special starting message to objects with body. This message contains an empty execution state associated with it and allows the object to run its body and receive other messages by preempting the previous execution state. The selection mechanism added to accept messages in a body plays the role of integrating return messages in the normal message flow.

3.2. Intermediate Language TOOL

3.2.1. General Description

TOOL is the Target Object-Oriented Language derived from the message driven object-oriented model described in the previous section. As a target language it is intended to be the code generated by higher level languages. TOOL is also a target for the design of architectures suitable for object-oriented computing. The definition of TOOL and VOOM is, consequently, the first step in the three level methodology used to define, design and implement the architecture described in this thesis.

A program in TOOL is a collection of objects. Objects are collections of message statements. Objects in TOOL can be declarative or executive, the former specifying objects containing data and the latter, objects containing methods. These objects form data or code images used as class templates and instance objects alike. Methods are defined by low level message statements that execute on VOOM predefined objects (like integers, characters, etc.) or are compiled into higher level messages, executed by user-defined objects. Messages determine the computational flow in TOOL. They are created within TOOL objects and start object processes as described in the previous section.
The declarative objects in TOOL are mainly specified by message statements called declarative-operations. Declarative-operations are instructions to the TOOL translator, interpreted at compile time, specifying placeholders for data and references to other objects. Objects in TOOL are created and deleted explicitly, and the definition of an object in TOOL implicitly creates one instance of it. Primitives are available to bind objects to a node or migrate them across the network. Objects are created latent unless they are associated with a body. The body is a stand-alone method and is started by an implicit message as soon as the object has been created.

Executive objects define methods containing code. They can be independent or be bound with an object to activate implicit migration of it when the object moves. The code is specified by a TOOL object containing executable message statements. These messages are classified in three types: primitives, built-ins and user-defined. Primitives and built-ins are executed by the virtual machine's predefined objects. User-defined messages are high level messages transported by the virtual machine to an object defined by the user. The user-defined messages are the effective messages that communicate across objects.

3.2.2. Syntax and Constructs

TOOL has a very simple syntax, similar to Unix shell statements. Each statement in TOOL is a message with an object destination, a method invocation and parameters. These message statements can be issued synchronously or asynchronously and incoming or outgoing parameters can be accessed through positional registers.

A program in TOOL is a list of object (declarative or executive) declarations. These object declarations are designated to be object data templates
or method implementations. The object declaration itself is a list of executable message statements. These message statements allocate space for data, assign symbolic names to entities or are symbolic representations of operations that will unfold at run time. A summary description of TOOL syntax in BNF notation follows:

\[
\begin{align*}
\text{<program> ::= } & \{\text{<object_def>}\}^* \{\text{<method_def>}\}^* \\
\text{<object_def> ::= } & \text{<object_id> }\{\{\text{<decl_part>}\}^* \{\text{<method_def>}\}\} \\
\text{<method_def> ::= } & \text{<method_id> }\{\{\text{<bind_op>}\}^* \{\text{<message_stat>}\}\} \\
\text{<decl_part> ::= } & \{\text{<object_id>.<declarative_op> }\{\text{<argument_list>}\}\}^*; \\
\text{<declarative_op>::= } & \text{instance | method | inherit | bind | start} \\
\text{<message_stat> ::= } & \{\text{<label>}:\}^* \{\text{<object_id>}+.\text{<method_id>}\{\text{<arg_list>}\}\} <\text{node}> \\
\text{<method_id>::= } & \text{<primitive_method> | <built_in> | <user_defined>} \\
\end{align*}
\]

In an object definition, methods can be defined in-line or off-line. In-line methods are bound to an object for migration purposes and off-line are independent, requiring a separate definition module. The declaration part of an object definition contains message statements that attribute symbolic names to instance variables in the object and declare methods available to the object. The declarative operation bind suggests a user-defined allocation for the object and start identifies the objects with bodies that will start when created. The declarative operation inherit defines a list of objects that are parents to this object.

Message statements (message_stat) in their more general form are used to specify the behaviour for methods. The object_id is the destination for the message and is an identifier declared as instance in the declaration part. The method_id identifies the method invoked to modify the destination object. In the next section, the different types of methods will be discussed. The argument_list contains constants, variables and labels passed as parameters to the method. The mode

\(^1\) the symbol * denotes one or more and the symbol + denotes one or zero
denotes whether the message is synchronous or asynchronous. In the asynchronous mode the method proceeds after the message is sent. In the synchronous mode the object is suspended and waits for the return message. The future mode is a particular case of the synchronous mode. It provides a special label in the argument_list and the execution of the method branches to the label upon the reception of the return message.

The argument list appended to messages constitutes what was defined in the model section as the execution environment. In TOOL there are two execution environments always active during the execution of methods: input and output. The input environment is received by the object when a message activates one of its methods. The contents of this environment can be accessed through positional parameters that follow the Unix shell notation and are preceded by a "$" ($1, $2, etc.). The input environment enables the access to incoming parameters and can be also a local dynamic store for temporary variables used by the method.

The output environment is used to build outgoing messages and its contents can be accessed through positional parameters preceded by "%." The input environment may persist throughout the execution of a method but the output environment is not guaranteed to survive the execution of a message. The output environment is designated to support manipulation of expressions and is also the place-holder for the result of message statements inside the method.

3.2.3. TOOL Message Statements

TOOL semantics is based on the concept that every statement is a message directed to an object. A message determines, through its method, the action to be taken. The nature of the target object and method operation allows methods to be classified into: declarative-operations, primitives, built-ins and user-defined. This
classification is related to the level that implements the function, and is a good guide for the design of TOOL supporting machines.

Declarative operators are methods used in an object declaration to define variables, identifiers and assign attributes to the declared object. The methods instance and method attribute a symbolic name to instance variables and member methods of the object being declared. The method bind suggests where to load the object in a multi-node network. It also defines the association of methods and objects for the purpose of migration. A method bound to an object will move with the object if it is moved to another node.

The start method will provide a starting message containing an empty environment that will enable the body of an object to start as an active process. The inherit method provides multiple inheritance as a list of parent objects. The list is checked at compile time to allow methods pertaining to parents to be associated with the objects declaring the list.

Primitive methods reflect the inherent facilities of the underlying architecture: control, transformation and transfer of the private data of an object. A summary of these methods is given below, in BNF notation:

```
<primitive> ::= <arithmetic> | <logical> | <relational> | <memory> | <control> | <select_op>
<arithmetic> ::= ADD | SUB | MUL | DIV | NEG | REM
*logical* ::= OR | AND | XOR | NOT | SHL | SHR
<relational> ::= EQ | LT | GT | LTE | GTE
<memory> ::= STORE
<control> ::= BR | BRT | BRF | RTN
<select_op> ::= SELECT
```

Message statements containing primitive methods normally omit the destination object since it is implicit. The destination object is one hardwired unit
of the virtual machine capable of the selected operation like the ALU, responsible for arithmetic and logical operations or the control, responsible for conditional and unconditional branches.

Built-in methods also reflect routines intrinsic to the virtual machine. These routines, however, apply to user-defined objects instead of the predefined object handled by the primitive methods. Built-in methods require the specification of the destination, otherwise the destination defaults to self, i.e., the object that is issuing the message. Some of the methods involve non-trivial operation that may be emulated by software drivers or may require special hardware to be executed:

\[
\langle \text{built\_in} \rangle ::= \text{new} | \text{delete} | \text{replace} | \text{rebind} | \text{revive}
\]

New and delete are the methods dedicated to the explicit creation and destruction of objects. The new method instructs the object to use itself as a template to create a new instance of its class. Arguments for the new method override the default values in the present state of the object. If the object declares a bind declarative operator, the presence of an argument in the bind position allows the new instance to be allocated to another node. Delete will reclaim the memory allocated to an object to be used by another instance.

The replace method is a variation of new and allows for the substitution of the contents of an object, even if the object is a method container. The replace method allows run time modification of methods. In the current version of the syntax, the first argument is a positional index indicating the starting point of the modification. The subsequent arguments are values or quoted messages (a message statement surrounded by quotation marks) that will be translated to substitute the existing ones. If the index is negative the next argument must be an object or
method reference and the whole object will be substituted by the referenced object.

The *rebind* method instructs an object to migrate to another node, regardless of the presence of a bind declarative operator. The *revive* method is used to restart the body of an object that has been terminated. It is equivalent to the declarative operator *start*, but is effective at run time.

User-defined is the highest level method definition and is fully programmable by the user. The valid *user* method references are those declared within the destination object declaration part or in any other object included in the destination inheritance list. User-defined methods must be implemented as in-line or off-line lists of executable messages. A user-defined method will be executed as an *object process* when a message invoking it is accepted by a dormant object.

### 3.2.4. Conclusions

**TOOL** was designed to be a practical representation of the message driven object-oriented model. Elements of the object driven model are also present to ensure compatibility with a wider range of languages. The central concept in **TOOL** is that computation is carried on through the execution of messages. This concept is the essence of the object-oriented paradigm, and grants **TOOL** a consistent uniformity and abstraction. The virtual machine underlying **TOOL** takes advantage of this concept, implementing methods in various levels of complexity that share the same *regular structure* of a message.

The regular design of **TOOL** played an important role in the definition of the abstract machine **VOOM**. There is a direct correspondence between the instructions on both representations. Registers and architecture organisation also
These correspondences make VOOM’s implementation a smooth translation of TOOL, preserving most of its characteristics.

Extensions were added to TOOL to cover a wider range of object-oriented languages (c.f. bodies). This reflects the view where TOOL is a target for compilation of high level languages. Other extensions can be added to TOOL to meet requirements of other paradigms. An example of it is IIR, a dialect of TOOL modified to be the target for a process-based language, PARLE [Refene89].

Some architectural aspects of TOOL may still need some refinement. It depends on a longer research on the virtual machine. Elements like the positional registers may be implementation dependant and some redefinitions can result in a better match for hardware implementations. The semantics and syntax of predefined messages may vary to accommodate advances in the understanding of the computational model. An example of this can be the development of optimised algorithms for object allocation that can change the mechanics of built-in methods [Balou91].

3.3. Virtual Machine VOOM

3.3.1. System and Node Architectures

VOOM is a network of processing nodes capable of multitasking objects activated by messages. The network architecture permits message broadcasting, although it is not currently implemented in TOOL. Hence, messages in VOOM have a single fixed destination. The register structure follows TOOL structure, with two register sets, one for incoming and another for outgoing parameters. VOOM is structurally organised in three parts, one dedicated to the handling of messages, one for the management of objects and the other for the execution of methods.
At system level, VOOM represents TOOL as a collection of objects distributed among the nodes of a scalable network of encapsulated processors. TOOL objects are allocated to VOOM nodes according to the suggested explicit placement or under a distribution algorithm [Balou91]. The nodes communicate through a broadcast network that interposes a single step between the sender and the receiver. The simplest implementation of this topology is a bus shared through an arbitration protocol. Better implementations can be achieved with frequency multiplexing or emulation through other topologies.

The VOOM node has a specialised organisation to support the object-oriented paradigm expressed by the TOOL language. The process flow in TOOL starts with the reception of a message by an object. This message selects a method that will produce modifications on the object state. The VOOM node incarnates a collection of objects resident in it. The VOOM node will receive and queue all the messages addressed to these resident objects. From this queue, VOOM will select the first message and activate the corresponding object, associated with the selected method. The method will be executed by the node processing facilities to completion or until being suspended. Once the execution of a message is suspended, another message will be scheduled. The VOOM node is organised in three units: communication unit, dedicated to the handling of messages; prefetch unit for linking messages to object and methods and the management of memory pages; and the execution unit for the execution of methods.

The communication unit is responsible for organising the messages inside the VOOM node and routing them across the network. It stores the messages generated by the node in an output queue and the messages received by the node in two input queues. The output queue buffers messages generated by methods.
executed in the node until these messages obtain access to the network and can be posted to the destination node.

The two input queues receive different type of messages: active and suspended. The active message queue collects the messages received by the node. The inactive queue saves messages that were suspended during execution. The messages in the active queue are divide into logical sub-queues according to the message priority and arranged in the arrival order. The queues are served in order of priority, but in a round-robin fashion to avoid starvation. The suspended queue contains messages of methods awaiting synchronous responses. When a return message is received it is matched with the corresponding suspended message and execution is resumed.

The prefetch unit links messages to the corresponding object and method. This unit searches the queues in the communication unit for the next message to be executed. It gets the message and tries to find the object and method associated with it. The node has three levels of storage: cache, local memory and global store. The first level is a cache memory, where execution of methods takes place. The second level is the local node volatile memory, where most of the resident objects are loaded. The third level is a system global shared mass store where all objects in the systems are stored. The prefetch unit searches these stores levels and whenever the object is absent in the top levels it starts a procedure to copy it to all levels above. This prefetch procedure proceeds until all messages in the queue have their objects and methods in the cache.

The object and method positions in the cache are stored within fields of their associated messages in the input queue. The execution unit is responsible for getting the first bound process in this queue and executing the associated method. Execution is only allowed on objects and methods residents in the node cache.
During execution, two register sets are accessible to the method: the $ set, associated with the received message in the input queue; and the % register associated with the outgoing message in the output queue. The method execution proceeds fetching two operand instructions from the cache and executing them into an accumulator (%1 register) based datapath. When the execution terminates, the object and method are purged from the cache into main memory and another process is started.

### 3.3.2. Object and Message Formats

Objects, methods and messages are encapsulated in a data packet used to represent these entities all over the storage and communication media. The packet formats are similar for the three entities, with an information header and a body containing data. The header is divided into several fields that identify the object, the class, size, and in case of messages, other fields are added or omitted, according to the data type.

The message has a variable format that accommodates several types of information transfer. The various formats, for different message types, are shown in figure 3.2. User messages are generated when user-defined methods are invoked. Return messages return result and control to suspended methods. Frozen object messages transport objects or object copies across the network. System messages are invocation of system routines, c.f. built-in methods.

The first record in a message contains its type. The type defines the rest of the message format. All messages contain a timestamp, used for system statistics and optimisation routines like load balance. Executable messages (not frozen objects) have a field discriminating the sender, that can be used to link a return message to it. The sender of a return message is a link to the suspended message.
The link in the frozen object is not related to any sender but is instead used to arrange correctly the sequence of packets when transferring large objects.

![Message Formats](image)

**Figure 3.2: Message Formats**

Formats for object and method packets (or pages) are shown in figure 3.3. Object and methods are referenced by a global identification contained in the object and method id fields. The node field identifies the node where the object was originally created, and is used as an extension to the object address space.

![Object and Method Formats](image)

**Figure 3.3: Object and Method Formats**

Objects and methods belong to a class declared as a template in TOOL definitions. Node, id and class fields together uniquely identify these objects and methods when a message carrying the same fields is evaluated to start a process. The mask field in the object is used for message selection, and the control field
stores system information concerning the packet state (c.f. suspended, active, size, etc.).

Object, method and message formats reflect in the internal organisation of data storage. These packet formats have a range of fixed sizes that vary in granularity and maximum capacity according to the entity type. Message packets are arranged in the node queues and the fields are reused to link sub-queues. Object and method packets are accommodated in the node cache or main store and fields are reused for memory management. To keep track of packet lists, VOOM has, in addition to $ and % registers, an internal set of pointers that are copies of fields necessary for packet handling.

3.3.3. Instruction Set

VOOM instructions are implementations of TOOL methods as described in section 3.2.2. VOOM instructions follow the same divisions as TOOL with the addition of explicit object support instructions (send, release, etc., implicit in TOOL). Primitive methods are simple implementations of logical and arithmetic functions. Built-in methods are complex instructions and are implemented as permanently stored routine codes. User-defined methods are responsible for object-oriented process activation and require special architectural design of queues, memory controllers and communication handlers.

Primitive methods in VOOM are represented by the same equivalent listing in TOOL. Arithmetic, logic, relational and memory operations are simply implemented by datapath operations involving the arguments of these messages. The messages are translated into VOOM instructions with one or two operands, and the result is returned to the accumulator (%1) or stored into a register or instance variable position in the object (STORE).
Control methods enable modifications in the instruction flow and operate on the program counter, testing the accumulator. The program counter points sequentially to instructions until a validated branching condition sets it to the value indicated by the argument. The RTN instruction can explicitly terminate the process, and/or just send a return message. If RTN terminates the process, the input message is scraped and the object is returned to main store, freeing the cache. When sending a reply message, RTN will fill in the link field with the sender field of the input message. This enables the system to track down and reactivate the suspended message waiting for this reply.

Apart from branching function, control methods are overloaded with process suspension facilities. Any control instruction other than RTN can implicitly suspend a process, if time slicing is required. When the execution is suspended, the input message is packed to the back of the inactive queue and a link to it is stamped in an outgoing message. This is a self reactivating message that restart the method to the interrupted state or to another state specified by a residual instruction.

Residual instructions are those that do not result in an immediate action, but set a state in the object. The residual instruction in VOOM is the SELECT method. SELECT modifies the mask field in the object header, establishing a pattern to be matched by reactivating messages. Apart from filtering messages, this residual instruction can force the method to resume at a different location than that expected at the moment of the suspension. This instruction is dedicated to the support of object-driven computational flow control.

Built-in instructions are dedicated to object management. Some object management functions can be highly complex and involve the description of non-trivial algorithms. Here, a very brief description will be given, but further reference can be found in [Balou91]. The NEW method creates a copy of a template object.
and packs it in a frozen message format. The identification is obtained from a system object reference allocator. The DELETE method will return the identification of the deleted object to the reference allocator for recycling.

REPLACE will use a frozen message to overwrite an existing object or method. REBIND will change the original allocation of objects. Messages will be relayed to the new destination by a shadow image of the object in its former node, until the reference allocator eliminates it, effectively rebinding the references in the original senders. REVIVE will just provide a context message for the reactivation of dead body method. This context message has a user format with an empty body.

User-defined messages are characterised in VOOM by the implicit generation of an explicit SEND instruction, inexistent in TOOL. Normally, this instruction builds up a message with user format. The two operands of this instruction are used to fill in the first and second records of the user message header, that are informations about the method. The destination object identification should be loaded previously to the SEND instruction. The object reference is normally retrieved through indirect access to the declared information stored in the object state. The body of the message is copied from whatever is in the % registers.

The SEND instruction is available in two basic modes: synchronous and asynchronous. Both modes associate the sender field in the message with the current context identification. This allows replies to both modes, but in the case of the asynchronous mode the reply will come through a future suspension of the method.
The synchronous variation will suspend the method immediately and all the protocol described in a branch suspension will take place. The difference is now that the reactivation message will be generated by another object. The whole mechanics of user messages depends on the cooperative activities of the three VOOM units, storing, transferring, binding and prefetching objects and methods for execution. A more detailed description of VOOM, explaining the role of each unit and discussing the architectural issues can be found in [Balou91].

3.3.4. Implementation

Two different approaches were taken towards VOOM implementation. One of them is a simulation of the whole architecture through a program written in C++. The other takes a simplified skeleton structure of VOOM and implements it in a VLSI integrated circuit. Both implementations are aimed at experimentation on the feasibility of the architecture at the correspondent level, not intending to develop a finished product. The implementations are tools for a study on fundamentals of computer architecture that can then be applied to the development of real products.

The C++ implementation features the most complete representation of VOOM architecture among the two, supporting at least a minimal functionality of each organisational unit. However, the focus of the implementation is to provide just enough mapping to experiment with the advanced features, namely the built-in methods and run time optimisation of system performance. The level of detailed implementation is then of a virtual machine running on the top of a C++ architecture. The more primitive instructions are not specified down to the hardware level and only the principles of the internal architecture are described in algorithmic level. The complete description and study of this implementation can be found in [Balou91].
The second approach to implementation of VOOM is described in full in this work. The functionality of VOOM was factorised to its most simple elements and a subset of it was chosen to be implemented in the Basic Regular Object-Oriented Machine (BROOM). The central point in BROOM is the message-driven model, so all support to other models, including procedural calls, were excluded from this implementation.

Most primitive instructions in VOOM map directly to BROOM but accommodated to a stack oriented organisation instead of register based. For the built-in set, there is only support for the frozen message format, allowing for the creation and migration of objects. Apart from it, the rest must be developed from the primitive set of instructions and based on conventional objects, since there is no support for system routines.

