An Object-Oriented Language and Architecture.

P. J. Stow

PhD Thesis

Department of Computing

Imperial College

London SW7.

I would like to thank Professor David Howarth for all his advice and suggestions; Roy Francis for discussions on Object programming; Duncan White for his practical ideas and finally Charlie the cat for knowing the answers anyway!

An Object - Oriented Language and Architecture.

This thesis argues that Objects are a natural way of providing many of the properties which we desire in programming languages, such as data hiding and abstraction, software reusability and concurrency. It also recognises the need for flexibility and the merits of block structure and procedural languages.

The *Primula* language incorporates an Object data type with a statement and expression structure similar to that of Pascal. Object definitions contain the declarations and code for that type, and may inherit characteristics from a previous object definition. Procedures may optionally be hidden, made available to any children of the type, and made public for use from outside the object.

Instances of the object type are created dynamically, and may be thought of as single tasking machines. Communication between instances is by calls to public procedures and synchronisation occurs as each instance can only handle one call at any one time. A guard facility is available to aid the writing of monitor type constructs.

Generic types are provided to increase the ability to write general, reusable software whilst maintaining the security of strong type checking.

The use of objects introduces an overhead in addition to that imposed by the block structure. Suitable architectural support is required to minimise this and take advantage of the concurrency offered by the language.

The choice of support has been addressed by analysis of the dynamic frequency of each source level action, including the effects of block and object structures. Increasing the speed of the most frequent actions has the best effect on performance, and this is

achieved by hardware support for some actions at the expense of others. The resultant design could justifiably be classed as a *RISC* machine.

Aside from the instruction set, the principal feature of the processor, named *RASP*, is a set of registers which hold the current stack frame. This is a compromise between rapid access to a large number of variables and rapid object calls which require storing of all registers on the current stack. Modified registers are written to the stack whenever the stack frame base register is changed.

An instruction cache, in conjunction with a short, two address, instruction format, tries to minimise instruction fetches to main store. There is also a small data cache, although because of the single word access to main store, neither cache is able to anticipate processor demands.

A single-pass compiler and a detailed *RASP* emulator have been constructed to aid performance evaluation. Benchmark tests using Modula-2 programs have been used to tune the architecture and give the *RASP* a seven to ten-fold speed advantage over an 8MHz M68000 processor, using similar memory speeds and anticipated chip complexity.

Contents.

Introduction

- Computing Development 7
- Software Engineering 10
- Programming Languages 14
- Machine Architectures 25

Object Oriented Programming

- Definitions 30
- Support for Software Engineering 39
- Object Oriented Languages 40
- Comparisons and Limitations 50

Primula - An Object Oriented Programming Language

- Aims of *Primula* 53
- The Design of *Primula* 53
- Limitations and Improvements 88

Execution of *Primula*

- Features of *Primula* Execution 98
- Analysis of Program Behaviour 111
- Evaluation of Architectures 117

The Processor Architecture

- Elements of Processor Design 121
- RISC vs CISC 127
- Hardware Implementation Technologies 133
- Architecture for *Primula* 135
- The *RASP* Processor 170

Evaluation

Compiler Construction 179

Run-time Kernel 184

Emulator 185

Implementation Performance 188

Strengths and Weaknesses 211

Effects of Compiler Optimisations 213

Primula for Software Engineering 214

The Design of an Object Oriented Operating System 216

Conclusions 219

The Overall Picture 228

Appendix 1 - *Primula* Syntax 233

Appendix 2 - Pascal Code Survey Results

Results from New Survey 236

Results from Schmidz' Survey 239

Summary and Conclusions 243

Appendix 3 - Abstract Machine Definition 251

Appendix 4 - Evaluation Benchmark Programs 254

References 275

Introduction.

Computing Development.

Since their conception, computing systems have been required to solve problems of ever increasing size and complexity. In response to these demands, a variety of tools and techniques have been developed to aid the production of reliable solutions and increase productivity. Such methods have helped lower the cost of computing, making it available to more people and so increasing the number of problems to be solved. They also offer the prospect of solving problems previously considered beyond reach and so the cycle continues.

The increasing power and sophistication of these tools and techniques can be demonstrated by examining the 'generations' of hardware and software. Maximum on-line storage and computational speed constantly increase while the costs per bit and per unit speed fall. New programming techniques allow a more expressive representation and the discovery of more errors before the program is ever run.

There is a fundamental basis, however, which has remained unchanged throughout, but is now under increasing attack. This is the concept of the Von Neumann architecture of which the critical feature is a single active unit, the processor, sequentially executing a series of program steps or instructions. Increasing the speed of execution requires either increasing the speed of each step or increasing the useful work done at each step.

Today's technology is approaching the physical limit of hardware speed and so further increases in this area will be small and expensive. The latter approach requires large amounts of hardware to perform more and more complex actions within a single step. Making effective use of such actions then becomes the critical problem.

More complex operations are generally less applicable in that there are fewer situations where the operation exactly matches the work to be done. Sometimes they may perform work not required by the program which then needs undoing by the next operation. Extreme cases are a fixed purpose machine, effectively having one instruction to execute, and the Turing machine [TURI] having two instructions which can be combined to perform any computable function.

In addition, the presence of complex operations slows the execution of the simple ones. This is partly because of the extra hardware slowing the progress of the instruction and partly because the cost of the complex operations has reduced outlay on the others. Hence the complex operations must be extremely well matched to the needs of the program, or there will be no benefit. An example of this is the ICL DAP [REDD] which can operate on an entire array in a single cycle, but provides no benefit if the program does not utilise arrays.

To provide more performance at an effective cost, therefore, needs a concept outside that of Von Neumann, and much research is being done in this area.

There must always be some sequential element, however, as the inputs to an operation may depend on the results of several others. Each of these inputs may be the result of some other operation, or series of operations, and providing these actions are not dependant on each other they may execute in parallel. It is this parallelism which fifth generation languages and architectures are attempting to exploit.

One example is the functional language HOPE [BURS], based on the lambda calculus of Churchill [CHUR]. A program is a series of mathematical function definitions, and hence executing a program is the evaluation of a function with the given arguments. Such an

evaluation may require the results of a number of other functions, the evaluation of which may proceed concurrently. As HOPE possesses the mathematical property of referential transparency, there is no dependency between these functions.

Referential transparency implies that every reference, hence expression, has a unique value. This disallows the Von Neumann based destructive assignment, where the value associated with a name or reference may change as instructions are executed over time. It permits more straightforward proofs of program correctness, and eases the implementation of automated program transformation software to improve efficiency [DAR1].

Similar ideas can be found in languages based on first order predicate logic, such as Parlog [CLAR]. Such languages are called 'declarative' as they specify what is to be done, rather than how it is to be done. They are also 'non-deterministic' as parts of the program may execute in any order. The type of parallelism is called 'fine-grain' as there are many, small concurrent units.

A number of problems have been found in the implementation of such languages. Input / Output, and in particular the graphics based approach, does not sit easily with such non-deterministic languages. There are many applications where a sequential approach is both apt and necessary, and to accommodate these, a meta-language has been developed to express time ordering on the underlying program [DAR1].

The performance of such languages on their parallel architectures has not been as good as often predicted. This has been principally due to contention between the processors for the main store which is used to hold results and synchronisation data. This is exacerbated by the large amount of data structure copying required, despite program transformations. An example of such an architecture is the ALICE

machine, designed for the execution of HOPE [DAR2].

Good performance has been obtained on architectures where processors have local memory and are connected by communications links, such as in the Transputer [INM0]. These work best in a statically configured system, as moving code or data between processor memories occupies the communications links of all intermediate processors. This is also a problem when the code and data size exceeds the local memory, and performance rapidly falls.

The problems of the fifth generation suggest that there may be an alternative approach which allows a neat, natural expression of both sequential and concurrent problems, and provides efficient parallel execution. Where the grain of the parallelism is reduced by the use of sequential programming, contention between processing units will be reduced and so throughput increased.

To be worthwhile, such a system must offer a significant advantage over those in current use in either or both of the areas of reliability and productivity.

Software Engineering.

In order to produce a system which fulfils the requirements for reliability and productivity it is necessary to examine the characteristics of existing systems, and determine those thought to be desirable.

The heavy demands made of computing systems has led to the introduction of software engineering principles to help a system achieve its objectives. This recognises several important factors to which little attention has previously been paid.

* No system is static, but evolves as its environment changes. The system itself may cause change in the environment.

* Changes to a system should be documented at all levels and not just present in the code.

* Some changes may best be performed by moving back to a suitable design stage and re-implementing rather than by going immediately to coding.

* Reliability and error detection / recovery must be a fundamental part of the design to be effective.

Consideration of changes to a system over time leads to the process model of software development [LEHM]. There are a series of representations of a system, beginning from some specification and proceeding to more detailed descriptions until a suitable implementation level is reached. A change to a system must begin at the appropriate level of design and propagate towards the implementation stage.

There are several Integrated Project Support Environments which support this sort of idea, although details obviously vary [CHAR]. One highly descriptive and intuitive approach is to treat each stage in the design process as a model of the entire system, including computer and users.

The model is described by the types of its elements, and the number of instances of those types. Each element type corresponds to an entity type in the system which is being modelled, so in the traditional airline booking system typical types would be passengers, clerks, tickets, flights etc. The choice of element types, or abstractions, has a profound effect on the resulting system and is a

non-trivial task [PARN].

Each instance may have a state and may communicate with other instances of the same or different type. The possible states and communications are defined by the type of the instance, as are the actions on receiving a certain communication whilst in a certain state. Error detection and recovery actions should also be included.

In the above example, the state of a flight will include the total number of seats and the number already reserved. A typical communication would ask to reserve a seat or to inquire the number of empty spaces.

The design proceeds, according to the process model, by refining these definitions and abstracting more element types as required. It is complete when all the states and communications are fully and formally described for all the types involved.

Such techniques may produce a seat type, which only has a meaning within the flight type, instances of which represent individual seats. Their state will indicate the reservation of the seat etc. Flights may be specialised into regular and charter types, which have much in common but some extra different state information for each.

Note that from the process model a design may be changed after the implementation it produces has been running for some time. Hence such abstractions occur within the context of an operational system and not merely as a paper exercise.

There are three generally accepted forms of abstraction and these are independent of the particular design methodology adopted.

* Classification is the notion of grouping instances of similar type. It goes further than the simple typing of instances as it requires that a specialisation, which is a different type, is also a member of the general, or parent, class.

* Aggregation is the combination of existing definitions to produce some new definition. A form of this is the specialisation of a class, adding extra definitions, to form a sub-class.

* Generalisation is the parameterisation of some definition in order to allow it to act in a variety of ways or situations. The most common form is the parameterisation of a procedure to allow it to be called from many places with differing values.

The use of such abstraction methods implies some features of the design method.

* Users of an abstraction have no need to know details of its internal implementation, but only its purpose and the communications it may receive and transmit. This information hiding protects against interference with any state data and permits implementation changes without affecting the usage.

* Individual abstractions may be implemented independently once their purpose and communications have been defined. Furthermore, such abstractions may be used in contexts not envisaged when they were first designed. The separation of abstractions is called modularity and leads to the idea of reusable software units.

* There must be some method of describing the types of communications between abstractions. As the messages may themselves be abstractions, abstractions and hence their state information must also be described in terms of type. Errors can be reduced if these type descriptions can be automatically checked.

* Abstraction instances may operate independently, hence concurrently, except where communication and synchronisation is required.

* The development of an abstraction often requires that it contains other abstractions which may only have meaning in the context of the containing abstraction. This is nesting, as generally associated with block structure.

A programming language which supports the techniques of abstraction, and provides the features these imply, reduces the step between design and implementation. This reduces costs, chances of errors creeping into the system, and eases subsequent maintenance and enhancement.

Programming Languages.

This is not intended to be a detailed language survey, but a brief overview of the basis of a variety of languages and their applicability to the above ideas of software engineering.

Fortran and Cobol [ANS1] [ANS2].

Between them, these languages make up the vast majority of code in use today. Although never popular for micro-computers and on the wane for mini-computer systems, they continue to dominate the mainframe market. This can be attributed to the vast store of subroutines, libraries and personal experience which has been built up, and any newcomer must offer sufficient advantage to make starting again worthwhile.

Both languages were produced for work in specific fields and provide facilities for those fields, eg. Fortran provides a complex number type for scientific use whereas Cobol includes many operations for file handling in the data processing area. Cobol in particular contains many complex 'verbs' making it a large language to fully learn and producing traps for the unwary.

Storage allocation is strictly static, the compiler fixing the addresses of all run-time entities. This immediately prevents recursion and the use of dynamically sized data structures. It does, however, allow efficient access to variables and separate compilation of sub-programs.

Sub-program names are globally available throughout the system and therefore there is no nesting. A limited form of parameter passing is available, but note that in particular arrays must be passed by reference as the size may not be known in the sub-program at compile time when the store allocation is performed.

In addition to parameters, sub-programs may access a 'common' data area, available to all program units. Each unit must have a statement giving the structure and names within the common area, but as they are all separately compiled these may differ and are a likely source of error. They also reduce the general applicability of the

independent sub-programs.

Control structures are limited and do not directly support the nesting of statements within the controlled part of a structure. Such facilities must be simulated by the use of labels and the 'goto', which leads to a flat list of statements where program flow is often unclear. Indeed, indiscriminate use of the 'goto' has been cited by Djikstra as a major source of program errors [DJIK].

In terms of ability to support the software engineering principles outlined above, both languages are restricted in the abstractions they make because of the static store allocation and limited data structuring techniques. Separately compiled sub-programs aid modularity and hiding, but any data to be kept between calls must be kept in the global 'common' area.

The Algols [WIJN].

Algol 60 was conceived as a rival to Fortran, and made great efforts to maintain the generality of its concepts. It is argued that generality eases the learning and use of the language, at the cost of a one-off expense for the implementor.

Some of this generality lead to problems, however. The control variable of a 'For' loop could be a subscripted array element, the subscript being evaluated on each access. The 'Step' and termination values are also evaluated on each iteration, which was a major factor in the language's reputation for inefficiency.

The major feature of Algol, however, was its adoption of a stack-based dynamic store allocation system. An Algol program consists of a block, which may contain declarations and statements, which may themselves be blocks. Stack space is allocated on block entry, and deallocated on exit. A block can access any item within its textual

scope, ie. any item declared in a block which encloses the current one.

Sub-programs are formed by declaring a block, which then has a name and parameters which may be passed by value (copying) or by name. The latter can be a confusing method, requiring the evaluation of the actual parameter expression whenever the formal parameter name is used.

The textual position of the sub-programs allows them to access global identifiers via the usual scope mechanism. It also allows sub-programs to be declared within sub-programs and so definitions which only have meaning within the context of another definition can be nested within that definition and made inaccessible from outside it. This is a powerful contribution towards the idea of abstraction.

The scope rules permit recursion and calls to other blocks declared at an outer level and this, along with dynamically sized arrays, prevents addresses being known at compile time. Instead, all identifiers can be uniquely accessed by using the lexical level of their declaration within the block structure, and their offset with their block [BRIN].

A drawback to this system is the difficulty of separate compilation, as a given sub-program only has meaning within its textual scope. Along with the need for global variables , this reduces the modularity of a program since there is no real distinction between the parts.

Block structure allowed the definition of control statements such that they controlled a single statement, which could of course be a block. This provision, together with 'If..Then.. Else', allowed a more descriptive representation of the control of a program and reduced the use of the general 'goto'.

Algol has only limited data types, and cannot conveniently perform string type processing. Another major drawback is the lack of any input / output definitions within the Algol 60 language. Implementors designed their own methods, and so the portability of programs between systems was severely curtailed. The problem was only partially solved in the definition of Algol 68.

Pascal [WIR1].

Pascal is currently the most widely used block structured language. It owes much of its inheritance to the Algols, but its definition also includes features to increase its applicability and standardisation.

The principal additions are a much richer variety of basic types, including 'records', 'sets' and 'files'. Records in particular allow the expression of the data structures which Cobol is based around, and so allow the language to be used in a more general environment. Strong compile time checking spots many errors before the program is run. String handling is still poor, however, although many implementations have non-standard extensions to help with this.

Simple input and output is defined, allowing programs to be ported from one system to another without change. Only sequential files are supported though, so any other type of access is dependant on implementation defined facilities.

Algol's block structure is modified in that a statement may not be a block, although it may be a compound statement. Only the main program and sub-programs are full blocks allowed to contain declarations. The statements are also changed to improve the clarity of expression within the program.

Two additional factors in the popularity of Pascal, especially

for micro-computers, are that it requires only single-pass compilation and that a compiler to an abstract machine was made available, reducing the implementation problem to writing a simple interpreter for this machine.

For the software engineering principles outlined above, Pascal is very similar to Algol except that it offers a much richer set of types.

C [KERN].

C became available at around the same time as Pascal, and is now arguably more popular, principally through its use in the Unix [AT&T] operating system for mini-computers.

It has the dynamic store management of the Algols, necessary for recursion, but does not allow the nesting of blocks. All sub-programs, and the special sub-program called 'main' are at the same textual level.

A program generally consists of several textual files, each of which is separately compilable. Names from other files can be accessed by including their definition (ie. name and type) in the current file. This is often done by the use of other files which simply list the names a particular file wishes to make available, and including these files into others as required.

This approach has increased program modularity and the hiding of implementation details, at the expense of removing the possibility of nested abstractions.

There is only a weak form of type checking in C, which is a potential source of error particularly when many type cohesions are performed automatically by the compiler. A pre-processor, LINT, is often used to filter out some types of error. There is no checking

across multiple files.

The control structures are brief, some would say terse, but flexible. C allows access to the low-level actions of the machine and is therefore well-suited to the needs of efficient systems software. A consequence of this is that high level concepts such as arrays must be implemented in terms of pointers to areas of store, and the checks associated with array access do not automatically occur. This is a problem for the implementation of large systems.

Although concurrency is not a core language feature, the ability to work at systems level allows implementation of the code necessary to support some degree of parallelism.

Modula-2 and Ada [WIR2] [ICHB].

These languages are designed with similar aims in that they support the development of large systems and give direct access to the machine resources when needed. Modula-2 is descended from Pascal, whereas Ada has no direct ancestors.

Both languages support nested block structure, a wide range of type structures and strong type checking. In support of large systems, Modula-2 'Modules' and Ada 'Packages' allow the specification of an abstraction, in terms of its external interface, to be separate from its implementation. There is full cross-checking of module usage and the details of implementation can be hidden from users.

Due to the problems of separate compilation within block structure, both mechanisms operate at the global level. That is to say that any name made available by an abstraction is available for use by any other abstraction. The problem of providing modularity whilst allowing nested abstractions has not been fully solved.

The generalisation of an abstraction is supported by generic

packages in Ada, where a compile-time instance of the package is created for each version of the abstraction. This then allows the same source code to act for a variety of different types. Modula-2 uses a simpler mechanism of opaque types, where the general module only deals in pointers to the actual items.

There is no general support for classification as modules and packages describe an abstraction and create a single instance. There is no ability to create multiple instances of the abstractions.

Ada provides 'Tasks' and the rendez-vous remote procedure call mechanism for concurrency and synchronisation. Modula-2 is again simpler in that it provides only a process type and procedures to switch execution between processes. The body of a process is specified to be a parameter-less procedure.

These languages have gone a considerable way towards meeting the needs of the software engineer and provide more powerful mechanisms than previously available. It must be noted that Ada is also a very large language, difficult to implement and, like Cobol, may prove too big to be comfortable.

Simula and SmallTalk [DAHL] [GOLD].

These are commonly known as 'Object Oriented Languages' because of the mechanism they use to provide abstractions. A more detailed discussion of the principles and terminology of object oriented programming is given in the next chapter.

An object is an instance of an abstraction which has a state and methods to affect that state. These methods are activated by messages from some other object and may return values to that object. A class is a collection of similar objects, and in general objects are created from a class definition or description.

A sub-class may be formed by inheriting the attributes, ie. state variables and methods, of an existing class definition and then adding to or modifying them.

Simula-67 is a superset of Algol-60 and so many of the comments on Algol are directly applicable. In Simula, however, each block is treated as a class description and calling that block creates an instance, or object, of the class. Variables within the block represent its state, and sub-programs are its methods.

Blocks can be named, in a similar manner to sub-programs, and then used to prefix other blocks. These other blocks then inherit the attributes of the named block, and add to or modify them.

Instances of named blocks can be created by a special 'new' procedure, which sets up the instance off the stack, and returns a reference to it. Hence an indefinite number of block instances, objects, may exist independently of the stacking mechanism. This calls for a much higher degree of storage management than Algol.

An object can be assigned to a reference providing that it is of that class - it may therefore be a sub-class. Note though that as the reference may be to any object of the class it cannot access any of the extra attributes or redefinitions of a sub-class. As an alternative to this, an identifier may be declared as 'virtual' in which case the youngest definition is always used.

SmallTalk-80 has similar notions of class and object, but goes further. Everything in the SmallTalk system is an object, and so eg. integers are objects with the state being their value and the methods being the usual arithmetic operations. Objects communicate by message passing, each message holding the sender and receiver references, method required and any parameters.

Class definitions are also object instances and so they too must

have a definition; this is called a meta-class. To support this, there are variables which refer to a class definition. The object concept aids abstraction by providing a method for classification. It also aids in aggregation through sub-classing and in generalisation by parameterisation of instances upon creation.

There is a limit on the aggregation in that inheritance can only occur from a single parent. Both languages limit generalisation by disallowing type parameters. Although general data structures can be built using sub-classing this is often unnatural and creates type insecurities as it is not possible to specify that all objects within the structure are of the same sub-class of the structure class.

Neither language provides selective hiding of methods or support for concurrent execution of objects.

The Fifth Generation.

The basis of these declarative or applicative languages was briefly described above. The abstractions these techniques support are based on logic or mathematics and so are highly general and theoretically powerful. There are difficulties, however, in the abstraction from some concept of a state, with actions that can change that state, to a system where state exists only as a value returned from a function which had a similar state as an input.

In a system where state, or the preservation of data over time, is a required part of the solution then this form of abstraction can be less than intuitive. These problems commonly occur where real world actions must be modelled.

It is argued that by eliminating state and, in particular, global variables, the problems of modularisation and hiding disappear to a large extent. This is not the case when a system state must be passed

through the functions of the system as then any of these functions may access and modify a part of the state before returning it.

Where typing is used, as in HOPE, some mechanism is necessary to ensure the consistent definition of types across separate modules.

A similar argument can be applied to logic or constraint oriented programming. Hence we conclude that whilst fifth generation languages have much to offer, they are not suited to all classes of problem, and additionally they have the execution problems outlined above.

Summary.

The imperative languages do not appear to provide sufficiently general support for abstraction. In particular there is the conflict between the nested abstractions of block structure and the modularity provided by the flat file structure of C. This has only been partially resolved in Modula-2 and Ada.

Declarative languages provide more powerful abstractive mechanisms, but the problems of modelling state require the use of non-intuitive programming techniques and give rise to many of the execution efficiency problems.

The object concept, as introduced by Simula and adopted by SmallTalk, appears to have the potential to provide a powerful abstractive capability, including the idea of state, and support effective parallel execution. Combination with block structure would provide a nesting of both object and imperative action abstractions.

Execution efficiency can still be a problem if, as in the SmallTalk model, everything including the simplest integer is treated as an object. A second overhead can be created if a fully transparent store management and garbage collection system is employed. This can only be avoided by the error-prone method of the programmer specifying

when store is no longer required.

Machine Architectures.

Increasing the abstractive power of a language generally implies the use of concepts which are further away from a basic machine architecture. This requires more work to be done by the machine and so results in reduced performance.

The relative costs of programmers, system failures and hardware indicate that it is viable to lose some degree of performance in order to produce more reliable, easier to program systems. To assess the likely losses involved in executing a block structured, object-oriented language some well-known processors are examined and likely problems outlined.

Experience has shown that the effective implementation of block structure is highly dependent on the efficiency of the procedure call and return mechanism. A call involves the stacking of the dynamic and static link information, the return address, and any registers which the caller requires to be unchanged. Finally a new stack frame must then be placed onto the stack. A return reverses all these actions along with placing any function results.

A second issue is the calculation of run-time addresses from the lexical level and offset information supplied by the compiler, and the speed of access, particularly to frequently used local variables.

Object accesses are similar to synchronous message passing and in a language where concurrency is permitted they must go via a run-time system kernel to provide the necessary synchronisation. On a uni-processor system, this can be implemented as an outer block object with access to system facilities.

It is unlikely that each object would be given its own virtual

address space on a machine supporting such concepts as the number of objects would be too large. Hence the compiler and run-time system must provide protection between objects by software.

M68000, VAX-11, IBM 370 [MOTO][DIGI][IBMC].

These are established processors in the micro, mini and mainframe areas respectively. Their common features are a large, complex set of instructions implemented in most cases by a microcode machine, a variety of addressing modes and a number of general purpose registers. All have a kernel or supervisor state and multiple priorities for exceptions.

Virtual address spaces are supported on VAX and 370, but not on the 68000, although they are available on the upwardly compatible 68020. VAX and 370 are 32 bit machines; the 68000 is classed as a 16 bit processor because although it has 32 bit registers, its bus is half this size.

The 370 has no knowledge of the stack concept, and so all addressing must be performed via a base register. As stack frame size may be unknown until run-time, a further register will need to be used as a stack pointer. The actual subroutine call stores the old program counter value in a link register, which must then be placed on the stack. There are multiple register store and load operations to speed the saving and return of these.

Access to the run-time system can be effectively done by a call to an outer block procedure, which must then save the registers necessary to restart the caller in a known location.

The VAX has an implicit stack pointer, which along with another register designated as the frame base register, can be used to speed the call. The program counter can be directly stacked by the

subroutine call instruction and retrieved by the return.

A similar operation exists on the 68000 which also provides 'Link' and 'Unlink' instructions. 'Link' stacks the frame base register, sets the value of that register to the current stack pointer, and then adds a constant to the stack pointer. This can achieve the saving of the static link and the initialisation of a new stack frame by one instruction. 'Unlink' does the reverse on procedure return.

B6700 and ICL2900 [ORGA][BUCK].

These architectures recognise the needs of block structured programs, and provided direct hardware support for the addressing required in addition to efficient procedure calls.

The B6700 was conceived as an Algol machine and contains 32 display registers, indexed by the lexical level required, which point to the appropriate stack frames. The lexical offset is then added to the register value to obtain the operand address.

To further ease the compilers task, operators work on the top of stack values, and words are tagged to indicate their type. This allows automatic selection of operation type, depending on the operators, automatic checking of array bounds etc. Saving the static and dynamic links for a procedure call is done in a single operation, which generates a tagged 'mark stack control word'.

There are no explicit registers, thus saving the work used in loads and stores. Process switch requires the display registers and stack pointer to be saved, and this can be done in a similar manner to the treatment of exceptions, which appear like forced procedure calls. Information necessary to resume is stored at the bottom of the process stack.

The design of the 2900 recognises that the majority of operand accesses are to the local stack frame, with the next popular area being the global stack frame. It has two base registers for these frames, plus a third for general use. Words are not explicitly tagged, but there are instructions which treat words as array descriptors, performing the necessary access checks.

Protection uses a flexible ring structure, with enforced vetting of calls to a more privileged ring. This mechanism could easily be used for protection of the run-time kernel.

Problems have been experienced on both architectures with the implementation of languages where structures do not match the descriptors and tags available. This results in a reduction in performance as combinations of simpler operations must be used and more complex facilities are ignored.

RISC-I and SOAR [KATE][PATT].

Both of these processors have been developed at Berkeley University, California, to execute specific languages. It was argued that compilers were unable to take advantage of complex instructions and addressing modes, and the figures quoted in the references seem to bear this out. Hence there is little to lose by eliminating the complexity, leaving a few simple fast operations, with limiting addressing capability.

RISC-I was designed with the objective of executing C programs. To this end, the display mechanism of block structure is simplified and for speed of access RISC-I keeps the current procedure frame immediately accessible in registers.

To speed procedure calls, the current frame registers are in fact a window into a much larger register bank. Procedure call simply

requires that the base of the register window is moved, equivalent to adjusting the frame base. An overlap between windows allows parameter passing, and the bottom ten registers are permanently assigned to global values.

Overflow or underflow of the register bank causes an exception which allows software to dump or restore registers to and from a main store stack. The nature of program execution is such that this is a relatively infrequent occurrence.

Implementation of general block structure is reasonable, but involves taking care over the location of the required word; is it in the register bank or has it been dumped to main store? Object programming will cause difficulty, however, due to the massive overhead in dumping the entire register bank on each object access.

The principle feature of SOAR, SmallTalk On A Risc, is the tagging of each word to indicate if it is an integer or otherwise. This considerably improves the efficiency of the frequent operations on integer objects.

SmallTalk also requires automated garbage collection, and the system uses a method called Generation Scavenging. This is an attempt to remove unused objects and keep frequently used objects in main store rather than on disk without imposing an unacceptable overhead on the system.

It is evident that none of the architectures discussed will fully support the requirements for block-structured object-oriented programming. In particular, none address the possibility of concurrent object execution.

Object - Oriented Programming.

The discussion on software engineering showed a need to support the three elements of abstraction - classification, aggregation and generalisation - at the programming language level. It also found that a logical part of this support includes features such a modularisation and information hiding, strong typing, nesting of abstractions and concurrency.

It is argued here that the object oriented programming paradigm, although not necessarily the implementations found in Simula-67 and SmallTalk-80, is a natural way to provide these concepts. As a precursor to this argument, some of the terms and concepts used in object oriented programming are described below.

Definitions.

Note that various languages use these many of these terms in a more specific manner than described here. Language implementations of these concepts were discussed briefly when examining the programming languages and are considered again below. Other views on object oriented programming and terminology can be obtained in [STEF] [NYGA].

Objects.

Objects are the basic element of object oriented programming and a typical system will consist of many such objects. Each has a state, represented by values within the object, and responds to communication from other objects by performing some action.

The values of an object are normally held in variables belonging to that object. The actions of an object many change this state, report it to some other object etc. In general, however, an object may

not directly access the state of another, whether to read or modify.

This forms an encapsulation similar to that of an abstract data type or state machine which provides certain operations of its data, but for security prevents direct access. It is also similar to a Modula-2 module, noting that a module may optionally allow direct access to its state by exporting its variables.

Note that the above description, whilst not preventing implementation in a declarative language by passing the state of the object around, suggests an imperative style with destructive assignment. We shall assume this form of implementation, although it is not important for much of the discussion.

Actions, or *methods* as they are often known, are thus implemented by imperative procedures or functions. These are called from other objects, and hence may call other objects, by some general message passing mechanism and may return results in the same way.

The resemblance of an object to a state machine has already been noted, and it follows that objects may execute concurrently, communicating with messages where required. This leads to the possibility of true parallelism on multi-processor and distributed systems, and is discussed below.

Class.

It is evident that many objects within a system will be similar, eg. all the 'flight' objects in our earlier airline booking system. Whilst they are all independent objects having potentially different states, they have identical sets of possible states (defined by the possible values of their variables) and identical methods. Such a collection of objects is called a class. The variables and methods are called *attributes* or *properties* of the class.

A class is therefore a definition of the objects, or class instances, within it. In this sense it is similar to the type declarations found in most procedural languages. Instances are created by the actions of some other object which has a knowledge of the class. The concept of class is more powerful than that of type, however, and this is discussed under class inheritance.

The ability to create an indefinite number of class instances from one definition is an important difference from the idea of the Modula-2 module, which only has a single, static instance. A simulation of the idea can be performed in Modula-2 using records and procedure variables and this is discussed under object implementations.

Messages and Synchronisation.

We have observed that an object can be treated as a state machine which interacts with its environment by the receiving and sending of messages. Such messages must generally contain a reference to the sender and some description of the method to be executed, along with any parameters. In a system where direct communication between objects is not possible, such as a distributed system, or undesirable for semantic or efficiency reasons, a reference to the receiver must also be included.

The methods of an object form a critical region, as to ensure consistency only one method can act on the state at any one time. Hence a synchronisation mechanism is required to uphold this rule.

A busy wait solution is not acceptable as the processor may be required in order to complete the method of the object causing the wait. Hence a strategy for maintaining waiting objects is needed, recursively implying a mutually excluded area of store in which to

hold the necessary information. Access to such an area will, however, be quick and not involve any other waiting and so the busy wait solution is acceptable.

A similar situation exists with the data structures required to implement the sharing of a real processor between several objects. The principle function of this area is to hold references to objects which are ready to execute.

A different form of synchronisation is required between objects to enable waiting for a message. This synchronous message receive primitive halts the object and enables the processor to continue executing another. Execution is resumed when a message is sent, as this checks the exclusion on the object itself.

Many languages hide the message passing by using a remote procedure call at the source level. Additionally, languages decode the message and call the appropriate method without the user actually seeing the message.

Class Inheritance.

A class may *inherit* attributes from a number of *parent* classes. The attributes of the parents then appear in instances of the class in addition to any extra attributes added by the definition. A class which inherits attributes is called a *child* class. Note that some forms of inheritance allow the child to over-ride inherited attributes with definitions of its own.

The family idea is carried on to define a *descendant* as a class which has inherited from an *ancestor* via an indefinite number of levels or generations. We shall initially consider inheritance from only a single parent and then generalise the ideas to multiple inheritance.

In single inheritance, a child class is obviously an extension or modification of the parent class. Instances of the child class form a *sub-class* within the parent class, which may be considered as a *super-class* of the child.

As a child instance clearly belongs to the parent class, it may then take the place of any parent instance. In general, therefore, a reference to a particular class of object may also point to any descendant instance of that class.

This is a powerful, abstractive, notion supporting the ideas of extendable type definitions and reusable software elements. There are, however, several views of inheritance which should be distinguished [HAIL].

* Type theory inheritance simply allows the addition of extra state attributes (variables) to the parent.

* External interface inheritance allows the addition of extra methods and, as the state is hidden, permits a complete change of implementation.

* Code sharing and reuse allows a sub-class to use or hide facilities provided by the parent.

* Simple polymorphism allows a child to specify a concrete implementation of a generic parent definition.

Most implementations allow a mixture of these forms, with differing implications and restrictions. Allowing inheritance produces another form of class usage, ie. usage by an inheriting class, distinct from the external usage. The view of a class from its sub-classes is often called the inheritance interface.

Binding.

An implication of class inheritance is that a reference to a given class of object may in fact refer to a sub-class of the class. If the child is allowed to redefine externally available attributes, which definition should be used?

Static binding insists that as the reference is to an object of a particular class, the definition available for that particular class is used and any sub-class redefinitions ignored. This allows the compiler to choose the youngest definition for that class.

Dynamic binding argues that if the object is a sub-class then it should be treated as such and so the youngest definition available within the class of the object itself, rather than the reference class, should be accessed.

The use of static binding inhibits the specialisation of classes by ignoring any new definitions. It also prevents a child re-implementing the parent methods whilst maintaining the same interface. It is, however, run-time efficient where the child is a simple extension of the parent and does need a change in the basic behaviour of the object.

Dynamic binding requires a run-time choice of the definition to be used. This permits the specialisation and re-implementation cases above as the older definitions will be hidden. If expensive run-time type checking is to be avoided, however, a redefinition will have to keep the same type *signature* as the original, thus reducing the ability to simply extend the actions of the parent and inhibiting the code-sharing aspect of inheritance.

Multiple Inheritance.

This is inheritance, as described above, generalised to allow more than one parent. There are several difficulties in this, which will be considered below, and some have questioned the need and advisability of such a concept [BER1] [SCHA].

An initial problem is the inheritance of multiple attributes with the same name. A simple solution is to provide some method to select them individually, typically by naming the parent.

If the parent attributes had different functions, however, external interface inheritance may not correctly occur as the functionality of the interface has changed. In addition, the use of the wrong attribute causes breakdown of polymorphic inheritance. The child's function may not perform the specialised actions which are generically specified by the parents.

Further problems arise when considering that the tree structure of single inheritance, where each class has a single inheritance path to the root, is generalised to a directed graph. This allows a child to inherit from the same ancestor via multiple intermediate routes. The generality of the problem can be shown by considering an ancestor attribute which is independently redefined in each of the parents before being inherited into the child.

Choosing one version implies breaking the external interface inheritance and ignoring any functionality in the unchosen versions, potentially leading to any state inherited along with these versions not being correctly set. Creating a new redefinition that accesses each of the inherited definitions runs into the problem of duplicating actions on the ancestor state in the differing branches, again leading to an incorrect state or result.

Whilst a skilled programmer with full knowledge of the

inheritance structure could avoid these problems, this reduces the apparent power of multiple inheritance. Perhaps more important is that full knowledge of the inheritance structure breaks the rules for modularity, and so small changes somewhere in the structure could have far reaching and unexpected effects.

A method of avoiding the above problems is to flatten the multiple inheritance into a single inheritance chain. Ordering between child and parent classes is maintained, but other classes may appear between them in the single ordering. The choice of inherited attribute is hence made by the single inheritance mechanism and no attribute can be accessed by differing routes.

The drawbacks are that there is now no choice of the attribute to be inherited and that a class may now not easily communicate with its real parents in order to access an attribute which could well have been redefined by some intervening class. Again the power of the inheritance is restricted and knowledge of class implementation required.

Delegation.

This is a technique used instead of the inheritance as described above. In essence, an object receiving a message may delegate responsibility for handling that message to some other object.

Note that the method revolves directly around run-time objects as opposed to compile-time classes. There is a potential for greater flexibility, and the inheritance described above can readily be simulated, at the expense of much less static control over the behaviour of an object class.

The idea is discussed in connection with objects as active units, or actors, in [AGHA].

Garbage Collection.

In most object oriented languages, objects may be dynamically created and referred to via some form of pointer. Sharing of objects occurs through having multiple copies of the pointer.

Some form of storage management is required in order to allocate space for new objects, and reclaim from defunct objects. If the programmer specifies when the space allocated to an object may be reclaimed then the problem is no more than that in languages such as Pascal.

Employing this method, however, introduces all the problems associated with the use of pointers in Pascal. In particular, disposing of an object to which other objects still have references may cause fatal, but hard to trace, errors.

To overcome this, most object oriented languages handle store disposal automatically. Store is reclaimed when an object no longer has references to it and is no longer executing any code. There is, however, a considerable overhead in the detection of such cases.

The methods used are implementation, rather than language, defined and there are two major techniques, both of which require help from the compiler and knowledge of the location of object references within objects.

The 'Mark and Sweep' technique operates by finding and following all references available to the program, beginning at one point and spreading in a tree-like manner. Hence all objects accessible from the program can be found and marked, and when this phase is complete all unmarked store can be reclaimed. The opportunity is often taken to compact store during the sweep phase. A major problem is that all activity must be halted whilst the garbage collection occurs.

'Reference Counts' operate by maintaining a count of the number of references to an object with the object. When a reference is removed, a check is made and if no other references exist the object store can be reclaimed. This does not work where rings of references occur and so must be backed up by another technique for completeness.

Support for Software Engineering.

It can easily be seen that the Class concept closely matches that of abstraction in a number of ways. An important factor is that it allows description of the abstraction and then creation of a dynamic number of instances of the abstraction.

Both classes and instances may be parameterised to provide full generality. The attributes of an object may contain other objects leading to a series of nested abstractions. These may be in the form of the state variables themselves being objects, or the methods containing a series of nested method abstractions.

Use of an object requires knowledge of its messages and methods as defined in its class. The implementation is, however, unavailable to the user and indeed with external interface inheritance sub-classes may implement the same actions in a different manner with no external effect. This imposes a strong form of modularity.

Abstraction specialisation can be performed by the inheritance mechanism, principally through the polymorphism style. Reuse of code is also provided for, and this is an important factor in programmer productivity. As noted in the definition of inheritance the use of a class by its children is different to its external interface use. Note, however, that implementations may not fully support all the inheritance forms.

To provide full modularity, ie. to allow re-implementation of the

class without regard to external or inheriting users, requires that details of the implementation are also hidden from children. As noted above, this is a problem with multiple inheritance.

The sub-class mechanism, and in particular the ability to assign a sub-class object to a reference to the parent class, is orthogonal to other type checking mechanisms employed by a language. It may also require run-time checking, the typical case being the assignment of a parent class reference to a child class reference. It is necessary to establish that the object referred to by the parent class reference is of the appropriate child class.

As objects communicate through message passing, this can be used to ensure synchronisation where necessary. Hence each object can be viewed as executing concurrently with other objects in the system, waiting for messages as required. This is a good model for communication sequential processes [HOA1], in addition to allowing exploitation of some non-obvious concurrency.

Object-Oriented Languages.

A brief view of programming languages, including two of the object oriented paradigm, was presented in the introductory chapter. Here we present further details of these two, together with some less well-known object based languages, to compare the implementations of the ideas outlined above.

Simula-67 [BAHL].

As discussed above, Simula is a superset of Algol-60 and so has the block structure, data types etc of that language. Blocks are regarded as class definitions, which are instantiated into objects when invoked. Named blocks, such as procedures, are parameterised, with the actual values being passed to the instance.

From the scope rules of Algol, no attribute of a block, ie. variable or procedure declaration, may be accessed from outside of that block. Hence the standard blocks, or class definitions, of Simula accept only a single message which causes execution of the block.

Again from scope rules, blocks and hence objects may be nested. An inner block or object has direct access to all attributes within its scope, and hence all global variables.

Simula defines a *class* type, the definition of which is a block. The predefined *new* function creates an instance of a class and sends an execute message, with optional parameters. It also returns a reference to the instance which may then be used to access any attribute. There is no ability to prevent access to any attribute[1].

Communication is hence by access to the variables of another object or by the execution of a procedure in another object. The logical view of this process is that a message is sent from the caller to the referenced object which then performs the required action. When complete, it sends a reply message to the caller which is kept in a synchronous receive state until the reply arrives.

The message passing in Simula is implemented by changes in the flow of control. In the case of access to a procedure of another

---

[1] Except by providing a substitute through use of the *virtual* mechanism.

object, control is transferred to that procedure and returns on its completion. Hence the language is single-thread and may only use a single processor.

Control is also transferred to a new object when it is created. The *detach* command then returns control to the creator, and the *resume* command allows switching of execution directly between objects.

Inheritance is from a single parent and specified by prefixing the child block with the name of the parent block. As with the external interface, the inherited interface contains all attributes of the parent. The default binding method is static, although a procedure attribute may be specified as *virtual* in which case dynamic binding is applied. Redefinitions of a virtual attribute need not have the same type signature as the original and so run-time checking is required.

Sub-classing is supported, so child class objects may be assigned to parent class references. Run-time checks where needed ensure that the object of a reference is always within the reference class. References may be qualified by the name of a sub-class in order to extend the flexibility of the static binding. Again, checks ensure the qualification is legal.

Execution of the main code body of an instance begins with the oldest ancestor and calls the next descendant at the end of the block or where an *Inner* statement occurs.

SmallTalk-80 [GOLD].

SmallTalk is a much more "pure" language than Simula and treats everything in the system as objects. This has the main advantage of simplifying the language to a single major concept but the disadvantage that everything must be forced to fit this concept.

This is no more evident anywhere than in the often unnatural

treatment of integers as objects, with the usual arithmetic operators available as methods. Hence addition in SmallTalk involves sending an 'Add' message containing one operand to the object representing the second operand and result.

A further consequence of the pure object nature is that class definitions are also treated as objects. This then produces a requirement to describe the class of the class - this is called the *meta-class*.

Every object can find its class object by executing the *class* method. Hence variables defined by a meta-class are instantiated as *class variables* and can be shared between all objects of that class.

Object attributes consist of object references to indicate the state of the object and the methods of the object. Note that the SmallTalk system provides the basic object types such as integer and character. The methods are described by a simple, non-nested statement list and consist of sending messages to other objects and executing primitives.

Likewise, object definitions may not be nested. Hence the scope of a method is simply the object attributes including class variables, any variables local to the method, and any global variables. The latter are defined in one or more dictionaries for each user and may be accessed by all methods.

Communication is by messages, of which there are three forms. These are basically similar, containing receiver, method selector and optional arguments, but have differing syntax for ease of use. There is a special form to allow a more convenient representation of the messages required for conventional arithmetic expression evaluation.

Inheritance is from a single parent[2] and the binding is dynamic

---

[2] Extended SmallTalk allows multiple inheritance.

with run-time checking. A special variable, *self*, is provided to allow a method to access other methods defined or inherited in the class. There is no hiding of attributes from inherited use, but as messages must contain a method in the selector field, the variable attributes are hidden from external access.

C++ [STRO].

Like Simula, C++ is an extension of an existing language, in this case C. Many of the earlier comments on C, particularly its machine-oriented and terse nature also apply here.

The stated aim of C++ is to "enable larger programs to be structured in a rational way". It is acknowledged that clean interfaces are necessary to allow several programmers to work on a project and that C does not provide the necessary modularity.

To this end, C++ provides objects which are defined by classes. Each class has optional public and private parts which define variables and methods which may or may not be accessed outside of the definition. Hence the interface to an external user is the same as that presented to an inheriting child.

When inheriting, a child class may specify whether or not the public members of its parent class should also be public for the child. This is good for code sharing, but destroys the usual class, sub-class relationship because a child can no longer act as a parent. Where a name has been redefined, the scope operator allows access to previous definitions.

The binding is static by default, but functions may be specified as virtual to obtain dynamic binding. In this case, redefinitions must have the same type signature as the original and are considered also to be virtual.

Inheritance is from a single parent, although the *friend* mechanism allows named functions to have full access to an object. This hence allows efficient implementations of functions requiring much access to greater than one object.

Constructor and destructor functions may be provided as part of a class, and will be executed on the creation and destruction of an instance. Arguments are passed to the constructor of a parent in the definition of the child. Note that a parent constructor executes before the child's, and the destructor executes after the child's.

Default function arguments, and functions with an unknown number of arguments, are also supported as an aid to specialisation of function. Overloading of names and standard operators is also permitted, with the definition to be used determined by the type signature of the arguments. This can be aided by specification of explicit type conversions in addition to those defined by the language. There are no generic type features, although the use of C style macros is suggested.

A class may be declared as *static* in which case a single instance is created when the program is executed. A class attribute declared as static is created in a similar manner, and may be shared by all instances of the class.

Objects are otherwise created by the *new* function and destroyed by delete. There is no garbage collection in the system, this being considered an unacceptable overhead for the production of efficient programs. Facilities are available for direct user management of store to allow program knowledge to improve allocation efficiency still further.

Nesting of class definitions is permitted textually, but is confusing as the nested class is still placed in the global scope. The

scope of names, as in C, is controlled through the use of separate files and *include* or *extern* declarations. No checks are performed between files so the programmer must ensure consistency.

Modula-3 [CARD].

Modula-3 is one of several developments of Modula-2 which make use of the object paradigm - see also [BER2] and [LOVE]. It is not a superset of Modula-2 and indeed has some significant differences.

A major issue is the use of structural type checking opposed to the more strict and common name equivalence of Modula-2. Extensive use is made of the sub-type concept, where all the values of a sub-type are contained within the super-type.

Multi-dimensional open arrays are permitted but variable declarations of these types are not permitted - parameters and references to open arrays only. There are no variant records.

References may to be a fixed type, an object type, or any type and may be *traced* or *untraced* for garbage collection purposes. Note it is unsafe for an untraced reference to lead to an item containing a traced reference. Each traced reference type has a type number associated with the referent and so the type of the referent can be determined by the programmer when accessing that type through an untyped reference by using the *typecase* statement.

The *object* type is a reference to a data record and a *method suite* which both contain at least those entries declared by the type. Hence sub-types of an object may extend the definition whilst maintaining compatibility with the super-type. A method suite is a collection of procedure type signatures which effectively act as place holders for the methods.

The object type declaration may assign a previously declared

default procedure to the method. Methods may also be assigned at run-time, hence providing explicit control over the inheritance of methods. A procedure to be assigned to a method must have a first parameter of the object type (or some super-type thereof) and the remaining parameters matching the type signature for that method.

Hence method procedures are not explicitly declared with the object type and once a method is defined then its type signature is fixed. There is no hiding of data or methods from external or sub-typing use. The language therefore supports the type theory view of inheritance, but does not provide the encapsulation for other forms although the flexibility of programmer assignment of methods allows simulation of such forms.

Procedure definitions support default parameters and keyword parameter bindings. Exception handling is provided, with a procedure header including the exceptions that may be raised within that procedure. The exception handling statement includes a statement body, exceptions raised in which cause the execution of a handling statement. Users may declare and raise their own exceptions, which may also have optional arguments. Exceptions propagate down the call chain until a suitable handler is found.

A standard module is used to provide concurrent *threads*, along with a *lock* statement to help give mutual exclusion to shared resources. Unsafe features, such as untraced references, are confined to *unsafe* modules to limit their use and visibility.

Beta [KRIS].

Beta is based on the idea of *patterns* which may be instantiated to become objects. Patterns may define variables, procedures, concurrent systems etc, and can be organised into an inheritance hierarchy. Dynamic binding is optionally provided, but checking is done at compile-time.

A pattern has a similar form to a block, containing parameters, declarations and code. Declarations define patterns or create instances of patterns, and hence full nesting and block structure is included.

Code is specified by simple imperative statements including conditional selection and iteration. Imperatives may be labelled and the statements *leave* and *restart* within a labelled imperative allow the immediate exit and re-entry of the construct.

There are three kinds of object, used for concurrent execution, co-routines and sequential actions. The latter are called *items* and themselves have three sub-forms. Static items are traditional declarations; Inserted items are created and destroyed during execution like nested blocks; Dynamic items are generated by the *new* imperative which may also return a reference. Note that creating a pattern instance causes its main body to execute.

Hence if no reference is returned when creating a dynamic item it is like a conventional procedure call; if a reference is returned then the item exists as an independent entity within the system.

Inheritance is again from a single parent, and all attributes are visible. Execution of the code body begins with the parent, with the child code being executed whenever an *inner* imperative is found.

Virtual patterns may be declared with a default qualifying pattern. The virtual pattern may be bound to any sub-pattern of its

qualifying pattern, and if no binding occurs the qualifying pattern is used. Binding may occur to a fixed pattern or to another virtual pattern qualified by a sub-pattern of the qualifying pattern of the pattern to be bound. All bindings can thus be compile-time checked for legality.

All attributes of a pattern are available at external interface. Protection at the external interface is provided through the mechanism for modularisation of large programs, which allows the separate compilation of sentential units of the program.

Communication between concurrent, or *system*, objects is by the requestor executing a request imperative and the receiver executing an accept imperative. It is similar to the ADA rendez-vous. A receiver may accept communication from one object or a class of objects.