The user-defined messages are well supported, with hardware implementation of most of the units' functionality. The send instruction is optimised to take a single cycle of execution unit, with the rest being executed in background by the units. However, only the asynchronous mode of SEND was implemented to reduce the complexity to an acceptable level of a first VLSI circuit implementation.
Chapter 4: BROOM System Architecture

This chapter describes the BROOM architecture, from the network topology down to node organisation. BROOM is a parallel object-oriented system where messages are processed by a network of nodes with distributed memory. The network topology is based on a tree of busses. Nodes are composed by a three stage pipeline of units handling communication, memory management, and message execution.

4.1. Machine Architecture

4.1.1. Introduction

BROOM is a parallel system designed to support the message-driven object-oriented computation model. It is a distributed architecture where messages dictate the computational flow. The message is both a scheduling token and a vehicle of object communication. Objects are incarnated by the nodes when messages bring them to activity, starting methods that carry transformations to their state. Each node has its own private memory and a unique network address that is associated with objects belonging to it.

The BROOM architecture originated from the VOOM virtual machine, with more emphasis on hardware implementation. Most features in VOOM were scaled down to meet the restricted allocation of resources in a hardware design. Regularity was applied throughout the design to reduce the design time and complexity, so the flexibility of variable formats was turned down to favour simple fixed formats. To keep design to a manageable size, the BROOM implementation gives hardware support exclusively to the message-driven model.

The architecture of BROOM is depicted in figure 4.1. A BROOM system is a collection of BROOM nodes interconnected by a network. The BROOM nodes contain the logic necessary to drive the protocol of a bus network. They are able to
store, route and transfer messages in the network and also handle an exception protocol for rejected messages. The nodes activated by messages prefetch methods and objects involved and bind them into an executable process. After modification the object is restored to the main memory.

Figure 4.1: BROOM System Architecture

Figure 4.1 shows that a BROOM node stores messages packed in queues inside its communication unit (CU). When transmitting, the message unfolds to a sequence of words, travelling in the BROOM network, controlled by the communication protocol. Messages are received and again stored in the destination node. While stored in the communication queues, the messages are decoded to determine the object and method associated with it.

In figure 4.1 we can notice that objects are stored in three memory levels. The first level is the node internal cache, the second is the node private, volatile, local memory and the third is a mass store. The mass store is normally in magnetic media, and is shared all over the system. In the cache, objects and methods are
organized as a list, in a correspondence to the messages in the communication unit. This list contains the processes bound by the action of the memory management, in response to the arrival of messages. This process list is a collection of object/method pairs, associated with a message in the input queue, waiting execution. In main memory, objects and methods are organized according to their classes and in the mass store the organisation is arbitrary.

The active process is formed by taking the first object and method in the cache and joining it to the activation message. This message is the first in the input queue and a message slot in the output queue is also added to the process. Only the first process entry of the list is active and being processed by the execution unit. The output slot can be used by the active process to send one or more messages. When the process finishes, the object is returned to main memory and the working method and messages are scraped. Messages generated by the process can stay in the node or be transmitted by the network, if addressed to another node.

4.1.2. Network Topology

The BROOM nodes communicate through a tree of buses topology, adopting the same strategy used in DICE [Delgad87]. The nodes are organised into clusters that communicate with each other by means of a switching element. All nodes in a cluster share a single bus, where messages are broadcast. Once a message is in the bus it can be received by its destination node, stored by a relay node or forwarded to another cluster by a switching element.

The implementation of full message passing support in hardware is an important design guideline in BROOM. Messages are handled automatically by hardware in a BROOM node. The sender stores the messages until they can be
transmitted. The node checks the destination of the message and then routes and sends it automatically.

The routing algorithm consists in checking if the message is addressed to the originating node or another. If addressed to the originating node, the message is moved from the output to the input queue of the node. A message addressed to another node is broadcast into the network and the destination node will recognise its own address and accept it.

Once in the network, the message is always received by one node, be it the initial destination or a relay node. A relay node will receive a rejected message whenever its real destination is busy. The relay mechanism stores the message in the input queue of a relay node. It will remain there until reaching the front of the input message queue. The message will then be sent to the network, towards its real destination or another relay node.

The bus topology was chosen because it is simple, have a small latency and is a good complexity vs. bandwidth compromise for VLSI implementation [Delgad88]. The bus topology consists simply of a bus shared by many nodes. The nodes need only to perform an arbitration protocol so that only one is writing on the bus at a time. On the reception side, all nodes can receive a message simultaneously, allowing a simple implementation of broadcasting.

The latency in the network is small, since the message reaches the destination in a single step. Moreover, since the message travels in a bus that is many bits wide (32 bits in BROOM) the message transmission will be around an order of magnitude faster than a serial link of the same speed. One limitation is that busses have a worse electrical profile for transmission than point to point connection. Adequate driving and shielding of the lines must be provided to avoid
cross-talk and signal distortion due to the irregular distribution of electrical parameters along them.

The bus topology chosen for BROOM simplifies the routing algorithms and often offers a better bandwidth compromise than Cartesian product topologies [Delgad88]. A bus topology uses all pins available for communication in a conventional IC to increase the link bandwidth. Cartesian product topologies (hypercube, hyperbus, etc.) have to split the available pin count among the various links. This reduces the bandwidth available for each one and influences the expansibility of the systems (more links means less speed per link).

The pin count is a fixed resource determined by the encapsulation technology used. The pin count also influences the silicon area necessary for the implementation. The silicon area determines the fabrication costs and the yield (percentage of working chips in a lot) of a silicon design. The bus network can be extended without being affected by pin count limitations, only electrical and performance limitations will apply.

Electrical properties and contention in a bus topology can limit the practical number of nodes allowed in a bus. A solution to this is to divide the whole network into clusters. These clusters can be then organised in any topology, but yet the simplest way is to connect them into another bus. Hence, for a generic number of nodes, a BROOM network is organised as a tree of busses.

Figure 4.2 shows that BROOM network is a regular recurring pattern. Nodes (N) are grouped around a bus forming clusters (C). The clusters group themselves around a connecting bus, forming higher level clusters. Cluster busses of both cluster or node groups are interconnected through a switching element (S) that decouples the effects of one bus onto the other and transmit the messages
from a cluster to another. Even a tree topology has advantages over Cartesian products when pin count is taken into account (see [Delgad88]).

![BROOM Network Topology](image)

Figure 4.2: BROOM Network Topology

4.1.3. System Modules

A BROOM system involves the combination of three modules: switching element, permanent store manager and the node processor. The switching element conveys messages across clusters and drains node queue overflows. The storage manager transfers pages between main memory and mass storage sites. The node processor has two busses, one exchanges messages between the node and the network and the other transfer objects between internal and external memory.

The switching element is a node composed of two communication units back to back. Its main task is to connect two busses, decoupling the impedance of a bus from the other. However, even in a single cluster system a switching element can be used to enhance the system's reliability. This is because it has an extra message queue that can be used to relay rejected messages.

A message is relayed by a node when its original destination is unable to receive it. The switching element does not process the messages, just passes it to
another bus, so messages have a small latency in its queues. Consequently, the switching element queues are less susceptible to overflow, with more space for message relay. The queues can also be designed longer in the switching element, using the space left by the omission of functions that would be otherwise implemented in a normal node.

The storage manager is an intelligent disk controller. Its main task is to substitute or retrieve missing pages in the main node memory. When the node processor decodes a message, the object associated with it is prefetched to the node cache. If it is not in main memory, it must be fetched from the system disk. The node processor sends the unbound message to the store manager, placing it on a prefixed place in memory.

The storage manager must decode the unbound message header, associating it with the correct missing object and method in the disk. It will then replace the objects in the main memory, storing the old ones and overwriting their image in memory. The unbound message is then sent back to the node processor, using a special request procedure, and put back in the communication queue. The storage manager must have multiple ports to keep the private memory busses of the nodes insulated. It will serve the nodes in a round-robin strategy, giving a fair share of disk resource to each node.

The main module in the BROOM architecture is the node processor, which constitutes the building block for the parallel object-oriented machine. Its main tasks are: to handle messages from their creation up to their destinations; to bind messages to the associated method and object; and to execute the process activated by the bound message. The other two units (switching element and store manager) are not fully described or implemented in this work and can be emulated by a convenient host. The node processor is the central idea of the work,
the hardware implementation of VOOM, and in consequence, of the message-driven object-oriented paradigm. The following sections will give a detailed description of the BROOM node processor and its operation.

The main idea behind the system modules is to apply the object-oriented methodology to the BROOM system design. Every module in the system should be a BROOM object, managing its own state and communicating with the others through a standard message passing protocol.

![Diagram of BROOM System Organization](image)

Figure 4.3: BROOM System Organization

Figure 4.3 shows the system organization with the BROOM node processor as the intersection of two bus systems. The inter-node bus system communicates objects that represent shared resources such as nodes, cluster services (printer, LAN gateway, etc.) and the switching element. The intra-node bus system interconnects private node resources. Private resources interact intensively among themselves, requiring a more exclusive and high performance bus system. Examples are main memory, mass storage manager and fast peripherals (video display unit, numeric co-processors, etc.). The mass storage manager is, in reality, a shared resource, but since it interacts heavily with the memory it is initially placed in the intra-node system.

The object-oriented methodology implies that all modules communicate using the same message protocol in both bus systems. This would simplify system
specification and reconfiguration, bringing abstraction and standardisation to hardware design. The interface between the system and component designers is the clean object/method specification. This abstracts the complexity of component or system details from each other, giving the choice of a range of implementations for both sides to use or design.

Peripherals and co-processors can be plugged in either of the busses, easily expanding the system or customising it to a particular application such as a graphical engine or an architecture emulator. The BROOM design is based in this regular approach but, due to implementation issues, the inter-node and intra-node protocols differ slightly. Moreover, there is a conventional memory interface to bypass the necessity of a memory controller module.

4.2. Node Architecture

4.2.1. Introduction

The BROOM node processor is the module that actually processes the messages and performs the transformations on the objects. It has two busses, one used to transport messages across nodes and the other used to fetch objects and methods associated with the messages and send them back to main memory after processing.

The previous section depicted the node processor in the confluence of two bus systems. The duality of the bus system not only reflects different communication requirements but also two levels of parallelism: the inter-node and intra-node parallelisms. These two levels are characterised by a predominance of different natures of parallelism. The inter-node bus interconnects a system that is predominantly homogeneous, with the BROOM nodes offering a normalised
processing capacity. The intra-node bus interfaces distinct personalised controllers, forming a heterogeneous system.

This duality is the starting point to understand the BROOM node architecture. On the inter-node side of parallelism, even nodes that have a certain specialisation such as a cluster server or switch can be considered as part of a homogeneous system. The homogeneity results from the fact that messages do not imply any pattern associated with its origin, i. e., messages coming from any node can have equal treatment on its destination.

On the intra-node side, modules are specialised objects that generate distinct message patterns. Messages originating in different modules have distinct meaning on the destination and may be treated in different ways. Messages coming from main memory, mass store manager or a numeric co-processor will be interpreted differently inside the node processor. The same messages received by the VDU may have a different treatment than the one given by the node processor. One noticeable characteristic of these messages is that they may address the same object in different representations (a file in the disk, a stored image in memory, a bitmapped picture on the screen).

This makes the front end of the intra-node bus different for each component module. In the node processor, in particular, it is different from the inter-node front end. As a consequence, the internal architecture of the BROOM node recurs from the system architecture, where a central unit interfaces two busses. Figure 4.4 shows that the node processor operative unit connects to the inter-node system bus through the Network Bus, and to the intra-node bus through the Memory Bus.
The Network Bus flows messages to and from the message queues. All messages received are queued in the input queue. The Memory Bus flows messages to and from the external intra-node bus. The route for internal messages depends on the message representation type. The destination of an incoming message can be the input message queue, object cache or method cache, for the corresponding representation types message, object and method. Outgoing messages are sent to the main memory if their representation types are object or method. The storage manager module will receive the message type messages, although a copy will be sent to memory as well.

![Figure 4.4: BROOM Node Processor Architecture](image)

### 4.2.2. Message and Instruction Format

Computation on the BROOM node occurs in two nested levels. The higher level comprises the message flow. The lower level is the programmed method instruction flow. The message has a fixed format consisting of a header tag followed by the message body, totalling eight words. The instruction is a scalable, fixed format, single byte, two level opcode. The top level divides the byte into halves for opcode and operand. The bottom level reserves the operand half to extend the operation code.

At message transfer level, the BROOM node processor moves and processes messages through the architecture shown in figure 4.4. Message
processing involves interpretation of information in it, correlated with its origin. Figure 4.5 lists the information contained in messages for all representation types recognised by the BROOM node.

A message is divided into header and body, the header contains control information and the body is the data contained in the message. The header is a word divided into five fields and the body has a range of seven words, totalling eight words in a message.

![Figure 4.5: BROOM Message Formats](image)

A message can represent different types of information and, for each type, the fields in the header are different. The type field distinguishes messages among its various formats. In analogy to the von Neumann machine, object and method representations are distinguished by context. The other fields are used for message identification and process binding and are further explained in the next chapter. The body contains parameters in a message or the data of method or object representations.

Message representations activate methods that are programmable, apart from few exceptions. The object and method message representations transport
objects from main memory to the cache. The object is the data to be modified and the method is the specification of the action operating on the object.

The method is coded in a sequence of bytes contained in its representation body. Figure 4.6 shows that the instructions are represented in two levels designated as top and bottom. The instructions in the top level activate register transfer and instruction flow control. At the bottom level, instructions activate transformations on data or message flow control operations.

![Top Level Instruction and Bottom Level Instruction](image)

Figure 4.6: Instruction Formats

There are sixteen instructions in each level, all with a fixed format. The top and bottom level have the same format, with two fields four bits wide. The top level interprets the highest field (bits 4 to 7) as the operation code. The lowest field is the operand. One of the codes (operate) in the top level identifies the bottom level mode. In the bottom level, the highest field is fixed and contains the code operate. The lowest field is then interpreted as the operation code.

4.2.3. Message-Flow Architecture

Following the message-driven model, messages are treated as the main information unit in the node architecture. To optimise the message flow, the node is organised in a pipelined structure to process them. In the front end of the pipeline is the communication unit, that generates and receives messages aligning them into two queues. The next stage in the pipeline decodes the messages while
they are still queueing for execution. The last stage executes the method associated with the message in the front of the queue and discards it after completion.

Message flow is the vital stream of information in the BROOM architecture and figure 4.4 in the previous section shows the routes available for message transit inside the BROOM node processor. We define here a *message-flow* architecture as a system that uses message as the information unit and provides specialised hardware structures for computation processing at message level. In the BROOM node processor, computation proceeds driven by the message flow and messages are processed as a single executable instruction. Architecture organization normally found in instruction driven machines are translated and applied to the message level in BROOM.

Instruction pipeline is an optimised machine organization that allows simultaneous execution of different instructions, overlapping different stages of each instruction [Kogge81]. One good advantage of the message-driven model is that message flow is analogous to instruction flow in conventional architectures and the pipeline organisation can be applied to a message-flow architecture. Figure 4.7 shows the three stage organization of the BROOM node processor message pipeline. Three units control the flow of messages in the pipeline: Communication Unit, Memory Management and Execution Unit.

![Figure 4.7: BROOM Message Pipeline](image)

The units perform a fetch, decode and execute sequence in the message pipeline. The action of the three units overlap, allowing the processing of three
different messages in parallel. The communication unit performs the equivalent of a fetch, collecting messages into the queues. The memory management decodes the message and associates it with the correct method and object. The execution unit operates on the bound process formed by message, method and object, producing the requested transformation or result message.

Message processing starts with a message arriving at the communication unit. Messages originated in the node go to the output queue and the input queue receives messages from the network. The memory management examines messages in the input queue determining the action requested by each one. Messages that are object or method representations are sent directly to main memory. Message representations have fields in the message header indicating the object and method addressed (see fig 4.5).

Object and method representations are searched or entered into the internal caches and lined up in correspondence to the message sequence in the queue. The associated sequence so formed is a process list of bound messages awaiting for a slot in the execution unit. The execution unit consumes this list, interpreting the message through the associated code in the method, modifying the attached object. A terminated process has the method and message parts deleted, while the object is returned to main memory.

The object-oriented message-driven model allows for some improvements in architecture organization and efficiency. In a message flow, unlike an instruction flow, messages are loosely coupled, reducing the chance of a pipeline hazard\(^1\). This also allows a more flexible structure for the pipeline where the units can work

\(^1\)A hazard is an interruption on the normal flow of a pipeline caused by a dependency between two contiguous instruction, e. g. a conditional branch, usually resulting in a flush of the pipeline and incurring in long recovery gaps in the execution.
asynchronously. This means that units ahead of the pipeline can advance processing new messages even if other units lag behind. The process list is an example, indicating that the memory manager has advanced many messages ahead of the execution unit.

Another benefit for hardware design originates from the encapsulation property. The object addressed by a message contains all the data necessary for processing. The message flow pipeline can pre-process data as well as code. The action of the memory manager can provide the inclusion of a requested object in the cache before execution needs it.

With the provision that object data is always in the cache during execution, it can be accessed at register speed. The method is fetched into the cache as well and can execute at microcode speed. The final effect is that the BROOM architecture executes messages as if they were native microcoded instructions and modifies objects as if they were hardwired into the register structure.

4.2.4. Instruction-Flow Architecture

The message pipeline decodes messages and activates the method that operates the transformations on the designated object. This is analogous to the operation of a microprogrammed machine that addresses the appropriate microcode to execute the decoded instruction. The process activated by a decoded message assembles designated information pages around the operative unit. One page contains the method specification that instructs how to process data contained in the other pages. Data is processed through the node operative unit, using the output message page as an operation stack. The input message page holds the input parameters and is used as temporary store. Most operations have an end target on
the object page that get modified and saved to main memory at the end of the process.

In the process of decoding a message, the pipeline stages use a collection of auxiliary registers concentrated in a block called operative unit\(^2\), shown in figure 4.8. The three pipeline control units share the operative unit and use its two busses to carry out transfers to the network and memory busses.

![Figure 4.8: BROOM Operative Unit](image)

The counter registers in the communication and management units play a dual role. First they organise data sorting and transfer in their respective units. Secondly, the execution unit uses them to refer to the assembled parts forming a process. The header shadow registers are used by the memory management to bind a process and will be explained in the next chapter.

A message arriving at the front of the input queue activates a process. This process is passed to the execution unit that controls the operative unit through the desired transformation. Figure 4.9 shows the information pages assembled in a process. The method page contains the programmed method code. The execution unit steps the program counter through the instructions in this page and decode them to perform the method. The instructions with operand have their operand

---

\(^2\) The name data path is more generic and the term operative restricts the range to parts that perform transformations like shift, add, etc.
fields sent to the operand register. Instructions without operand operate on the top elements of the stack, represented by the output message page.

The ALU performs arithmetic and logical operations on the top elements of the stack. The stack top element is kept in the operative unit stack top shadow register. The execution updates it with the stack page at the end of every operation, pushing or popping the result. A send instruction moves the stack page ahead in the message queue and substitutes it by another empty slot.

The input message contains the parameters received with the message request. They can be accessed as registers, representing the $ registers in the VOOM machine. Empty positions can be used as temporary storage for method processing. Both input message and object page can be accessed for read and write by means of instructions with operand. The operand represents the offset from the start of the page of the addressed element. Different instructions are used to address the two pages and the offset in the operand register is used by branching instructions as well.

The instruction format is derived from the Inmos Transputer [May86]. The small size of the instruction offers a well balanced compromise between code

Figure 4.9: Assembled Process Pages
storage and flexibility. Even the small range available for the operand serves well objects designed in a fine granularity as proposed in BROOM.

The small size of the instruction also implies simple execution and economy of data path resources such as busses and register ports. This results in a compact design for the operative elements that enables instruction execution in a single cycle, and the use of fast clocks. The limited range of instructions and address space can be compensated with prefix instructions. This will slow down the execution to more than one cycle, but the slower, longer codes can be judiciously assigned to less used instructions.
This chapter describes the BROOM node architecture. The BROOM node processes messages as a single atomic instruction. The messages are first collected and queued in the communication unit. The messages in the queue are decoded to bring the corresponding method and object to the internal processor store. Finally the message is passed to the execution unit that accesses its contents and performs the action stated by the associated method. These three actions are executed in an overlapped way in a high level pipeline.

5.1. Communication Unit

5.1.1. Organisation

The main task of the communication unit is to manage two queues, one for outgoing messages and one for incoming messages. Each queue is a FIFO stack with capacity for eight messages. Messages for the input queue are received through the network bus from other nodes, or through the memory bus, delivered by other node modules. Messages in the output queue are generated only by the execution unit and are transferred to input queue or set to wait for a slot in the network, if they are outward bound.

The communication unit controls two queues (as shown in figure 5.1), one for input messages and one for output messages. The queues are implemented as two circular lists enclosed in two separated memories. The list is based on contiguous, wrap around positions of memory. Self-increment registers point to the front and back of the queue.

A better implementation for these queues is in the form of two linked lists pointing to pages in a shared circular buffer. In this case, the transfer direction with largest demand can allocate most of the space available. Transfers between queues do not involve page copies, but only pointer manipulation. Messages terminated by
the memory management can be discarded from the queue, passing its pointer to the preceding message.

![Figure 5.1: Message Queues](image)

The linked list implementation requires two ports for the memory buffer, [Anido89] to enable the operation of the execution unit on both stack and parameter accesses. Due to limitations of the library available for VLSI design, the implementation was done with two separated memories.

5.1.2. Protocol Arbiter

One important task of the communication unit is to control the arbitration protocol that guarantees the right of writing on the bus to one single node. This arbitration is obtained through a daisy chain with a circulating master.

When the system is started, a programming pin determines which node will start as chain master. A prospective sender precedes the transmission with a request pulse. This pulse is fed into the daisy chain by the node assigned as master.

The first transmitter candidate to get the permission holds the bus, extinguishing the signal forward. This node is then assigned as the chain master. This provides that the current sender has the least priority for getting the next transmission slot.
The schematic diagram in figure 5.2 shows the sequence of operation for the arbitration protocol. Node A has been programmed to start with the bus master, but has lost it already to node B in a previous transaction. Node B retains the master card that feeds the request pulse into the daisy chain. Node C is idle and passes the grant signal forward. Node D requests and traps the grant signal, preventing E from receiving it. As D receives the grant, it will receive the bus master, rendering it with the least priority to hold the bus next time.

![Figure 5.2: Protocol Arbiter](image)

This protocol requires less hardware complexity than a collision detection protocol, with no need of rescheduling counters or special drivers/detectors. One drawback is the cascading of serial logic that limits the number of nodes in a chain, but it is coherent with the same limitation already imposed by the bus topology.

### 5.1.3. Message Handling

The main task of the communication unit is to control the queue pointers and to signal the overflow and underflow events to other units. There are eight competing message request types to be handled: four transmission related and four reception related. Transmission requests are: normal, overflow, relay and local; Receptions requests are: normal, relay, local and interrupt. The requests are
arbitrated by a state machine that mutually exclude the calls and establishes their priorities.

The communication unit is part of the message flow pipeline. Since the pipeline can work asynchronously, it must be synchronised with other units. The other units work concurrently with the communication unit, requesting slots and consuming messages. Simultaneously with that, messages are received and despatched to the inter-node bus.

The memory management interacts with the communication unit consuming the input queue to pre-process the messages. When the queue is exhausted, the memory management waits for a signal from the communication unit warning for the reception of new messages. Memory management processes immediately *init* and *relay* messages, sending requests to move the messages in the input queue directly to memory or inter-node bus.

The execution unit uses the pointers to the front of the input queue and end of the output queue to address the parameter registers and the operational stack. The parameter or "$" registers persist throughout the method execution and are discarded when it terminates. The stack page is used to build outgoing messages and is substituted by an empty slot when the send instruction is issued. The send and release instructions rely on the action of the communication unit to do the message and process switching and update the queue counters.

Apart from serving other units, the communication unit has its own task to execute. It switches to receive mode whenever another task is not obstructing it. It can receive normal and relay messages from the network, local messages from its own output queue, or interrupt messages from the intra-node bus. The relay messages use the protocol arbiter to decide which will be the relay node. After the
header is sent and no one indicates the reception of the message, the nearest node available in the daisy chain will be assigned as the relay.

The greatest priority is to receive messages from the inter-node bus, to reduce traffic overhead. The next priorities for message handling are the reception of interrupt and local messages, in this order. Reception of relay messages has the smallest priority and will be performed only if the node is idle. Reception is only possible when there are slots free in the input queue and there is no other reception going on. Messages have to be relayed to other nodes if these conditions don't hold.

Transmission is only possible when the node is granted with a slot in the arbitration protocol. Only local messages can be transferred from output to input queue disregarding the protocol. Transmission slots are always requested for external messages in the output queue. In the case that the output queue is full, even local messages are ejected to the inter-node bus to be relayed, generating an overflow transmission request. Relayed messages stored in the input queue also generate transmission request when processed by the memory management.

5.2. Memory Management

5.2.1. Organisation

The memory management unit is responsible for decoding and binding messages to the respective objects and methods. Consequently, this unit manages all page transfers involving the memory bus. The unit keeps a pointer to the input queue and uses three registers to decode the message pointed by it. The unit then allocates or transfers objects and methods bound to the message. The control of the memory management presents a three level hierarchy. The top level decodes
the purpose of the message; the middle level checks the headers of the three elements; and the bottom level transfers pages that fail to bind.

5.2.2. Message Decoder

The Message Decoder is the highest level of memory management control. Its main task is to pre-decode the messages in the input queue and to interface the communication and execution units. The Message Decoder checks the messages and decodes its type and destination, producing processes. The communication unit signals the message decoder when the input queue is not empty. The execution unit flags when a process is consumed, freeing positions in the cache, where new process can entered by the memory management.

Messages are examined, one by one, as the decoder pointer traverses the input queue. As soon as there is space in the local object cache, a new message is analysed to check its purpose. There are three types of messages: normal, relay and init. After processing, normal messages are simply passed forward to the next control level and the other two types are discarded.

Normal messages contain an object address and activate a method for this object. If there is space in the cache, a normal message can be assembled into a process, otherwise, the memory management will wait for a free page. After identifying a normal message, the decoder leaves the message in the queue untouched. The other levels of memory management control will then take over to complete the process binding.

Relay messages are signaled to the communication unit, that will request a transmission slot for it. When the slot is granted, the relay message is sent back to the network, in an attempt to reach the original destination. The relay message will
block processing in the memory management until dispatched. An extra queue or list would be necessary to skip them, to schedule their processing.

An init message is a message whose representation is of type object or method. Init messages are sent to the main memory with the intervention of lower level controllers. Relay and init messages are marked invalid after being processed, preventing them from being processed by the execution unit.

The message decoder performs only a pre-decoding, leaving the rest of the task to the message binder. It eliminates non-executable messages from the process list, leaving only appropriate messages for the next controller. The message decoder advances the pipeline asynchronously, on demand by the communication and execution units. The communication unit signals the arrival of new messages and execution unit signals when it frees a slot in the internal cache.

The asynchronous behaviour of the pipeline is a major advantage for the architecture design. Messages are processed in advance and in parallel with execution. It minimises the impact of process switching, bringing information to the fast access cache in a forward-looking manner [Pacheco89].

5.2.3. Message Binder

This level processes messages recognised as normal by the decoder. The header of the message is first copied into a shadow register. This register addresses object and method, bringing their headers into the other two shadow registers. The fields are checked to confirm the binding. If binding is confirmed, the message is left on the queue for execution, otherwise the message is sent to memory and invalidated at the queue.
Messages decoded as message representations are left in the input queue for binding. In the binding process, the addresses contained in the message are decoded to bring the other components of the object-oriented process. To decode the component addresses, the message header is loaded into its shadow register into the operative unit.

The shadow registers contain some extra logic that allows clipping, shifting and comparison of header fields. They are designed to reduce bus usage for multiple field checks. They also reduce the number of cycles for these checks with use of purpose designed parallel hardware.

The fields in the message were designed to address the process components with a reduced control information overhead. They are tightly packed in the message and must be manipulated to yield the correct addresses. They are concatenated, realigned and clipped by the registers to produce the cache or main memory address for the object and method.

The binding takes place when all headers are loaded in the shadow registers. The first step of the binding process is to check the object and method of the process currently present in the cache. If the identification of process components is wrong or if there is a mismatch in the class field, binding fails. The missing pages are signaled to the next level, that will fetch them into cache or report the failure to the mass store module.

Bound messages remain in the queue with the respective components in cache waiting execution. An unbound message is copied to the mass store module and invalidated in the queue. When the components are retrieved by the mass store controller, the message is sent back to the input queue.
Message binding conforms with a relaxed garbage collection scheme for persistent objects. Objects in BROOM are persistent and deleted only under explicit demand. Hence, the Message Binder, through the identification checks, can spot in the memory pages containing objects reminiscent of previous processing. These pages normally are idle and are swapped with the requested components of an active process.

The Message Binder also reduces the overhead caused by slow disk access, eliminating unbound messages from the queue. Other messages in execution condition overtake an unbound message, while it is being retrieved from mass store. Given that the memory management is able to analyse messages in the input queue, pages requiring disk transfer can be detected well ahead of being requested by the Execution Unit. This gives time for disk transfer to occur in parallel with execution of other messages.

5.2.4. Page Transfer Handler

The Page Transfer Handler manages page exchanges between internal cache and main memory. It receives request from the execution unit to restore pages to main memory and the message decoder requests to load pages into the cache. Page addressing and allocation depends on the cache size and organization. Hence, three cache models were studied for the project and a dual page cache was selected.

The action of the Page Transfer Handler is related to the cache model of addressing and slot allocation. The original proposal for cache organisation was a direct-object sequential-method model. In this model, objects in the cache are addressed directly, clipping or hashing the object address until it fits the cache address range. The resulting address points to the position that would match the
desired object. If the pointed position does not match the object, the original address is rehashed a predefined number of times to try a match (originally, no rehash would be attempted). If all attempts fail, the page is considered missing and should be fetched.

Methods are not searched in the cache, but always fetched and entered into the cache in the order of message arrival. This strategy is simple and imposes little hardware overhead. The major drawbacks are waste of object cache space and waste of time to bring methods already in the cache.

The second model involves an auxiliary list of associative addresses. The list organises the methods in the order of message submission and associates the respective object with them. The list is implemented with a specialist hardware that responds to a message header with two signals indicating the presence of object and/or method in the cache.

The associative list contains three fields, the message header, object cache address and method cache address. When presented with a message header it responds indicating the presence of the elements and stores the required combination into a new entry. If any of the pages is missing, cache addresses are fed sequentially to the list until it responds indicating a slot. The missing page can then be fetched into the indicated address.

This scheme optimises memory usage in the cache but still requires that present items in the cache have their headers checked for class matching. Unnecessary duplication or copying procedures for methods are avoided but there is still a time overhead for slot searching. The major drawback of the model is the hardware cost, for it requires an extra silicon area and the design of complex associative memory devices.
The third model reduces the cache to a simple dual buffer. There are only two pages in the cache, one is busy with the execution process while the other is being fetched by the memory management. This simplifies the searching process in the cache, since there is only one position to check. The major drawback is that it restricts the advance of memory management, since there is only one slot available for the process list.

This last model was adopted due to area restrictions for the silicon implementation. The page transfer control consists of only one state telling if the current object is being reused for the next method or not. The address for the method page is simply toggled between one page and the other and a method is always fetched into the cache even if it duplicates the existing one. New messages are processed on the demand of the execution unit. When it terminates the execution of the current one, the object in the cache is transferred to memory and a new object can be brought into the slot created.

5.3. Execution Unit

5.3.1. Organisation

The execution unit runs the processes in the list produced by the memory management. It steps the program counter through the method decoding and executing each instruction. The instructions involve data transfers, functional transformations, branches and message generation. The execution cycle has two phases: fetch/decode and execute. In the first phase the unit retrieves and decodes the instruction, in the second it activates all transfers and operations determined in the first phase.

The BROOM node is a stack based machine and the stack top is always involved in a transfer as origin or destination. Functional transformations are
operated by the ALU and control instructions can modify the program counter
instead of the stack top. A two operand machine was also studied, but it increased
method size and was less flexible for a reduced instruction set.

The execution unit executes all instructions, including load and store, in a
single cycle. The cycle is divided into two phases, the first used to decode the
instruction and the second to execute it. In the first phase, an instruction contained
in the method cache is fetched through a dedicated bus into the instruction register
inside the execution controller. It is pre-decoded and passed to the decoder sub-
unit.

The Instruction decoder is activated during the second phase, activating
the signals necessary to carry out the operation. These signals operate the transfer
gates that allow data exchange through the operative and internal node busses.
They also enable or select the operation of functional units attached to the
registers or present in the ALU. For the special case of ALU operations, the cycle
is extended to a third phase, overlapping the next instruction.

5.3.2. **Execution Architecture**

The execution architecture is designed to interpret the method indicated
by a message and operate the desired transformation on the selected object. The
four cache pages assembled by the memory management into the process list form
the backbone of the execution architecture (refer to figure 4.9). The control part of
the information flows from the method page into the execution decoder. The data
part is contained in the other pages and are exchanged with the elements of the
operative unit, namely the stack top, operand, program counter registers and the
ALU.
Figure 5.3 represents the organisation of the operative execution elements and the data busses associated with it. The active process is composed of two message pages, one object page and one method page. The message pages connect to the network side of the operative unit and the input page also connects to the memory side. The object page connects with the memory side and the method page connects directly to the operand register.

![Execution Architecture Diagram](image)

The input page is organised as registers that can be accessed by a set of transfer instructions, as detailed in figure 4.9. The output page is organised as an operational stack and access to it is sequential, through pushes and pops. The object page can be accessed as memory positions offset by the operand register.

The current instruction is indicated by the program counter, pointing to a position in the method page. The instruction is a single byte and is transferred through a dedicated bus into the decoder. The operand part is also transferred into the operand register. This operand is then used as an offset to address positions in the input and output pages. The operand is also used as direct data pushed into the stack or to indicate the offset of a branch instruction relative to program counter.

The stack top position is detached from the output page and stored into the operative stack top register. This register is source or destination for any data
operation and connects directly with the ALU. The ALU can source operands from both operative busses or from the stack top and can result back into the stack top or into the program counter for branches.

The busses are allocated to permit instruction execution in a single cycle. Direct inter-register connection also contributes to increase communication resources for parallel transfers, as for example, the detached stack top connection with ALU. The connection of the input page with two busses not only serves the transfer of init messages, but also leaves the network bus free for stack updating during input parameter access (see also figure 4.4).

5.3.3. Instruction Set

The BROOM instruction is encoded in a single byte, with two four bit fields. The opcode is represented in the upper field and the operand in the lower. The instruction set is divided into two levels: sixteen instructions at the top level and sixteen more encoded in the opcode field at the bottom level. There are instructions for data transfer, branching, arithmetic, logic and object-oriented support.

Table 5.1 gives a condensed description of the top level instruction set. It shows the operation code, operands, destination and ALU operation for each instruction. It also shows how the stack is affected by the operation.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Source1</th>
<th>Source2</th>
<th>ALU op</th>
<th>Destination</th>
<th>Stack Ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>with</td>
<td>0</td>
<td>opr</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>at</td>
<td>top</td>
<td>opr</td>
<td>add</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>push</td>
<td>0</td>
<td>IMP[opr]</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>pop</td>
<td>0</td>
<td>top</td>
<td>add</td>
<td>IMP[opr]</td>
<td>-1</td>
</tr>
<tr>
<td>from</td>
<td>0</td>
<td>OP[opr]</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>to</td>
<td>0</td>
<td>top</td>
<td>add</td>
<td>OP[opr]</td>
<td>-1</td>
</tr>
<tr>
<td>brx</td>
<td>prog cnt.</td>
<td>opr</td>
<td>add</td>
<td>prog count.</td>
<td>nop</td>
</tr>
</tbody>
</table>
The operation of the top level instructions is described below:

**with, withnot**  The operand is pushed into the stack, negated in case of *withnot*.

**at**  The operand is added to the top of the stack. This instruction is used to add the method selector to the message header. The object reference should be previously loaded with a *from*. The object reference has the lower byte blank, where the addition operated by *at* will set the method field.

**push, pop**  transfer instructions between the input message and the stack top.

The operand indicates the position to be transferred.

**from, to**  transfer instructions between the object page position pointed by the operand and the stack top.

**brf, brt** if the top of the stack is zero (non zero for *brt*) add the operand to the current program counter and branch to the resulting position.

**brb, bra**  subtract (add in case of *bra*) the operand from the current program counter and branch to the resulting position.

**opr**  interpret the operand as a bottom level instruction.

**upon**  substitute the stack top by the value at the position indicated by the stack top in the object page.

---

Table 5.1: Top Level Instructions

<table>
<thead>
<tr>
<th>opr (1)</th>
<th>top</th>
<th>opr</th>
<th>top</th>
<th>nop</th>
</tr>
</thead>
<tbody>
<tr>
<td>opr (2)</td>
<td>top</td>
<td>second</td>
<td>opr</td>
<td>top</td>
</tr>
<tr>
<td>upon</td>
<td>0</td>
<td>OP[top]</td>
<td>add</td>
<td>top</td>
</tr>
<tr>
<td>pack</td>
<td>0</td>
<td>OP[top]</td>
<td>inc</td>
<td>second</td>
</tr>
<tr>
<td>onto</td>
<td>0</td>
<td>second</td>
<td>inc</td>
<td>OP[top]</td>
</tr>
</tbody>
</table>

---

Chapter 5: BROOM Node Architecture
pack push into the second position in the stack the value at the position indicated by the stack top in the object page and increment the value in stack top. This instruction is used in sequences to pack the contents of an object into a message.

onto pop the second position of the stack at the position indicated by the stack top in the object page and increment the stack top. This instruction is used in sequences to save the contents of the stack into the object before flushing it for use by a message. The contents of the stack can be restored with pack instructions.

Table 5.2 presents in the same way the instructions of the bottom level.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Source1</th>
<th>Source2</th>
<th>ALU op</th>
<th>Destination</th>
<th>Stack Ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>add, sub</td>
<td>top</td>
<td>second</td>
<td>add, sub</td>
<td>top</td>
<td>-1</td>
</tr>
<tr>
<td>or, and</td>
<td>top</td>
<td>second</td>
<td>or, and</td>
<td>top</td>
<td>-1</td>
</tr>
<tr>
<td>xor, xnor</td>
<td>top</td>
<td>second</td>
<td>xor, xnor</td>
<td>top</td>
<td>-1</td>
</tr>
<tr>
<td>dup</td>
<td>0</td>
<td>top</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>flush</td>
<td>0</td>
<td>top</td>
<td>nop</td>
<td>none</td>
<td>reset</td>
</tr>
<tr>
<td>not, neg</td>
<td>0</td>
<td>top</td>
<td>not, neg</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>inc, dec</td>
<td>0</td>
<td>top</td>
<td>inc, dec</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>shl, shr</td>
<td>0</td>
<td>top</td>
<td>shl, shr</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>send</td>
<td>0</td>
<td>top</td>
<td>nop</td>
<td>OMP[stk]</td>
<td>reset</td>
</tr>
<tr>
<td>rel</td>
<td>0</td>
<td>top</td>
<td>nop</td>
<td>none</td>
<td>reset</td>
</tr>
</tbody>
</table>

Table 5.2: Bottom Level Instructions

The operation of the bottom level instructions is described below:

and, sub arithmetic operations that add or subtract the second position of the stack from the top. The result is left on the top and the operands are consumed.
or, and, xor, xnor  logical operations between the second position of the stack and the top. The result is left on the top and the operands are consumed.

dup  the stack top is duplicated.

flush  the stack pointer is reset to the top of the output message page. The contents of the stack are void. This operation is used to force this position to hold the message header. The next push will set the contents of the message header and subsequent pushes will complete the message.

not, neg  unary inverse or negative of the stack top.

inc, dec  increment or decrement the stack top.

shl, shr  arithmetic shift to the left or to the right.

send  the current stack is sent as message. In the top of the output message page must be the message header (see flush instruction). The stack pointer is reset to zero and the output queue back pointer is incremented, leaving a fresh empty stack replacing the one previously sent. The contents of the new stack are void.

rel  releases the processor. Aborts the processing of the current method, without leaving any trace of the previous environment (PC, Input Message). The input message is discarded by incrementing the input queue front pointer until a valid message is reached (the header is not void). The cache front pointer is incremented once to place the new object/method pair. If the back of the process list is reached, the execution unit comes to a halt.
5.3.4. Operation Control

The operation control is determined by activation of the lines set by instruction decoder. The action takes the form of data transfer through the operative unit and node busses. Functional units incorporated into registers or enclosed into the ALU are responsible for transformations in the data. Transformations are normally sourced and returned to the stack top shadow register. In parallel with the transformations, the shadow register and the stack (output message page) are updated to restore the correct values resulting from the previous operation.