Vulcan [KAHN].

It is argued that whilst concurrent logic programming can support an object oriented programming style, it has not provided a suitable notation for expressing the object abstractions.

The conflicting requirements of object state and the declarative nature of logic programming are overcome by associating a communication channel with an object. When a Concurrent-Prolog process receives a message it reduces to other processes, one of which now represents the new object state. This recursion can be optimised and produces a perpetual process.

A process waiting for a message uses a read-only annotation to suspend waiting for that message. An object hence consists of a series of logic clauses, incorporating the necessary features to obtain the object nature. In a language such as Concurrent-Prolog this is verbose and error prone.

Vulcan overcomes this by providing a syntax to specify the instance variables of a class and associate methods with the class. A method body consists of a series of message sends to other objects, or to self. These statements are then translated to the Concurrent Prolog equivalents.

Vulcan also manages the actions necessary to allow an object to be shared between others. This requires a splitting of the communication channel into distinct messages, as a second message will not generally unify with an already instantiated message variable.

Multiple class inheritance is supported by expanding the definition of the sub-class to include the instance variables and methods of the parents. Hence there is code reuse at the Vulcan level but not in the resultant translation. Delegation is also supported by passing messages on using instance variables to hold references to suitable objects.

Communication strategies such as broadcasting can be supported, and look-ahead in the message stream can be performed. It is possible for objects to dynamically change their class, for example from rectangle to square.

The system is at an experimental stage and so features such as method hiding, choice of binding style etc. could be added.

Comparisons and Limitations.

It has been established that the object concept can provide good support for the ideas of software engineering. The general characteristics of object oriented programming have been described along with a number of current implementation languages.

We now compare these implementations against each other and against the general definitions. This will then give an idea as to the

benefits and problems of each, with emphasis again on the support of software engineering via objects.

Simula and C++ are essentially super-sets of common procedural languages, and so inherit their problems and benefits. In so far as the object implementation is concerned this is reflected in the inability to support parallel execution of independent objects. Although designed specifically as an object language, SmallTalk also has this limitation, as has Modula-3.

Support for the hiding of object attributes is limited and there is often little distinction between the external and inheritance interfaces. C++ is probably the most flexible in this respect with respect to individual objects, but is let down by the lack of checking between differing files of the program, a feature from C. Modula-3 does not encapsulate methods with the object definitions, allowing flexible but potentially confusing assignment of procedures to methods at run-time.

Again with the exception of Vulcan, the languages allow only single inheritance. Simula, C++ and Beta have the more efficient static binding as default, whereas SmallTalk and Modula-3 insist on dynamic binding. C++ is the only language to allow the hiding of a public attribute by a child, hence improving the security of code reuse inheritance, although note it destroys the usual class, sub-class relationship.

In terms of run time efficiency, C++ also scores with its machine oriented features from C and its user managed store allocation. These are the same features, however, which allow errors to enter the system. A major problem here is the lack of inter-file checking and absence of nesting facilities as required by the software engineering principles outlined earlier. Modula-3 can also be efficient, noting

that use of the garbage collector can be controlled, and is the only system to provide a system of exception handling.

SmallTalk and Beta both attempt to use a single underlying concept to define the whole system. In SmallTalk the use of objects forces an unnatural style in many simple cases, whereas the pattern concept of Beta is divided by storage class and then by concurrency potential. Along with the arguably unclear control flow, this can lead to confusion over the precise intention of the program.

In summary then, current object oriented languages have been found lacking in their support for concurrency, distinction between external and inherited interfaces, and in some cases control over the inheritance mechanism. The question of efficiency may be one of application, as there is an inevitable trade off between garbage collection and user managed store.

Primula - An Object Oriented Language.

Aims of *Primula*.

The potential of the object oriented approach to support software engineering has already been established and some problems of existing languages discussed. The principle aim of *Primula* is to support software engineering by a fully general implementation of the object oriented concept, offering some facilities noted as lacking in other languages.

In brief this requires full support for the notions of abstraction - classification, aggregation and generalisation - through the use of objects. As outlined in the discussion on software engineering this leads to a need for nested abstractions, hiding and modularity, and allows parallel execution. Current languages have generally failed to fully support these needs and take advantage of the object concept.

A further requirement is that the language should be suitable for a wide range of applications, including low level systems programming. Economics dictate that some reduction in code efficiency can be tolerated in return for increased programmer productivity, but lower level code is more sensitive to efficiency losses and these must therefore be minimised. Hence the language must allow the production of efficient code where required, and in particular not introduce overheads unavoidable by the programmer.

The Design of *Primula*.

Primula is to support the concept of objects as outlined in the previous chapter. This implies a number of features that must be provided within the language.

In support of abstraction, objects must

* belong to a class
* be capable of inheritance
* be parameterised.

The discussion on software engineering indicated that abstractions may be nested for maximum clarity in the model being constructed. This requires some form of block structure.

Objects are a natural form of modularisation, and inheritance allows the derivation of new modules from old. To provide maximum support for modularisation, objects must

* have user definable interfaces to public users and children. It follows that parts of an object will be private to that object.

* support the four styles of inheritance discussed earlier, ie. type theory, external interface, code sharing and polymorphism.

* be capable of being separately compiled and later linked to others, with full cross checking of semantics at link time.

An implementation of objects should also make use of their potential for parallel execution by the use of some synchronised communication protocol. Note this does not require parallel execution, but simply that the language semantics permit it.

Aspects of *Primula* are now discussed in more detail. Readers with a knowledge of Pascal will recognise many of the constructs, although often generalised in some manner. Pascal was chosen as a language known for its clarity of expression, wide user base and acknowledged

drawbacks which could be eliminated. *Primula* is not a superset of Pascal.

General.

This section outlines some of the basics of the language which are relevant to the other sections. To avoid the need for a highly theoretical approach to describing the language, appeal is made to common practice for such things as the use of white space to separate reserved words etc.

Comments are bracketed by << and >> symbols, or may alternatively begin with an exclamation mark, !, and terminate at the end of the line. They are not nesting.

Identifiers consist of a letter followed by optional letters, digits or underscore. Recognition of identifiers and reserved words is not case sensitive, as it is felt this leads to confusion. In the examples, however, reserved words are in upper case in order to clearly describe the program structure.

The use of commas is as a separator between items in a list, whereas the semi-colon is used as a terminator after an item.

Grammatically it was intended to keep *Primula* simple so as to allow common compiling techniques to be used and hence ease implementation. This has resulted in an LL(1) grammar, suitable for the recursive descent style of parsing.

Block Structure.

The need to nest abstractions has already been determined and block structure is a powerful, well known technique for achieving this. For further information on block structure, and its origins in Algol, consult [WIJN].

The major benefit of the scheme is the ability to restrict the availability of definitions intended only for local use. The otherwise massive global name space is reduced by the removal of identifiers which have little meaning in a global context and program reliability is thus improved.

In a block structured system, blocks may be defined within, ie. wholly enclosed by, other blocks. Each block has a lexical level associated with it, such that the lexical level of an enclosed, or nested, block is one greater than the immediately enclosing block.

Blocks may contain identifier declarations which are numbered independently of all other blocks; the number of an identifier within a block is called its lexical offset. Note that a given identifier may only be declared once within one block, although it may be redeclared with some new meaning in another, perhaps nested, block.

Where an identifier is used, the search for its declaration begins in the current block and moves through textually enclosing blocks until the declaration is found. If no declaration is found when all blocks have been checked, then the use of the identifier is illegal.

Note from this that each identifier currently in scope, ie. those found by the above search, can be uniquely identified by its lexical offset and the lexical level of the block in which it is declared. This forms the basis of the addressing scheme used in block structured languages such as *Primula*.

The principle objections to block structure have been that redefining a name in an inner block causes confusion; use of variables through the scope rules is unclear, and the textual structure as used in Pascal etc. separates the top of the block from the actual code at the bottom by a large number of inner blocks thus reducing

readability.

All these objections can be removed by syntactic or semantic changes without affecting the principles of block structure. There is good reason to consider such changes if blocks were the only method of structure in a language, but in *Primula* they are secondary to objects and so really only used for "programming in the small". Hence the familiar rules of the Algol family are kept.

Primula follows the Algol, rather than Pascal, approach of allowing any statement to be a block. It is felt that this offers a more flexible approach to block structure and locality at little cost. The Pascal combination of blocks and compound statements can be viewed as unnecessary and restrictive.

A block is delimited by **BEGIN** and **END** keywords and consists of declarations - constants, types, variables and procedures in any order - followed by statements. Expressions may occur in constant and type declarations and are evaluated once on each entry into the block in the order in which they appear.

If there are no declarations then the block looks like the compound statement of Pascal. If declarations are present, then the **EXECUTE** keyword must be used to introduce the statement part of the block.

Strong Types.

The merit of strong typing is that having given a particular interpretation to an identifier the compiler can ensure it is adhered to. Many errors can be detected in this way without the need to run and debug the program, and the compiler can provide run time checks where there is possibility of an error occurring.

Furthermore, an optimising compiler can use type information to

allocate space and reduce the amount of run time checks, multi-word length arithmetic etc. This is particularly the case where subranges have been specified.

By defining the meaning of the basic types provided by the language, program portability is increased as, for example, differing sizes of machine word length do not reduce the range of integers available. A well-written program will not lose out on efficiency from this technique as the use of subranges, where appropriate, will allow the compiler to eliminate extra work on smaller word length processors.

The fundamental types of *Primula* are :

Integer	:	A 32 bit signed value.
Real	:	A 32 bit value corresponding to the IEEE standard; 23 bit mantissa, 7 bit exponent, 1 sign bit [IEE1].
LongReal	:	A 64 bit value corresponding to the IEEE standard; 52 bit mantissa, 11 bit exponent, 1 sign bit [IEE1].
Char	:	An unsigned value representing the character set of the host machine. Generally 8 bits.
Boolean	:	The enumeration (false, true) which must be defined internally.[3]
Word	:	A machine defined, unsigned, quantity.

Note these definitions are the minimum requirement for the type. For example, an implementation may choose to increase the size of the Real type to suit the particular hardware, but it must provide at least the range and precision specified here.

The usual range of infix operators are available for these types and are discussed under 'Expressions'.

Automatic conversion of types to suit the situation is regarded as dangerous in that a type may be unexpectedly changed and an error

---

[3] eg. to allow checking the type of expression in a conditional statement.

go undetected. For textual convenience in situations where the meaning is clear, however, a small number of conversions are permitted. These are :

```
Integer --> Real --> LongReal.
Integer --> Word
```

Users may define two additional forms of scalar type :

(id_0, id_1, ... , id_n)
: The enumeration declares the identifiers which have the ordinal value of their place in the list.

type RANGE expr .. expr
: The base type must be a scalar type and the expressions be of that type. The resulting type may only take the values in the given range. The expressions are calculated on block entry and an error occurs if the upper bound is less than the lower. Other than for range checks, the type is taken as equal to the base type.

The definition of 'scalar type' includes all the types defined above. The definition of 'index type' is the scalar types without Real, LongReal and the subranges based on them.

Three structured types are provided. Whilst the record type can be viewed as a simple case of an object class it is included for cases where a class is inappropriate, perhaps for efficiency reasons. A record definition forms a block for scoping purposes. There is no variant form as these more complex and potentially type insecure structures are better handled by classes.

Array index types must, naturally, be index types as defined above. Bounds may therefore be dynamic in so far as they may be fixed at run time when subrange expressions of the index type are evaluated on entry to a block. Sets are intended to be represented using a single bit to represent the presence, or otherwise, of an element. This implies a finite ordering of the elements, and so only index types are permitted as base types. A more general set implementation can be programmed by the use of classes.

RECORD $f_0, f_1, \ldots, f_a : t_0$;
$g_0, g_1, \ldots, g_b : t_n$; END

ARRAY [$t_0, t_1, \ldots, t_n$] OF t

SET OF t

The problems associated with the use of pointers, eg. dangling references, have already been noted and it is expected that the structures that are normally implemented via pointers will instead use the more secure class mechanism. Hence there is no explicit pointer type, although an ADDRESS type is provided for low level programming. Its use is restricted to system type objects, as noted below.

With this exception, all addressing is done at a logical level and so there is no possibility of illegal or meaningless access to store. Hence in a compiler only system, there is no need for hardware protection between objects, where objects may be different users, providing of course that the use of system type objects is restricted.

Code types are provided to allow the procedure variables and parameters. The syntax is similar to a procedure or function header except that no names of local parameters are required. Typical examples are :

PROCEDURE (Integer; REF Complex;)

FUNCTION (LongReal;) : Integer

An issue when allowing procedure variables is avoiding the execution of a procedure, via a procedure variable, when the scope it expects is no longer available. This occurs when the block in which the procedure itself was defined exits.

Modula-2 adopts the limiting solution of only allowing global procedures to be assigned to procedure variables. A second option is to prevent the assignment of a procedure to a variable at an outer

scope level, but this is also unsatisfactory as it prevents global structures holding procedures whose execution may be legitimate.

Primula tends towards the Modula-2 approach but allows any procedure declared at the outer level of an object definition to be assigned to a procedure variable. Such a variable holds a reference to the instance from which the procedure was assigned in addition to information identifying the procedure itself. This ensures that the instance will not be deleted, hence removing the environment required by the procedure, and that the procedure call is made to the appropriate object.

Type checking occurs wherever an operator is used, on assignment and parameter passing. An error occurs when the actual type is not equivalent to the formal type (the formal type of an assignment is the type of its left-hand side). Note that for reference parameter passing the types must be equivalent in both directions.

Types are said to be equivalent if they were defined by the same definition. Therefore they must have the same name, or in the case of anonymous types have been defined in the same declaration, or have had the names equated by a type declaration of the form **type2 = type1**[4]. This is a similar scheme to that used in Pascal and is a variant on strict name equivalence.

A subrange type is defined to be equivalent to its base type and hence can generally take the place of the base type. Note though that a subrange value cannot be an actual parameter where the formal parameter is **REF** mode of the base type.

Name equivalence is chosen in preference to the weaker structural equivalence, which requires only that the types are similar in form

---

[4] This form of type equality is transitive, so if C = B and B = A then C = A.

and therefore may be differing actual declarations. It is felt that the scheme chosen provides more of the benefits of typing at little extra inconvenience to the programmer.

Some language designers have advocated a differing form of type equality which effectively creates a new type from an existing definition. The application of this, for example, is to prevent two integer quantities representing fundamentally differing ideas being assigned.

The drawback to this is in the assignment of implicitly typed constants and other values having a standard integer type such as the result of a 'max' function. In general this requires the heavy use of type casting functions which, in the authors opinion, produce more textual clutter than positive benefit. The complex types which generally at the heart of a program are usually independently defined anyway.

A **SEQUENCE** type was considered to allow the implementation of strings, sequential files and other one dimensional dynamic items. It was rejected, however, because complex items would be implemented via objects and simpler ideas by single dimensional arrays, perhaps with some arbitrary maximum as is commonly used in string processing.

A final proposal to allow recursive type definitions was excluded as such structures can be implemented using objects with far greater flexibility. The extra user code needed to implement the recursion by object references is small and can be hidden in a parent class for each type of structure.

Statements.

In keeping with the idea of any statement being a generalised block, the statement syntax looks more like Pascal than Modula-2 [WIR2], as the latter has an implicit statements-only block built into most statements.

Within a list, statements are terminated, rather than separated, by semi-colons in a manner consistent with the use of punctuation within *Primula*. Note though that the recursive use of statements within other statements does not require a semi-colon to terminate each individual statement - merely where a statement is part of a list.

Assignment, procedure call, IF and WHILE statements are identical to their Pascal equivalents. As an aid to clarity, actual reference parameters must be indicated as such and are checked by the compiler.

```
value_location := expression
proc_name( value_param, REF reference_param, ... )
IF boolean_type_expression THEN statement
IF boolean_type_expression THEN statement ELSE statement
```

The CASE statement is generalised in two ways. The selector expression may be of any type, not merely an index type, and secondly the case labels may be general expressions rather than constants. If the case is of an index type then a range of values may be specified using ...

On execution, the selector is evaluated and then each of the case labels in turn until a match is found. The statement with that label is then executed and the case statement ended. If no match is found, then if an OTHERWISE case is specified this is executed else a run time error is signalled.

```
CASE selector_expression OF
     == expression : statement;
     == expression : statement;
     OTHERWISE statement;
END
```

The Repeat statement of Pascal is replaced by a general loop construct similar in principle to that of Modula-2. The statement within the construct executes iteratively until an EXIT statement is found, where upon control returns to the statement following the loop.

Note that an exit for the loop may not be placed in any procedures that are called or defined within it. The simulation of a 'Repeat' style loop is :

```
LOOP BEGIN
     statement;
     statement; ... ;
     IF NOT Repeat_Until_Condition
          THEN EXIT;
END
```

Pascal's 'For' statement is also generalised, principally to allow differing sizes of step. For clearer reading, there is no 'DownTo' version of the statement, this being replaced by specifying a negative step. This results in a form similar to that of Modula-2.

The restriction of the control variable to be of index type, the inhibition of assignment to it within the loop and its undefined value after the loop are kept from Pascal.

Initial, final and step expressions are evaluated once on entry to the construct and the initial value assigned to the control variable. The loop body is executed whilst the following condition holds :

```
If step < 0 then initial >= control >= final
            else initial <= control <= final
```

The value of step is added to the control variable after each iteration. A general form of the loop is as follows :

```
FOR i := expression TO expression STEP expression
     DO statement
```

Finally, a Return statement is used to assigned the result value of a function. Unlike the Modula-2 statement, it does not cause execution of the function to terminate. Each function must have at least one Return statement, which may not be within procedures called or defined by the function. It is an error if a function completes with no value assigned.

```
RETURN expression
```

Expressions, Constants and Variables.

The syntax, and hence operators and operator priority, is taken directly from Pascal. Overloading of the standard operators has been extended to include the **LongReal** type where 'real's are provided by Pascal. Standard operators also act on Word and, in system objects, Address types. Word type arithmetic does not perform any overflow checking to allow direct use of the machine representation where required.

Extra bitwise operators, BitAnd &, BitOr !, and BitNot ~ are provided for operands of type Word. The logical boolean operators, ie. AND, OR, NOT, use the partial evaluation scheme of Modula-2 as opposed to the full evaluation of Pascal. This implies that evaluation halts when the final result of an expression can be determined, rather than at the end of the expression. For example :

```
A OR ( B AND C )
```

If A is true then B and C need not be evaluated. If A is false and B is false, then C need not be evaluated. In this way, code can be

more concise, in particular where Pascal must use nested If's to avoid Nil pointers and subrange bounds checks, and more efficient.

The following standard functions and procedures are provided for use with Index types, with meanings similar to those of Pascal and Modula-2.

Pred(x) :	Returns the next lower value to x within the index type. It is an error if x is the lowest.
Succ(x) :	Returns the next higher value to x within the index type. It is an error if x is the highest.
Ord(x) :	Returns the integer ordinal value of x within its type.
Low(x) :	Returns the lowest value allowed within the type of x.
High(x) :	Returns the highest value allowed within the type of x.
Cast(x, REF i) :	Assigns a value to i such that Ord(i) = x.

Low and High are particularly useful where generic types are being used (see below) and as boundary values for loops. Use in the latter manner avoids modifying code when extra elements are added to the end of enumeration or subrange types.

Cast is implemented as a procedure in order that the return type is well defined. If it were a function then, for example, the meaning of Low(Cast (x)) is indeterminate.

Variables also have the syntax and meaning of Pascal. Simple constants are extended, however, to allow specification of integers in binary and hexadecimal by prefixing them with % or $ respectively. The rare use of other bases made a fully general representation seem of little value when the extra clutter of expressing frequent cases is considered.

Primula also allows structure constants by prefixing them with # followed by the type name. For example :

```
#record_type RECORD f0; f1; ..., fn; END
#array_type [ e0, e1, ..., en ]
#set_type { e0, e1, ..., en }
```

Identifying the type of a set allows problems with set constants in Pascal to be overcome and is the same technique used in Modula-2. Expression ranges, using .., may be specified for set elements.

The parts of a structured constant are general expressions which are evaluated each time the structure is used. Therefore a constant declared at the top of a block can be guaranteed constant for that block as the expressions will be evaluated on block entry. A structured constant in the statement part of the block will have its expressions evaluated when that statement is executed and so the exact value is only determined upon use.

A revision not currently implemented suggests generalising the **Succ** and **Pred** functions to take an integer argument indicating the number of increments or decrements required. The **Cast** procedure, it is suggested, should become a function still taking a dummy type parameter to indicate the type of the value to be returned.

Procedures and Functions.

These are similar except for a function is called from within an expression and must return a value via a **RETURN** statement whereas a procedure is called as a statement and may not return a value via a **RETURN** statement. Unless otherwise specified, the term procedure applies to both functions and procedures.

Procedures are named, parameterised blocks. The name of the procedure is in the scope of the declaring block and the names of the parameters are local to the procedure block. The block is defined as starting at the opening bracket of the parameter list.

There are three parameters modes, of which value and reference will be familiar to Pascal users. Value is the default mode and

produces a local copy of the result of the actual parameter expression. Reference mode is specified by **REF** in both formal and actual parameter lists and passes a reference to the location of the actual parameter which must therefore be a value location in a similar manner to the left hand side of an assignment. Dereferencing is done automatically wherever the formal parameter name is used.

Reference can hence be used for variables which the procedure may like to change. Note that reference parameters to a function should only be used with care as functions are often thought of as pure and so not causing changes to global data. No language restrictions have been enforced, however, for generality with procedures.

The third parameter passing mode is **CONST**, which need only be specified in the formal parameter list as the semantics are identical to the caller. Like a declared constant, however, a constant parameter may not be assigned to and may not be passed as a reference parameter to any other procedure.

As the parameter may not now be changed it could be implemented by a reference to the actual parameter expression in some cases, thus saving on the copying of large structures. Care is required, however, as a concurrent object accessing the actual parameter could change the 'constant' value!

Functions may return values of any kind, including structures.

Default parameter values are not currently included, but are contained in a revised version to allow a more flexible form of inheritance - see below. The proposal is that parameters with default values must not be reference parameters and must occur at the right hand end of the parameter list.

A variable number of parameters, as allowed in C by use of '...' should not be allowed as the handling of such parameters is unclear

and not in keeping with the name and type principle of the remainder of the language.

The aspects of the language relating to Pascal have now been described. A couple of examples show the flavour of this side of *Primula*.

```
FUNCTION Factorial ( x : Integer; ) : Integer;
BEGIN
  IF x = 0
    THEN RETURN 1
    ELSE RETURN x * Factorial ( x - 1 );
END;

PROCEDURE Sieve( CONST Sieve_Size : Integer; );
<< Benchmark program - doesn't output results. >>
BEGIN
  VAR
    i, Prime, Multiple : Integer;
    Flags : ARRAY [ Integer RANGE 1 .. Sieve_Size ]
              OF Boolean;
<< Flags [ i ] remains set iff 2 * i + 3 is prime. >>
  EXECUTE
    FOR i := 1 TO Sieve_Size
      DO Flags [ i ] := true;
    FOR i := 1 TO Sieve_Size DO
      IF Flags [ i ] THEN BEGIN
        Prime := 2 * i + 3;
        FOR Multiple := ( Prime * Prime - 3 ) div 2
                      TO Sieve_Size STEP Prime
          DO Flags [ Multiple ] := false;
      END;
END;
```

Object Types.

We now examine the object nature of *Primula*, beginning with object definitions. It has already been established that an object is defined by its class, and so in some sense its class is its type. The idea of class is more extensive, however, because of the ideas of inheritance and sub-classes.

An object has a state and methods of using that state, so a class definition must describe the possible states and the methods of objects within the class. In *Primula* a class definition is a block

containing variable declarations for the state of the objects and procedure declarations for the methods. Instances of the block form each object.

The block may be parameterised, in the same manner as procedure blocks, to provide parameterised objects. As a new object may execute concurrently with its creator - see later - the use of reference parameters is dangerous as more than one executing object may be able to affect the data values.

Upon creation of a block, or object, the main code body of the block is executed. This can be used to guarantee the initialisation of the objects variables *before* any other actions are allowed.

When the main body has terminated, the object is available to receive messages from other objects requesting that a given method be executed. An object is treated as a critical region and so only one request (or the main body) may execute at any one time. Messages are queued until the object is able to deal with them.

When a request terminates, a message containing any results is sent to the caller, which may then proceed with its actions. The called object is now available for other requests which may already have been queued.

At language level, these messages take the form of remote procedure calls to the method procedures of other objects. This fits well with the style of many objects and can implement low level message passing send and receive primitives by creating an object to act as a message buffer or communication channel. An example of this is given below.

Note that remote procedure calls are synchronous and that although a caller is suspended waiting for the call to complete it is still effectively executing its method (or main body) and so is

unavailable to other requests.

In support of the implementation hiding discussed earlier in terms of general objects, all identifiers of an object are by default kept private. Procedure names may be made externally available by declaring them as **PUBLIC**, and hence these define the interface to the object.

To allow the writing of monitor type constructs [HOA2] and prevent deadlock situations, a **GUARD** clause may be specified for any methods which may be requested by another object. On receiving such a request, the guard expression is evaluated and if true the method proceeds. If it fails, however, then the request is queued and the object made available to any other requests.

Note that requests may not be serviced in the chronological order in which they are made from other objects. This is in any case a requirement for use in distributed systems where communication delays may alter the arrival times of requests to particular nodes. Also note that a guard expression for a waiting request is evaluated whenever another request completes and so may be executed an indeterminate number of times.