The two phase cycle allows sharing the processor internal pages (message and cache) with other units. During the fetch/decode phase, the instruction is transported through an eight bit dedicated bus, freeing both internal busses for the use of the other units. In this first phase the instruction is loaded into the operand register and execution decoders, setting the lines that will be activated in the next phase.

The system clock is responsible for activating the decoded lines. These lines operate the read and write controls on the registers. They also control gateways that interconnect the operative unit, internal and external busses of the processor. These lines select the path through which data transfers take place.

The control lines also determine the operation selected for functional units such as counters, comparators and shifters. The ALU is commanded to perform one of the operations that can be programmed through its fourteen selection lines. Control lines also dictate the sources and destination of ALU operations.

The main flow of operation during execution is a simultaneous transfer from and to the stack top register. In most transfer operations the stack top is
source or destination. When the state of the stack is affected, there is an update between the old value in the stack top register and the stack page. At the same time, a new result is transferred into or from another port of the stack top register. For example, in a push, the previous value in the stack top is sent to the stack while the new value enters the stack register by another port.

ALU operations are often sourced from the stack top and one of the busses. The result is also returned to the stack top unless the operation is a branch. Branches source the operand and program counter and return directly into the program counter. When the operation in the ALU results into the stack top, the effective result storage is delayed to the next first phase. The delay was designed to give a better margin for the propagation in the 32 bit carry chain.
Chapter 6: Architecture Simulation

This chapter describes BROOM architecture simulation. The architecture operation was simulated in C++, seeking to assess the feasibility of the proposed computing model. Mainly the instructions and operations used for object and message support were modelled under a message based timescale. The simulation was used as a laboratory to develop the architecture and organisation design.

6.1. Introduction

The BROOM architecture simulation was the first step towards its implementation. The simulation was used to test the preliminary design and to consolidate features. The simulation centred on the message flow level, depicting mainly the message pipeline. The design of the communication protocol, memory management and object-oriented support instructions were consolidated through the simulation design.

Implementing the multi-processor parallel architecture in software required writing a program that simulates the system components (i.e., processors, network and system modules) as individual instances. These instances were then assembled into a configurable cluster, holding up to 16 nodes. The node processor itself is composed of smaller concurrent units, arranged as encapsulated modules (C++ classes) to allow easy reconfiguration.

A large part of the program was dedicated to the user interface. The importance of the user interface arises from the purpose of the simulation as an interactive platform for system evaluation and iterative design. For this, a set of unique tools was designed and incorporated to the simulator through a user-interface module. Much of this code is required to provide observation portholes and managing the occurrence of timed events in the simulated parallel network.
A set of programs that simulates the average workload in such system was distributed over the network to drive the simulation. The activity of each component was observed through on-line indexes, suggesting variations in the organisation to achieve the best use of parallel resources.

6.2. Program Structure

The simulator itself was written in the programming language C++ and consists of about 2000 lines of code, distributed over five specification modules and eight implementation modules. The program structure in the simulation mimics the regular organisation of the BROOM architecture. It is organised in a hierarchical structure, with each level subdivided into encapsulated units. The organisation in modules was intended to accelerate the iterative procedure of testing and redesigning the simulated architecture. The five specification modules are listed below:

- **MAIN** Tables with configuration constants.
- **BROOM** high level module containing the network and node description.
- **UNITS** intermediate level module containing the general organization of the message pipeline units.
- **BASICS** low level module containing sub-systems internal to the units.
- **WINDOWS** this module contains the simulation interactive user interface.

The MAIN module contains constants to configure resource size for the various units. The constants are used to experiment with different number of nodes, size of queues, memory, caches and pages. The values were adjusted to fine granularity object-oriented computation, directed to the best utilisation of VLSI resources.
The BROOM module contains two classes: BROOM_NET and BROOM_NODE. These two classes are encapsulated in a single implementation module. BROOM_NET represents a single BROOM cluster implemented in the simulation. It accounts for the message transfers across the network and schedules instances of the node processor.

BROOM_NODE is the class representing the node processor. It receives and delivers messages to the network and holds an instance of each unit class. The main task of the BROOM_NODE structure is to schedule the operation of units according to the established priority. The dual level scheduling provides two perspectives of system activity, one with global message traffic and the other showing the detailed processing.

The UNITS module contains the classes representing the stages of the message pipeline and emulations for the system modules. The classes described in this module are: Queue_manager, Message_Unit, Execution_Unit, Operative_Unit, Mass_Storage and Monitor_Shell. The first three are implemented in separate modules corresponding to the pipeline controllers: the communication, memory and execution units. The last two modules are system modules corresponding to the mass store controller and switching element and are implemented together with the communication unit, since they interact directly.

Classes in the UNITS module model the algorithms assigned to each of the pipeline controllers and system modules. They also generate statistics about the relative performance between the units, taking into account estimated time for the completion of tasks assigned to each unit. The observation of these statistics allows changes in design decisions for the algorithms and structures of the units.
The BASICS module encapsulates classes of basics elements in BROOM design. It defines the formats for messages, object and method pages, queues, caches and instructions. Each class in the module represents the functionality of a unit controller or resource. The grouping of definitions in this module simplifies program upgrading. Hence, parts can be redesigned without interfering with the rest of the system.

The WINDOW module provides an interactive windowed display facility, using the Unix Curses package. The implementation was divided into two modules, one with advanced function supporting the classes I/O and other in the lower level, interfacing the Curses library. The windowing facility was very useful in observing simultaneously all the parallel activity of nodes and units.

The instruction set is not evaluated on the simulation, since it runs at the message transfer level. The BROOM architecture targets mainly the message processing level and the instruction set derives from the transputer, a well proven architecture [Barron83]. Instructions are modelled as C++ methods associated with the class memory_cell, that represents a 32 bit word. The purpose of instruction implementation is to simplify the programming of test routines. These routines are implemented in a separate module and emulate the expected run time behaviour of the machine.

6.3. Message Driven Timing

The simulator works mainly at message flow level, and message transfer is the time step. The message is used in the same sense as in the BROOM abstract architecture. The message is the process activation token and carries the parameters for the method. The simulation is advanced on a message unit basis, each unit running its whole task at the reception of a message. The parallelism is
emulated through the control of a network scheduler and a node scheduler, tracking the message route independently at these two levels.

Message and simulation timing flow together in a cascade of two level task schedulers. The system level schedules requests of nodes with pending message processing. The node level schedules units that are assigned with slots to perform operations associated with a message. As the message proceeds into the pipeline it enters tokens into the node scheduler, requesting slots to the unit responsible for the next processing step. These entries are forwarded to the system scheduler, requesting slots for node processing.

Once a unit has started to perform its task, it proceeds to the end and accounts for the time spent by updating its own timer register. The time is estimated in number of machine cycles necessary to complete the operation. The timing of all units is collected in a time credit register in the node. This enables an idle unit to start in the middle of another operation by using the credit available.

After a full round, when all pending nodes were served, the time in all nodes and units is synchronised by the maximum time in their registers. This timing scheme is a cross between clock and event driven simulation. It is coarse and imprecise for an absolute analysis but is good enough to detect problems in the relative activity of units. Another justification for the scheme is that it renders the simulation fast enough so it can be used interactively.

6.4. Simulated Architecture

The simulated architecture differed slightly from the adopted BROOM architecture. This is due to the original purpose of it, i.e., to be the testbed for design development. The definitive BROOM architecture evolved from the
primitive simulator design, considering the observation of its features and its behaviour.

The system level in the simulation is the same described in chapter four, with the network restricted to a single cluster. Messages have the same format with a seven word body and the header addressing the destination node, object and method. The protocol was simplified with the omission of the relay exception protocol. The sequential nature of the program was used to implement the bus arbitration.

The system modules were implemented within the units that interface them, just to allow the system to operate in a minimal configuration. The switching element was not necessary since there was a single cluster and there was no relay protocol. The switching element was substituted by a monitor module, working as an independent node, containing a terminal connected directly to the cluster bus. The monitor node allows I/O interaction with the simulation user interface.

The node architecture is similar to the one described in chapter five. One characteristic is that the memory unit was not well defined and its duties were blurred with those of the communication unit. The execution unit was also affected by a simplified instruction implementation and register organisation.

The register organisation is one of the major differences from the description in chapter five. There are two sets of register buffers that interface the communication unit queues with the network and operative unit. One of the buffers stores messages *en route* between the queues and network bus. The other buffer holds a copy of the input message and plays the role of a combined $ and % register set. The purpose of these buffers is to give an extra degree of freedom to message operation and enable more parallel action of the units.
The most simplified unit is the execution unit, where only send and release instructions are simulated. All the other instructions are directly encoded in C++ equivalent operations. The combined $ and % registers have the advantage that input parameters are already loaded into the stack, saving instructions and machine cycles for the operation. The major drawback is that these parameters disappear when a message is sent, forcing the programmer to save them in the object space.

Another interesting feature is that the register $0 is reserved to store the output message header. The advantage is that the send instruction can be issued before or after the parameters are loaded in the $ set. However, the alignment of the parameters with the message header is complex in a circular stack mechanism. For this reason, this scheme was abandoned in the implementation.

The memory management was extended in the simulation to incorporate the functions of the mass store system module. The cache strategy used was the direct-object sequential-method model, described in chapter five. This strategy works well for the small working set used in the simulation. Consequently, some forced page faults were introduced to evaluate the cache operation.

The communication unit includes the control to transfer messages from the two buffers. The relay procedure is not implemented, therefore a message sent to an overflown queue will cause the simulation to abort. Because of that, the test routines were tuned not to overload the system to cause panic.
6.5. Simulation Procedures and Results

6.5.1. User Interface and Simulation Procedure

The simulation was used to exercise the primary realisations of the BROOM architecture. Two test routines were programmed to emulate the opposite regimes of local and distributed processing. The system behaviour was observed by meters showing statistics of each component. Messages could be exchanged with nodes to intervene into the workload profile.

The two programs used to activate the simulation were simple message generators. They would generate messages within a set of intervals, ranging from two to eight cycles. This emulates the profile of a heavy communication load because these are the minimum number of cycles necessary to generate a message. The local program always sends messages to the same node and the distributed program sends messages to a random destination. Instances of these programs can be loaded interactively in any memory position in any node to shape the desired workload.

To observe the system behaviour, the user interface displays a dynamic picture of the system consisting of windows with information for each component. The user interface was based on the C Curses library to format the screen into windows. There was one window for each node and global system windows. The system windows are the network scheduler, the monitor and input and output slots. Each window is subdivided in fields presenting the collection of information pertinent to the component.

The main observed measurements were:

**Maximum Latency** the time elapsed from message creation to its acceptance in the destination communication unit (average).
Message Lifetime the time elapsed from message creation to its destruction (average).

System scheduler display of nodes current scheduled.

Input Queue display of message headers in the queue of a node.

Queue Sizes current sizes of input and output queues of a node.

Units Activity percentage of activity of each pipeline unit in a message step. A low value means that the unit is idle.

More information was also available through reports given by the monitor system when required, accounting for page faults, node activity and error conditions. The monitor also maintains an interface that accepts interrupts from the operator to change the speed of simulation, send messages, or request reports on resource usage.

6.5.2. Simulation Results

Unfortunately, the simulation could not produce concrete statistics results due to the discontinuation of software and hardware support. The machine used for the simulation was exchanged for another model and the C++ compiler supplied proved to be completely incompatible with the original code. Only the first tests were available and all enhancements inferred from the first tests were carried direct to VLSI implementation without being reiterated in a new simulation run.

The results assessed from the simulation were used to guide the improvements to the architecture described in the previous chapters. Despite the few iterations allowed to the simulation, it has been enough to indicate several modifications on the basic simulated architecture. Some of the problems exposed were the register organisation, relay protocol and cache strategy. The modifications
were mainly directed to organisation of units, since the configuration of the elements (size of messages, queues, etc.) remained constant throughout the simulation.

At system level, the main observation was on the behaviour of the network traffic. In the configuration tested, the network consisted of four nodes. The workload was incremented interactively until the network was saturated. The average message latency and lifetime measured were around 32 and 80 machine cycles respectively.

The buffer transfers were found responsible for an unnecessary extra delay, specially in the message lifetime. The solution was to eliminate these intermediate registers and route message access directly to the message queue memories. On average, this eliminates 32 cycles in the message lifetime.

The queue sizes were observed to keep a size around 2 messages long. This is mainly due to the self-controlled nature of the message generators, emulating "well behaved" algorithms. The good behaviour of these algorithms avoids the generation of a new instance of itself for each message sent. In contrast, "greedy" algorithms can multiply instances exponentially and overflow the queues. Well behaved algorithms indicated that the relay mechanism would be well tolerated as an exception handling mechanism, introducing an affordable overhead, since it would have a low occurrence profile.

The indexes measuring the relative interdependence between message pipeline units indicated a good balance in resource contention. They were marking around 90 percent activity most of the time. However, there were occasions when the execution unit activity dropped very low, in the occurrence of a page fault. Page faults were induced by moving objects across the memory.
The memory bus bandwidth is insufficient to restore cache positions before very small granularity programs exhaust the input queue. A program is considered small in terms of BROOM architecture when the code is just enough to receive and forward a single message. This can be done within six to twelve cycles. The memory bandwidth allows the update of a cache position within 20 or 28 cycles. Since BROOM proposes to support small granularity, the cache hit rate had to be optimised. This led to the development of a better caching strategy to make best use of on chip memory, to reduce message decoding times and to increase the chance of a cache hit.

The associative memory list cache strategy mentioned in section 5.2.4 would reduce the number of times that page faults would occur. However, in case it happens, there is still the possibility of rendering the execution unit idle. The only envisaged remedy would be to increase the memory bus bandwidth with large buses and external caches. None of the solutions could be adopted due to implementation restrictions.

The simulation was mainly an exercise in architecture design. The result was an evolution from a primitive implementation version to a more adequate design, aware of VLSI implications. The simulation enabled the observation of designed algorithms in motion, attesting their feasibility and proper behaviour. The empirical results obtained served as guidance to redirect resources from where they were introducing overhead, to an area were they can output full parallelism.
Chapter 7: VLSI Implementation

This chapter reports the architecture implementation into a microchip, using the CMOS 2microns technology. It describes the tools and libraries used for the design. The design approach takes into consideration the available resources for the project. A modular cell library was designed to implement the specialised operative parts. The strategies for control and operative parts are discussed for the resulting implementation.

7.1. VLSI Design Approach

The role of a good VLSI design is to reduce complexity and to assure a final product, consistent with its architectural and functional specifications. Before embarking on the physical implementation, one must consider a structured approach that trades the expected results with the available tools and design methodologies for a particular chip [Weste85]. The relevant attributes of a VLSI design can be summarised as follows:

- silicon area
- performance: speed and power
- implementation time
- testability

The first two attributes are particularly influenced by the silicon processing technology, while the last two attributes depend most on the design approach, design environment and on the designer's expertise.

We investigated several design methodologies and estimated their impact on the design attributes. For silicon area, the primary objective was to pack multiple processors onto the same silicon die, to evaluate future implementations with denser technologies. To meet this requirement, we designed an optimised cell library, which could provide minimum area for the chip datapath. The
performance constraints were also rigorously observed in the conception of these cells. The cell library was a joint work and provided support for two additional VLSI projects, also part of PhD theses [Vellas91] [Pachec91]. (Section 7.2 describes the design of the cell library).

Given our limited experience with VLSI design and the timetable constraints of the thesis, we sought to make use the available CAD tools. This would reduce the handwork involved in the design. This approach offered the best prospects of dealing with a large design in a short time. When we started the implementation, the available VLSI design tools were: the set of Berkeley design tools [Scott86] (in particular Magic) and the CAD system Solo 2000 from European Silicon Structures [ES2-87].

Magic is an interactive editor for creating and modifying VLSI circuit layouts [Ouster84]. As a symbolic editor (it has knowledge of design rules), Magic provides better design resources than a simple layout editor. The disadvantage of using Magic is that it does not incorporate a complete design, verification, and test system, to attend all phases of an IC design. For instance, Magic offers limited support for automatic routing (it only deals with local routing) and does not work with schematics. On the other hand, SOLO provides a complete and integrated tool set, with a unified data structure, that includes: a schematic graphics editor; module generators for parametrical RAM, ROM and PLA; automatic placement and (global) routing facilities; and standard cells libraries for 1.5\text{m} and 2.0\text{m} CMOS technologies [SDA88].

The option for SOLO seemed obvious, except that the available version of SOLO did not include the layout graphics editor. Consequently, there were two alternatives for implementing the processor's datapath: firstly, to use Magic and integrate the layout into the SOLO environment; and secondly, to use the standard
cells supplied by SOLO. From the viewpoint of simplicity and design automation, the alternative of using standard cells combined with the schematic editor, was certainly the most attractive one. However, silicon area would increase significantly.

To evaluate this alternative properly, we designed and implemented a toy microprocessor using standard cells all over the design. The microprocessor was proportionally sized to provide a precise idea of the effect standard cells would produce in our designs. The "toy" microprocessor was a 16-bit RISC containing: 2 instructions (add\&write and sub\&write); a set of 8 (16-bit) registers; 512 words of RAM; and 64 words of ROM.

The results obtained for this chip were extremely discouraging. The device measured 7.9x5.2mm, with 42% of its area occupied by standard cells. Considering that the complexity of our actual designs was at least three times greater than that of the "toy" microprocessor, the use of standard cells for the datapath was out of question. We therefore decided on the alternative that would result in the smallest size of die and that could provide an adequate basis for assessing the performance and chip area of implementation with modern technologies.

Before starting the layout, we carried out an extensive analysis of Magic's technology file to determine how manageable would be converting it from the available $2\mu$ to the $1.5\mu$ technology. The analysis showed that the intended modification would demand a laborious work for reprogramming and testing, which we considered out of the scope of the thesis. For this reason the chip was implemented in $2\mu$ CMOS.

To achieve a highly integrated chip layout and to shorten the turnaround time in the layout design, we effectively applied the automated design system in all
design levels. The flowchart diagram in Figure 7.1 describes the design environment used for the chip implementation. The chip layout consisted of a six-stage design process:

![Flowchart Diagram]

**Figure 7.1: VLSI Design Environment**

1) **Cells Design:** This involved the manual layout of bit cells and the automatic generation of the \( n \)-bit slices for each type of cell in the cell library. Cells were designed with Magic and their layouts exported to the SOLO environment using a software tool to convert formats (Calma to SDA). In the SOLO environment, three representations were created for each of the \( n \)-bit slices: a symbol for manipulating the slices in the design of the cell-blocks; a schematic, describing the slices' logic behaviour (to be used for functional simulation); and abstract representation with pin and cell boundary information for placement and routing.
2) **Cell-Blocks Design:** Cell-Blocks are functional units that combine the logic features of many slices. They implement registers, ALU and other functional elements, present in the datapath. Cell-Blocks were designed in three phases: firstly, they were laid out by abutment to check for DRC errors and for the need of blank-jumper cells between slices; secondly, schematics were produced for netlisting and simulation purposes; finally, symbols were automatically created for later use in the chip integration.

3) **Datapath Design:** This stage involves the combination of several cell-blocks for implementing datapath segments. Their design followed the same steps executed for the cell-blocks and included the generation of a textual placement of cells. This text records geometric characteristics related to cells abutment, rotation, and overlapping.

4) **Macroblocks Design:** Macroblocks are RAM, ROM and PLA modules created automatically by SOLO's module generators. The generation of RAM and ROM blocks and their representations is straightforward. For the PLAs, the process also involved truth-table generation (Eqntott) and Boolean minimisation (Espresso), before the final generation.

5) **Random-logic Design:** These are parts of the circuit that were designed with standard cells, using automated design programs for schematic editing and capturing.

6) **Chip Integration:** Producing the final chip layout involved several steps, supported by the automated design tools. In the first step all components were integrated in a single chip schematic (Schematic Editor) for generating netlist and layout representations. Cells and blocks were placed using a semi-automated method for controlling terminals position and direction, to reduce the
interconnection area. Routing was performed by an advance routing program supporting automatic channel generation and control functions to specify routing objectives. In this process, layouts were checked for DRC errors and the design process repeated from the beginning. The final optimised and correct layout was then converted to Calma and sent to fabrication.

7.2. Datapath Cell Library

The cell library contains a total of 64 different cells, organised into three classes: modular register building blocks, functional elements, and interface & switching elements. The design of these cells follows a design philosophy that seeks a balance between the minimum silicon area and high performance requirements.

7.2.1. Design Philosophy

As the complexity of VLSI chips increases, the need for a common shared cell library becomes essential. Sharing cell library helps to prevent duplicate development efforts and promotes exchange of ideas for new cell architectures. In addition, it contributes to a rapid prototyping and allows designers to explore alternative architecture designs by combining different cell structures with various design constraints, such as area, speed, fan-out and power consumption [Foo90].

Designing a cell library involves choosing a number of features and standards for the library. These include: cell selection; performance attributes; topological, electric and temporal characteristics; drive and interface capability; and a standard format for documenting cells. The design philosophy emphasises modularity and flexibility and attempts a trade-off between performance and silicon area. The cells satisfy a consistent set of design rules that provide a near-optimum process for synthesizing different functions. In this process, cells
concatenate linearly to form multiple-bit slices and slices are arranged in stacks to customise registers and functional units to the design purpose.

The first design decision for the library was the choice of the logic clock regime. From the three classical choices, dynamic, pseudo-static and static, [Mukher86] the pseudo-static was considered the most suitable. Dynamic circuits are smaller but unstable; static circuits are more reliable but yield complex realisations for functional implementations; the pseudo-static is a compromise. In a pseudo-static clock regime, the logic state of a register is temporarily stored in the gate capacitance of its transistor. The continuity of the clock guarantees the restoring of the logic levels. Compared with the static logic, this scheme is disadvantageous, since the clock can not be stopped during chip tests for examining the machine state, or for reducing power consumption in stand-by operation. The main advantage of the pseudo-static logic is that it provides a simple and small design for a master-slave store. The circuit consists of a pair of inverters separated by pass transistors (static logic requires twice as much) [Oliver87].

The second important design choice concerns the functional coupling of cells, conditioned by the way that transistors are arranged to synthesize a function. The coupling used is based on the "ratioed" logic [Sutton83] [Weste85]. With this logic, the coupling to the register core can be made through pass transistors, thus reducing the number of physical control lines. In addition, some simple functions can be implemented by using NMOS switching logic, which is highly area efficient. Another consequence of this technique is the use of limited voltage-swing bus, discussed later in this section.

The combination of pseudo-static and ratioed logics is the key to an effective modularity. The pseudo-static output stage provides the robustness of an open-loop, dual rail drive that eliminates electrical hazards in module interfacing.
The ratioed logic enables a wired-or input interface that reduces the interdependence between modules and simplifies the logic circuitry.

The cells in the library are designed to interconnect with each other by abutment or overlapping, with coincidence of terminals. This requires a topological and geometrical standardisation of control, data and supply lines and a fixed pitch in one direction. The implications for inter-cell routing were studied to establish guidelines, yielding a balanced relation of area economy, layout complexity and aspect ratio.

The cell frame is structured in an orthogonal grid of rails and tracks, with bits concatenating alongside tracks and registers stacking alongside rails (Figure 7.2.a). Tracks carry control lines within standard geometric patterns for each cell family. This allows the division of a register into specialised fields of any bit count. The number of tracks depends on the control needs for a particular circuit and seldom determine the size of the cell.

Figure 7.2: The Interconnection Structure of Cells

The rails determine the standard pitch of the cell library so that cells can stack without the need of terminal routing. Studies with circuit sketches resulted in the establishment of seven data/supply rails and two connection rails, totalling
nine rails in a 64 lambda cell pitch. The supply rails are large current metal strips for powering the circuits along the datapath. The five data rails cover the intercommunication needs of datapath segment, while the connection rails are reserved to internal circuit routing and contacts between layers.

The concatenation of bit cells is made by overlapping adjacent supply rails alongside the tracks. To match the corresponding supply polarity, the cell must be flipped around the rail axis (see Figure 7.2.b). This reduces the total block area and alleviates the overhead caused by the large supply rails. Moreover, this strategy reduces the chances of well-substrate’s conflicts and their electrical problems.

Three additional standards complete the set of design rules: the transistor banding, the transistor size and the bus voltage-swing. Transistor banding organises the transistor locations alongside the connection rails, reducing routing and supply lengths. The transistor banding also contributed to the standardisation of transistor width in 4 lambdas. This small size for transistors contributes to speed by limiting the load of control drivers.

The voltage swing of signals that move data between resources in the datapath affects the bus delay in proportion to the bus capacitance. Bus precharging is one of the possible techniques to address this problem. It can be used in designs where the bus is idle every other phase or cycle due to the organisation of the processor [Hennes84]. For designs such as BROOM, in which the node processor uses the bus on every clock phase, the limited voltage-swing bus is a more suitable approach. The use of this technique (granted by the NMOS pass transistors used as register access gates) required careful design and tests, and contributed to minimise the machine cycle time.
7.2.2. Modular Register Building Blocks

The main requirement in the storage cell design was to attain a flexible set of modules that could be assembled according to the desired functionality. To meet this requirement, we have designed a set of modular building blocks that can be assembled to create registers with the desired number of ports and reset/preset features.

The register building blocks were designed using the "ratioed" logic coupling technique for minimum area consumption [Sutton83]. With this approach, the register terminals can be accessed through pass-transistors, reducing considerably the necessary number of control lines. The use of "ratioed" logic halved the number of control lines, since only one line is required for each pass-transistor, instead of the two as with the transmission gate. In addition, "ratioed" logic grants smaller machine cycles due to its intrinsic limited voltage for the level "one," caused by the pass-transistors [Glasse85]. The fundamental limitation in the "ratioed" logic technique is the asymmetry in the rise and fall timings.

The register building blocks can be classified into four categories: basic register, bus read and write, preset and reset and register-to-register connections cells.

**Register Cells** - The basic register design was centred on the semi-static (or pseudo-static) approach where the charge of the dynamic memory element is refreshed every cycle [Mukher86]. The circuit, shown in Figure 7.3, implements a bit storage with two inverters and two transmission gates. The transmission gates TG1 and TG2 control the transmission and restore phases, respectively; the inverters provide complementary contents. For this circuit, the input data in X is transmitted to the output terminal Y during $\varphi_1$, and it is restored back into the
input stage during $\varphi_2$. With this scheme, it is possible to read and write a register in the same phase, by using a two-phase non-overlapping clock.

![Diagram of the basic register](image)

**Figure 7.3:** Schematic of the basic register.

Input data as well as reset and preset commands are fed into the X terminal via pass transistors, making use of one control line per input. Likewise, the register’s output is read into the datapath buses (or into another register) through pass transistors. The design should prevent activation of TG2 simultaneously with an input pass transistor, to avoid data conflict between an input data and the restored value.

This basic register has been developed using two different layout designs. The first layout provides access to the input/output terminals from both top and bottom cell boundaries. This feature permits direct data communication through the X and Y terminals, avoiding the unnecessary use of the common datapath buses. However, this cell is not transparent to bus Z, supplying terminals to buses A and B only. The second retains transparency to all three datapath buses A, B and Z but requires a larger area. Besides increasing the cell height, this configuration loses the access to X and Y terminals from one of the cells boundary (X and Y terminals are available at the top, while their complements are provided at the bottom of the cell). Although this feature can be viewed as a limitation, it can be
quite useful in cases where the previous and current cell's output values are required simultaneously, or when a master-slave configuration is desired.

**Buses Read and Write Cells** - These cells comprise a single pass transistor that connects the X and Y terminals to one specific bus. A simple design has been achieved by selecting the "ratioed" logic coupling mechanism. These cells provide read and write facilities to all existing buses, namely A, B and Z. Any combination of these buses can be used, allowing the basic register to have up to 3 bus ports.

**Preset and Reset Cells** - These are simple cells used to force a specific value into the register. They are also implemented with pass transistors plugged into the register's X terminal.

**Register-to-Register Connections** - To increase the register's connectivity, two additional ports have been provided as register-to-register connection cells. These cells contain one pass transistor that connects the X terminal of one register to the Y terminal of another register. The coupling can be made at the top and bottom boundaries of the register cell. Therefore, by coupling the appropriate building blocks, the basic register can communicate through up to 5 ports.

Single-bit modular register cells can be arranged into a full datapath width module or can be split in fields with individual bus accesses and reset/preset configurations. Apart from having up to 5 ports and reset/preset features, the basic register can perform extra functions with the addition of some functional elements such as counters, comparators and shifters.
7.2.3. Functional Elements

Functional elements are special cells designed essentially to provide extra functionality to the basic register. They include comparators, counters, shifters, and a general-purpose arithmetic and logic unit (ALU). All these functional cells can also be organised in fields of any length or even mixed, to provide the necessary functionality to the datapath.

Comparators - The cell library provides a set of four different comparators to conform with the required number of ports and functionality. Their design follows the same approach used in the basic register: some comparators offer transparency for the two basic datapath buses (A and B), while others include bus Z terminal using a taller cell. The library also includes dedicated comparators that test registers contents, either to zero or one values.

Comparators in the cell library adopt a single design strategy. Their common structure is based on the traditional exclusive-or (XOR) design, implemented with AND/OR gates. Smaller circuits could have been achieved by using the transmission gate XOR logic [Weste85]. However, switch level simulators such as Silos, [SDA88] have problems with this design, hence we decided not to use it.

Counters - The counter cells supplied in the cell library were designed to integrate easily with the basic register. The register's output is used as the counter's input; after evaluating the new value, the counter directs the result to the register's input. Three different configurations are supplied: an UP counter, a DOWN counter and a combination of both—an UP/DOWN counter.

The method adopted for the counter's carry circuit design is based on an array of pass transistors, controlled by the register's output. The counting is
accomplished through this dual carry chain (carry and its complement), similar to the one used in the ALU (see subsection Arithmetic & Logic Unit).

**Shifters** - The shifter functional element has also been specially designed to interface directly with the basic register. It is based on the barrel shifter approach [Mead80] and provides left and right shifting operations. Due to the register's "ratioed" logic design, the basic shifter has been implemented using simple pass transistors to attain a smaller silicon area. To afford special facilities for arithmetic shifts as well as for shifting concatenated register, specialised shift cells have also been designed.

**Arithmetic & Logic Unit** - The important features in the ALU design are: the type of functions it can perform, the word width, and the operating speed. To accommodate multiple VLSI projects, we designed a general-purpose ALU, with a configurable width and an optimised carry chain. The structure of the ALU resembles that presented by Mead and Conway, [Mead80]. It contains: three generic function blocks for generating P (propagate), K (kill) and R (result) signals; and the carry chain circuit. These function blocks provide inputs for the carry chain and for the ALU output stage. Twelve control wires (four per each function generator) are used to select among the various logic functions of three variables (A, B and Carry).

The carry chain is typically the most speed-critical component in the design since it must take the carry information across the width of the ALU [Glasse85]. In the standard implementation of the Manchester carry chain, a single carry chain (often using an active-low carry signal) runs through a series of pass transistors with a precharge circuit at each stage. A pair of cascading inverters is inserted at regular intervals to overcome the quadratic delay originated by the series of pass transistors. [Mead80] We have established a variation of the Manchester original
design by adopting a dual carry chain with asynchronous charge. The circuit for one stage of the modified Manchester carry chain is shown in Figure 7.4. The circuit activates the asynchronous charge by "generating" a carry. Thus, when \( P=K=0 \) (corresponding to the inputs \( A=B=1 \)), carry lines are charged properly. The dashed lines represent how the carry chain is buffered. At restoring stages, an inverter is introduced in each chain and their outputs are interchanged between the two chains.

![Figure 7.4: Carry chain circuit for the ALU](image)

The techniques used in this design serve a twofold purpose: it speeds up the ALU operation and provides design compatibility with our strategy for machine timing. The gain in performance is obtained using a single inverter rather than two at each restoring stage, and by adopting asynchronous operation for the carry chain. This avoids using a whole phase for precharging and the following phase for calculations. The design combines both operations concurrently in a full "shorter" cycle. It uses more area than the single pre-charged chain design, but this is justified to compensate the lack of look-ahead circuitry.

An additional characteristic of the ALU allows writing the result of the ALU operation without consuming datapath resources. The ALU contains an internal path that takes its outputs back to its inputs location, so that it can be loaded into one of feeding registers. This feature accounts for the overlapping of the instruction fetch and ALU operations.
7.2.4. Interface and Switching Elements

Interface elements are cells designed to improve flexibility and compatibility among operative blocks. They include: bus drivers, data-alignment converters, and padding cells. Switching elements are simple arrays of pass transistors for interconnecting bus lines.

**Bus Drivers** - These are more complex interface cells and were designed to match the electrical characteristics between internal and external buses. The bus drivers amplify the limited-driving signals of the register drivers and simultaneously convert them into full CMOS range. The bus driver has two stages: the first senses the ratioed levels; the second restores the signal level and supplies a fan-out sufficient to drive three standard cells. These drivers are designed in two types — unidirectional and bidirectional — and incorporate controls to put their outputs into high impedance.

**Data-alignment Converters** - These cells provide extra flexibility in the datapath design by exchanging the topological assignment of buses for specialised connection. These converters transpose busses lines across bit boundaries, diverting the bus route. They are used as barrel shifters, that shift in steps of eight bits, aligning bytes packed in multi-field words.

**Padding Cells** - Padding cells are interface units, used to separate modules with incompatible borders. In the cell library, modules were designed to optimise the area used for each function. However, modules are conservative in respect to the margin to avoid border conflicts. The avoidance of all possible conflicts would penalise excessively silicon area, hence the necessity to design a variety of padding cells. These cells interface contiguous modules, both horizontally and vertically, interrupting the appropriate lines.
**Switching Elements** - These elements provide a simple gating to connect or isolate two busses under demand. No signal amplification is performed and the switching elements add their own load to the busses burden. These elements can substitute full tri-state buffers in cases where minimum area is decisive. Switching elements also provide pseudo-static storage to combinational circuits. For instance, they are positioned at the input of combinational PLA's to incorporate dynamic retention to the input lines. These elements do not comply with the topological standards of the other cells but instead assume the characteristics of the external elements they connect.\(^1\)

7.3. **Chip Organization**

7.3.1. **Operative Part Organization**

The operative part of an integrated processor establishes the hardware implementation and the topological structure of the elements defined at the functional level. There are several criteria for the organisation of the operative part available to the machine designer [Anceau86]. Here, the design of the operative part is based on the principle of information locality for increased performance, with local memory and registers providing multiple accesses.

The operative part of BROOM is organised in five blocks: two message queues, two page caches and one operative unit. The locality principle was applied by the use of on-chip memory, forming data caches and message queues. The operative unit makes use of data locality by concentrating the functional elements of the various control units. These functional elements share the support environment of the single operative block, thus avoiding the duplication of circuits.

\(^1\) note that this *switching element* is a small logic device and is completely different from the switching element system module mentioned in section 4.1.3
One of the roles of the message flow pipeline is to bring information to a point where it can be readily accessed when needed. The targets for the information are the internal memory stores, built with fast static cells. They are organised in four blocks: input queue, output queue, object cache and method cache. The message queues have eight pages of eight words (of 32 bits) each and the caches, only two pages of the same size. They are positioned in the layout so that data lines face the side of the operative unit where the data will processed.

The operative unit forms a single block congregating all the registers necessary to the operation of the control units. The registers are all built from the modular cell library. Registers serving the execution unit are 32 bits wide, except for the Program Counter that has 8 bits. The registers for the other units are slices of either eight or four bits (three bits plus a driver interface cell) wide. They are concatenated to form 32 bit words.

The advantage of concatenation is that all registers share clock and preset lines, with economy of area. Their presence in the same block gives access to the common data bus, allowing exchange of information between units. For example, the Processor Identification register can be copied to or from the operational stack to assess or establish the node identification number. The counters can be dumped on the external intra-node bus for observation.

Figure 7.5 shows the block diagram of the prototype. The operative part, on the right shows the message queues on the top, the caches at the bottom and operative unit between them. On the top of the operative unit are the communication unit counters used for queue management. On the bottom, just before the Operand Register, are the memory management counters that supervise the cache and memory transfers. They use the extra cell rails to send control lines
straight across the Operand Register. The Bus Drivers perform a level conversion for bidirectional data and unidirectional control.

![Diagram of BROOM Block Diagram]

Figure 7.5: BROOM Block Diagram

7.3.2. **Control Part Organization**

Control operations are executed in a timing scheme where each phase of the clock is equivalent to a conventional machine cycle. Any machine instruction takes only one cycle with two phases. The complexity of specialist instructions is assumed by the pipelined units working in parallel in the background. The control of these units is implemented with PLA's and is encoded in a hierarchical organisation of non-deterministic finite state machines. This strategy minimizes the control size so that control part takes only one fourth of the prototype silicon area [Obrebs82].
The global strategy for control timing is based on two phase non-overlapping clock used for the pseudo-static circuits. A full cycle comprises the two phases, but each half cycle holds a stable state stored dynamically in the transistor gates. In the global control timing, this stable state is used to perform the equivalent to a full machine cycle, initiating and completing transfers within the phase transition edges.

This strategy speeds up the instruction flow, enabling instructions to execute in a single cycle. Other advantage of the strategy is that a slower clock is distributed over the circuit, reducing phase shift problems and the dead time zone between phases. One drawback is that some delays must be introduced to guard asynchronous sequential events from switching hazards. These delays are physically dependent and can induce failures on extreme operating conditions. This may reduce the yield of reliable or fast components in a fabrication lot.

This timing strategy applies to all units, that cooperate sharing resources on alternative half cycle basis. Resources used by a unit in a phase are released to the use of others in the consecutive phase. This reduces the necessity of complex bus systems with multi-ported stores.

A counter-proposal to resource alternation is to use each phase as a whole cycle, doubling the machine performance. A machine with a single phase cycle would require far more silicon area for doubled resources and a complex self synchronised control for resource contention. The extra silicon space necessary for this approach would be better used to duplicate the whole node processor, keeping the simplicity of control.

Each control unit in the message level pipeline is organised in a hierarchical structure with variable number of levels. Communication between
levels takes a whole machine cycle in the two first pipeline units. In the execution unit, where the instruction must be restricted to a single cycle, there are only two levels, operating in counter-phase.

The levels in the hierarchical control structures are closed finite state machines. These machines are designed in a non-deterministic style [Silva86], where a transition can occur to many active states simultaneously. This design approach normally results in the best silicon implementation [Floyd80].

The control is implemented with PLA's, occupying less than 25% of the chip area. The PLA's are pure combinational blocks and the states are stored in gates using switching elements adapters and static flip-flops. The flip-flops retain the output state of the FSM's and are concatenated in a scan path for monitoring of the internal processor state.