An example of this is in the message buffer used to provide low level message passing to other objects.

```
MsgBuff = CLASS BEGIN
            VAR
              Msg : MsgType;
              Sent : Boolean; ! Msg available

            PUBLIC PROCEDURE ASynch_Send( AMsg : MsgType; );
            BEGIN
              Msg := AMsg;
              Sent := true;
            END;

            PUBLIC FUNCTION Receive() : MsgType;
            GUARD Sent; ! make sure msg available
            BEGIN
              RETURN Msg;
              Sent := false;
            END;

            EXECUTE ! Main body initialisation
              Sent := false;
          END;
```

A further difference between ordinary types and class types is that declaring a variable of a class type does not actually create an instance of the type. Instead, it creates a reference of the appropriate type, which is initially given a Nil value.

Actual instances of a class are created by the New function, which obtains store for the object, passes the required parameters and commences its execution. The creator may continue asynchronously once execution has begun.

Assignment between variables of the class type actually copy the references, so allowing the sharing of an object between many users. As an object is an individual entity with its own state, including program counter etc., the precise nature of making an exact copy is unclear, although a user may of course define a copy procedure and create a new object into which to place the copy.

An executing object need have no references to it, ie. it may be anonymous. When such an object terminates, however, it cannot be accessed by any others, has no further use, and is therefore destroyed. Garbage collection of an object must occur only if it is in

the available (terminated) condition and there are no references remaining to it. The implementation of this is not defined by the language.

As class definitions are blocks they may be nested within other class definitions and hence instances of a nested class only have meaning in the context of an instance of the outer class. To prevent erroneous garbage collection of such an outer instance, nested instances contain references to the enclosing instances. Note that references to a nested object cannot occur outside of the main object.

Class definitions are also nested within ordinary blocks and so a similar constraint must be applied to ensure that the environment expected by an instance is always available. Hence a block containing a class definition will not exit whilst instances of that definition are active. Again, from the scope rules, a reference to such an object cannot occur out of this block.

Object Concurrency.

Objects may be thought of as independent state machines, with their own data and instruction sequence. They may synchronise and communicate by using the remote procedure call mechanism.

This method removes much of the burden of thinking about concurrency from the programmer. Each object is strictly sequential, with concurrency occurring naturally where more than one object is in use. In particular, true parallel performance is obtained when objects may execute independently of one another.

In scientific processing this occurs when vector or array operations are performed - each element may be a separate object performing the actions required in parallel with other elements. Graphical programming can represent each icon as an object, and

windowing systems can have each window as an object within which other objects representing the contents may be nested.

A typical window object might look like this, synchronising with others for input and output by shared devices, represented by objects, but internally allowing full concurrency with other windows.

```
Window  = CLASS ( CONST Window_Id : Id_Type; );
          BEGIN
            VAR
              Window_Info : Window_Record;
              Command_Msg : Msg_Type;
            EXECUTE
              LOOP BEGIN
                Command_Msg := Input_Device^.
                  Wait_Command( Window_Id );
                << Process command. >>
                Output_Device^.Do_Redraw(
                  Window_Id, Window_Info.Redraw );
              END;
          END;
```

Object Inheritance.

In support of the four types of inheritance outlined earlier - type theory, external interface, code sharing and polymorphism - *Primula* provides two principle mechanisms, outlined below. An important part of these is to reduce or remove the need for a parent to be designed and implemented with specific forms of inheritance in mind. This then increases the ability to use that parent, via the various forms of inheritance, in situations which were not foreseen when the parent was produced.

Following the discussion on multiple inheritance, and in particular the problems of isolating the implementation of an object from its children, *Primula*, provides inheritance from only a single parent. This also has the side effect of allowing a more efficient implementation of access to variables and methods.

The first mechanism is called REFINEMENT, reflecting the most common use of the facility which is to specialise, or refine, an

object type by adding or modifying the facilities provided by the object. This is the essence of the polymorphic form of inheritance and also provide the Type Theory and External Interface forms.

The basic syntax is :

```
child_class = REFINEMENT OF parent_class
```

Dynamic binding of methods to accesses is used throughout, following the earlier reasoning that a specialisation is a specialisation regardless of whether it is accessed as a more general object.

This also provides some support for external interface inheritance, as by redefining every public method of the parent in the child, the parent methods will never be accessed. The support is not complete, however, as instances of the child will include the parent variables which are redundant in pure external interface inheritance. Defining a class which sets out the public methods but provides no data or real implementation can be used to overcome this. Children of the class then contain just their own data and methods. Methods in the defining parent class should return errors if called, indicating that the interface has not been fully implemented by the child.

In correct support of the class concept, objects of a descendant class may be assigned to references to an ancestor class. Run time checks are provided where an ancestor reference is assigned to a child reference.

This requires, however, that the public interface of a child contains at least the interface of its parent. Otherwise assigning a child object to a parent reference could cause the child to receive an access which it does not understand.

Therefore it is necessary that a child class :

* Inherits the full public interface of its parent.
* May not hide any of the public interface of its parent.
* May not redefine the type signature of any part of the public interface.

As with the external interface, object attributes are private by default and so not visible to an inheriting child object. Attributes may be made visible by prefixing their declaration with the **BEQUEATH** keyword, which then allows a child to use, but not redefine these attributes. Public attributes are automatically bequeathed to a child in order to maintain public interface compatibility.

Procedures may be declared with the **DELEGATE** keyword, with or without being public, allowing a child to access *and* redefine that procedure. From the above though, a redefinition of a public method must have the same type signature to maintain sub-class compatibility, and this is checked by the compiler.

The extension of parameter definitions to allow default values on the right hand end of the list would allow this rule to be relaxed to say that the type signature of a redefinition must be the same as the original or extend it using parameters with default values. Hence when a child instance is called from a parent reference, the default values are substituted in the extra formal parameters.

The original definition may be called as a procedure from a descendant by the syntax

```
PARENT :: procedure_name ( parameters )
```

A child redefinition may alter the availability of a name, noting that, as above, public methods must remain public for sub-class compatibility.

The second inheritance mechanism is introduced by the keyword **USES** and provides support for code sharing and re-use. This form of inheritance cannot be suitably implemented by refinement as it requires that some methods are hidden from users and may wish to redefine other methods with a different type signature. These needs contradict the rules of refinement and dynamic binding as outlined above.

Cases where this is applicable are the derivation of a simple queue from a priority queue or the derivation of a queue from a general linked list structure. Using refinement, methods of the parent class must be bequeathed to the child but not made public. A child can then modify and make public those methods it wishes. This requires that the parent is explicitly created with this form of inheritance in mind, which goes against the idea of abstraction and code re-use. It also prevents code sharing from the child, as it has made methods public and so fixed the type signature.

USES inheritance creates a child which is treated as a new type rather than as a sub-class. Attributes are inherited as normal, but not made public - the child must specify the entire public interface itself. Assignment to a parent type reference is not permitted as the child is not acting as a sub-class.

As the child is now released from the constraint of the public interface of its parent it can define an entirely new interface, whilst using the facilities of the parent to implement this interface in an encapsulated, modular manner. Additionally, the parent need not have been explicitly designed for re-use of code, which therefore widens the scope for this activity.

The following applies to both forms of inheritance. Names are inherited into the main block of a child definition, and are

effectively declared immediately after the BEGIN keyword. Object instances are created by the New function, which passes parameters to the instance and returns a reference to that instance. A child may pass actual parameters to its parent with the syntax below. Note that when initialising, parent code always executes prior to the child's code.

```
REFINEMENT OF parent_class ( child_formal_parameters )
                        :: ( parent_actual_parameters );
```

Code execution begins when parameters are passed to the oldest ancestor. The INNER statement causes a procedure call type transfer of control to the child of the current block, unless no child exists in which case it is a null statement.

All objects have the 'Basic_Object' as an implicit, oldest ancestor and hence a 'Basic_Object' reference may hold a reference to any object instance. The class is used to set the basic characteristics of an object.

A form of static binding can be provided by a parent by not allowing the delegation, hence redefinition, of a method. All descendants will then be forced to use this definition, regardless of exact descendency.

A proposal to improve the support of external interface is to provide a third inheritance form. The action would be to not directly inherit any attributes from the parent, but to check at compile time that the full public interface of the parent is provided by the child. Dynamic binding and the sub-class facility would be unchanged from the standard refinement.

This has not been included as it is felt that there is to much overlap with the refinement concept and that refinement is an adequate

technique considering the frequency with which true external interface inheritance is required.

Examples of the various forms of inheritance are based on a general list structure being inherited for a number of different reasons, illustrating the inheritance forms.

```
Node = CLASS BEGIN
       << Inherit from this class to include data into
          structure by type theory inheritance. >>
         VAR
           Last_Node, Next_Node : Node; ! private data

         PUBLIC FUNCTION Last() : Node;
         BEGIN
           RETURN Last_Node;
         END;

         PUBLIC FUNCTION Next() : Node;
         BEGIN
           RETURN Next_Node;
         END;

         PUBLIC PROCEDURE Set_Last( A_Node : Node; );
         BEGIN
           Last_Node := A_Node;
         END;

         PUBLIC PROCEDURE Set_Next( A_Node : Node; );
         BEGIN
           Next_Node := A_Node;
         END;

         EXECUTE
           Last_Node := Nil; ! Initialise
           Next_Node := Nil;
           INNER;            ! Initialise child
       END;
```

```
List = CLASS BEGIN
       << Main structure referred to by users. >>
         VAR
           BEQUEATH First_Node, End_Node : Node;
           ! Allow children to use variables.

         PUBLIC PROCEDURE Insert_Back( A_Node : Node; );
         BEGIN
           A_Node.Set_Last( End_Node );
           A_Node.Set_Next( Nil );          ! for safety.
           IF End_Node = Nil
             THEN First_Node := A_Node      ! Empty list.
             ELSE End_Node.Set_Next( A_Node );
           End_Node := A_Node;
         END;

         PUBLIC PROCEDURE Insert_Front( A_Node : Node; );
         BEGIN
           A_Node.Set_Last( Nil );
           A_Node.Set_Next( First_Node );
           IF First_Node = Nil              ! Empty list.
             THEN End_Node := A_Node
             ELSE First_Node.Set_Last( A_Node );
           First_Node := A_Node;
         END;

         PUBLIC FUNCTION Get_Back() : Node;
         GUARD End_Node <> Nil;
         BEGIN
           RETURN End_Node;
           End_Node := End_Node.Get_Last();
           End_Node.Set_Next( Nil );
         END;

         PUBLIC FUNCTION Get_Front() : Node;
         GUARD First_Node <> Nil;
         BEGIN
           RETURN First_Node;
           First_Node := First_Node.Get_Next();
           First_Node.Set_Last( Nil );
         END;

         EXECUTE
           First_Node := Nil;
           End_Node := Nil;
           INNER;
       END;
```

```
Priority_Node = REFINEMENT OF Node( Priority : Integer; );
<< Inherits node properties. >>
       BEGIN
         PUBLIC FUNCTION Get_Priority() : Integer;
         BEGIN
           RETURN Priority;
         END;

         EXECUTE
           INNER;
       END;

Priority_List = REFINEMENT OF List;
<< Has all facilities of a list, plus priority ordering. >>
       BEGIN
         PUBLIC PROCEDURE Priority_Insert(
                             A_Node : Priority_Node );
         BEGIN
           IF First_Node = Nil
             THEN Insert_Front( A_Node )
             ELSE BEGIN
               VAR
                 This_Node : Node;
               EXECUTE
                 This_Node := First_Node;
                 WHILE ( This_Node <> Nil ) AND
                       ( This_Node.Get_Priority() >=
                           A_Node.Get_Priority )
                   DO This_Node := This_Node.Get_Next;
                 IF This_Node = Nil
                   THEN Insert_Back( A_Node )
                 ELSE BEGIN
                   A_Node.Set_Last( This_Node.Get_Last() );
                   A_Node.Set_Next( This_Node );
                   This_Node.Get_Last().Set_Next( A_Node );
                   This_Node.Set_Last( A_Node );
                 END;
             END;
         END;

         EXECUTE
           INNER;
       END;
```

```
Stack = USES List;
        << Only requires access to front of list. Reuse of
           list and node classes. >>
        BEGIN
          PUBLIC PROCEDURE Push( A_Node : Node; );
          BEGIN
            Insert_Front( A_Node );
          END;

          PUBLIC FUNCTION Pop() : Node;
          GUARD First_Node <> Nil;
          BEGIN
            RETURN Get_Front;
          END;

          PUBLIC FUNCTION Is_Empty() : Boolean;
          BEGIN
            RETURN First_Node = Nil;
          END;

          EXECUTE
            INNER;
        END;
```

Generic Types.

Generic types are used in functional languages such as HOPE [BURS] to provide the ability to write higher order functions, whilst maintaining the strong type checking. In an object based language, it is useful to be able to define the actions of a general data structure without specifying the actual data type to be stored.

Languages such as Modula-2 support this idea to a limited extent by the use of *opaque* pointers. The data structure operates with pointers without knowing details of the 'pointed to' structure - hence any pointer type can be used without change or recompilation.

Object oriented languages can offer general data structures by use of the inheritance mechanism. By inheriting the structure class into each of the data classes, the data classes can then be a part of the structure. This only provides weak typing, however, as differing children of the data structure could be a part of the same structure. In addition, the inheritance tree becomes very unclear and non-intuitive when a data type wishes to be allowed into more than a

single structure.

Primula provides a generic type facility which allows the production of parameterised data structures which can later be instantiated with the appropriate data type. Strong type checking is present as only that type is allowed into the structure, and there is no complication of the inheritance tree.

Type declarations may contain formal type parameters whose scope is that type declaration. Variables may not be declared as of a parameterised type, as the type description is not complete and therefore the meaning of the variable unknown.

As a class definition does not declare a variable, however, it may be parameterised, and hence any code within the definition will be generic. Variables may be declared of the generic type as it is not itself parameterised.

Like other variables, however, a object reference may not be to a generic type or class. Hence the class must be instantiated with actual types before object instances or references can be created.

As nothing is known about a generic type at compile time the only operation available on it is copying (by assignment or parameter passing) to the same formal type. This is sufficient to produce strongly typed generalised data structures.

A particular case of generic type occurs when the formal parameter is declared as of **INDEX** class. In this case the actual parameter must be an index class type and the formal type may have the usual index type functions listed above, eg. **Succ**, **Pred**, applied to it.

```
List( alpha ) = CLASS ( Init_Value : alpha );
                BEGIN
                  VAR
                    This_Node : alpha;
                    Next_Node : List( alpha );

                  PUBLIC FUNCTION Node : alpha;
                  BEGIN
                    RETURN This_Node;
                  END;

                  PUBLIC FUNCTION Next : List( alpha );
                  BEGIN
                    RETURN Next_Node;
                  END;

                  EXECUTE
                    This_Node := Init_Value;
                    Next_Node := Nil;
                END;
```

Exception Handling.

The handling of run time errors has been neglected in many high level languages, often being left to individual implementors to devise a system of handling I/O and other errors occurring as a result of calling facilities of the underlying system. Typically such checks require the programmer to explicitly test return codes - tests which are often omitted.

Language defined errors occur where some run time check fails such as value out of subrange bounds or no matching selector in a case statement. User errors are explicitly detected and signalled by the user code.

Primula provides an enumerated error type containing all the language defined errors which can occur, plus an entry for user error. The 'Basic_Object' type contains a variable of this type which is bequeathed into every object. The declarations are as follows :

```
TYPE Error_Type = ( No_Error, Range_Error, Case_Error,
                    Return_Error, Overflow_Error,
                    Class_Error, Access_Error, Object_Error,
                    User_Error );
VAR  Error_Var  : Error_Type;
```

The error variable is initialised to No_Error and then set when an error occurs. An object can force this by assigning to the variable. Whenever the variable is set, outside of an error handler, a jump is then performed to the 'nearest' error handler.

'Nearest' is defined as being the error handler found by checking each block on the stack for an error handler, beginning with the most recent. If a handler is found it is executed, otherwise the block is removed from the stack and the process repeated.

Any block may optionally contain an error handler to deal with errors caused within, or propagated to, the block. The error handler can test the error variable and take whatever action the programmer deems necessary.

On exit from the handler the variable is tested again, and if set to No_Error then the block exits as normal and processing continues. If some other value is present, then it is propagated to the 'nearest' other error handler.

Where an error is propagated past a public access to an object, the error variable of the caller is set to an Access_Error and propagation continues. An object which allows an error to be propagated out in this way has its error variable set to Object_Error. Subsequent calls to the object will immediately raise the Object_Error within the called procedure and enter the error handler.

It is the programmers responsibility to maintain data to allow multiple user errors to be distinguished. The use of user errors can avoid highly nested IF..THEN..ELSE constructs which occur where many error conditions are being tested for and where use of the 'goto' may

have been considered in languages such as Modula-2.

```
FUNCTION String_Length( String : String_Array ) : Integer;
BEGIN
  VAR
    Count : Integer;

  EXECUTE
    Count := 0;
    WHILE ord( String [ Count ] ) <> 0
      DO Count := Count + 1;
    RETURN Count;

  ON ERROR DO
    IF Error_Var = Range_Error THEN BEGIN
      Output.Text( " No terminator on string. " );
      Error_Var := No_Error;
    END;
  << Outputs message but does not return a result from the
     function. Hence a 'No_Return' error will be raised in
     the caller when leaving by  this route. >>
END;
```

System Facilities.

Primula provides as standard most of the facilities to write low level systems code. In particular, the **Word** type allows bitwise operators and unchecked arithmetic. Compilers should support subranges of the **Word** type to define bytes and may also provide long word types, dependent on the underlying machine.

The **Address** type may only be used in an object prefixed by the **SYSTEM** keyword. Within such objects, integers may be type coerced to addresses and addition and subtraction are defined between addresses and integers. Subtraction of two addresses yields an integer result.

Address variables are dereferenced using the ^ operator and may be declared as an address of a given type, in which case dereferencing the address yields an item of that type. To be more precise it produces a store address the data at which will be treated as of the addressed type. Address variables may be freely assigned, unlike Pascal pointers, to provide flexibility in these system objects.

To aid the use of addresses, the **Addr** and **Size** functions are

provided to return the address and size of their single REF class parameter.

Store allocation and return is the responsibility of the programmer, but may use standard calls to the system store manager and garbage collector to obtain and return store. The use of standard calls should also allow programmers to implement their own store management and garbage collection policies, replacement the implementation provided method.

Separate Compilation.

As a class definition is essentially an encapsulated and reusable unit, it is the natural unit of compilation. There are problems, however, in checking the consistency of classes to be linked together to form a program and in the scope rules of block structure.

These problems are not unique to *Primula* and are dealt with in a number of ways by existing languages. C relies on the programmer to maintain header files to be textually included into the compilation of files using the facilities of others, and does not provide block structure. Modula-2 restricts separate compilation to modules at the global level and uses separate definition and implementation parts.

A possible method is to define an 'External' block which specifies the minimum set of names and types that the definition will expect to find in its environment. This is a generalised form of the Modula-2 'Import'.

Separately compiled objects may be introduced with the keyword 'Separate' and then specifying the required interfaces, public and / or inherited. The linker can check that the object provides this interface and that the user provides the required environment.

This is not a very satisfactory solution as it requires much

extra textual specification of external items which whilst acceptable for large objects may tend to 'drown' simpler ones. It also requires a considerable amount of work by the linker, where reporting errors to the programmer is difficult to relate to any source text.

The solution proposed here, although not currently implemented, is to adopt an object workbench / library approach. All objects required by the object to be compiled must be already compiled, at least sufficiently specified to fix the interfaces.

When the compiler encounters the use of an externally defined object type it can search the library to find the details of its interface and load them into its data structures, thus allowing full checking. Information can be placed in the output file to allow a relatively straightforward link to occur.

So far, this method has assumed that block structure is limited to within a separately compilable unit, in a similar manner to Modula-2. Block structure can be accommodated by a second pass at the compilation process, this time starting from the top level.

This must be performed before the bottom-up method described above comes into play. The first part of compilation simply establishes the lexical relationship of nested blocks and builds a sub-directory of classes required by the class being compiled. The information is simply the lexical level that the nested class is to be compiled at, as all other checks can be made when the main compilation pass is performed.

Limitations and Improvements.

An assessment of *Primula* must compare it against the theoretical objective of software engineering support and against existing languages in terms of expressive power.

Some suggestions for improvements have been included above and should be considered for inclusion in any following versions of the language. For a variety of reasons, they have not been included in the current compiler.

General.

Primula has a similar general appearance to most imperative languages. One simple improvement, not found in most languages, would be the provision of nested comments to allow chunks of code, themselves containing comments, to be taken out for debug purposes.

The restriction of the grammar to be LL(1) is again a common feature to many other languages, although it clearly prevents a more free and expressive notation.

Block Structure.

If the merits of block structure are accepted then *Primula* provides a general Algol style implementation. The extra clutter where declarations appear in in-line blocks must be traded with the less flexible approach of the Pascal family which restrict declarations to named blocks.

There may be merit in adopting this type of policy in which case it is sensible to follow the Modula-2 route and provide the statement list as the basic non-terminal occurring within other statements, eg.

IF <expression> THEN <statement_list> END

Types.

It can be argued that the strong type checking causes a programmer to work to get around it in some cases. With the exception of some low level code, however, it is felt that this is the result of static types and would not be necessary with the use of objects.

The **Address** type is provided for low level programming and allows unchecked use of addresses and dereferencing. Its use is potentially dangerous and therefore should be restricted, but it does provide some flexibility to deal with store and type conversion issues - compare with 'C'.

Specifying the basic types in terms of minimum range and accuracy may produce inefficiencies on unsuited hardware, but the compiler should check for subranges to allow use of a smaller representation range. This also overcomes the lack of an explicit **Byte** type.

The structures are familiar to many languages. Although use of arrays is strongly checked to avoid out of bounds accesses, a good compiler should be able to remove many of these - most occur within loops where the possible values of the index are explicitly stated by the programmer. A similar argument applies to multiple evaluations of the index expression.

The restriction on set base types to be index types is also common and stems from the 'bitset' implementation. Any attempt to extend it runs into severe implementation difficulties and the possibility that changing the base type would completely alter the performance characteristics of the type, which is not acceptable.

Code types are implemented in as general a manner as possible without excessive overhead or reduction in security. However, there is not the complete ability to call at any time a procedure which does not need anything from its environment. Whilst this could be overcome,

the complexity of discovering where a procedure can be executed are considered too great for the benefits.

Statements.

The generalised **CASE** statement does not allow the jump table implementation as often used in Pascal without incurring compiler complexity in detecting the special case where the labels are constants.

It is arguable that the **REPEAT** statement should have been retained from Pascal and Modula-2, as its exclusion was only on grounds of relatively infrequent use and ability to achieve similar results using the generalised **LOOP**.

Other than this, *Primula* has followed the general form of clear and simple statements adopted by Pascal.

Expressions, Constants and Variables.

The expression syntax could have been modified to remove the parenthesis required in cases such as :

(a < b) AND (b < c)

Allowing functions to return any type is a major strong point in improving generality and clarity of expression. A further small improvement could be made if, for example, an **Address** returning function could be called and dereferenced on the left hand side of an assign statement to allow assignment to the referenced location.

A good compiler should include the ability to detect fully constant expressions at compile time and fix the value to the benefit of run-time performance. This will involve the evaluation of general expressions, although it would be reasonable to exclude function calls

even though they may not depend on any other values.

The in-built functions are a small, but essential set. Changes to **Succ**, **Pred**, and **Cast** as indicated previously would permit wider use of such functions.

Procedures.

It is a run-time task to check that the return value of a function has been assigned, although some cases may be optimised out by flow analysis of the function code.

The use of reference parameters to avoid copying of structures as done in Pascal is made possible by the **CONST** mode, and hindered by problems with concurrently executing objects accessing the pointed to structure. Some technique to reduce structure copying, other than by the use of **REF** could bring useful performance and security benefits.

Allowing default parameters as described above would be of value to the inheritance mechanism. Specifying that a procedure should be executed in-line, or at least providing some hint to the compiler, could give a useful performance boost.

Object Types.

The disguise of message passing by the remote procedure call mechanism is not always appropriate. Although synchronous message passing can be provided by the mechanism, the implementation is not 'clean' due to the need to keep either sender or receiver waiting on guard conditions.

```
SynchMsgBuff = CLASS BEGIN
                 VAR
                   M : Message;
                   Full, Receive_Waiting : Boolean;

                 FUNCTION Set_Receive() : Boolean;
<< Set flag to allow sender to proceed and deposit Msg. >>
                 BEGIN
                   Receive_Waiting := true;
                   RETURN true;
                 END;

                 PUBLIC PROCEDURE Send( A_Msg : Message; );
                 GUARD Receive_Waiting AND Not Full;
<< Proceed when receiver has arrived and evaluated its
   guard. Not Full ensures against multiple senders. >>
                 BEGIN
                   M := A_Msg;
                   Full := true;
<< Set flag to allow receiver to pick up message. >>
                 END;

                 PUBLIC FUNCTION Receive() : Message;
                 GUARD Set_Receive() AND Full;
<< Set flag to allow sender to proceed then proceed here
   when sender has finished. >>
                 BEGIN
                   Full := false; ! reset
                   Receive_Waiting := false;
                   RETURN M;
                 END;

                 EXECUTE
                   Full := false;
                   Receive_Waiting := false;
               END;
```

There is also the issue of evaluating guards in public procedures called from within the object, typically by a child. The child should not have to rewrite the guard condition and if the parent does not bequeath the data it may not be able to.