Figure 7.5 shows the control part, on the left, representing the hierarchical divisions of the control units. The controller blocks represent: Communication Message Exchanger, Communication Queue Handler and Communication Protocol Arbiter for the Communication Unit; Execution Instruction Fetch and Execution Instruction Decode (two blocks, 1 and 2) for the Execution Unit; Management Process Control, Management Message Decoder, Management Transfer Sequencer and Management Transfer Control for the Memory Management.

7.3.3. Implementation Considerations

The major trend in the design effort has been focused on two implementation considerations: the optimised use of the silicon area for improving parallelism of integrated nodes; and a short implementation time, compatible with
the thesis schedule. The composition of VLSI design tools used allowed us to complete the chip design and to meet partially these consideration points.

We experienced some difficulties for not using an integrated development system. It took longer than expected to produce the final layout and some planned tasks could not be executed. For instance, a consistent functional specification for all cells and blocks was produced to run a functional test. However, the lack of functional models for nMOS and pMOS transistors in our version of the SOLO system, added to our limited time, prevented us from performing such test. Also, design tools provided insufficient support for layout measurements and therefore delivered poor statistics about performance and transistor count. Nevertheless, the automated place and route system played an important role for floor planning and area reduction.

The floor plan organisation of the prototype takes into consideration pin access and terminal proximity for size and routing optimisation. Figure 7.6 shows the schematic floor plan for the prototype. The control part is all aligned to the left while the operative part is to the right. The left border contains control pins, while the busses encircle the other borders to the right. The relative placement of the blocks was designed to minimise the distance between connecting terminals, reducing routing lengths and propagation times.

Between the PLA column and the operative blocks lays a collection of random logic comprising drivers for the registers and flip-flops storing machine states. These flip-flops are arranged as shift registers to enable a scan observation of the internal state together with some important control signals. The drivers are qualified with the global phases to activate access lines in the operative part.
The prototype was implemented using $2\mu$ CMOS process technology with double-metal interconnection layers. It was designed towards an academic multi-project service, with a limitation of 100 mm$^2$ maximum size. The features of the BROOM node for the current implementation are summarised in Table 7.1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Processors</td>
<td>1</td>
</tr>
<tr>
<td>Data Length</td>
<td>32-b</td>
</tr>
<tr>
<td>Instruction Length</td>
<td>8-b</td>
</tr>
<tr>
<td>ALU</td>
<td>32-b</td>
</tr>
<tr>
<td>External Bus Length</td>
<td>32-b each (Network and Memory)</td>
</tr>
<tr>
<td>Method and Object Cache</td>
<td>32x32-b (total)</td>
</tr>
<tr>
<td>Message Queues</td>
<td>128x32-b (total)</td>
</tr>
<tr>
<td>Control PLAs</td>
<td>10</td>
</tr>
<tr>
<td>Machine Cycle</td>
<td>100ns (estimated)</td>
</tr>
<tr>
<td>Total I/O Bandwidth</td>
<td>40Mbyte/s per bus (estimated)</td>
</tr>
<tr>
<td>Package</td>
<td>100 pin PGA (92 used)</td>
</tr>
<tr>
<td>Number of standard gate cells</td>
<td>743</td>
</tr>
<tr>
<td>Number of full-custom cells</td>
<td>179</td>
</tr>
<tr>
<td>Number of transistors</td>
<td>50k (estimated)</td>
</tr>
<tr>
<td>Device technology</td>
<td>$2\mu$ CMOS</td>
</tr>
<tr>
<td>Die Size</td>
<td>7.6x8.5mm (64.6mm$^2$)</td>
</tr>
</tbody>
</table>

Table 7.1: BROOM Node Processor Features

The chip contains about 50K transistors (the precise number of transistors could not be established), 743 standard cells, and 179 customised cells. The layout
is 7.6 x 8.5 mm giving a total of 64.6 mm$^2$ die area, within a good margin from the limit. The total number of 92 pins includes 64 I/O data pins, the others representing control, supply/clock and test.

In the experimental evaluation for operation speed, the program SPICE was used to estimate the performance of possible critical paths in the processor's circuit. One possible critical path is the long carry chain in the arithmetic logic. It has the delay equivalent to 16 cascaded inverters. Operations involving 32 bit propagation in the chain are allowed two phases to complete. This extends the safety margin for the completion of long additions or subtractions. The other suspect path is cache to register transfer, involving decoding of cache address and delays in the internal and operative busses. The estimation with SPICE tests and data from the standard library is that both paths can operate within a clock of 10Mhz.
Chapter 8: Assessment

This chapter presents the goals proposed to this work, and assesses the results obtained. The goals are discriminated in three levels: the virtual machine, BROOM architecture and VLSI implementation. The main criteria for the assessment consider the suitability, parallelism and integration of the diverse levels of the project.

8.1. Targets Review

The central goal of this work is the design of a parallel machine that could support the object-oriented computational model. The whole project encompasses study of the model, specification of a target language, an intermediate level virtual machine, a parallel hardware architecture and its implementation in VLSI. The message-driven object-oriented model can offer the best approach to the proposed goal, with ease of hardware implementation and contributing to enhanced performance.

The message-driven object-oriented paradigm drives the whole project. It is in the base of formulation of an equivalent message-driven architecture and is behind the design philosophy itself. The horizontal project philosophy is to decompose an entire level into objects and consider the messages that will communicate them. The vertical philosophy is to design each level using the object-oriented methodology and regard the levels as different representations of the same object-oriented system.

The suitability goal proposes that the architecture should be the closest match to the object-oriented model, providing support for its primary features. This is obtained with the studies of the object model, to understand its internal mechanisms. The result is then carried though the three level methodology,
keeping the virtual, logical and physical implementations tuned to the object-oriented model requirements.

The project approaches parallelism with the object-oriented perspective in mind, taking into account the autonomous and encapsulated nature of objects. Beyond exploiting the intrinsic parallelism of the model, the approach encompasses other level of parallelism available in the message-driven object-oriented model. Asynchronous message passing and message-driven multitasking represent the architectural level parallelism, while pipelined interpretation of messages carries the parallelism to the implementation level.

Integration in object-oriented methodology means that representation levels share structures common to them all. This smooths the development path between levels and promotes coherence of functionality across the representations. The positional parameters in TOOL message statements exemplify that. They are represented by equivalent registers in both VOOM and BROOM architectures.

8.2. Virtual Machine

The first sub-goal was to formalise the object-oriented model into a target language and respective virtual machine. The message-driven paradigm was developed behind TOOL and VOOM, but object-driven operation was supported as well. The model was realised through standard message formats and procedures to interpret and execute message requests.

TOOL and VOOM were developed with VLSI implementation in mind. The choice of the message-driven, object-oriented model as the mainstream computing model reflects this. This model introduces parallelism in a flexible way, with asynchronous messages, equivalent to procedure calls that do not return values and do not block the caller.
For some parallel programs, specially requiring synchronization, the object-driven model offers a more convenient programming model. Therefore, TOOL and VOOM support partially the object-driven model. This implies a more complex architecture, supporting multiple queues and message polling to process returning and masked messages. The overhead, however, is restricted to objects operating in this mode, since the object-driven model is supported using the message-driven model.

TOOL suitably maps the message-driven model in its syntax and semantics with a *message statement* language definition. A program in TOOL is a collection of objects defined in terms of message statements. VOOM translates these statements decomposing the message in its tokens and associating an instruction with each one. The registers in VOOM actually map the respective positional parameters of the message statement in TOOL.

Most primitive statements have an equivalent VOOM instruction implementing it. However, some high level statements are implemented as calls to programmable routines. In the case of commonly used object management methods such as `new` and `delete`, these procedure calls may affect execution efficiency. Implementation of high level statements in the virtual machine is complex and requires more study of system behaviour, corroborating for a definitive hardwired routine.

Other high level construct, the inheritance list, may not satisfy the requirements of many high level languages. This is partially due to the blurred definitions of its semantics, varying largely from language to language, even among versions of the same language. We adopted this simple model of inheritance based on convenience to implementation.
Load balance is a facility partially supported by the virtual machine, with the intervention of the programmer. Migration of objects and methods is possible with virtual machine facilities like the frozen object message format. It makes good use of the standardised element formats, that allows object, methods and messages to be used interchangeably. However, frozen objects provide only means of transportation, and conversion of object references is to be made by the programmer, as well as the decision of when and where to move them.

The virtual machine supports object persistency, i.e. objects are only deleted explicitly by the user. However, the machine can detect and remove garbaged dynamic objects if their references are only kept in environments (suspended input messages) [Balou91]. Even undetectable garbage objects do not hamper memory usage, since the persistent object management swaps them to disk under demand for memory.

8.3. **BROOM Architecture**

In BROOM's architecture the message-driven model is developed into its architectural equivalent. This comprises the identification and implementation of the transport, decode and execution stages of message processing. In BROOM's architecture the intrinsic object-oriented parallelism is exploited at system level and message processing is pipelined at node level. The object-oriented methodology is applied to the design, dividing the architecture into objects that communicate through message passing.

8.3.1. **System Level**

The system architecture effectively implements object-oriented parallelism, with a distributed execution of objects multitasked in its nodes. The inter-node parallelism is based on objects running in parallel in the nodes. The
encapsulated nature of the objects justifies the distributed design of the system. Encapsulation also implies locality of access, which contributes to a better node performance.

The intra-node parallelism exploits the loose coupling between messages, processing them in a pipeline. This processing reduces the overhead imposed by the object-oriented paradigm, that slow down software implementations. Objects are multitasked by the scheduling of messages in a timeshared emulation of parallelism. The node restricts multitasking to message switching, with no time slicing. This leaves the management of processor time exclusively to the program, that is both a flexibility and an extra responsibility, even sometimes a burden. However, this design relieves the architecture from extra control to support interruption context switching and entitles the program to chose the time when switching will pose the smallest data transfer cost.

Communication between objects is carried exclusively by messages. The message plays the dual role of information transfer and computational drive. It conforms with the adopted message-driven model and is the source of distributed parallelism. Messages are always generated asynchronously, with no support for synchronous message passing. Synchronization must be obtained by releasing the processor to receive a reply message. This simplifies context handling and compensates in part the lack of time slicing. The main inconvenience is that context is discarded when the object releases the processor and must be saved by the program within the object space.

Messages address directly the object that will receive them. This simplifies and accelerates the decoding of message headers but restricts object migration and the range of the working set. For object migration all references to migrated objects must be updated in the caller objects. The working set is then limited by the
size of fields capable of uniquely identifying classes, objects, methods and nodes. Each of these fields is eight bits long, being only sufficient for a simple multiprocessor workstation. Larger multi-user systems would require a version with extended addressing capability.

Messages cross node boundaries through the system network, based on the bus topology, with a possible extension to a tree of busses. This choice reflects the expected locality and low traffic density associated with a message-driven user parallelism. The coarse time granularity of message generation, associated with the large bandwidth and small latency combine to justify the restriction that only one message at a time can travel in the bus.

The bus topology well suits VLSI implementation, optimising the bandwidth/pin-count relation. However, the original configuration limits the operation of massive program parallelism. Nevertheless, clusters can be reduced to avoid message delays and suitable inter-cluster communication can be used, for example, a hyperbus topology [Seitz84].

8.3.2. Node Level

The node architecture is a strict realisation of the message-driven flavour of the object-oriented paradigm. The design is even more strict since synchronous messages are not supported. Although TOOL is still a workable language within these restrictions, most languages depend mostly or support only synchronous messages.

BROOM design concentrates in the asynchronous messages because they constitute the basis of message-driven parallelism, while synchronous messages are the equivalent of procedure call in sequential languages. Emulation of synchronous messages may incur programming overhead in this version of the architecture, so
its implementation is desirable for fully integration to the higher level. VOOM already establishes the basis of its implementation, but timetable restrictions induced its omission so to reduce the development time.

Messages enter the node processing environment through the first stage of the pipeline, the communication unit. The two queues in this stage are the main resource of the message pipeline. They decouple the pipeline stages, allowing asynchronous operation of them. They loosen the coupling between objects in the same or different nodes, enhancing parallelism. Objects can send many messages, continue operation or yield the processor without waiting for the reception of sent messages.

The organisation in two queues has the drawback of introducing an extra copy operation for transmission of local messages. The organisation as a single shared circular buffer could avert this problem but would require specialised hardware implementation. The extra necessary complexity would only be justified with multiple on-chip processors, where many queues can be shared and the load distributed among the available processors.

Memory management is the next step in the pipeline and responsible for the message decoding. Message decoding originates many critical points on the architectural design. One already mentioned is the limited size of the working set and the object direct addressing, that are actually handled in this stage.

Most of the system functions, such as new and delete, introduce irregularities on the system design and difficult VLSI implementation. Consequently, these functions were abandoned and still require the study of alternative implementations, more suitable for silicon design. This probably could affect TOOL level, inducing redefinitions of syntax and semantics.
One recognised problem is the inefficiency of the current cache design. The suggested associative list mechanism is far better, covering also the possibility of cache size expansion. Memory bus bandwidth should also be increased to cater for finer object granularity, trading off with prototype package pin count. However, even with limitations, the cache mechanism is a strong point in the BROOM architecture. It enables full on-chip operation for methods and objects.

The execution unit finalises the message operation, causing the required transformations on the object. Execution of messages is highly improved by the high level pipeline, with a scalable, memory efficient, single cycle instruction set. The pipeline eliminates longer cycled instructions by executing them in background (like send) or by prefetching data into the fast internal caches.

The compact instruction set can closely match TOOL statements and make rational use of internal caches. The data contained in objects comprises of 32 bit words, that sometimes can waste more memory then needed. A suitable way of dealing with smaller data portions would reduce considerably memory waste. Eventually, 32 bit wide operative parts are required only for precision arithmetic operations, modestly supported in current BROOM implementation. A narrower data-path could also be used for simple algorithmic computation and in return permit more processors to be integrated in a single chip.

Parallelism is highly enhanced by the asynchronous node pipeline. The three stages of the pipeline handle three different messages in parallel. This organisation suits satisfyingly the object-oriented computational needs, managing by hardware transport, decoding and binding of messages.

There is a synergistic combination between the computational model and the machine architecture. The model provides the convenience of small
encapsulated information packages, activated by an orderly flow. The architecture responds with fast local data access and a high level overlapped processing.

However, start up latency is an inconvenience associated with any pipelined system. The mechanism is only efficient for parallel processing after the filling of the pipeline stages. Purely sequential processing of local message streams is heavily penalised by the combined length of the three stages processing time.

8.4. VLSI Implementation

The main target of VLSI implementation was to integrate an architecture that supports the object-oriented model and takes advantage of it in terms of silicon economy and performance. The design supports the model with hardwired execution of message handling tasks and the effect of its high level organisation is equivalent to microcoded method execution. Encapsulation and fine granularity of the model result in both silicon economy and high performance of small fast access on-chip memory.

The design approach involved a compromise of silicon economy, device performance and development timescale. Considering the limited men/year resource available for a Ph.D. project and our limited experience in high performance VLSI circuitry, the design effort was directed mainly to complete a deliverable prototype in due course. To cut down implementation time, area economy was sacrificed with use of standard cells and customised cells were not fully optimised for performance.

The operative part was designed with a custom designed cell library. Register address decoder/drivers and PLA state registers were designed with the SOLO/EDGE standard cell library. PLA and RAM were produced by the CAD
automatic generators and the floor-plan and routing were completed with the aid of placement tools and automatic global/channel routers.

The data-path cell library was organised modularly, with stacking horizontal slices enabling easy reconfiguration of registers and counters. This flexibility allowed fast design iterations with the area economy of full custom design. The performance of registers just matches the standard cells, but for counters and functional units there was no time or area available to design fast look-ahead carry chains. These fast carry chains are necessary to boost the performance of long 32 bit arithmetic operation and probably the ALU is in the critical path of cycle timing.

Logical design was hierarchically structured with many levels of control, based on non-deterministic FSM, but still a compromise between the project timescale and highly optimised complex logic. The combination of hierarchical and non-deterministic design styles improves the parallelism of control. It allows the distinct levels to operate concurrently, specially that non-determinism implies that a machine can transit to more than one state simultaneously.

Organisation of floor plan was directed to minimal interconnection overhead with terminals facing the routing channels, but suffering the interference of random logic implemented with standard cells. Blocks were allocated when possible to restrict signal routing to a single channel. However, an automatic algorithm controls standard cell allocation, making it difficult to control the interference of stretched control routing. This extra routing can contribute with unaccounted control delay and together with the conservative standard cell design increase the chip area around 15% over an equivalent custom implementation.
The circuit, due to resource limitations, was designed with the outdated technology of 2 micron. Even though, with a possible clock frequency of 10 Mhz, it will deliver 10 Mips, with the single clock period machine instruction cycle. This is comparable with current workstations like the SPARC SF9010IU 10MIPS [Namjoo88]. A four node station could fit a board equivalent to the SPARC, increasing the output to 40Mips, if full parallelism was exploited.

Using the current state of art 1 micron technology, clock speed can double and four enhanced nodes could fit in a single chip. A four chip workstation would have the processing power of a 16, 20Mhz node cluster, bringing the combined performance to 320MIPS. This accounts only for the raw processing power, that may not be achieved due to a lack of parallelism or sub-utilisation of the pipeline. In the other hand it does not consider the high expressivity of the node processor, executing object-oriented support operations in hardware. These operations may account for more than 50% of processing overhead when emulated in conventional machines [Ungar84].

The message-driven model is highly supported within the node processor, suiting both system requirements and low level performance enhancements. However, some sacrifice was made on expressivity to satisfy area and complexity constraints. Operations such as new and delete, which may constitute a large percentage in programs with dynamic object allocation, are not properly supported by BROOM. A more advanced VLSI implementation of VOOM should consider to support these system methods.
Chapter 9: Conclusions and Future Work

This chapter presents general conclusions. It starts giving a summary of the thesis, highlighting its main points. The research contributions to the area of object-oriented computing and architecture follow. Finally, the prospects for future research and development are outlined.

9.1. Summary

The primary goal of this thesis was to investigate and develop a parallel architecture and node microprocessor for building a general-purpose object-oriented computer. The motivation for the BROOM node and system architectures was to develop a cost-effective hardware, suitable for supporting the object-oriented paradigm, exploiting the intrinsic parallelism of the model.

The BROOM architecture has demonstrated the feasibility of programmable, highly parallel, general-purpose, object-oriented computers. The expressivity of node architecture and consistency of the system architecture can provide effective support for languages and applications, promoting the development of the object-oriented systems research field. The effectiveness of the BROOM architecture results from a coordinated effort in computational model research, architecture design and VLSI implementation.

Under the computational model research, the result comes in the form of a language and respective virtual machine definitions. The definition of TOOL and VOOM not only outlines the essential constructs present in most object-oriented models but also represents an intermediate level which considers the potentialities of hardware implementation. This was possible due to a close interaction with the design of architectural and silicon levels, guiding design decisions in the higher level towards optimised hardware utilisation.
At the architectural level, the study of model features led to exploitation of available parallelism in three levels. The system architecture defines a simple, linear, cluster-based, network for modular construction of parallel object-oriented computers. In this distributed control architecture, objects running in distinct nodes accounts for the first level of parallelism.

The node architecture fully supports in hardware a message-driven non-preemptive multitasking of objects and handling of multiple asynchronous messages. This represents an inter-message parallelism. Further down in the system operation, the architecture processes messages as high level instructions and overlaps, in a pipeline, the tasks of transport, decoding and execution of messages, exploiting an inner level of parallelism.

The VLSI implementation demonstrates the integration of hardware requirements and model features. Carrying ahead the concept that everything in an object-oriented system is an object, leads to fine granularity of information. The immediate consequences of the concept are regularity of formats and locality of access.

In BROOM implementation, object-oriented design results in the regular representation of messages, objects and methods in standard interchangeable packages, neatly stored in fast access on-chip memory. The normalised treatment of these entities is also reflected in the simplicity of the operative part and in the efficient hierarchical organisation of control. Locality of access implies small size of address registers, cache pages and effective code compaction. Small size and regularity have a significant impact on silicon economy and performance. This is attested in BROOM by the potentiality of integrating several nodes into a single chip and by the processing rate of one instruction per cycle.
Although BROOM has been designed to incarnate the message-driven, object-oriented paradigm, it is based in a conventional architecture. For practical purposes, BROOM can be considered a RISC with communication and cache facilities. The communication is essential to a parallel architecture and the cache is included for improved performance.