Conversely, if the child executes code and then calls the parent, whereupon the guard is evaluated, it must be aware that all code before that call may be executed an indeterminate number of times. Alternative semantics for GUARD could require that execution restarts at the guard condition, overcoming this problem.

Although the main code body of an object may be regarded as its initialisation, there is no corresponding termination code. This

imbalance prevents an object taking actions such as writing out its final state or performing a clean-up in a user provided store allocation system.

Garbage collection is a complex task and depending on the method used may impose important performance penalties. The difficulty in allowing user store allocation is compounded by the lack of termination code. Nested objects cause further difficulties and may cause unexpected effects.

Consider a procedure which declares a class type and creates an instance of that class. The instance requires the environment of the procedure in which to execute and so although the code may have been executed, the procedure cannot exit to its caller until the nested instance has completed.

Object Concurrency.

The central issue in concurrency is how best to exploit a number of processors. *Primula* provides no facilities for the programmer to indicate the assignment of objects to processors and assumes a dynamic allocation scheme. For applications such as vector processing, better performance is likely to be obtained by fixing objects to processors by a similar method to that used in Occam for Transputer arrays [INMO].

Object Inheritance.

The refinement method of inheritance and its associated dynamic binding imposes the severe restriction that a child must mimic its parent's public interface in terms of type signature. Although default procedure parameters help to a large extent in this respect, it still does not give the generality of name overloading as presented in C++.

Overloading allows a choice of methods of the same name by comparing type signatures. Hence a child could inherit the parents method and have it accessed wherever that form of type signature was used, eg. from a parent type reference, but also define its own with a different type signature.

Objections to overloading contend that it causes confusion when the same name is used to denote differing operations and that particularly in conjunction with automatic type conversion rules an unexpected definition may be used.

There is no provision for static binding in *Primula*. Such bindings can be more efficient and remove the possibility of a child changing the behaviour of a parent such that a sequence of actions carried out on behalf of some user produces an unexpected effect.

USES type inheritance does not maintain the sub-classing hierarchy of conventional inheritance, although it does inherit attributes from the parent. This again is due to the dynamic binding mechanism requiring that a sub-class provides the full external interface of the parent.

Although external interface inheritance can be performed using the REFINEMENT mechanism, it is not enforced by the compiler. In addition, attributes of the parent are carried over, although perhaps unused, into the child. Hence the pure form can only occur where the parent has been specifically written with this inheritance form in mind. This restricts the generality of the parent and reduces opportunities to fully perform this style of inheritance from a general class.

Generic Types.

The applicability of generic types is limited to the parameterisation of object types. It would be useful if they were extended to procedure parameters, for example allowing a function to return the middle element of an array independently of the element type of that array.

Index generic types are really substitutes for subranges and are principally used where open arrays are found in Modula-2. A more powerful notion of dynamic arrays would remove the need for this form.

Exception Handling.

Unlike other recent languages that feature exception handling, *Primula* does not allow the declaration of exceptions. Instead there is a single user exception and the programmer must maintain the necessary data to discover the reason for the exception.

Exceptions are raised and then the system informed they have been fully handled by assignment to the error variable. In the context of raising an exception, and therefore changing the flow of control, this is not clear or sufficiently explicit.

The propagation of exceptions is along the dynamic chain of calls. Whilst this suits the block structure, it would be useful to have an error handler associated with an object which could attempt to handle exceptions within that object.

In addition there is currently no provision for the explicit handling of external interrupts, although a low level handler could disguise them as remote access procedure calls.

System Facilities.

The **Address** type is the sole extension to the language for low level programming. It provides the necessary generality, but some restriction such as a type cast function would help identify errors caused by inter-assignment of differently typed address variables.

Separate Compilation.

Little can be said about this at the moment as the current implementation supports only rudimentary separate compilation.

Execution of *Primula*.

This chapter is concerned with the needs and characteristics of a *Primula* program at run-time. It examines the mechanisms needed to implement language features, at the logical level, bearing in mind that the availability of a given feature should not penalise the execution of a program that does not use the feature.

To evaluate the performance of an architecture executing *Primula* it is necessary to establish a typical workload. This is done by developing an estimated profile of the static and dynamic (run-time) frequencies of various *Primula* constructs. As few programs have been written in *Primula*, the basic data of the profile is derived from surveys of Pascal and modified to compensate for the differences between the languages.

This information is then used to briefly examine the likely performance and problems of executing *Primula* programs on the common architectures described in the first chapter.

Although aimed primarily at *Primula* the ideas developed will be applicable to other languages which share particular features.

Features of *Primula* Execution.

This section forms the outline of a general implementation model. Language characteristics are examined and the run-time support described.

Block Structure.

The principles of block structure were outlined in the previous chapter and it has been established that each block has a lexical level, each identifier has a lexical offset within the block of its

declaration, and that at any time an identifier in scope can be uniquely addressed by its lexical level and offset, known here as an address couple.

Block structured languages are based on the idea of a stack [WIJN]. On entry to each block, a frame is placed on the stack containing, in general, the values of all identifiers declared in the block. When the block completes, this frame is taken off the stack and the top frame is now that of the block which called this block.

To implement this, a *dynamic* pointer is stored at a known location within each stack frame, pointing to the previous frame. It is set on entry to a block and used to determine the address of the previous frame on exit.

The problem is then to use the address couple to access the correct value. This involves run-time computation as recursion and calls to outer block procedures prevent the compiler knowing the location of the frame of a given block - there may be many instances of a given block within the stack only one of which is accessible by an address couple.

Hence the lexical level of the address couple must be mapped to the correct stack frame. There are two established techniques for this.

* Each stack frame contains, at a known lexical offset, a pointer to the stack frame at the next lower lexical level. This is the *static* pointer. When a block is entered it is set to point to the frame in which that block is declared. Each block knows its own lexical level and so may follow the appropriate number of pointers to find the frame at the required level. A fixed location, typically a register, must be used to hold a pointer to the current frame.

* An array of pointers to frames, the *display*, is indexed by lexical level. On block entry, the entry at the level of that block is saved at a known offset within the frame and a pointer to the current frame stored in that display entry. When a block is completed, the old value is restored to the display. Access is then simply a matter of indexing the display by the required lexical level.

Having found the stack frame an item must be selected from it. If all items within a frame are of fixed size then the compiler can calculate the lexical offset of an identifier in terms of store address units leading to simple base and displacement (frame and offset) addressing as found in the basic addressing modes of many processors.

If the size of items may vary at run-time, however, the compiler cannot perform such a calculation and a more complex technique, similar to that for stack frames is required.

In *Primula* most items are of a known size with merely dynamic arrays and generic types being run time dependent. In accordance with the principle stated above, access to fixed sized items ought not to be penalised by the presence of such things.

The technique, explained in more detail later, is to divide the stack frame into fixed and dynamic parts. Structures whose size can only be determined at run-time are stored in the dynamic part and address through a fixed size reference, often called a dope vector, held within the fixed part. Hence the efficiency of fixed sizes and offsets is maintained, whilst the dynamic items require an extra pointer dereference.

Types.

Implementation of the basic types is highly machine dependent. In particular, **Real** and **LongReal** operations may be implemented in hardware, via extra-codes or by software. The chosen method must at least match the IEEE requirements for range and precision.

For optimisation of store use, and arithmetic speed on many machines, subranges should be used to reduce the size of basic types, particularly of **Integer** and **Word**. This is only practical where the maximum subrange bounds can be determined at compile time, such as when specified by simple constant expressions. Checking of bounds on assignment (taken to include parameter passing) and indexing can be similarly optimised.

Using the idea of fixed and dynamic parts of the stack frame requires that on block entry, the stack pointer is set to the top of the fixed part. Each dynamic structure in turn begins at the next location on the stack, setting its own pointer in the fixed part and updating the stack pointer according to the structure size to ensure it refers to the top of the allocated stack. Note *Primula* fixes structure size on entry to a block and does not allow it to change within the block.

On ending a block, the stack pointer must be reset to the value it had when the block was entered. This is the same value as the frame pointer of the called block which must therefore be stored into the stack pointer.

Checking the bounds of access to a dynamic structure requires that the expression values used in its creation must be stored within the stack. As the number and type of bounds expressions is determinable at compile time they may be stored in the fixed part of the frame along with the pointer.

Note the technique for procedure types as outlined in the language description. The necessary information can be kept as a record structure within the fixed part.

Expressions, Constants and Variables.

As the use of block structure implies an execution model containing the stack concept, expression evaluation is also based on this model. In essence, evaluation occurs by using locations accessed through the stack pointer at the top of the stack frame, effectively extending the dynamic part as required.

In practice, many architectures will opt to use internal registers for computation and storage of intermediate results. This can be viewed as an optimisation of the stack strategy, with the compiler effectively simulating the stack by the use of a fixed register set, with overflow to stack.

For optimisation in *Primula* it is important that a compiler distinguishes between constant expressions which can be evaluated at compile time, and can therefore be built directly into the code, and the more general expressions which must wait until run time.

Statements.

As the statement syntax is essentially recursive it could be implemented by stacking a return address on entry to each nested statement and then taking it off the stack on completion. This involves more work, however, than the simple use of branches to implement control structures and in-line coding for the remainder.

Within an object all branches may be relative to the current location as the relative positions of the various pieces of code are fixed at compile-time.

Note that care is required in the implementation of the **EXIT** statement as it may be used to exit a number of nested statements before finding the **LOOP** to which it refers. Hence the stack pointer must be reset to its value at the level of the loop and any intermediate values discarded. An **EXIT** is therefore generally a non-trivial operation typically implemented by the compiler calculating the number of locations stacked between the loop and exit statements and modifying the stack pointer appropriately.

Procedures and Functions.

Having established mechanisms for block structure and expression evaluation based around the stack there is little extra needed for calls to named blocks with parameters. Actual parameter expressions are evaluated on the stack and the new stack frame is placed so that these values are within its fixed part.

Note though that the dynamic pointer and display data must be stored in locations known to the called block. A return address must also be held to allow execution of the caller to resume correctly after the call. Because of recursion, this must again be held within the stack frame at a known offset.

Possible options are to leave space for this information before evaluating the parameters or to store it immediately above the parameters. The choice is essentially dependent on any mechanisms provided by the underlying architecture to support procedure calls.

Functions must return results which can immediately be used as part of expression evaluation. These must therefore also be in a location known to the caller and the called function and to save copying must appear on the top of stack when the call is complete.

An obvious policy is to leave space before evaluating actual

parameters for the result and housekeeping information. The called function then has the result space at frame offset zero, effectively as a hidden parameter. When the stack is collapsed after the function call, the result is left on top.

It is evident that parameters are placed in the fixed part of the called frame. Hence when value parameters are of a dynamic size they cannot immediately be copied. The method is to copy in the fixed pointer information and then the called block copies in the actual structure when it begins to set up the dynamic part of its stack frame.

Parameters passed in **REF** mode require an extra dereference to follow the pointer to the real data. **CONST** mode parameters may be passed by reference or copying - this is not apparent at the source level and so implementations may choose when to copy and when to pass a reference on efficiency grounds.

Except where a procedure is **DELEGATE** and therefore the exact procedure to be called is unknown at the point of call, default parameters can be most simply implemented by examining the formal declaration and having the caller place the default values onto the stack as actual parameters. A more complex method for **DELEGATE** procedures is described later.

Object Types.

An object is regarded as a collection of data with procedures to act on that data. Procedures may either be private to the object or made available to other users. Objects may also be viewed as independent abstract machines, communicating by remote procedure calls.

Object instances may be dynamically created within the scope of

their type definition, the block of which may not exit until all objects of types declared in that block have terminated.

The dynamic, separate nature of objects rules out a simple stack implementation although all names must still be accessible by the use of address couples. The logical solution is a cactus stack, where each object stack is an offshoot of the stack of the creating object at the lexical level of the objects type declaration. Links from offshoots to existing stacks allow the address couple mechanism to operate down to the root object at level zero.

Information such as current frame pointer, stack pointer, display etc. are local to an object and so if objects share processors then the state must be saved and loaded on each object swap, which is therefore effectively a process switch. Further details on object concurrency are given below.

The more complex store organisation leads to the need for a storage allocator and garbage collector. The allocator maintains a list of free store blocks and allocates them on creation of object instances. The amount of store given to an object is a pragmatic issue which is not fully addressed by *Primula* at present. One highly flexible policy is to organise the stack such that each frame may be an independently allocated store unit.

Garbage collection may return store to the allocator as outlined in the language description when no references exist to the object and it is no longer executing. The two common forms of garbage collection are noted in the chapter on object oriented programming in general.

The method preferred is that of reference counting, because the overhead is a small amount throughout execution rather than bringing the system to a complete halt for a period of time in order to execute a Mark and Sweep algorithm.

To implement reference counts, each object maintains a count of references to it which is updated whenever reference assignments take place. When a count is decremented or an object terminates execution a check is made on the other condition and if successful the garbage collector is called. Note that the references counts of an object being collected must also be decremented and checked - this is implemented by a hidden procedure or data added to every object type by the compiler and used by the collector.

Simple reference counts do not catch the situation where a circular list of objects has built up, but there are no external references and no object is active. More complex forms require significant processor time to chase all references checking for circularity.

The current *Primula* implementation does not handle such cases but relies on the programmer to break the circle and hence trigger the collection. If this does not occur then blocks containing the declarations of the objects will not exit by the rule stated above.

The standard function **New** calls the storage allocator, initialises the system data structures of the object and transfers the actual parameters to the stack of the new object. It then returns to the caller and the new object may begin concurrent execution.

Remote procedure calls to public methods are performed by some form of message passing. Hence the requested method and parameters, or pointer there to, must be identified in the message and interpreted by the receiving object.

Identification of the method is by specification of a stack frame offset where a pointer to the method can be found. This allows simple inheritance and dynamic binding by changing the pointer value - see below.

Object Concurrency.

Each object has its own state and instruction stream and may therefore execute concurrently with other objects, communicating and synchronising via remote calls to the public access procedures of other objects. As noted previously, these calls are the high-level representation of a synchronous message passing system.

To provide mutual exclusion each object has a lock which is set whilst it is executing either an access for some other object or the main code body. A calling object must check this lock and, if set, queue along with any other callers.

This separate data structure implies the need for a separate lock, as many callers and the called object may simultaneously require access. Such accesses are brief, however, and so a busy wait with some form of arbitration for multiple waiters is a sufficient method to check the queue lock and gain access.

As, in general, there will be more objects than processors, there must also be a data structure allowing a processor whose current object has been forced to wait to find another object to execute.

Possible implementations of this strategy depend on the machine architecture involved. A uni-processor system does not require locks on the queue structures or the executable object list, although they are needed on the objects to detect calls to an object which itself is waiting for a call to complete. Note that if this situation occurs, and there are no objects on the executable list, then the program has deadlocked.

A multi-processor architecture will require the full set of locks, and must additionally ensure that multiple copies of the locks do not occur through the use of cache memory systems. One way of

achieving this is to hold all locks, and probably some data too for efficiency, in a kernel area to which mutually exclusive access is guaranteed by the hardware.

A distributed system has similar needs, but must communicate by message passing along some network. Arriving messages will cause an interrupt and must be received so a non-interruptable kernel is needed for each distributed unit to hold locks for the objects held by the unit.

Whilst a calling object maintains its lock set to prevent being accessed by other callers, it must be able to respond to a signal to indicate that the call has terminated and it may proceed.

When calling another object, the caller must wait for the request queue lock to be available. It then sets the lock, queues the request and releases the lock. If the lock on the called object is not set, the object is available and so the lock is set and the object added to the executable object list. The caller now suspends and allows the processor to find another object from the executable list.

Whenever an object terminates a method and has returned any results, it puts the caller back onto the executable list. It then checks its own queue (using the queue lock appropriately) and if empty, suspends and allows the processor to find another object from the executable list.

If the queue is not empty, then the front request is executed, beginning with the guard construct if present. When a guard fails, that request is marked and put to the back of the queue. The next is then tried and so on until the request is accepted and execution proper begins or the marked request is found, in which case the object suspends. Whatever the outcome of this search, the marked access must be unmarked before execution proceeds.

When an object is restarted, by the lock being set and being put on the executable list, it checks the queue as above to find which object made the request and the action required.

Inheritance.

REFINEMENT inheritance from a single parent requires the creation of an object instance containing all the names of the parent and new names defined by the child. Note that all parent names and not just bequeathed names are needed as a parent method may refer to a private identifier. Bequeathing of an identifier is a compile time issue and has no run time consequences.

Delegation does require run time action as the method to be called both from within the object and from an external reference is dependent on the run time type of the object. This may be implemented by providing a stack frame offset for delegated methods in which a pointer to the current method is kept.

All calls to the method go via the stack frame entry, in the case of remote accesses by the call message including the offset as specification of the method to be called. This method is used for all remote procedure calls and so every public method has a frame offset pointer although internal calls need not use it.

The **USES** inheritance form is similar in its extension of the instance record to include child attributes. The frame offsets must be kept in the record as parent methods will use them to access other methods, although the child may not update the pointers.

Where default formal parameters appear in delegated procedures the procedure itself must fill in any missing values as the call does not know the exact type signature of the method. The called procedure can detect the number of parameters provided by finding the size of

the parameter area, between the frame base and the stack pointer.

Generic Types.

The dynamic store management required for generic types has already been described in the context of dynamic arrays. Rather than storing bounds information along with the pointer, the size of a generic can be determined from the instantiation of the type and stored as hidden data within that type. This allows the required copying of such types to occur, including the case where a formal generic type is called with an actual parameter of some defined type. In this case it is the responsibility of the caller to detect the case and generate the pointer to the actual parameter.

INDEX generics are stored in a fixed size record with fields for low and high bounds in addition to the value. The bounds can be accessed using the standard Low and High functions.

Exception Handling.

On most architectures the detection of errors is explicit and must be provided through conditional tests provided by the compiler. After setting the error variable the problem is to find the 'nearest' exception handler, using the dynamic pointer chain.

Alternative implementations are to provide a pointer to the error handler at a fixed offset within the block, this being Nil if no handler is present, or to define an error handler label in each block, which may simply be at the end of the block, and check the error status on return from each block.

Analysis of Program Behaviour.

To provide some measure of the effectiveness of the support an architecture provides it is necessary to have an idea of the typical workload imposed by the use that language.

Some of the nature of the load can be found from the language definition, and in the case of *Primula* there is the need to support objects and block structure as primary language features. More detailed information on the load imposed by the language, in terms of statement frequency, expression and variable complexity etc, can only be gained by analysing a sample of programs.

As *Primula* is a new language, such a sample is not available. The non-object parts of *Primula*, however, are based around Pascal, and so this makes a good starting point.

The PhD thesis of E. Schmidz [SCHM] contains the results of a survey on the use of Pascal, and provides much useful information. Static data was collected by inserting code into the compiler, and dynamic data by generating extra run-time code to collect information during execution.

His work, however, was on systems software, and therefore may not be representative of the true workload. Hence a survey of user Pascal code was made, but because of the size of the survey, and the varied nature of the source code analysed, no dynamic information was collected. A good match between the static figures and those of Schmidz would, however, tend to indicate that his dynamic data was also acceptable.

The survey was performed using the front part of a Pascal compiler to produce an abstract syntax representation of the source program which was then examined to extract the required information. All the source programs examined were taken from the archive system on

the IBM 370 in the Department of Computing, Imperial College. They consisted of student projects, staff and research work, and systems software.

The principle areas chosen for examination were statement mix, declarations within a block and the form of data types, simple, hence optimisable, forms of expressions, and complexity of access to values.

Derivation of Survey Results.

The appendix gives a full listing of the results, together with a summary of Schmidz' results and a comparison of the two. It concludes that they are consistent in most cases, with some accountable and a few unexplained differences, and so present a reasonable picture of Pascal usage.

It must be emphasised that the survey is looking for trends and for situations where one particular case is highly dominant. To this end, the accuracy of results is not crucial; hence this rather crude, but effective, analysis.

Where the surveys are in close agreement, the static results quoted below are the arithmetic mean of the two, rounded to the nearest percent. Where results differed, a compromise value, based on the reason for the difference, is preferred.

As, for reasons outlined above, this survey did not collect dynamic execution data, these results are based around those of Schmidz, modified proportionally where the static result quoted here differs from his original result. Where direct data does not exist, inferences and estimates are made as outlined below.

Schmidz does not table data for the dynamic number of single constant or variable expressions. The static and dynamic figures for operators per operand (0.21, 0.69) allow a reasonable estimate to be

made, however. The remaining expressions can be catagorised by the proportion of relational and other operators (61.77, 38.33), leading to approximate values of 16 / 52 / 32 for none / relational / arithmetic operator expressions.

Comparison of Primula and Pascal.

Before tabulating the results of the survey, a brief discussion on the expected differences in the usage of the languages will allow the anticipated *Primula* figures to appear alongside their Pascal counterparts.

The most significant language difference is the object-oriented nature of *Primula*, as against the purely procedural Pascal, so some object calls will replace global procedure calls and record structures in Pascal. Additionally, objects will be used for dynamic and shared data structures, and to replace large and variant records.

Considering this and the general trend of an increasing call to assignment ratio in higher level languages it is likely that more calls will be made as a result of this mechanism being available.

The compound statement of Pascal will be replaced by blocks in *Primula*. From static figures it seems that about two thirds of the control statements control a block, noting that the If..Then..Else has two controlled parts and a Case has as many as it has selectors. To these must be added the number of procedure and function declarations.

Dynamically the conditional statements can execute a maximum of one block depending on the condition, but the iterative statements can execute a block many times. The calculation assumes that two thirds of executed control statements control a block. To this number must be added the number of procedure and function calls.

Differences in other statements are small, and coupled with low

usage these, mean little change in the overall pattern.

There are also few differences in expression evaluation except that Pascal always fully evaluates boolean expressions, whereas *Primula* only evaluates as far as necessary. This will reduce the dynamic number of complex expressions, although as the most common form seems to be 'WHILE x AND y' the savings will not be large. This method also allows easier writing of some loop constructs, but the effect on overall expression and statement mix is likely to be small.

The use of objects for data structures will tend to increase the number of function calls made during expression evaluation, at the expense of variable accesses. It will also reduce the number of large record structures, and hence the average number of fields per record.

The use of non-local names will be reduced, as items needing to be shared between program parts can be passed as local references to an object. This will also increase the average amount of dereferencing required.

Modification of the Pascal figures to produce *Primula* values has been done on the above basis in a fairly arbitrary but hopefully reasonable manner. In brief, the use of objects decreases the use of non-local variables and increases the amount of dereferencing and procedure calls.

Result Summary.

Primula values are taken from the Pascal and modified in the manner suggested above. Note that procedure and function call values include object calls in *Primula*.

	Pascal		*Primula*	
	Static	Dynamic	Static	Dynamic
Statements.				
Procedure Call	37	24	39	27
Assignment	40	41	38	38
If..Then	6	9	6	9
If..Then..Else	9	20	9	20
Case	1	1	1	1
While	2	2	2	2
Repeat (Loop)	1	0	1	0
For	2	0	2	0
With	3	3	3	3
Goto	0	0	0	0
Average iterations : While - 6; Repeat - 8; For - 10.				
Average successes : If..Then - 34%; If..Then..Else - 25%.				
Selectors per Case : 7				
Block	9	30	35	60
(figures are relative to statement percentages.)				
Expression factors.				
Variables	42	42	40	38
Constants	50	43	50	43
Function Calls	7	16	9	19
Sets	0	0	1	1
Factors per expression :	1.2	-	1.2	-
Expression operators.				
Relational	50	60	50	60
Add class	25	20	25	20
Mul class	14	12	14	12
Not	11	8	11	8
Expression forms, including sub-expressions.				
Var only (Copy)	16	5	16	5
Const only (Use Const)	34	11	34	11
Var op Var	3	2	3	2
Var op Const	8	10	8	10
Other op	14	20	14	20
Var relop Var	5	13	5	13
Var relop Const	18	38	18	38
Other relop	2	1	2	1
No other specific form exceeds 1%.				
Name Access.				
Scope	33	-	28	-
Local	67	-	72	-

Dereferencing.

No dereferences on	72	-	65	65[5]
Derefs per access :	0.37	0.5	0.4	0.7
Field	39	40	30	30
Array	35	26	35	25
Pointer[6]	26	34	10	15
Object	-	-	25	30

Static averages of various occurrences.

Names per block	: 3.4
Variables per block	: 1.6
Code names per block	: 1.1
Value parameters per defn	: 0.9
Reference parameters per defn	: 0.4

Array index size	: 820	-	820	-
Array dimensions	: 1.1	-	1.1	-
Variants per record	: 11	-	-	-
Fields per record	: 4.8	-	4	-
Subranges (Max<256)	: 78	-	78	-
Struct value params	: 19	-	19	-
Constants > 255	: 1	-	1	-
Constants > 15	: 7	-	7	-

Call forms.

Procedure	81	na	81 assume	80
Function	19	na	19 "	20

Relative Numbers.

Statements	100	100	100	100
Operators	33	61	33	61
Designators[7]	77	158	64	161
Dereferences	44	87	44	87
Expressions	125	73	125	80

Notes.

Integer typed variables are easily the most common.
Almost all pointers point to records.
Packing is rare except for arrays of char.
Arrays of char account for almost half array usage.
String constants are, on average, 17 characters long.

---

[5] Assumed value.
[6] Includes file buffer references.
[7] Only variables used in expressions. Reference parameters and left hand side of assignments are not included.

Comments on results.

A few important factors have been shown by this and other surveys. It is been noted that higher level languages tend to increase the ratio of calls to assignments and here Pascal has almost a one to one ratio. This places great emphasis on the call mechanism, and for *Primula* the object access mechanism. Other statements, with the exception of 'IF' have little overall effect, taking into account their relative complexity.

About half of all expressions consist of a single constant or variable, thereby making this case worthy of optimisation, particularly with regard to the typical small size of constants. Evaluation of complex expressions seems of little importance. A further optimisation target is the high number of relational operations between a variable and a constant.

Emphasis should be placed on variable dereferencing and local variable access, the latter particularly in view of the low average number of declarations per block.

Evaluation of Architectures.

From the above results it is readily seen that a suitable architecture will efficiently support procedure call, remote procedure call and block entry and exit as comprising a major part of the workload.

Expressions tend to be simple (1.2 factors per expression) with a high dynamic proportion comparing a variable against a constant - presumably as the control expression of a loop. Variables themselves tend to be simple and local to the current block.

This suggests that a suitable architecture should provide direct access to simple local variables in order to perform the simple

expressions directly within a single instruction. The small typical size of constants enables good use to be made of any short literal addressing mode.

The conventional architectures discussed in the introduction (IBM370, VAX-11 and M68000) provide varying degrees of support for procedure calls and process switching, the procedure call aspect in particular improving in the later designs.

However, they are all register based machines generally requiring at least one operand of an instruction to be in a register. Use of the registers as purely temporary locations for intermediate results is not particularly effective as most expressions are too simple to benefit. Stack frame access is by base and displacement type addressing modes.

Some compilers perform a complex register allocation algorithm to maximise register use. The choice of technique and implementation is not a simple problem and leads to a variety of widely differing solutions.

The use of registers also requires that they are saved and restored for procedure calls and process switches. Where some registers are used for variables the multiple register move type instructions cannot be used as the locations in the frame will not generally be contiguous.

To overcome some of these problems, the B6700 and ICL2900 do not use an explicitly addressed register set. The B6700 is a stack based machine and all operators take operands from the top of the stack. Stack frame operands are accessed using address couples, described earlier for block structure, which are interpreted using a hardware display implementation.

ICL2900 is similar in using an implicit accumulator register and

providing local, global and extra frame base registers to aid frame access. Machines such as these are easier targets for the compiler and provide some advantages, but make performance improvement by pipelining difficult, as the next instruction tends to depend heavily on the last.

The tagging of Burroughs and descriptors of the ICL is also a problem where the concepts of the source language do not map exactly onto those provided by the machine. This was a problem for Pascal implementors on the 2900. Similar difficulties can arise with the complex instruction sets.

The latter consideration was a motive behind the development of the RISC-I architecture which provides excellent performance executing 'C' programs by a combination of simple, fast instructions and providing a sufficiently large set of addressable registers for a full stack frame and furthermore providing several such sets within a large register stack to allow procedure calls to simply involve changing the base of the addressable register window.

Such a large register stack implies, however, that task switching will be slow as a large number of words must be contiguously saved to store. There is argument on the relative contributions of the instruction set and register structure to RISC-I performance, but it must be noted that space for the register file is essentially created by not implementing complex instructions [KATE].

SOAR, designed for SmallTalk, was the reverse of this in that it provides a rapid context switch between objects but little support for the notion of procedure calls and block structure. It tags integer type words to optimise the SmallTalk treatment of integers as objects.

None of these machines provides full support of the basic implementation ideas outlined for *Primula* or the optimisations found

desirable from a look at run time characteristics. We therefore proceed to consider how such support could be provided in a suitably cost effective manner.

The Processor Architecture.

The above discussion has outlined some of the requirements for the effective support of *Primula* and the problems involved on a few common processors. It appears that a more effective design could be found and so to derive it we begin by examining some of the basic requirements and constraints of architecture design..

Elements of Processor Design.

A processor design must, in general, attempt to satisfy a number of fundamental requirements.

* It must provide the best possible price / performance ratio where the performance is as seen by the user under working conditions and not by artificial benchmarks.

* It must be possible to construct the design for some predefined cost. This may not be the cost required for optimum price / performance, but may be greater or less depending on the application. Hence the first requirement is extended to become obtaining the best performance at the given price.

* It may need to be compatible with some existing design in order to produce a range of designs across a cost band by varying transparent features such as cache design and hardware implementation. The usefulness of this has been amply demonstrated by success of the DEC VAX range [DIGI].

As the eventual target here is an architecture to support *Primula*, there is no need for compatibility with any existing designs.

It should, however, be possible to produce a range of machines based on the same architecture.

The first two considerations do apply, however, and are now examined in greater depth.

Performance.

The requirement to produce a maximum performance / cost ratio leads to the problem of determining end user performance as opposed to artificial benchmarks and the like. This can be approximated by noting that :

* The more frequently a user executes a task the more important its execution time becomes.

* The more complex the task the more machine resources it will require to achieve a performance in keeping with its execution frequency.

Hence machine resources allocated to a task should be in proportion to the frequency and complexity of that task, in order to achieve maximum performance.

To ease the allocation problem, tasks could be collected into similar groups such as systems software, Pascal programs, *Primula* programs etc, on the basis that members of such groups behave in a similar manner and that the complexity of a task is partly determined by the language in which it was written.

The requirements of each of these groups then indicates how machine resources should be divided between them. For example, if the most predominant group is of Fortran programs then there is no point to providing machine support for block structure which is then only used a small fraction of the time.

Tasks can also be divided into logical sub-tasks which are common to many tasks, eg, procedure call and return. Machine resources required for each sub-task can then be determined and this can lead to the derivation of the overall architecture, as performed below.

The measure of performance could cover a number of properties, including such things as direct support for the structures and semantics of the language, ease of compilation, representation of program in store, operands fetched per operator etc., which are sometimes used to evaluate an architecture. It is the view of this thesis, that particularly with increasing personnel costs and decreasing hardware costs, the most relevant measure is the speed of execution, although some note must be taken of response times in an interactive type environment.

A common objection to this approach is that emphasis on direct execution speed leads to the structures of the original program being lost, thus making debugging more difficult. This may be overcome by the use of software debugging tools aided by code output by the compiler, which then hides the problem from the end user. The method also removes the overhead of maintaining the source program structures which, in most cases, are never examined.

Note that the requirement for fast program execution will also have a limiting effect on program size as increases in program size tend to slow execution through extra accesses to main store.

For the design of the processor, the definition of speed of execution is delimited to exclude input / output operations, which, with the exception of the processor interface, are beyond the control of the processor architecture. For simplicity, exceptions other than those caused directly by the program are also ignored, although it is clear that they must be efficiently handled in practice.

The performance, ie. time taken, of a task is the sum of its sub-tasks, as noted above. By grouping similar sub-tasks, the total time is a summation over the sub-tasks of the sub-task frequency multiplied by the average time required for its execution. The task of the architecture design is to reduce this total. Note that sub-tasks could be individual instructions.

Resource Sharing.

Given fixed resources, as implied by the fixed cost outlined at the beginning of this chapter, it is necessary to allocate these between the various components of the system to obtain maximum performance. The criteria of performance is taken to be speed of execution, as indicated above. As noted above, input / output devices are excluded from the discussion as being relatively independent of the processor.

A significant factor though is the cost and performance of the main store. Whilst an increase in the resources spent on store will improve access speed, either by faster immediate memory or by provision of more real memory in a paging system, the size and hence cost of main store relative to the processor is such that a substantial increase in resources are required, in general, to have a significant effect.

Furthermore, the size of real store can be varied almost independently of the architecture design allowing the user to choose this aspect of resource allocation, ie. large store with cheaper processor or vice-versa. Although a less common approach, store speed can also be varied in a similar manner.

The characteristics of cache stores can be determined, to a large extent, outside the definitions of the visible architecture and hence

another variable factor of resource allocation which need not be fixed by the architecture design itself.

Note though that the block size of cache is generally related to the width and interleaving properties of the main store. The use of cache is also an inherent part of many processor designs in that performance is highly dependent on the use of cache.

For a given main store performance, therefore, resources must be balanced between caches and processor in order to obtain best results. Note that although for a given processor, increases in cache resources (hence overall resources and cost) will provide increases in performance, the balance between cache and processor may not be optimal.

Hence the resources allocated to the processor, and the associated level of cache needed to make the design effective, can be considered fixed relative to main store and Input / Output. A similar view can be arrived at by considering the processor to be the fixed cost part of the machine with main store and input / output being bolt-on, cost extra, facilities.

The problem is hence reduced to allocating fixed resources amongst the various parts of the processor and any required cacheing in order to achieve best performance, ie. minimise execution time.

The average execution time of an instruction can be viewed as proportional to the resources allocated to it and its inherent complexity. A first attempt could therefore divide resources between instructions such that the amount of resource was proportional to the frequency multiplied by the complexity.

It is evident, however, that a proportion of the resources for each instruction must be used for the actions they all require such as decoding, operand fetch and store etc. It follows that if some

instructions can share some resource eg. an ALU, then the sum of these resources could produce a quicker action for all the sharers. The price is the use of some resources in organising the sharing.

Microcode is a common technique which allows complex instructions to share the control sequencing mechanisms and a number of very fast, basic, resources. It also allows the total resources required to be reduced, allowing many otherwise impractical processors to be built. On a typical microprocessor such as the M68000 [MOTO], control circuitry occupies 60% of the silicon resources.

In addition to the control and execution circuitry, instructions must also share operand access and storage circuits. In most processors, a small set of registers is provided, optionally with a cache store buffer to main memory. The cache is not generally in the CPU design and so is not fitted onto the VLSI chip.

It can be argued that the sharing of operand storage has been overlooked in many designs, particularly with respect to the difficulty in compiling code to efficiently use a small register set, and the large number of 'Move' type instructions this tends to generate. Following the argument for sharing resources amongst the units needed by several instructions, it is apparent that the operand storage area should receive a substantial amount.

A consequence of this is a reduction in the resources available for the execution part of instructions, placing greater emphasis on the way they are shared. It may not be possible to implement some instructions within their resource limits, and so they must be emulated by other instructions, the resources being redistributed appropriately.

Reducing the resources required for instruction execution can be achieved by a more thorough compile time analysis of the problem,

leading to a series of simpler instructions. This reduces execution time by reducing the number of general cases that the original instruction would have to check for. In some cases, the re-allocation of resources may require that the general instruction is emulated by others, following the reasoning above.

The elimination of some complex instructions places the burden of microcode resources onto fewer instructions, and this tends to continue until microcode does not provide sufficient sharing between these simpler instructions to justify its resources. At this point, the design reverts to hardware control and becomes a RISC[8] style machine.

Deriving the instruction set for a *Primula* machine begins with a high level abstract machine, and ends with a series of simple operations, which could be shared by microcode or at object code level. Following the above argument, when a reasonable amount of resources are allocated to operand storage, and using a compiler which can identify simple cases and produce appropriate instructions, the costs of a microcode machine outweigh potential benefits.

RISC vs CISC.

These approaches reflect the two routes to achieving the resource sharing discussed above. RISC machines provide simple hardware primitives and require the compiler to build up language actions from these. CISC[9] machines implement the more complex higher-level operations, generally by the use of a micro-code interpreter.

Much has already been written on the relative merits and problems

---

[8] RISC - Reduced Instruction Set Computer.
[9] CISC - Complex Instruction Set Computer.

of both approaches [IEE2][SIG1] and so only a summary of the benefits and problems of each is presented here. It is intended to be general and so some comments may not apply to all architectures.

Note in particular that many contemporary architectures have not been highly oriented towards high-level languages and so many of the potential positive aspects of CISC have not been realised in these designs.

RISC.

+ Lack of control circuitry allows more space for fast operand storage.

- Simple instructions imply more are required to express high-level language semantics.

+ Simple instructions allow compiler to build up high-level language semantics and so allow effective implementation of differing languages.

+ Compiler can optimise out checks where the result can be determined at compile time, eg. indexing into an array within a 'For' loop.

+ All facilities heavily used, ie. tending towards optimal balance of resource usage.

- Assembly level programming is difficult and so optimisation must be performed by the compiler. Systems programming difficult unless supported by high-level language.

- Low level interface definition limits ability to produce a range of compatible machines, differing only in their implementation.

+ Direct hardware execution of instructions, generally in a single cycle, leads to fast execution.

- Increases in performance can only be made by hardware improvements or addition of transparent items such as cache.

- Strictly sequential operations, as instructions and control mechanism not sufficiently complex to deal with implicit parallelism.

+ Single cycle operation allows fast interrupt response time and easy handling of virtual store exceptions.

CISC.

+ Complex instructions allow shorter programs, hence less instruction fetches, by modelling high-level language semantics.

- Instructions cumbersome or unusable where not exactly matched to high-level language semantics. A CISC which is highly oriented toward a particular language is often reduced to using only the simpler instructions to implement a different language.

- Optimisation by eliminating checks where the result is known at compile time can only be done using a sequence of simpler instructions unless a special instruction for that type of optimisation is available.

- Many rarely used instructions and addressing modes.

+ Assembly language programming is usually well-supported for use in optimisation or systems type code.

+ Complex interface allows differing underlying implementations to provide a fully compatible range of machines.

- Micro-code is inherently slower than using the compiler to generate the simple actions as it interposes an additional level of run-time interpretation.

+ Hardware implementation of complex instructions likely to be fast, although highly expensive.

+ Horizontal microcode and hardware implementations can take advantage of implicit parallelism within the instruction.

- Long instruction execution times slows interrupt response. Complex instruction state causes problems for handling virtual storage exceptions as it must be saved or the instruction arranged so that it can be restarted from the beginning.

Multiple Languages.

In general, an architecture must be capable of efficiently executing more than one language, although it is reasonable to restrict this requirement to languages of a basically similar nature (block-structured etc). This requirement can be joined with the need to efficiently execute systems programs.

The CISC approach is to provide the necessary instructions to

support the operating system and all languages that the machine is designed to efficiently execute. In practice, this would lead to a large number of similar instructions to cope with the differences in language semantics.

An extreme example of such a machine is the SWARD [MYER] which is designed to directly execute Pascal source code.

If compiler optimisation is to be performed, then further instructions are needed, for each language, to allow the given optimisation without requiring the use of a sequence of simpler operations.

In a general purpose system then, this argues against the support of high-level languages by providing complex instructions which have limited applicability. Such instructions take up valuable resources with limited returns.

The architecture must provide sufficient support to allow the construction of an operating system kernel. Other systems software may be written in a language which the machine is designed to support on a user level, which therefore minimises the types of support required and so concentrates resources for better performance.

Except for the features which the high-level language cannot support, of which there should be only a few providing a suitable language has been chosen, assembler programming should not yield a significant performance benefit over a compiled program if the architecture is correctly supporting the source language.

The advantages of having the majority of the system in high-level languages are in terms of potential reliability and portability. Unix, written in 'C', is a classic example.

Compilation and Optimisation.

An architecture can aid a compiler in two ways. In the CISC idiom, it can provide complex operations and data structures which correspond to those of the language concerned. It may also provide variants which an optimising compiler could use when it spots a potential simplification.

Secondly, the processor can provide a regular set of instructions and addressing modes to allow the construction of code sequences to provide the necessary semantics. The regularity is of great help as it reduces the number of special cases to be dealt with. Simplification with this form of support is simply a matter of omitting unnecessary instructions.

Many conventional processors fail to support high-level languages on both these grounds. The complex operations do not match the required semantics and the instruction set is not orthogonal to the addressing modes so that some modes only work with some instructions etc.

In addition, the many complex instructions and modes which the compiler cannot utilise are a waste of resources as far as the particular language is concerned.

Another feature is the ability of the compiler to understand and use the pipelining of the processor for maximum performance. CISC pipelines are often complex and change between differing implementations of the same basic architectures. RISCs, in general, have a very simple system which the compiler can use to advantage. Indeed, the MIPS processor [HENN] has no pipeline reservation in the hardware, relying on the compiler to provide explicit control and hence effectively making it a part of the architecture.

Hardware Implementation Technologies.

A range of hardware technologies are available for the implementation of an architecture, and it is worth briefly examining the relative characteristics of these. Further discussion can be found in [RAND].

* Small Scale Integration (SSI) is the method most often used on large computers. The large number of chips required for a design mean a high production cost, the main advantage being that fewer components on the chip allows each to dissipate more heat and hence operate at a higher frequency. Against this must be set the time required to pass signals between chips.

* Very Large Scale Integration (VLSI) is the common implementation for microprocessors. A single chip, currently up to 500,000 transistors, can hold an entire complex processor. The drawback is that the heat dissipated restricts the speed of operation. Some processor designs may also encounter a problem with the limited number of connections available to the chip.

* Wafer Scale Integration (WSI) is an emerging technology which uses fault detection and redundant circuitry to increase both component count and percentage of working units (yield). The latter is a limiting problem for VLSI.

The emergence of VLSI has brought about a large increase in the performance / cost ratio, as the communication delays between system components have been reduced by the close proximity of the components.

If the system is too large for a single chip, which is typically

the case with processor, memory, I/O circuits etc., the discrepancy between on-chip and input / output speeds requires that functional units are arranged to minimise communication delays. Commonly, the processor, I/O controller and bit slices of memory reside on different chips.

The division of function becomes more prominent with the use of Gallium Arsenide fabrication, which offers a ten-fold speed increase in return for a tenth of the workable chip size, and a greater on-chip to input / output speed difference [IEE3]. As processor components are generally tightly linked, it appears advisable to keep them in as close proximity as possible for maximum speed.

A further issue here is the compromise between speed and packing density, due to increases in both producing extra heat to be dissipated. This restricts the use of VLSI as sufficient speed cannot be obtained for some applications and so SSI is preferred. One approach is the use of VLSI main memory and SSI for processor and cache circuits, hence providing a high performance / cost ratio.

It must be noted that the repeated use of a simple circuit allows a small optimisation of that circuit to have a large overall effect, and hence increase the level of integration. The results of this are clearly shown by comparison of the transistor counts of memory chips against conventional VLSI processors. The yield of operational chips is also a problem, and this can be increased by the use of error detection and redundancy. Here, the use of repeated small units allows greater flexibility in the provision of redundancy [RAND].

Looking forward, the likely needs of Wafer Scale Integration, WSI, must be considered. WSI provides a silicon area may times that of VLSI, hence minimising input / output requirements of large circuits. It relies heavily on the error detection and redundancy ideas noted

above, and so despite the apparent potential to build more complex designs, WSI may be more suited to repeated small units such as memory and simple processors.

This thesis will initially develop an architecture suited to a VLSI implementation. In particular, VLSI is the only cost effective method to satisfy increasing main memory requirements at almost all levels. Although initially for VLSI, the processor could be adapted to SSI for higher performance or incorporated into WSI as a multi-processor system.

The cost effectiveness of VLSI has led to many researchers investigating the potential of multi-processor architectures based around a number of VLSI processors.

Architecture for *Primula*.

Having established the general principles and criteria of architecture design, we turn to the design of an architecture to support *Primula*.

As *Primula* is capable of effectively expressing systems programs there is no need for any extra systems or assembler level support, other than that needed for the kernel. In particular, as the language expresses concurrency and requires task switching as part of the implementation, the requirements of a multi-tasking architecture can be derived directly from the language.

It was noted above that architecture performance would be based on speed of execution with some note taken of response times. Response time is generally a function of the overhead in performing a task switch and as such is recognised by the need to support the object switching within *Primula*.

The sharing of resources between competing, as opposed to co-

operating, tasks or objects can be performed by the operating system, again written in *Primula*, with no extra support needed at architectural level, other than provision for external interrupt handling.

It is assumed that protection between objects is provided by the compiler and hence for this design no hardware storage protection is implemented. Virtual storage is also not specified in the main design, but could be added as a unit between the processor and main store. This requires the facility for instructions to be halted on any store accesses and is briefly covered below.

Execution of other languages is not an important factor but as *Primula* is block structured other languages of this nature, such as Pascal, will be supported in this respect. Such a program can be viewed as a single object in the *Primula* system.

As *Primula* defines its integer type to be 32 bit, the architecture does likewise. This is also a reasonable size for the addressing range and is convenient. To allow VLSI implementation and minimise costs, a single word width store is used, with caches as described later to bridge differences in processor and main store speeds.

A multi-word width store and complex instructions were used in machines such as the Control Data STAR [HINZ] to obtain performance with a relatively slow main store. Complex instructions reduced the number of instruction fetches and were often vector type operations lending themselves to a block cacheing type strategy.

Whilst simple instructions increase the fetch rate, they also tend to increase the usefulness of cache as loops are held in the cache and so main store is not then accessed. Non vector processing tends to increase randomness of data structure access, again reducing

the effectiveness of wide stores.

Current technology has also narrowed the difference between processor and store speeds and so whilst the use of wide store with cache is undoubtedly a performance aid, it is an expensive option and its advantages are not directly relevant to this design.

Note though that by use of an external cache, the processor could be used within a wide store system.

Instruction Set Derivation.

The derivation of the instruction set is essentially an iterative process. Each step is an abstract machine capable of performing the operations required by *Primula*. By using the dynamic execution data from the source code survey in the previous chapter, relative frequencies can be assigned to each abstract machine operation.

As previously noted, the resources of the processor must be shared out for minimum execution time, and this implies that each instruction receives resources in proportion to its frequency and inherent complexity.

At each stage, it may be possible to emulate some of the operations using a sequence of the others, or devise some new operation which is then used in several sequences. The resultant design may produce a faster net execution time due to the sharing of resources between operations. Note this itself does not imply either a microcode or a RISC type implementation.

Note the process is not likely to go so far as to produce a Turing type machine as the number of simple operations required to implement the more complex operation would be so large as to outweigh the benefits produced by sharing the resources.

The process halts when no better arrangement of resources and

operations can be found.

Language Translation Fragments

Abstract Machine 1.

The initial abstract machine can easily be specified from the language design and in particular from the chapter dealing with the run-time needs of *Primula*. Note that the architecture must tag data according to its type, and performs all the actions, range and error checking etc, implicit in a language statement.

```
EVAL_DESIGNATOR <Designator> { <Expression> ! <Designator> }
     Returns the value of the designator.
BLOCK <stack_space> { <Statement> }
     Allocates stack space and execute statements.
END <size>
     Deallocate stack space and continue calling block.
ASSIGN <Designator> <Expression>
IF <Expression> <Statement> <Statement>
CASE <Expression> { { <Expression> } <Statement> }
       <Statement>
WHILE <Expression> <Statement>
LOOP <Statement>
EXIT
FOR <Designator> <Start_Expression> <End_Expression>
      <Step_Expression> <Statement>
WITH <Designator> <Statement>
```

The level of control flow instructions prevents a simple, linear instruction stream implementation, and it appears that a stack based call and return scheme will be needed for most statements. [SIG2]

Note that the EVAL_DESIGNATOR instruction is only needed for procedure calls and object accesses, other expressions and designators being evaluated as part of some other action. Object accesses require the synchronisation protocols to be observed, but at this level we assume that the machine itself handles this.

Although required to complete the machine, details of the tagging mechanism and the representation of complex types and subranges are

not necessary for this argument and so are not elaborated.

By a more detailed definition, this abstract machine could be used to model or define the *Primula* language itself.

Frequencies 1.

These figures are taken directly from the previous chapter. The inclusion of BLOCK and END values means that the figures are no longer percentages of the abstract machine instructions executed.

Fragment	Static	Dynamic
EVAL_DESIGNATOR	39	27
ASSIGN	38	38
IF	15	29
CASE	1	1
WHILE	2	2
REPEAT	1	0
FOR	2	0
WITH	3	3
BLOCK	35	60
END	35	60
Total	171	220

Abstract Machine 2.

It is readily evident that the 'EVAL_DESIGNATOR', 'BLOCK', 'END', 'ASSIGN' and 'IF' are the fragments to be examined more closely and given the best possible support as they account for 97% of dynamic instructions and none of the other operations is of sufficient complexity to cause significant load on the machine.

As the instructions stand, the arguments are complex recursive data structures, implying that an instruction execution could need to execute other instructions, perhaps recursively, needing use of a stack to save intermediate instruction states. Whilst this achieves sharing of execution resources, heavy use of the stack and the number of calls and returns to / from code blocks produces an unnecessary

load on the operand access resources. This suggests breaking down operations into non-recursive, shared parts.

Expression evaluation has several opportunities for recursion, eg. in function, array index, and set constructor evaluation. These could be removed by evaluating the constituent designators of an expression, storing the values, and then applying a sequence of operators to them.

A similar idea can be applied to designator evaluation, with each part of the designator individually evaluated and then a sequence of operators applied. Whilst possible, this method has little to commend it, especially when the high proportion of simple expressions and designators is considered.

A further simplification places each operator immediately after the evaluation of its operands in the linear instruction stream. This results in instruction sequences similar to those found on many current machines.

The final step in removing the recursion is to introduce more explicit control of program flow. Control statements in particular no longer need to include the statement which they are controlling. The number of branches from sequential execution is drastically reduced with all statements and most blocks being coded in line. This has, however, necessitated the use of an unconditional branch, or 'Goto', which also allows the implementation of all control constructs in terms of 'If'.