The Execution Unit is the kernel RISC processor, with a minimal instruction set derived from the Inmos Transputer. The few specialist instructions substitute the equivalent process based architecture by the object based paradigm. These instructions are tuned to serve message passing and object switching in replacement for procedure call and process switching.

The communication unit just provides the node with means to exchange information with other nodes. The communication protocol is designed to match the object-oriented model. This provides a simple protocol that can be totally handled by hardware, avoiding software interruptions. Since communication facilities is primordial to a parallel architecture, automatic handling of messages only adds to the overall performance, at a minimum cost.

The Memory Management basically manages the cache space, dealing with object and method faults. This is the equivalent of page fault handling in conventional architectures, only tuned to support the object-oriented environment. This tuning mainly involves the size of the memory page and cache algorithms in consonance with message traffic and object activity.

Included in the cache management algorithm is a specialist hardware for process binding. It performs the task equivalent to a scheduler in a process based machine. The binder includes a test to check class consistency of methods.
Hardware assistance for binding is commonly elected as an important contribution for object-oriented operation [Ungar84].

In summary, on the framework of a conventional architecture, the selected specialisations towards object-orientation do not overload the basic project or degrade the design time and machine performance. The tuning applied to favour an object-oriented design is in conformance with the general rules of RISC design. Specialist hardware is only used with parsimony, where its presence greatly improves the performance. None of the hardware interferes with the critical path of instruction execution. On the contrary, the applied hardware contributes to shorten the clock optimally. The tuning applied to BROOM results in an equivalent RISC architecture best adapted to the new way programs will be written.
9.2. Contributions

The research in this thesis spans through three areas of knowledge, namely computational models, computer architecture and VLSI design. To coordinate and concentrate the efforts in such vast fields, a prime consideration has been to provide answers to the following questions:

- is it possible to design a machine that congregates the main characteristics of object-oriented computation and that can be the target to a wide range of object-oriented languages?

- how the intrinsic parallelism of object-oriented computation can be applied and supported by an architecture implementable by a custom VLSI design?

- is there any mutual benefit in the interaction of object-oriented characteristics and its architectural support and implementation?

The thesis gives a positive answer to these questions. We felt that the collection of research contributions presented in this thesis can be summarised by the following list:

- study of the object-oriented computational model, discriminating the object and message-driven sub-models.

- definition of a target language capable of porting a range of parallel and sequential object-oriented languages into parallel computer architectures.

- design of a virtual machine encapsulating the major object-oriented supporting operations translated into the incarnation of the message-driven
model. These three first contributions were the result of a cooperative research and constitute the main issues of another thesis [Balou91].

- elaboration of the message-driven architectural concept that embodies three level of parallelism: inter-object, inter-message, and intra-message. This architectural model is a powerful extrapolation of techniques used in conventional machines applied to messages, at a higher level than instructions and operands.

- design of an object-oriented, message-driven parallel architecture (BROOM). BROOM implements a distributed parallelism activated by the message passing mechanism. The BROOM node applies the potential parallelism of messages processing them in a high level pipeline.

- VLSI implementation of the processor node. The node design follows the object-oriented methodology, featuring easy expandability. It also takes advantage of the model characteristics, providing fast access on-chip caches, updated by a forward looking strategy based on message passing.

- development of a full custom library for the design of registers and operative part. The library is modular and flexible, allowing easy assembling of many different register configurations. It was also a joint development [Pachec91], [Vellas91]. It also can be used as a library for silicon compilation of operative parts.

The target language definition was adapted to stand as intermediate layer at the SPAN project, funded under the European Strategic Programme for Research in Information Technology (ESPRIT Project 1588). The results of the research have been propagated by papers presented in two VLSI design conferences [OliRef89a], [OliRef89b].
9.3. Future Work

This work organises a number of concepts in object-oriented computing in a structured way. These concepts were applied throughout the project and lend themselves to be further developed or applied in other projects.

The first concept is the object-oriented virtual machine VOOM. It is fully described in [Balou91], where special features as garbage collection and load balancing are studied. The virtual machine is a model to guide future implementations of object-oriented kernel systems or an advanced hardware machine. A full implementation of the virtual machine has the advantage of its wide coverage of programming models, being adequate to support various parallel and sequential object-oriented languages.

The second proposal is a refinement of BROOM’s architecture, as a second version of the machine definition. This would involve the implementation of the other system modules, mentioned in section 4.1.3, and modifications on the message protocol and node architecture. The node architecture would benefit from a better queue organisation, a more efficient cache mechanism and special instructions to handle lower level messages.

The system modules were only proposed in this work and not fully designed, although part of their function was simulated. The development of the switching element and the mass storage controller is necessary for a full scale implementation of BROOM. The protocol on the two BROOM busses must be normalised to be common standard, understandable by all modules, and probably a memory controller module will also be needed.

The node must be upgraded to offer a better support to non-pure message-driven languages. This starts with the support for return messages, providing
retention of released environments and mechanisms to bind return messages with them. This suggests the use of linked lists to organise the queues, arranged as a single circular buffer. The linked lists would provide for easy repositioning of messages, creation of priority queues and would enhance the disposal of messages processed directly by the memory management.

The memory management should be upgraded to use the associative memory list cache organisation. This would expand the working set available in the cache, reduce the need of swapping cache pages and increase the buffering between page transfers. Exploiting the linked list queue alternative organisation, memory management can extend asynchronous processing, advancing over discarded messages in the queue.

At the execution architecture, new send instructions could improve the expressivity and performance. The implementation of a synchronous send would by-pass the overhead of emulation of synchronous protocol with asynchronous messages. The synchronous send can instruct the communication unit to retain the environment for future reference of a return message, avoiding its storage within the object page.

A lower level send instruction could also be implemented to provide communication with on-chip servers or coprocessors. This can be an option for the implementation of system routines to support TOOL object management methods such as new and delete. It would also provide a path to on-chip floating point coprocessors. It would be equivalent to a procedure call to a hardware or firmware implemented routine.

The VLSI implementation can also be upgraded to a new version of the prototype. This new version would benefit from a denser technology (probably
submicron). This will make room for the large cache implementation, alongside with the access list. The inclusion of a floating point coprocessor would supply for the need of multiply and divide operations of various flavours.

The available area will be more than fourfold the current implementation, and an obvious utilisation is to integrate more nodes in the same chip. The network bus may keep its bandwidth, due to the bus network topology. However, the memory bus needs expansion to cope with more page traffic. There is the possibility to expand it to 96 bits with current packages. The number of nodes can be increased to four or three with FPU.

Altogether, the future work points to the development of a full fleshed BROOM system. This system would require the funding of a large project, involving from the development of application systems to the production of integrated circuits and assembled BROOM workstations. It would also involve research on basic software such as operating systems and compilers to TOOL. The result would be a powerful platform were a new generation of object-oriented systems could be developed.
REFERENCES


REFERENCES


References


REFERENCES -149- REFERENCES


Appendix 1 Cell Library

Basic register

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Basic register controlled by phi1L, phi1H, phi2L and phi2H.</td>
</tr>
<tr>
<td>Location:</td>
<td>~marley/chip/mag/cells</td>
</tr>
<tr>
<td>Size:</td>
<td>h = 60 w = 68</td>
</tr>
</tbody>
</table>

| Author: | Carlo Oliveira |
| Last Updating: | 27/06/90 |
| Last Upd. done by: | Carlo Oliveira |
| Input Signals: | X |
| Output Signals: | Y |
| Control Signals: | phi1H phi2H phi1L phi2L |

Signals:

| Up/Layer | A1 Y1 Xpoly B1 |
| Down/Layer | A1 Y1 Xpoly B1 |
| Right/Layer | phi1H2 phi2H2 phi2L2 phi1L2 |
| Left/Layer | phi1H2 phi2H2 phi2L2 phi1L2 |

Comments:

It is transparent to buses A and B only.

Variations:

- reglr.mag is composed of two reg.mag cells, which are mirror images of each other.
- reg16.mag is composed of an array of 8 reglr.mag cells, side by side.
- reg4_4_4.mag is composed of (from left to right): 2 cells reglr.mag, 2 cells nullreg.mag, 2 cells reglr.mag, 2 cells nullreg.mag and 2 cells reglr.mag.
- reg7_7.mag is composed of (left to right): 3 cells reglr.mag, one cell reg.mag, 2 cells nullreg.mag, one cell reg.mag (sideways) and 3 cells reglr.mag.

Layout: regz.mag
ALU

**Cell Names:** nalu.mag, nalur.mag, nalubuf.mag, nalu4bits.mag, nalu16.mag, nalu16p.mag

**Description:** Arithmetic and Logic Unit, controlled by 12 bits: K0-K3 (Kill carry), P0-P3 (Propagate carry) and R0-R3 (Execution code).

**Location:** ~marley/chip/mag/cells

**Size:** h = 452  w = 68

**Author:** Marco Pacheco

**Last Updating:** 13/09/90

**Last Upd. done by:** Marco Pacheco

**Input Signals:** cinL cinH alua alub

**Output Signals:** coutL coutH aluout aluoutb

**Control Signals:** K0-K3 P0-P3 R0-R3

**Signals:**

| Up/Layer | A1 | alub1 | aluout1 |
| Down/Layer | A1 | coutH1 | aluout1 |
| Right/Layer | K3-K0ndiff | P3-P0ndiff |
| Left/Layer | K3-K0ndiff | P3-P0ndiff |

**Comments:**
- These cells are transparent to buses A and B only.

**Variations:**
- **nalur.mag** is basically the same as **nalu.mag** cell. The difference is in the layout of the carry-in and carry-out signals, in order to still have them in the correct side of the cell (despite the sideways movement), that is, CIN coming in from the right and COUT going out from the left.
- **nalubuf.mag** is the **nalu.mag** with 2 additional inverters for cinL and cinH signals, in order to recover the signals to continue the transmission through the cell.
- **nalu4bits** is composed of (left to right): 1 cell **nalubuf.mag**, 1 cell **nalur.mag**, 1 cell **nalu.mag** and 1 cell **nalur.mag**.
- **nalu16.mag** is composed of an array of 4 **nalu4bits** cells, side by side.
- **nalu16p.mag** is the same as **nalu16.mag** with the additional plugs - **aluplugl.mag** and **aluplugr** - to transform ndiff and poly layers to metal 1.

**Layout:** nalu.mag
Bi-directional tri-state buffer

**Cell Names:** bi3st.mag, bi3stA.mag, bi3stAlr.mag, bi3stA16.mag, bi3stB.mag, bi3stBlr.mag, bi3stB16.mag, bi3stZ.mag, bi3stZlr.mag, bi3stZ16.mag

**Description:** Bi-directional tri-state buffer, with different control lines for each direction.

**Location:** ~marley/chip/mag/cells

**Size:** h = 131 w = 68

**Author:** Marley Vellasco

**Last Updating:** 13/08/90

**Last Upd. done by:** Marley Vellasco

**Input Signals:** see comments

**Output Signals:** see comments

**Control Signals:** dpout dpoutb

**Signals:**

<table>
<thead>
<tr>
<th>Up/Layer</th>
<th>A₁</th>
<th>Y₁</th>
<th>X₁</th>
<th>Z₁</th>
<th>B₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Down/Layer</td>
<td>A₁</td>
<td>Y₁</td>
<td>X₁</td>
<td>Z₁</td>
<td>B₁</td>
</tr>
<tr>
<td>Right/Layer</td>
<td>dpout₂</td>
<td>dpout₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>soloinb₂</td>
<td>soloin₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Left/Layer</td>
<td>dpout₂</td>
<td>dpout₂</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>soloinb₂</td>
<td>soloin₂</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Comments:**

- The input and output signals depend on which bus (A, B or Z) is being buffered.
- Each cell is transparent to all buses, except to the one which is being bufferised.

**Variations:**

- bi3st.mag is the basic cell that is used in all variations, namely bi3stA.mag, bi3stB.mag and bi3stZ.mag. This cell does NOT contain any bus.
- bi3st?.mag buffers the bus ? and is transparent to all other buses (where ? can be bus A, B or Z).
- bi3st?lr.mag is composed of two bi3st?mag cells, which are mirror images of each other.
- bi3st?16.mag is composed of an array of 8 bi3st?lr.mag cells, side by side.

**Layout:** bi3st.mag

---

Appendix 1

-153- Appendix 1 Cell Library
Comparator of two inputs

**Cell Names:** comp.mag, compu.mag, complr.mag, comp16.mag, comp4_4_4.mag

**Description:** an exclusive-or that compares two inputs (Y1 and Y2) and puts the result in the compout output.

**Location:** ~marley/chip/mag/cells  

**Size:** h = 64  w = 68

**Author:** Marco Pacheco
**Last Updating:** 04/07/90
**Last Upd. done by:** Marley Vellasco
(compu.mag)

**Input Signals:** Y1  Y2
**Output Signals:** compout
**Control Signals:**

**Signals:**
- Up/Layer: Aᵢ, Y₂ᵢ, Y₁ᵢ, Bᵢ
- Down/Layer: Aᵢ, Y₁ᵢ, Y₂ᵢ, Bᵢ
- Right/Layer: compout₂
- Left/Layer

**Comments:**
- The compout output of comp.mag cell is of "wired-or" type, which means that the output has only the pull-down n-type transistor. When designing a comparator of n bits, the output of n-1 comp.mag cells are wired together, with only the nth cell containing the pull-up p-type transistor. The nth cell should be compu.mag.
- This cell is transparent to buses A and B only.

**Variations:**
- compu.mag is exactly the same as comp.mag with an additional pull-up p-type transistor connected to the compout output. (h = 64, w = 66).

**Layout:** comp.mag
DOWN Counter

**Cell Names:** dnl.mag, dnlbuf.mag, dnr.mag, dnrbuf.mag

**Description:** Gets the Y output of a register as the input, counts down, and writes the result in the X output, whenever the wtX signal is active.

**Location:** ~marley/chip/mag/cells

**Size:** h = 88  w = 68

**Author:** Carlo Oliveira

**Last Updating:** 13/09/90

**Last Upd. done by:** Marco Pacheco

**Input Signals:**
- Y
- cinL
- cinH

**Output Signals:**
- X
- wtX

**Control Signals:**
- wtX

**Signals:**
- Up/Layer: $A_1$, Y, $B_1$
- Down/Layer: $A_1$, $Y_1$, $X_1$, $B_1$
- Right/Layer: $cinL_2$, $cinH_2$, wtX
- Left/Layer: $coutL_2$, $coutH_2$, wtX

**Comments:**
It is transparent to buses A and B only.

**Variations:**
- **dnr.mag** is basically the same as **dnl.mag** cell. The difference is in the layout of the carry-in and carry-out signals, in order to still have them in the correct side of the cell (despite the sideways movement), that is, CIN coming in from the right and COUT going out from the left.
- **dnlbuf.mag** and **dnrbuf.mag** are the same as **dnl.mag** and **dnr.mag**, with 2 additional inverters to buffer the coutH and coutL signals.

**Layout:** dnl.mag

![Diagram of DOWN Counter](image)
### SHIFT left/right

<table>
<thead>
<tr>
<th><strong>Cell Names:</strong></th>
<th>shift.mag, shiftr.mag, shiftlr.mag, shift16.mag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description:</strong></td>
<td>Gets the Y output signal of a register and puts either in the X input of the register adjacent to the right, if SR is active, or in the X input of the register adjacent to the left, if SL is active.</td>
</tr>
<tr>
<td><strong>Location:</strong></td>
<td>~marley/chip/mag/cells</td>
</tr>
<tr>
<td><strong>Size:</strong></td>
<td>h = 22  w = 68</td>
</tr>
</tbody>
</table>

| **Author:** | Carlo Oliveira |
| **Last Updating:** | 27/06/90 |
| **Last Upd. done by:** | Carlo Oliveira |
| **Input Signals:** | Y |
| **Output Signals:** | X |
| **Control Signals:** | SR  SL |

#### Signals:

| Up/Layer | A₁  Y₁  X₁  B₁  |
| Down/Layer | A₁  Y₁  X₁  B₁  |
| Right/Layer | SR₂  SL₂  |
| Left/Layer | SR₂  SL₂  |

#### Comments:

- It is transparent to buses A and B only.

#### Variations:

- shift16.mag is composed of an array of 8 shiftlr.mag cells, side by side.

**Layout:** shift.mag
Appendix 2: Hardware Manual

BROOM HARDWARE MANUAL

BROOM Microprocessor

The BROOM node is implemented as a silicon microprocessor. It has two busses, both with 32 pins, one to communicate with the local memory and the other with the communication network.

![BROOM Node Pin Diagram](image)

The circuit is powered by the supply pins VDD and GND. The RESET pin accounts for initialization, resetting all registers and fetching the initialization object (in absolute position zero) together with the initialization method (absolute position on) and executing the start up procedure. The signal GRABO selects the node that will be the initial master. The general clock is provided by PHI1 and PHI2 that must provide a two phase non-overlapping clock timing. The other signals control the memory and network bus protocols.

**Network Protocol**

- **PROPAGATE**: This line works both for transmission request and transmission acknowledgment.
- **GRANT**: Is generated by the current master to be daisy-chained by all processors in order to arbitrate the one which has the priority to use the network bus. The signal is generated as a copy of GRANTED if the current node is not interested in transmission or is not in condition of serving as relay.
- **GRANTED**: when active, informs the node that it has being granted the permission to use the network bus. It is valid only if PROPAGATE is off, indicating that no transmission is currently going on. If activated together with PROPAGATE, indicates that the node has being chosen to relay a rejected message.

**Memory Protocol**

- **MESSAGE**: requests the node to receive a memory message. The message is expected to be generated by the object manager restoring a unbound message that have now being installed to the main memory. The full protocol requires the implementation of a watch dog to prevent cached positions from being swapped to the disk.
MESSACK instructs the sender of the MESSAGE signal to start the transmission of the message. The message must be the equivalent to a network message with the same timing and length.

HOLD informs the node that the memory bus will be in use for the time the signal is held active.

HOLDACK states the time the memory bus is not being used by the node so that external bus activity requested by HOLD can be started.

ALE address latch enable. Memory bus contains the address to be used in the next memory transaction.

RD memory read strobe. Memory contents addressed by the previous latched value are being requested by the node.

WT memory write strobe. Memory contents addressed by the previous latched value are being changed by the node.

2.1. Communication Unit

![Communication timing diagram](image)

Figure 2.1.1: Communication timing diagram
Message Exchange Control

This unit controls the network bus transfers, assigning an alternating priority to output and input of messages. It controls the packet transfer length both for transmitting and receiving.

**STATE**
- RXBusy: input queue is receiving a message.
- RXHead: the header of a message is being received.
- IDLE: both reception and transmission are idle.
- TXRelay: a relayed message is being transmitted.
- TX Busy: a relayed message is being transmitted.

**INPUT**
- RXAbort: combination of signals that indicates that a reception has been aborted.
- MCEnd: the message word counter has reached its end.
- Message: external request to send a message to input queue.
- Propagated: sampling of the external propagate signal.
- Granted: sampling of the external Grant signal sent by another processor. Indicates that the current node has the privilege to transmit or relay a message.
- LocalDest: comparison of the contents of the Network Bus with the PID register.
- IQFull: the input queue is full.
- OQFull: the output queue is full.
- OQActive: there are messages in the output queue waiting to be sent.
- RESET: general initialization pulse.
- TXGrant: the sampling of the daisy chain indicates that this node had got the privilege to transmit.
- ForRelay: a message for another node has being relayed by this node. It will be forwarded from the input queue directly to the correct destination node.
**OUTPUT**

- **IncMC, IQwtP1**: increments the message word counter, writes the message word into the input queue.
- **RstMC**: resets the message word counter.
- **RXOK, IncIB**: a message has been received to completion, increments the input queue back pointer.
- **IQisMB**: input queue receives data from main memory through Memory Bus.
- **NDInP1**: the external network bus driver is selected to admit data from the network during the first phase.
- **SetProp**: activates the external propagate signal.
- **TXReq**: request to the network protocol arbiter to grab the next slot to transmit a message.
- **OQrdP1**: reads a word from the current message in the output queue.
- **KillGrant**: kills the propagation of the Granted signal, signaling that the node is ready to relay a message.
- **NDOutP1**: the external network bus driver is selected to submit data to the network during the first phase.
- **TXOK, IncOF**: a message has been transmitted to completion, increments the output queue front pointer.
- **IQrdP1**: reads a word from the current message in the input queue.
- **FWDOK**: a relayed message has been forwarded.