More explicit control over space allocation and display information is also required. There are several possible strategies for organisation of stack frame data etc, but the choice is limited by the possibility of calls to procedure variables, holding a procedure of unknown lexical level. By placing all stack frame handling within

the called block code size is reduced and compile time constants can be utilised.

Calls proceed by stacking parameters, performing the 'Call' to stack PC, and then the block uses 'Alloc_Stack' to set up the new frame and allocate workspace. On completion, 'De_Alloc_Stack' deallocates workspace and resets the frame data, allowing 'Return' to find the old PC. A further call to 'De_Alloc_Stack' then removes the parameters, leaving any function result value.

In-line blocks use 'Alloc_Stack' and 'De_Alloc_Stack' for their workspace management. These operators must be sufficiently sophisticated to allow for the presence of generic types whose size is unknown until run-time. The tagging information can be used for this purpose.

Tagging is also used to identify basic types and pointers etc, enabling correct choice of operation and automatic dereferencing, and to indicate a complex type descriptor which holds the type length to allow generalised comparison operators etc.

Note that the logical operators 'And' and 'Or' may cause partial evaluation of a boolean expression and so are implemented using the standard conditional 'If' and logical 'Not'.

The instruction set now looks like the following, noting that arguments beginning in 't' are temporary and may be held in some architecture defined location eg. explicitly in registers or implicitly on the stack. Arguments beginning 'n' and the lexical information are all literal constants.

Instructions, and in particular their operands, are more tightly defined to give some indication of complexity.

```
CALL t
     Stack PC, execute from location t.
```

RETURN
Return control to address left on stack.

ALLOC_STACK <lex_level>, n_{params}, n_{locals}
Stack frame data, new frame has n_{params} already on stack. Allocate n_{locals} stack locations.

DE_ALLOC_STACK <lex_level>, n_{locals}
Deallocate n_{locals} stack locations; Iff LL <> 0 pop frame data.

ASSIGN t_{data}, t_{dest}
Assign t_{data} to t_{dest}^, noting that structures must also be copied, some object references require type checking, and some scalars require range checking. Also used for parameter passing etc.

IF t, <location>
If value is 'true' then execute from location.

GOTO <location>
Switch execution to the given location.

ADD t_1, t_2, t_{result}
SUB t_1, t_2, t_{result}
SETUNION t_1, t_2, t_{result}
SETDIFF t_1, t_2, t_{result}
NEG t_1, t_{result}
MUL t_1, t_2, t_{result}
DIV t_1, t_2, t_{result}
RDIV t_1, t_2, t_{result}
MOD t_1, t_2, t_{result}
SETINT t_1, t_2, t_{result}
NOT t_1, t_{result}
BITOR t_1, t_2, t_{result}
BITAND t_1, t_2, t_{result}
BITNOT t_1, t_{result}
EQ t_1, t_2, t_{result}
Set tresult according to equality of t_1 and t_2.

NEQ t_1, t_2, t_{result}
LE t_1, t_2, t_{result}
LEQ t_1, t_2, t_{result}
GT t_1, t_2, t_{result}
GEQ t_1, t_2, t_{result}
IN t_1, t_2, t_{result}
RECORD_ACCESS t_{base}, t_{offset}, t_{result}
ARRAY_ACCESS t_{base}, t_{index}, t_{result}
Index into array and return result. Range check data and size are found from descriptor.

POINTER_ACCESS t_{addr}, t_{result}
Return contents of location t_{addr} in t_{result}.

OBJECT_ACCESS t_{ref}, t_{entry}, t_{params}
Access object given by t_{ref}, using entry point t_{entry} and passing parameters at location t_{params}.

RECORD_WRITE t_{base}, t_{offset}, t_{ref}
Add t_{offset} to t_{base} and return in t_{ref}.

ARRAY_WRITE t_{base}, t_{index}, t_{ref}
Calculate address of element, check against descriptor.

REFERENCE <lex_level>, <lex_offset>, t_{ref}
Put runtime address of name into t_{ref}.

Depending on architectural details, extra operations may be

needed to manipulate the temporary values, t. This style of architecture bears resemblance to the Burroughs B6700 [ORGA], where the t operands correspond to the B6700 stack.

Frequencies 2.

These are derived from the first set of frequencies, adding extra information from the code survey as required.

The number of calls is found from the number of procedure calls added to the function calls, which is in turned derived using the function (20) / procedure (80) call percentages in the survey results. This is also the dynamic number of returns.

The static number of callable blocks, and hence by subtraction in-line blocks, is found by using the Pascal block count quoted in the results, as Pascal only has callable blocks. This figure is the static number of 'Return's. The static number of 'Alloc_Stack's is the same as the number of blocks and the number of 'De_Alloc_Stack's is this number added to the number of calls.

Each dynamic call requires a 'Return' and a 'De_Alloc_Stack' in addition to the 'Alloc_Stack' and 'De_Alloc_Stack' required by all blocks. The 'Alloc_Stack' count is the same as the number of blocks entered, and the 'De_Alloc_Stack' this value plus the number of calls.

'If' instructions are now used to implement all control constructs, so the number is the sum of all such constructs. Note the average of 7.3 selectors in the 'Case' statement implies that about 7 'If's are needed statically, and it is assumed that 3 are used dynamically. Each loop requires a static 'If' and a number of dynamic 'If's depending on the average number of iterations.

If..Then..Else and the loops require a single 'Goto' statically, whereas a 'Case' has one per branch. Dynamically, the If..Then..Else

executes its 'Goto' only when the Then branch is chosen (25% of If..Then..*Else*), loops execute them on each iteration and the 'Case' just once.

The operator values are found by taking the percentages for each class, as found by Schmidz, from the total number of operators given in the relative numbers section of the source survey. Set operations are included in these values. A similar principle is used for the access operations.

A 'Reference' operation is needed before each use of a variable, so the figure is the sum of variables used in expressions, as reference parameters, and on the left hand side of assignments. The instruction takes a compile time offset and adds it to the run-time stack frame base indicated by the lexical level operand to obtain a reference to the variable.

Total Counts Fragment	Static	Dynamic	
CALL	49	34	
RETURN	9	34	
ALLOC_STACK	35	60	
DE_ALLOC_STACK	84	94	
ASSIGN	38	38	
IF	27	44	
GOTO	21	18	
ADD/SUB/NEG/ SETUNION/SETDIFF	8	12	
MUL/DIV/RDIV/MOD/SETINT	4	7	
NOT	3	5	
BITOR/BITAND/BITNOT	0	0	
EQ/NEQ/LE/LEQ/GT/GEQ/IN	17	37	
RECORD_ACCESS	13	26	(read and write)
ARRAY_ACCESS	15	22	
POINTER_ACCESS	4	13	
OBJECT_ACCESS	11	26	
REFERENCE	125	290	(64+23+38,181+21+38)
Total	463	760	

It is evident here that the support emphasis must be on the call and stack allocation operations, and the primary 'Reference' instruction. Additionally, the large use of temporary locations requires these to be effectively implemented.

The 'Reference' mechanism must be able to return the base address of any stack frame entry. This is straightforward where the compiler knows the lexical level and offset but note the actions required to accommodate dynamically sized structures as described in the previous chapter.

From the survey, about 65% of variable use requires no dereferencing, and from the declaration types we can assume that a majority of these variables will be of simple type. This suggests that a reduction in the number of 'Reference' instructions could be brought about by allowing the operator instructions to directly access variables, requiring a choice of addressing modes. Except for a reduction in instruction fetches, partly nullified by extra instruction length, there is only the limited saving in not requiring a temporary location.

Noting, however, that 67% of variable accesses are to local variables suggests that providing an easier access to just these variables would have benefits. It would require the stack frame manipulation instructions to additionally maintain a local frame base register, with the displacement provided within the instruction.

Adding a local variable addressing mode to the operator instructions would now save more space and time, as the generalised action of the 'Reference' instruction is no longer required. This, in essence, is the method used by many current processors which permit base and displacement addressing of operands, in addition to register,

or temporary, locations.

Note though that the method will only be successful if the displacement field is large enough to hold the majority of accesses required, but small enough not to cause a large increase in instruction size.

This has placed greater emphasis on the access to the current stack frame, with the number of temporary locations in use severely reduced. As these temporary locations are few and local in use, if they could be made part of the local stack frame, the operand storage requirement would be firmly based on the local stack frame.

Dynamic types, however, require an unknown amount of space within the stack frame, and so temporary locations must be addressed via the stack pointer if they are to be part of the stack frame.

Turning to the structured variable access instructions, it is evident that these too could have displacement type operands. A Record access is an 'Add' followed by a 'Pointer_Access', and array access contains comparisons for range checking, subtraction and multiplication followed by a 'Pointer_Access'.

From the code survey it is evident that many arrays are of simple elements, thereby removing the multiplication by element size. If the word size is greater than the addressing unit, then shifts may be required to implement multiplication by two or four. Much of the range checking can also be found unnecessary as indexing commonly uses subrange variables of the same type as the index.

A clever optimising compiler could also reduce array accesses in a tight loop down to increments on a pointer followed by referencing that pointer. Without this level of sophistication, however, it is evident that savings can be made by performing array access with the simpler operations.

Although no figures are available, it is likely that use of the full array access operation would drop to a point where the resources it requires are uneconomic compared to its use. It is hence excluded from the instruction set.

This removes the need for an architecture defined array descriptor and must call into question the need for a range descriptor to allow the 'Assign' operator to check scalar assignments. 'Assign' can be simplified and range checks inserted where necessary by the compiler, using existing comparison operators.

Remaining uses of tagging or descriptors are to identify the type of a scalar when using an expression operator instruction, and to hold the size of a structure for copy and equality type instructions. The former can simply be replaced by the compiler generating the appropriate instruction for the types, generally resulting in more effective use of main store bits[10].

The use of a fixed size descriptor for dynamic and generic types has been noted, and a size descriptor for structures is a useful feature for set, copy and equality operations in that it avoids the need for an assembly level loop with counter. Scalar assignment can be optimised with 'Move' byte, word or double word (for the LongReal type) instruction; structure assignment is renamed 'Copy'.

Note that the 'Move' and 'Copy' instructions have indirect addressing variants to allow them to operate on structures found through pointers, such as items in the variable part of the stack frame.

---

[10] Does each addressable store unit, ie. byte, have its own tag or does it use that of the whole word? If the latter, how are space efficient structures having two types of data in separate parts of the same word handled.

Arithmetic operators are basically the same set, but extended to cover the primitive types supported by the language. This also involves the provision of conversion instructions between these types. 'Neg' is replaced by 'Sub' and the bitwise operators by the set operators - 'BitNot' is equivalent to 'BitDiff' where the first set operand is full.

Relational operators are also extended to cover the primitive types. Set relations, except for 'In', are infrequent and are easily simulated using 'BitDiff' operators and a conditional branch. Short operand operators are not provided, as they can be effectively performed by moving bytes into and out of words.

The instruction set is now as follows, where x indicates a choice of displacement from implicit base or from stack pointer.

CALL t
    Stack PC, execute from location t.
RETURN
    Return control to address left on stack.
ALLOC_STACK <lex_level>, n_{params}, n_{locals}
    Stack frame data, new frame has n_{params} already on stack. Allocate n_{locals} stack locations.
DE_ALLOC_STACK <lex_level>, n_{locals}
    Deallocate n_{locals} stack locations; If LL <>0 pop frame data.
MOVE.B/MOVE.W/MOVE.L x_{data} | x_{dest}^, x_{dest} | x_{dest}^
    Move byte, word and double word of data.
COPY x_{data} | x_{dest}^, x_{dest} | x_{dest}^
    Copy structure at x_{data} or x_{dest}^ to x_{dest} or x_{dest}^.
IF x, <location>
    If value is 'true' then execute from location.
GOTO <location>
    Switch execution to the given location.

ADD.I/ADD.R/ADD.L x_1, x_2, x_{result}
SUB.I/SUB.R/SUB.L x_1, x_2, x_{result}
SETUNION x_1, x_2, x_{result}
SETDIFF x_1, x_2, x_{result}
MUL.I/MUL.R/MUL.L x_1, x_2, x_{result}
DIV.I x_1, x_2, x_{result}
RDIV.I/RDIV.R/RDIV.L x_1, x_2, x_{result}
MOD.I x_1, x_2, x_{result}
SETINT x_1, x_2, x_{result}
NOT x_1, x_{result}
ITOR/ITOL/RTOL x_1, x_{result}

STREQ X_1, X_2, X_{result}
Set xresult according to equality of structures X_1 and X_2.
STRNEQ X_1, X_2, X_{result}
EQ.I/EQ.L X_1, X_2, X_{result}
NEQ.I/NEQ.L X_1, X_2, X_{result}
LE.I/LE.L X_1, X_2, X_{result}
LEQ.I/LEQ.L X_1, X_2, X_{result}
GT.I/GT.L X_1, X_2, X_{result}
GEQ.I/GEQ.L X_1, X_2, X_{result}
IN X_1, X_2, X_{result}

ASHIFTL X_1, X_{result}
OBJECT_ACCESS X_{ref}, X_{entry}, X_{params}
REFERENCE <lex_level>, <lex_offset>, X_{ref}

Frequencies 3.

The assignment operation now occurs implicitly as the final step of expression evaluation, as operators can now directly address the stack frame. 'Copy' and 'Move' are used whenever an expression has no operator (50% Of 125 static, 16% of 80 dynamic), wherever a reference parameter is used, or where addresses are found at run-time, as in array accesses, pointers and dynamic structures.

The total found by adding these parts is divided between the two instructions using Schmidz' data on assignments. He found that only 14% static, 3% dynamic assignments were of structures and so required the 'Copy' operation.

Many record accesses can be replaced by compile time arithmetic to produce a frame offset for direct use in the instruction. This cannot be done, however, where the record is accessed via some other structure and so it is arbitrarily taken that half the record accesses require an 'Add' instruction.

The full, checked array access requires a 'Sub', two 'If's and a 'Mul', in addition to the move operation included above. It is assumed, however, that optimisation will remove the need for half the 'If's and two-thirds of the 'Mul's. A third of the 'Mul's are assumed to be replaced by 'AShiftL's, although if the multiplication was slow,

a greater proportion would be replaced in practice with shift and add sequences.

The number of type conversion instructions is assumed to be negligible. Structural equalities are also likely to be insignificant as most equality testing occurs as checks on scalars for loop control etc.

The 'Reference' operation is now only required for non-local name access and so from the survey is reduced to 28% of its previous use.

Total Counts Fragment	Static	Dynamic
CALL	49	34
RETURN	9	34
ALLOC_STACK	35	60
DE_ALLOC_STACK	84	94
MOVE	71	47
COPY	11	1
IF	34	55
GOTO	21	18
ADD/SUB/SETUNION/SETDIFF	30	47
MUL/DIV/RDIV/MOD/SETINT	9	14
ITOR/ITOL/RTOL	-	-
NOT	3	5
STREQ/STRNEQ	-	-
EQ/NEQ/LE/LEQ/GT/GEQ/IN	17	37
ASHIFTL	5	7
OBJECT_ACCESS	11	26
REFERENCE	35	81
Total	424	560

Abstract Machine 4.

The remaining instructions using the structure size information are used very infrequently, suggesting that they could be replaced by a loop constructed from existing subtract and compare instructions.

This raises the question of a specialised 'Subtract&Compare' operation, but except for the few structural operations and a small proportion of the relatively uncommon 'For' loops it would seem difficult for a compiler to take advantage of this operation and

therefore its frequency would be insufficient to be justifiable.

Bitwise logical operations are brought back to be used in the loops implementing set operations, in addition to as operators in their own right. Their use is infrequent, but the the inclusion is justified by extremely low complexity and the common need for such instructions in low-level programming.

Examining the types of arithmetic operands in the survey, it appears that the proportion of 'Real' arithmetic will be low, and presumably the 'LongReal' type will also be uncommon. It is therefore questionable whether the high complexity of 'Real' instructions can be justified.

An additional factor, not brought out by the survey, is that heavy use of such operations occurs in relatively few programs, these generally being of a scientific or engineering nature. The majority of users make little use of 'Real' arithmetic and so inclusion of the facilities is definitely not justified. Inclusion is necessary for intensive users as the frequency is sufficient that distributing the resources saved by emulation would not make up for the subsequent losses.

The solution adopted is that 'Real' operations are not included in the principle instruction set of the processor but are included via a special 'Extend' instruction whose operands specify the required action.

These instructions could then be interpreted by hardware within the processor (perhaps included by reducing on-chip cache sizes), by an external co-processor, or by a software trap. By defining the instructions they become a fixed part of the architecture, but need not be hardware implemented in all cases.

A similar argument can be applied to the integer mathematical

operators which, apart from multiplication, see little use. Whilst hardware support for a step of Booths' multiplication algorithm cannot be ruled out, the resources required to implement integer division and modulus in particular are far in excess of their frequency of use.

Without special support, software emulation of multiplications requires two branches, two shifts, add and move per bit; division (and modulus) requires two branches, two shifts, three subtracts and an add. Hence a 32 by 32 multiply will typically require 68 instructions to perform as the add will be performed for half the bits and the multiplier will be 16 bits long, on average. Extra simple shift and rotate operations are provided for these emulations.

Another candidate for emulation is the 'In' operator, which again is infrequent relative to its complexity, requiring the ability to select a specific bit from a word by its number. As it is the only use of this sort of facility, 'In' is better implemented by a loop, or where the number is a constant, use of a compiler generated mask.

The relational operators currently return a boolean value, which is almost always used immediately by an 'If' instruction. Most architectures have a similar system of a special instruction to set flags which are then tested by a variety of branching operators to determine the action to take. These also require two instructions and still need extra code to turn the flags into a boolean value when needed.

The method suggested here is to convert all the relation operators into branches conditional on their operands, so that 'Eq' becomes 'BrEq'. This reduces the instruction count for the most frequent case, although it does have drawbacks, discussed later, in relation to pipelining. Having adopted this technique, logical 'Not' can be performed by reversing the condition and hence the instruction

is redundant.

Boolean values are obtained by using a 'BranchTaken' bit in the status register, set according to the action of the last branch instruction. Hence by branching to the next sequential instruction this bit can be set and then used as the boolean value.

A remaining issue is stack management and the accessing of operands. Whilst by making the stack pointer and display information more visible the stack management and reference operations could be performed by more primitive actions, a differing storage policy could overcome the need for this.

Additionally, the current scheme requires specification of a displacement for each stack frame operand, and that displacement field must be able to hold all variables, including full structures, of all stack frames, or some other overflow technique will be required.

There is also the fact that the large frame size resulting from including structured variables directly within the frame reduces the locality of store references, thereby reducing the efficiency of any cache memory.

A policy to address these issue develops the idea of fixed and variable parts of the stack frame, as outlined in the previous chapter. One view is that the fixed part is further divided into scalar and structure parts, with pointers in the scalar part to the structures. The second view is that structures are removed from the fixed part and placed at the bottom of the variable part, again pointed to by the fixed part.

Only the scalars, and pointers, are directly addressable by most instructions, allowing small displacements hence smaller instructions thus increasing locality and cache efficiency. The extra store reference required to access structures will be able to proceed at

maximum speed in the cache, and overall performance should improve.

An easy overflow strategy for the scalar part is to build a structure of scalars in the remainder of the fixed part that have overflowed.

By making the stack pointer and frame registers accessible as operands to normal instructions, the need for the remaining special stack and block structure handling operations is removed. They can be emulated by a couple of other operations with the advantage that all remaining instructions except the special 'Extend' are single cycle.

The potential losses of providing an extra addressing mode to access these and any other special purpose registers required can be overcome by reserving some displacement values which are detected by hardware and diverted to access these registers.

The instruction set has now reached the stage that, with the possible exception of splitting the branches into relational operators and 'If', no more sharing and emulation can be achieved without greatly increasing the dynamic instruction count, with little return in improved resource usage. This set is arguably then the most effective way of distributing resources between instructions for the support of *Primula*.

It must be noted that each instruction will require a single processor cycle, making hardware control easy and throughput rapid. The addition of any complex instructions to this core would require resources both for the operation and for the extra control hardware. It would therefore be difficult to justify without a high frequency of usage.

CALL d, t
    Store PC at d, execute from location t.
RETURN d
    Return control to address at d.

MOVE.B/MOVE.W/MOVE.L d_{data} | d_{dest}^, d_{dest} | d_{dest}^
    Move byte, word and double word of data.
GOTO <location>
    Switch execution to the given location.

ADD.I d_1, d_2, d_{result}
SUB.I d_1, d_2, d_{result}
BITOR d_1, d_2, d_{result}
BITAND d_1, d_2, d_{result}
BITNOT d_1, d_{result}

BREQ.I d_1, d_2, d_{result}
BRNEQ.I d_1, d_2, d_{result}
BRLE.I d_1, d_2, d_{result}
BRLEQ.I d_1, d_2, d_{result}
BRGT.I d_1, d_2, d_{result}
BRGEQ.I d_1, d_2, d_{result}

ASHIFTL d_1, d_{result}
ASHIFTR d_1, d_{result}
ROTL d_1, d_{result}
ROTR d_1, d_{result}

EXTEND

Details of the 'Extend' instruction and its use in implementing 'Real' and 'LongReal' operations are not important to this discussion and so are not included. In the above figures it is used to implement the 'Object_Access' operation by a software trap into the run-time system.

Frequencies 4.

As all structure copies must be done using a loop around a 'Move' instruction, the dynamic instruction frequencies are dependent on the average structure size. Little information is available on this, although average record field count and array sizes indicate a value of ten words would not be unreasonable. Each iteration of the copy involves a move, an addition and a comparison.

The stack allocation operation requires three moves and an add, the corresponding deallocation needing two moves and a subtract. In the frequent case of an in-line block which declares no local variables, however, these actions can be optimised out altogether by,

in effect, compiling the block at the lexical level of its enclosing block.

Hence only those stack management operations connected with named blocks need to expanded and included in the instruction count. Note also that the deallocate operation following a return from a procedure can reduced to a subtract. The number of named blocks can be found from the number of 'Return' operations.

The branch operations now take over the 'If's, with the difference in numbers of 'If's and relational operators in the above frequencies being primarily due to the use of boolean flags in loop conditions etc. Such flags are now tested using a relational type operator. To this figure must be added the tests required for the loops around structured operations.

Multiply emulation uses the instructions specified in the derivation above and also assumes the typical 32 by 32 bit case with average values as given above. Such emulation increases instruction counts considerably and can only be justified by the simplification of control circuitry. Note the replacement of many multiplications by a constant with shift and add sequences, by the compiler, will produce substantial benefits.

As noted, object access now go via the 'Extend' operation, as do real number operations.

Total Counts Fragment	Static	Dynamic
CALL	49	34
RETURN	9	34
MOVE.B/MOVE.W/MOVE.L	127	227
GOTO	21	18
ADD.I/SUB.I	66	411
BITOR/BITAND/BITNOT	-	-BREQ.I/BRNEQ.I/BRLE.I/
BRLEQ.I/BRGT.I/BRGEQ.I	56	427
ASHIFTL/ASHIFTR	7	184
ROTL.ROTR	7	184
EXTEND	6	19

Operand Access and Addressing Modes.

The above derivation has established the hardware instructions which must be provided and argued that the limited resources are better spent on operand storage than control circuitry. It also advocated addressing operands by a displacement from an implied stack frame base register, and further, that by accessing structures by pointers the maximum displacement could be reduced, allowing smaller instructions and increasing locality of accesses within the stack frame, hence increasing the efficiency of any cacheing system.

Katevenis [KATE] argues for a large register file as opposed to cache in the RISC II design. The author points out that whilst caches are dynamic and registers static, because of the block nature of caches, words may be wasted, and useful words overwritten. He also indicates the need for a full width adder, comparison hardware, and the physical size of a multi-port cache.

To reduce the overhead of register loads and stores on procedure call and return, RISC II provides register windows within a very large register file. Each window is equivalent to a stack frame, and is indexed by the displacements contained in the instructions. Procedure call and return simply involves changing the window base, with some

overlap for parameter passing, unless the file overflows.

Where a process switch is required, however, the large register file takes a long time to be dumped into store. This would is not acceptable in an a system based on communicating objects, as such switching is likely to occur often. For these reasons, a smaller amount of processor state is desirable. This implies keeping only the most frequently accessed values in the processor, ie. the current frame.

A register set would incur the overhead of multiple register loads and stores for each procedure call and process switch, taking time and extra instructions. Using a more dynamic approach would allow only those values necessary to be loaded and stored on each occasion, and would also spread the demands on store helping to avoid a bottleneck on each call and return.

The potential inefficiencies of a cache can be removed and the amount of hardware reduced to little more than required for a register based system, by the idea of a directly addressable cache. In this system, the address tag of the base of the cache block is the frame base register and the displacements are used to index into the block. The idea can also be viewed as a set of registers (the block) holding the current stack frame which would begin at the store address given by the frame base register.

It follows that the cache block, or register set, is made an integral part of the processor. Operand access is similar to a conventional register machine and just as quick, because the displacement can directly index the register set.

To implement the cache effect, each register has a validity flag associated with it. Each Read access must check this flag, and if not set, a store read is performed to the address found by adding the

displacement to the frame base register. The validity flag is then set. Each register also has a WriteBack flag, which is set when the register is written to, and reset only when main store has been updated with the new contents.