**Queue Handler**

![Queue Management Controller Diagram](image)

The Queue Management controls the sizes of input and output queues, avoiding underflow or overflow. It also holds a state controlling the availability of messages to the Execution Unit.

**STATE**

- **IQFull**: the input queue is full.
- **IQPend**: there are messages in the output queue waiting to be analysed.
- **IQActive**: there are messages in the output queue waiting to be processed.
- **OQFull**: the output queue is full.
- **OQActive**: there are messages in the output queue waiting to be sent.

**INPUT**

- **RXOK**: a message has been received to completion.
- **TXOK**: a message has been transmitted to completion.
- **ExSend**: the Execution Unit processed a send instruction.
- **ExRel**: the Execution Unit processed a release instruction.
- **OBeqOF**: the output queue back pointer is equal to the output front pointer.
- **IMeqIB**: the input queue middle pointer is equal to the input back pointer.
- **IBeqIF**: the input queue back pointer is equal to the input front pointer.

**OUTPUT**

- **IncIB**: increments the input queue back pointer.

---

Appendix 2
Protocol Arbiter

The protocol arbiter algorithm circulates the transmission priority among the nodes. Transmission is enabled to the node that captures the daisy chain grant when it is waiting to transmit.

**STATE**
- **TXWait**: a message is waiting a slot in the network bus to be transmitted.
- **SetProp**: activates the external propagate signal.
- **Master**: this node is the current master of the network bus arbitrating protocol.

**INPUT**
- **TXBusy**: a relayed message is being transmitted.
- **RXBusy**: input queue is receiving a message.
- **TXReq**: request to the network protocol arbiter to grab the next slot to transmit a message.
- **RESET**: general initialization pulse.
- **GRAB0**: selects the node that will be the initial master
- **MCEnd**: the message word counter has reached its end.
- **Propagated**: sampling of the external propagate signal.
- **Granted**: sampling of the external Grant signal sent by another processor. Indicates that the current node has the privilege to transmit or relay a message.

**OUTPUT**
- **KillGrant**: kills the propagation of the Granted signal, signaling that the node is ready to relay a message.
- **SetGrant**: generates the Grant signal, sampled by all nodes in sequence to arbitrate the privilege of transmitting or relaying a message.
- **TXGrant**: the sampling of the daisy chain indicates that this node had got the privilege to transmit.

Figure 2.1.4: Protocol Arbiter Controller
2.2. Memory Management

Cache Strategy (version 3)
There are two pages for objects and two for methods. The memory unit will try to bring a new object page and check against the one already in cache. When an object is already in cache, the memory unit will only bring the method and mark the object for replay. The replay mark means that the object and method will be cross positioned across object and method caches. They will remain like this until another cache hit happens and returns them back to their original position.

When the Execution Unit terminates the current method, the current object is restored to its original position in memory. After that, the next message is processed to bring a new object to cache. The method cache will work as a list of objects to be processed together with the message queue.

The Memory Unit will stop for the following conditions:
- the message queue is empty
- the object exists already in the cache memory
- the object cache is full
The Message Processing Control sequences the actions necessary to bring the message and respective object and method together.

**STATES**

- **Init**
  - processing a init message. It will be copied to memory.
- **Trap**
  - the current message is unbound. It will be sent to memory location zero to be processed by the Object Manager.
- **Fetch**
  - the current message is bound. Object and method will be fetched from memory if not already in the cache.
- **ForRelay**
  - a message for another node has been relayed by this node. It will be forwarded from the input queue directly to the correct destination node.
- **HoldAck**
  - the memory bus has been request for external use. While it is been used the Memory Unit will remain in this state.
- **Analyse**
  - the headers of message, object and method are being analysed.
- **CacheVoid**
  - this state is set during reset to indicate that the contents of the cache are void and should not be restored to memory. Remains until the cache is filled.
- **CCFull**
  - indicates that the code cache is full.

**INPUT**

- **CloseTrans**
  - message has been transferred and invalidated in the input queue.
- **TransferOK**
  - associated object and method already installed in memory.
- **FWDOk**
  - message already forwarded to destination node.
- **IQPend**
  - there are messages to be processed by the memory unit in the input queue.
- **HOLD**
  - external bus request.
- **ISInit**
  - header indicates that message is of init type.
- **BindFail**
  - method or object associated with object could not be found.
- **ISRelay**
  - header indicates that message is for another node.
- **IMEqIB**
  - input queue middle and back pointers coincide.
- **CBeqCF**
  - code cache back and front pointers coincide.
When a message arrives, it is queued by the communication unit. The memory management is signaled to start analysing the messages in the input queue. There is a pointer, IM, that points to the next message to be analysed. The message pointed by IM is loaded into MSH and used as a memory address to get the headers for the correspondent object and method. These headers are fetched from the memory during idle bus cycles and checked against MSH. If they match, the message is bound, else an exception is generated.

**STATES**
- **LDMes** the message header is loaded from input queue.
- **LDObj** the object header is loaded from the memory.
- **LDMet** the method header is loaded from the memory.
- **KeepObj** the object is already in memory, only the method pointer will be advanced.

**INPUT**
- **MSeqPI** the message header PID is equal to the node PID.
- **MSeqIN** the most significant bit of the message header PID indicates that it is an init message.
- **MSeqMT** the message header and method header have the same method field.
- **MSeqOB** the message header and object header have the same object field.
- **MTeqOB** the method header and object header have the same class field.
- **StartDec** the previous page transfer has ended. Another message header can be analysed.

**OUTPUT**
- **BindFail** method or object associated with object could not be found.
- **EndDec** end message decodification analysis.
the object field read from the message header addresses what will be written into the object
header register.
MSMrd.MTHwt the method field read from the message header addresses what will be written into the method
header register.
IsRelay header indicates that message is for another node.
IsInit header indicates that message is of init type.
BoundOK object and or method were found in main memory and can be fetched.
IAisIM.IQR,MSHwt the input middle pointer addresses what will be read from the input queue and written into the
message header register.

Page Transfer Sequencer

When fetching pages to the internal cache, a sequence must be established due to the sharing of a single bus to
memory/cache transfers. Apart from the reset time, when all cache pages are empty, the current page must be restored to
memory before being overwritten by a new one. After the object page is restored, the object and method pages are fetched
from the memory.

STATES
RestObj an object in cache will be restored to memory.
FetchObj an object in memory will be installed into cache.
FetchMet a method in memory will be installed into cache.

INPUT
CCEnd the cache counter has reached to its end.
Fetch the current message is bound. Object and method will be fetched from memory if not already in
the cache.
KeepObj the object is already in memory, only the method will be retrieved.

OUTPUT
CAisCBP1 the cache will be addressed by the cache back pointer in phase one.
CAisCBP2 the cache will be addressed by the cache back pointer in phase two.
IncDB increase the data cache back pointer.
IncCB increase the code cache back pointer.
TransferOK associated object and method already installed in memory.
Figure 2.2.4: Page Transfer Control

Five different page transfers are controlled by this unit. Init messages, page fault messages and object pages to be restored are transferred from internal cache or message queue to memory. Object and Method pages are fetched from memory to cache.

**STATE**
- **PutAddress** selects the source address and copy it to the memory address register.
- **MoveData** transfers pages from main to local memory and vice-versa.
- **ClearHead** invalidates the message headers in the message input queue to avoid misinterpretation by the execution unit.

**INPUT**
- **RestObj** an object in cache will be restored to memory.
- **FetchObj** an object in memory will be installed into cache.
- **FetchMet** a method in memory will be installed into cache.
- **Init** processing an init message. It will be copied to memory.
- **Trap** the current message is unbound. It will be sent to memory location zero to be processed by the Object Manager.
- **ForRelay** a message for another node has being relayed by this node. It will be forwarded.
- **EndDec** end message decodification analysis.
- **CCEnd** the cache counter has reached to its end.
- **BBBM, MBB** datapath bus or memory bus is busy, been used by execution unit.
- **MM, MO** movement of Method or Object page.
- **(R + O)** a Restore or Object fetch has been completed.
- **PHI1, PHI2** general clock phases.

**OUTPUT**
- **CloseTrans** message has been transferred and invalidated in the input queue.
- **IncCC** increases the cache word counter.
- **MDin** selects the memory bus interface to admit data into the internal bus.
MDout selects the memory bus interface to submit data to the external bus.
IQRead the data will be read from the input queue.
MPwt the data will be written into the message page.
OPwt the data will be written into the object page.
MTALE address latch enable. Memory bus contains the address to be used in next memory transaction.
MTRD memory read strobe. Memory contents addressed by the previous latched value are being requested by the node.
MTWT memory write strobe. Memory contents addressed by the previous latched value are being changed by the node.
MAisMSO1 memory address register will be loaded with the object field of the message header register.
MAisMSM1 memory address register will be loaded with the method field of the message header register.
MAisZero memory address register will be loaded with the absolute value zero.
MAisOP memory address register will be loaded with the header word of the object page.
MAisBB memory address register will be loaded using the data path bus B to source the contents of the message header register.

The diagram above depicts the page transfer operations and the registers used to address these pages. Most transfers occur between the cache and the memory, but init and page fault pages are transferred from the input queue to memory. This requires a connection of the Network Bus not only with the external Network Data Bus, but also with The Memory Data Bus. The diagram shows the alternation between the Memory Management and Execution Unit address registers.
Figure 2.2.6: Message Decode Transfers

This diagram shows the transfers which are necessary to gather the identification data from the pages and the register that contain the addresses for these operations. The comparison signals that validate the correct binding are also described in the diagram.

2.3. Execution Unit

Instruction Set

The instruction set has a two level structure with general transfer and control instructions on the top level and arithmetic and special instructions on the lower level. Top level instructions have a single four bit immediate operand and the lower level has no operand. The instructions are stack oriented with source in the two top level stack entries and the destination being the stack top.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Source1</th>
<th>Source2</th>
<th>Operation</th>
<th>Destination</th>
<th>Stack Ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>with</td>
<td>0</td>
<td>opr</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>at</td>
<td>top</td>
<td>opr</td>
<td>add</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>push</td>
<td>0</td>
<td>R[opr]</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>pop</td>
<td>0</td>
<td>top</td>
<td>add</td>
<td>R[opr]</td>
<td>-1</td>
</tr>
<tr>
<td>from</td>
<td>0</td>
<td>M[opr]</td>
<td>add</td>
<td>top</td>
<td>+1</td>
</tr>
<tr>
<td>to</td>
<td>0</td>
<td>top</td>
<td>add</td>
<td>M[opr]</td>
<td>-1</td>
</tr>
<tr>
<td>br</td>
<td>pc</td>
<td>opr</td>
<td>add</td>
<td>pc</td>
<td>nop</td>
</tr>
<tr>
<td>opr1</td>
<td>0</td>
<td>top</td>
<td>opr</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>opr2</td>
<td>top</td>
<td>sec</td>
<td>opr</td>
<td>top</td>
<td>-1</td>
</tr>
<tr>
<td>upon</td>
<td>0</td>
<td>M[top]</td>
<td>add</td>
<td>top</td>
<td>nop</td>
</tr>
<tr>
<td>pack</td>
<td>0</td>
<td>M[top]</td>
<td>inc</td>
<td>sec</td>
<td>+1, keep top</td>
</tr>
<tr>
<td>onto</td>
<td>0</td>
<td>sec</td>
<td>inc</td>
<td>M[top]</td>
<td>-1, keep top</td>
</tr>
</tbody>
</table>

Table 2.3.1: Main Instruction Patterns

In the table above, R[ ] represents the contents of the starting message stored in the input queue buffer. Object instance variables are accessed in the Object Page Buffer when M[] is indicated as operand.

<table>
<thead>
<tr>
<th>Code</th>
<th>ANB</th>
<th>AMB</th>
<th>BA</th>
<th>BB</th>
<th>NetBus -&gt; Dest</th>
<th>MemBus -&gt; Dest</th>
<th>AB</th>
<th>AA</th>
<th>AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>with</td>
<td>stk</td>
<td>-</td>
<td>top</td>
<td>opr</td>
<td>BusA -&gt; OQ[stk]</td>
<td>-</td>
<td>BB</td>
<td>top</td>
<td></td>
</tr>
<tr>
<td>at</td>
<td>-</td>
<td>-</td>
<td>opr</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>BA</td>
<td>top</td>
<td></td>
</tr>
</tbody>
</table>

Appendix 2 -168- Appendix 2: Hardware Manual
Table 2.3.2: Main Instruction Paths

AMB: address used to select the data deposited on the Memory Bus.
ANB: address used to select the data deposited on the Network Bus.
BA: contents of the datapath Bus A
BB: contents of the datapath Bus B
NetBus -> Dest: Network Bus transaction, from source to destination
MemBus -> Dest: Memory Bus transaction, from source to destination
AA: connection to the ALU input A
AB: connection to the ALU input B
AR: destination from the ALU output R (Result)

During Instruction Fetch, the program counter (PC) is deposited in the special data path bus Z. It is used to address the Method Page and the Byte Selector to fetch the operation code into the Instruction Register Bus, fed into the Execution Control machine and then into OPR through the data path bus X.

Table 2.3.3: Register Allocation

PID: Processor IDentifier
IF: Input queue front
OF: Output queue Front
OB: Object Header
**O: Header Object code
CC: Cache word Counter
PC: Program Counter
IM: Input queue Middle
MC: Message word Counter
OB: Output queue Back
MT: Message Header
MT*: Method Header
**P: Header Processor id
**U: Header User code
**C: Header Class code
DB: Data Bank
DF: Data Flag
CB: Cache Back pointer
CC: Cache Counter
CP: Cache Pointer
OPR: OPerRand Register

Appendix 2: Hardware Manual
Table 2.3.4: Top Level Instruction Encoding

<table>
<thead>
<tr>
<th>Code</th>
<th>Name</th>
<th>Kill</th>
<th>Prp</th>
<th>Res</th>
<th>Cy</th>
<th>Code</th>
<th>Name</th>
<th>Kill</th>
<th>Prp</th>
<th>Res</th>
<th>Cy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>add</td>
<td>1</td>
<td>6</td>
<td>6</td>
<td>0</td>
<td>0000</td>
<td>not</td>
<td>0</td>
<td>5</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>sub</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>1</td>
<td>0001</td>
<td>neg</td>
<td>10</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>or</td>
<td>0</td>
<td>14</td>
<td>12</td>
<td>0</td>
<td>0010</td>
<td>inc</td>
<td>5</td>
<td>10</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>0011</td>
<td>and</td>
<td>0</td>
<td>8</td>
<td>12</td>
<td>0</td>
<td>0011</td>
<td>dec</td>
<td>10</td>
<td>5</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>0100</td>
<td>xor</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td>0</td>
<td>0100</td>
<td>shl</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>xnor</td>
<td>0</td>
<td>9</td>
<td>12</td>
<td>0</td>
<td>0101</td>
<td>shr</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0110</td>
<td>dup</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>0110</td>
<td>send</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>flush</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>0111</td>
<td>rel</td>
<td>0</td>
<td>12</td>
<td>12</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.3.5: Lower Level Instruction Encoding

dup   duplicates the stack top. Increments the stack pointer and copies the top register into stack.
flush clears the stack. The stack pointer register is reset to zero.
send  the current stack is sent as message. In the absolute position zero must be the message header.
       The stack pointer is reset to zero and the output queue back pointer is incremented, leaving a fresh uninitialized stack replacing the previously sent.
rel   releases the processor. Aborts the processing of the current method, without leaving any trace of the previous environment (PC, Input Message). The input message is discarded by incrementing the input queue front pointer until a valid message is reached (PID matches). The cache front pointer is incremented once to place the new object/method pair. If the back of the queue is reached, the execution unit comes to a halt.
## Instruction Decode

<table>
<thead>
<tr>
<th>Top/Low Level</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trans,02/ctr,01</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>direct/indirect</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>const/variable</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SIGNAL NAME</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>BBBusy</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MBBusy</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDQisMB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDQrdNM</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IncSTK</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DecSTK</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ReactSTK</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDNBOut</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDNBin</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOPwtA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOPrdA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOPwtB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOPrdB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TOPwtZl</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUawtB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUawtY</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDKillCarry</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDPropagate</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDSetCarry</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDResult</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OPRrdA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>OPRrdB</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDMBIn</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDMBOut</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUisOPR</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ALUisBusA</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDOPrdM</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>IDOPwtM</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Table 2.3.6: Execution signals for complete instruction set

Branch instructions are decoded in the execution control FMS to provide the PC controls.

IAlisOPR \(=MB\): PCCwZP \(=\) PCrdA; ALUawtB \(=\) !ALUawtZ;

ALoisOPR \(=\) !ALoisBusA

---

Appendix 2: Hardware Manual

Appendix 2: Hardware Manual

-171-
Instruction Fetch

![Diagram]

Figure 2.3.1 Execution Unit decode and control

The execution of messages is controled by the FSM above. Execution is interrupted by the Memory Management when it is analysing message headers (PAUSE). When execution of a message is completed, EU starts browsing into the input queue for a new message to execute (SKIP). It goes into an idle state if there is no more messages to process in the input queue or if there is no more space in the output queue when it tries to send a message (HALT).

**STATES**
- **EXEC** an instruction will be executed during phase two.
- **HALT** the execution of a method has been interrupted because there is no stack available.
- **PAUSE** the execution of a method has been interrupted because the internal busses are being used by the Memory Management.
- **SKIP** the execution of a method has terminated. The execution unit now searches the queue for a new message that can be processed.

**INPUT**
- **IMeqIF** no message in the queue to be processed.
- **CBeqCF** the object cache is full.
- **KeepObj** the object in the cache will be reused.
- **StartDec** Memory Unit is going to decode a new message.
- **EndDec** Memory Unit has finished to decode a new message.
- **PIDeqNB** the contents of the network bus is equal to the node PID.
- **OQFull** the output queue is full.
- **NoBranch** the decoded instruction is not a branch.
- **PuPo** the decoded instruction is a push or a pop.
- **OPR3** the operand register bit 3 is one.
- **NoOprinst** the decoded instruction is not operate.
- **Oprinst** the decoded instruction is operate.
- **OpCodeCD** the decoded instruction is shift right.
- **OpCodeCF** the decoded instruction is release.
- **OpCodeCE** the decoded instruction is send.
- **brt, brf, bra,brb** the opcode decoded is a branch instruction.
DTOPisZ the contents of the stack top register are zero.

OUTPUT
- IncIF increases the input queue front pointer.
- IncDF increases the object cache front pointer.
- IncCF increases the method cache front pointer.
- TopLevel decode the opcode as a top level instruction.
- LowLevel decode the opcode as a low level instruction.
- NoOperation instructs the instruction decoder not to perform any operation.
- IncPC this performs an increment operation on the Program Counter.
- PCrDA copies the result of the operation in the ALU to the Program Counter.
- ExTOPsr the instruction decoded is a shift right.
- ExSend the instruction decoded is a send.
- ExRelease the instruction decoded is a release.
- ExPushIQ the instruction decoded is a push or a pop using IQ as source or destination.

The following diagram depicts the paths used to address and transfer data between the various storage units. The rounded corner rectangles represent busses (MD and ND are external).

![Figure 2.3.2: Execution data path](image)

Appendix 2 -173- Appendix 2: Hardware Manual
Appendix 3: Layout and Circuit Schematics

Full Layout Mask
Appendix 3: Layout and Circuit Schematics

Lower Datapath (16 Bits)
Arithmetic and Logic Unit (4 Bits)
Packaging Pin Diagram
Appendix 3: Layout and Circuit Schematics
Appendix 3

Appendix 3: Layout and Circuit Schematics