Any attempt to modify the frame base register causes execution to halt until all the WriteBack flags are reset, hence ensuring no values are lost. Such changes may be caused by procedure calls, process switches and exceptions. Some performance may be gained to allowing the user to reset the flags and so prevent the writing back of temporary values and local variables on return from a procedure call.

The pause in execution whilst registers are being written to store could be reduced by using a write through policy, where the cache immediately writes to store any value placed in a register. However, the total number of writes would increase dramatically, for example where a loop variable is constantly being incremented, increasing the likelihood of store contention and thus reducing performance. As noted above, allowing the user to mask the write back flags also significantly reduces store writes.

Note that because the cache is always an executing process' stack there is no need to compare general memory accesses to discover if the cache holds the required location. Such accesses are not allowed by *Primula* and are defined by the architecture not be be synchronised.

Having established this mechanism for local scalars and pointers to structures, a method for accessing non-locals and the structures themselves is required. These are less frequent, and generally more widespread in nature.

A base and displacement mode, where the base could be any register, would allow access to record structures, but other types require more complex action. Base and displacement could also be used

to index into other stack frames, the base being found from the display information.

Note, however, that unless the displacement field is larger than that used for local operand access, another method will be needed for large record sizes. With multiple operands per instruction, this implies that a more complex, variable length format would be required. Even if the displacements are limited, instructions will be longer because of the need to specify a base register.

An alternative approach is to provide 'Load' and 'Store' operations in a similar format to existing instructions, using local registers for the address and operand. A three operand format would allow base and displacement or indexed operation by adding two operands to form the address.

It is debatable which approach yields the shorter programs, as much depends on the implications for the instruction set as a whole and on the number of accesses needed. It is, however, evident that the second method provides a simpler format and so reduces decoding overhead overall.

Accesses to non-local, non-global variables, ie, those at an intermediate level in the block structure, is complex in a register file system, as it is impossible to tell at compile time where they are within the register file or if they have overflowed into main store. Only keeping the local frame within the processor eliminates this problem.

To speed access to frequently used structures, an external cache can be added between processor and main store. Depending on resources available, it could be placed on or off the processor chip with the on-chip variant giving faster access on a cache hit at the expense of limited size and the problems of loading a full block where the chip

pin count to main store is limited.

Pipelining and Instruction Cache.

Pipelining is a technique which splits instructions into sequential parts which can be performed by independent hardware units. Each unit receives information from the previous unit, processes it and passes information on to the next. Hence an instruction may begin execution as soon as its predecessor has completed its first stage, rather than when it has completely finished.

Increasing the number of stages in the pipeline allows greater overlap of operations at the expense of extra resources for pipeline control, and greater losses where branch type operations, and in particular conditional branches occur.

This last factor is highly significant, as the proportion of branches is likely to be around one in five, and has been the subject of much study [MCFA]. Many methods have been tried to reduce its effect, including fetching both possible routes, delaying the effect of the branch until after the next instruction, and more elaborate schemes involving remembering the route taken last time the branch was executed.

The latter is a dynamic version of the simple technique proposed here. Analysis of program control structures indicates that branches used to control loops usually cause execution of the loop to continue, whereas from Schmidz' figures show that only 28% of those controlling selection fail and allow control to continue normally. The compiler can use this information, with any other it can gather, to set a bit in the instruction to show the likely flow of control.

This can then be used to determine the next instruction to be fetched. Execution can proceed until the instruction requires to

modify something, at which point it must wait for the result of the branch. If the other route is required, the pipeline is emptied and the new instruction fetched. From the numbers of conditionals and loops, it would appear that the method will be over 75% effective, at little cost in pipeline resources.

The delayed branching technique employed by RISC I [KATE] saves a wasted cycle by always executing the instruction following the branch, and then allowing the branch to occur. Hence the ALU can do work regardless of the previous stages need to load another instruction. In general, if there are n pipeline stages before the ALU, a branch needs to be delayed by n operations to ensure the ALU is fully occupied.

Although [KATE] reports that the instruction following the branch did useful work on 80% of occasions, it will become more difficult to fill the extra cycles if the branch is further delayed.

To prevent errors caused by accessing a register before the previous instruction has written its new value, a reservation bit is associated with each register. These are set when an instruction is found that writes a value to that register, and reset when the write occurs. An instruction requiring a register value can only proceed when the reservation flag is not set.

The MIPS processor [HENN] omits this hardware and entrusts the compiler to determine the reservation needs, and arrange code such that the errors do not occur. This accurately follows the philosophy of doing as much work as possible at compile time, thereby minimising that required at run time. It has not been adopted here as the hardware requirements are quite simple and small, and by performing reservation beneath the visible level, there is opportunity to rearrange the pipeline to suit differing implementations whilst keeping the same object code. Compile time knowledge of the pipeline

will still help to improve performance, however.

With the simple instruction set suggested earlier, there appears to be little gain in pipelining the basic ALU operations. This leaves the task of matching the other pipeline stages to this speed, in order to maintain an even performance along the line. A four stage system, consisting of instruction fetch, operand fetch, ALU execution and store result should provide the necessary balance. Note that operand fetch will have to deal with two operands, and will also execute simultaneously with the store result stage. This produces a need for a multi-port register set.

Note that for best performance, all ALU operations must execute in a single cycle. This places a severe limit on the complexity of operation possible, and so works against the inclusion of instructions such as multiply and variable shifts.

The arrangement also imposes a demand that the instruction fetch unit can keep pace with the ALU as, without due care, this could form a bottleneck in the system. To help with this demand, an instruction cache is an essential, integral part of the processor.

Given fixed resources for a cache, the principal variable factors are the number, and therefore size, of the blocks, the addressing organisation and replacement policy. More, smaller blocks reduces unused locations at the expense of having less locations in total because of the amount of control circuitry. Fewer, larger blocks increase the number of cache words but increase the number of unused locations and reduce the ability to cover differing areas of store. An associative organisation increases flexibility but requires more control resources than a modulo method [POHM].

A block replacement strategy must also be decided upon and again the choice is not clear cut. The Least Recently Used policy seems

intuitively acceptable, given that the aim of cache is to dynamically follow program activity. It does, however, require more implementation hardware than some other strategies and has the problem that if a loop is just larger than the available cache then the next block is never in the cache and so performance degrades to that where there is no cache present. An interesting alternative is a random replacement method as used in the VAX [DIGI] architecture. Whilst this may occasionally swap out a required block, the performance will not degrade asymptotically in the manner of LRU.

The optimal arrangement is difficult to determine theoretically, and an analysis of possible systems is contained in the following chapter.

Instruction Format.

The instruction set established can be divided into two classes - operational and branches. The operational class generally require two sources and produce a result, whereas the branches have two sources and a destination for the branch.

From the source code survey it is evident that complex expressions are infrequent, and so the emphasis is on expressions with zero and one operator. A quick examination of likely code sequences shows that zero operand stack machines and single operand accumulator designs will produce longer code sequences for these operations, without savings in the number of operand specifiers.

```
Zero Operand    One Operand     Two Operand     Three Operand

C := A + B
PUSH A          LOAD A          MOVE A,C        ADD A,B,C
PUSH B          ADD B           ADD B,C
ADD             STORE C
POP C

C := A
PUSH A          LOAD A          MOVE A,C        ADD A,#0,C
POP C           STORE C
```

Whilst a design may incorporate different numbers of operands per instruction, this complicates decoding and effectively increases the number of instructions requiring resources. It is also difficult to achieve benefits by pipelining zero and one operand machines as the previous instruction must have written its result before the next can proceed.

The minimalist approach of the RISC design adopted the simple three operand format, where each instruction occupies a full 32 bit word. Except in the rare cases where large literals are included in the instruction, many bits[1] of the word are unused, suggesting that a more compact form may be possible.

It is contended here that the small number of instruction codes, together with the small displacements required to address the stack frame, would allow a two operand instruction to fit within a 16 bit half-word. The use of such a format may require the occasional extra instruction, but there will be very significant savings in code size and hence instruction fetch demand on store.

The restriction on field size limits the size of literal operands, but the source code survey indicates that the vast majority of constants are small values and so will fit in the available space.

Branch type operations still occupy a full word, with the first

---

[1] On RISC I eight bits are unused if the instruction specifies three register operands.

half having an identical format to the operational form and the second half specifying a relative offset, and the compiler determined likely direction of branch bit.

The exceptions to this are the jumps to absolute locations, which have the same function code field as other instructions, but the remainder of the word is an absolute address value in words. This is effectively multiplied by four to get the byte address, and it follows that procedures and other code requiring access via an absolute address must begin on a word boundary. This is simply to maximise the range of possible jumps.

Registers and Exceptions.

In addition to the general registers, or direct access cache, some special purpose registers are required to allow the architecture to function. To avoid extra addressing modes, these are accessed in a similar manner to the operand registers, but are unaffected by changes to the Frame Base Register. This has already been discussed as indicating the base of the direct access cache. Writing a value of this register causes execution to halt until any write backs are completed.

The stack pointer is used principally by software, with its hardware use being as a save area for the program counter and status register when an exception occurs. It could also be linked to memory management and protection if this were included in the design. To aid this type of facility, external signals should indicate program and data accesses, in read and write modes.

The Carry flag is made available as a register containing values zero or one depending on the last ALU operation. This allows it to be accessed in a similar manner to other operands, hence allowing easy

extended arithmetic without special instructions.

Overflow detection must be performed explicitly in software as it will be possible for the compiler to eliminate many of the checks that an hardware system would have to make. It also avoids the need for a method to switch off the hardware method where non twos complement arithmetic is being performed. The flag is held in the status register and can be tested by use of the 'BRIN' instruction.

The processor has a kernel state which provides a number of extra facilities. The Status Register contains flags to indicate mode (kernel or user), stop (stopped or running) and priority level (0 to 7), in addition to the branch and overflow bits, and may only be modified whilst in kernel mode. Exceptions are only allowed to proceed if they have a higher priority than the current level held in the status register. Level 7 is reserved for the processor reset and hardware failure exceptions. The system is based on the M68000 [MOTO].

When an exception occurs, the existing program counter and status register are stored on the stack, the program counter is set to zero and the status register is set to kernel mode at the priority of the exception. The Exception Value field of the status register holds further data as to the cause of the exception, which for external interrupts is read off the data bus.

This structure requires a first-level exception handler to examine the exception value to determine the exact nature of the exception and hence which handling routine to call. Return from interrupt is done by moving the status word from the stack to the status register and then executing a return instruction to reset the program counter.

Virtual storage is not included in this design, although it would present little difficulty as only a few instructions make main store

accesses. These are all sufficiently simple to allow them to be restarted if unable to complete because of a virtual store exception. To facilitate exception handling, the first-level handler must be kept in real store and the method of storing the PC and Status register on the user stack must be modified as it may be access to the stack page which has caused the exception.

There is no store protection in the current design, although there is no impediment to it being added. Within the *Primula* system the compiler can provide the checking necessary to prevent accidental store violations such as stack overflow and bad pointers.

Multi-Processing.

The use of a processor in a multi-processing system requires facilities to coordinate use of shared facilities such as store and communications. *Primula* requires a mechanism to synchronise object communications and dispatch objects on processors. Note also the need to access all names available in scope, which may poise a problem for some implementations.

In distributed systems, for example, such accesses may be implemented by message passing. Note that the language requires that such accesses are asynchronous to the object which contains the accessed item.

In general, a kernel type area is required to provide the synchronisation primitives for object communications and dispatching. Depending on the implementation this may provide simple semaphore operations, message passing, or directly implement the language actions. The minimum architectural provision for this is an indivisible 'TestAndSet' operation.

It is expected that the mapping of many objects to few processors

will be highly dynamic and therefore the communication links approach to inter-processor communication, as used by the Transputer [INMO], may not be suitable. The links would become saturated with the transfer of code and data in such a system although the method would yield excellent performance for a more static organisation.

With the processor design proposed here, the links approach could be implemented as an additional feature by memory mapping the channels.

The design itself is more suited to a common main store implementation as a tightly-coupled multi-processor. Store contention is reduced by extensive use of cache, both for instructions and data. In particular the architectural measures adopted to reduce main store accesses play a major part in reducing contention.

Synchronisation is not provided by the indivisible Test and Set operation because the use of cache allows more than one copy of the semaphore to be accessible from differing processors leading to inconsistency.

This could be overcome by reserving a certain address range within the architecture for semaphores and ensuring these are never cached. An alternative solution is to define a kernel object containing all the semaphores and associated operations. Synchronisation of access to the kernel object is controlled by the hardware using a kernel state control line. This is must be ∧ (in reset state) before a processor may enter kernel mode whereupon it is immediately set high. In effect it is a hardware version of a semaphore with the indivisible 'TestAndSet'.

External interrupts attempt to cause entry into kernel mode and are hence delayed when the kernel is already active. Software entry to the kernel is by the 'Extend' operation.

In the semaphore solution the processor status flag acts as a semaphore controlling access to the interrupt handler. External interrupts must perform a 'TestAndSet' type operation before being allowed to proceed.

Note that a high priority process may execute in user mode, thereby allowing exceptions to occur on other processors, or indeed on this one providing the exception has sufficiently high priority. There is a distinction between kernel mode and process priority.

The RASP Processor.

This section gives details of the actual *RASP* processor design, based on the architectural considerations already discussed.

In accordance with the integer type of *Primula* the word size is 32 bits, although there is limited support for handling 8 bit bytes. Due to cost and VLSI pin-out limitations, the data bus to main store is a single word wide. This does not, however, prevent the use of an external cache and a wider main store width.

Local operand storage is provided within the processor using the direct access cache method, described above. The code survey gave an average of 3.36 name declarations per block, so a total of 24 register locations, addressed as 0 to 23, should be adequate for almost all blocks. Hence a 5 bit address field is required.

Register addresses 24 to 31 are special purpose, and not a part of the cache system. Their functions are as follows :

24 : Reserved - currently unused.
25 : Reserved - currently unused.
26 : Dummy register - no reservations[12], value always zero.

[12] Hence avoids pipeline delays when the register is used as a dummy to fill the instruction format.

27 : Display register - available to user.
28 : Status Register - as described below.
29 : Stack Pointer - also used to save PC & Status.
30 : Frame Base - Address of direct access cache block.
31 : Carry Register - Carry from ALU in ls bit.

The status register is made up as follows, with numbers denoting bit positions increasing from right to left :

<31 Kernel mode if set, else user mode >
<30 Stopped if set, else executing >
<23:8 Exception value of latest exception >
<4:2 Current priority, 7 maximum >
<1 Overflow bit >
<0 Branch bit; set iff last branch was taken >

Note that the stack pointer and frame base register hold addresses in word form, the hardware multiplying them by four to obtain the byte address when these registers are used *implicitly* as opposed to directly in an instruction.

The Display register is provided as a user register which is not affected by the operand frame cacheing mechanism. It can hence be used to point to a display table, which is of course modified by procedure calls but not copied.

There are three instruction formats, the first requiring only a half-word, the remainder a whole word. Note that for cache efficiency, whole word instructions may span word boundaries.

F1 : <15:11 op_code>
<10:6 destination register; may also be a source>
<5 literal_flag; set for literal>
<4:0 source register or unsigned literal>

F2 : <31:27 op_code>
<26:22 source register>
<21 literal flag; set for literal>
<20:16 source register or unsigned literal>
<15 likely direction flag; set for branch>
<14:0 relative offset to PC. Twos complement form>

F3 : <31:27 op_code>
<26:0 absolute value - addresses in word form>

Most of the instructions are taken directly from the instruction

set derivation, but there are some omissions and modifications. The rotate left is omitted because the simple carry register access allows it to be simulated by arithmetic shift left followed by adding in the carry. All other shift type operations can be manufactured in a similar manner.

Two forms of Call are provided; one for normal calls to absolute locations and the other for calls to a variable held in a register. There are also two forms of Goto; one to an absolute address and the other an unconditional relative branch. The latter also performs an addition, which could save an instruction in some loop forms or become a no-op by using the dummy register.

To increase the range of addressing, the direct Goto and Call instructions jump to a word, rather than byte, address. This implies that the target code of these instructions must be on a word boundary.

Call and Return use the stack for the stored PC value, as storing it in a register would cause it to be immediately written out when the new frame is set up. Additionally to allow a choice of register would mean a change to the standard call format, F3. As the stack pointer holds word addresses there are no problems with the alignment of the stored registers on the stack.

Attempting to return to an address of zero will cause an exception, and this can be used to signal the end of a *Primula* object access. As no special instruction is required, the code can be identical to that of a normal procedure, avoiding the need for an explicit check.

Stack frame storage allocation is performed by a series of simple instructions acting on the stack pointer and frame base registers. Referencing of operands through scope is also performed by a series of instructions, allowing the compiler a choice of strategies. The

Display register is included to help with this.

As noted above, the generalised move operation has been replaced by 'Load' and 'Store' for main store operations. These use byte addresses and so an alignment exception may occur if the address is not word aligned. To ease string handling in particular, 'LoadB' and 'StoreB' variants are provided, which access the specific store byte and the least significant byte of the register. Loads clear the remainder of the destination register. Careful use of the byte instructions with regard to the data cache can produce highly efficient string searches etc.

The 'Extend' operation causes an exception and hence provides the ability to effectively extend the instruction set by providing hidden routines to service these exceptions and enter kernel mode for general system calls. They could also be used to trigger the use of numeric co-processors for example.

The 'BrIn' operation is included to support 'In' type comparisons, principally because of a gap in the instruction codes which should be a branch operation. It also allows quick testing of bits in the status register etc.

'Stop' and 'Reset' are only available in kernel mode (in user mode these bit patterns will cause an illegal instruction exception) and are included in F1 to ease decoding, although the operand fields are ignored by the instruction.

'Push' and 'Const' make the stacking of parameters and use of large constants much easier. Whilst the use of the stack by the 'Const' operation is not desirable in many cases it does allow the specification of a 27 bit constant within the existing instruction format. The 'ASL2' instruction is included as a quick way of changing word to byte addresses.

The function of the 'MWBF', Mask Write Back Flags, instruction is explained above whilst describing the directly addressable cache. Suffice to say here that the direct access cache write back flags are bitwise anded with the operand value with the least significant operand bit corresponding to cache location zero.

The actual operations available are the following :

F1.

```
MOVE    <s> <d> ; d := s
LOAD    <s> <d> ; d := s^ ( ls 2 bits of s ignored )
STORE   <s> <d> ; d^ := s ( ls 2 bits of d ignored )
LOADB   <s> <d> ; d := s^ ( single byte )
STOREB  <s> <d> ; d^ := s & 255 ( single byte )

ADD     <s> <d> ; d := d + s
SUB     <s> <d> ; d := d - s
OR      <s> <d> ; d := d ! s
AND     <s> <d> ; d := d & s
NOT     <s> <d> ; d := ~s

ASL     <s> <d> ; Shift s left one bit, ls bit := 0,
                    carry := ms bit, store in d.
ASR     <s> <d> ; Shift s right one bit, sign extended,
                    store in d.
ROR     <s> <d> ; Shift s right one bit, ms bit := carry,
                    carry := ls bit, store in d.

CALLR   <s> <d> ; SP := SP + 4, SP^ := PC; PC := s.
RET     <s> <d> ; PC := SP^, SP := SP - 4.
                  If PC = 0 cause exception.
EXTEND  <s> <d> ; Cause exception s.
ASL2    <s> <d> ; ASL followed by ASL.
PUSH    <s> <d> ; SP := SP + 4; SP^ := s.
STOP    <s> <d> ; Stop execution; wait for exception.
RESET   <s> <d> ; Cause reset exception.
```

F2.

```
BRA   <s> <d> <relative> ; d := d + s; Branch PC relative.
BREQ  <s> <d> <relative> ; If d - s = 0 then branch.
BRNEQ <s> <d> <relative> ; If d - s <> 0 then branch.
BRLT  <s> <d> <relative> ; If d - s > 0 then branch.
BRLEQ <s> <d> <relative> ; If d - s >= 0 then branch.
BRGT  <s> <d> <relative> ; If d - s < 0 then branch.
BRGEQ <s> <d> <relative> ; If d - s <= 0 then branch.
BRIN  <s> <d> <relative> ; If d & ( ~s ) = 0 then branch.
```

Symbols '>' and '<' are used at the start of the relative field to set or reset the Branch Expected bit respectively.

F3.

```
GOTO  <location> ; PC := location * 4.
CALL  <location> ; SP := SP + 4, SP^ := PC, PC := loc * 4.
CONST <value>    ; SP := SP + 4; SP^ := value.
MWBF  <value>    ; WriteBackRegister := WriteBackRegister
                     & value.
```

A four stage pipeline is implemented with instruction fetch, operand fetch, execute and store result stages. To allow simultaneous fetching of operands and storing of results, a multi-port register system will be needed. Each register, including the special registers, has a reservation flag operating as described earlier. Additionally, each pipeline stage has available and done flags to communicate with the previous and following stages respectively.

The instruction fetch stage contains an 'AdvancedPC' register which holds the address of the instruction currently being fetched. When a 'Call' or 'Goto' is found, the new address is calculated and the pipeline continues uninterrupted. On branch instructions, the likely flag is used to choose the path on which to set the AdvancedPC. If the flag is set, the branching option is taken. If the guess proves to be incorrect, the pipeline is flushed, with the wrong instruction only having progressed as far as the operand fetch stage.

The instruction cache is a major part of the fetch stage, and as noted earlier its ideal arrangement is difficult to determine theoretically. As a starting point, addressing is associative to select the block, with the least significant bits being used to select a word within that block. This is a similar scheme to that used on the Cray-1 [RUSS], and provides a high degree of flexibility.

Sample code seems to indicate that the tightest loops will occupy between 4 and 8 words each. Hence a block size less than this would have little benefit for the cost, and larger would increase unused information and decrease flexibility. As a first attempt therefore

four blocks of four words each seems a reasonable compromise.

As the cache has only a single word width access to main store, each word of the cache has an individual validity bit. If the required block is available, but the validity bit for the individual word is not set, a store read is made and the bit is set.

If the block is not available, then one block must be chosen for replacement. Again as an initial choice, this is done on a Least Recently Used basis, implemented by every block having a two bit counter associated with it. When a block is accessed, its count is set to zero and any counts less than its old value are incremented. When a valid block must be replaced, the chosen block will have a count of three. Note that the Cray-1 has a cyclical replacement strategy. Random replacement was not chosen as it was felt that with only four blocks it would make too many bad choices.

The current design does not include a large general cache as resources may be limited, and in any event it would need to be organised like the instruction cache in that only a single word could be accessed from store at once. Data accesses tend to be much less iterative than instruction accesses and so the performance of a data cache is restricted as no pre-loading of whole cache blocks can occur.

The exception to this is in string operations, where sequential access is required on a byte basis, hence causing up to four accesses to the same word. Such operations can be made much more effective by providing a single word associative cache. To allow efficient string comparisons etc. four such blocks are provided. A similar replacement strategy to the instruction cache is also used here, and a write back on replacement policy similar to the direct access cache is adopted to minimise store traffic.

Pipeline stages and the direct access cache require access to main store, and this is provided through the Store Access unit, which is configured in a similar manner to a pipeline stage with input and output buffers.

To access store, a requestor must wait until the input buffer flag is not set, and may then set this flag, and place data in the buffer. At the end of the clock cycle, providing the Access unit is free, this is transferred to the unit itself which then begins operation.

When complete, the unit sets a flag and at the end of the clock cycle, providing the output buffer is free, as indicated by its flag, any results are transferred. The requesting unit may then pick up these results during the following cycle, remembering to release the output buffer reserved flag.

The initial state, mentioned by the RESET instruction, is kernel mode, tracing off, running, priority 7, program counter set to zero, other registers undefined.

Further Considerations.

It has been noted that the cache configurations are initial attempts and must be checked by practical tests and possibly modified as a result. This is done in the following chapter.

Also worthy of investigation, although there is insufficient time here, would be different pipeline organisations, changes in the 'extra' instructions added to the basic derivation, and adopting a more conventional branch on status flag approach, possibly with a delayed branch as with RISC-I.

The actions and facilities for exception handling also need further clarification and the writing of some low-level software would

give a good guide as to the actions required from the hardware.

Evaluation.

Compiler Construction.

The principal aims of the compiler are to be experimental vehicle to discover the compilation complexity of *Primula*, to uncover any apparent language anomalies and to determine the difficulties of the *RASP* architecture as a target machine. It is not intended that it should be a highly optimising compiler but its construction should give an idea as to the complexities of producing efficient code for the processor.

With a view to the overall simplicity of the program, a single-pass design is adopted using a recursive descent parser. The lexical analyser, semantic checks and code generation procedures are all driven by calls made from the parser.

Pascal was used for the first implementation but the recent availability of Modula-2 means this will be used for future versions.

Symbol Table and Associated Data Structures.

For efficient use of storage a large character array is used to store all identifiers encountered in the program, with the length stored as the first byte by use of a type cast operation. Identifiers are then referenced by their index within the array.

To provide fast look-up of identifiers a hash table is used, the key being the product of the ordinal character values of the identifier, modulo the table size. Overflow is dealt with by the chaining method, where each table entry is the head of a dynamic linked list.

On insertion, identifiers are checked against all those in the chain to prevent duplicates being stored in the character array. Hence

the indexes into this array have a one-to-one mapping to the source program identifiers.

Program structure, and hence identifier scope, is kept by a dynamic structure of block records. Each block has pointers to the enclosing block, to its parent block (if any) and to a list of identifiers declared within the block. Searching through the structure for an identifier uses first the list of parents and then the list of enclosing blocks.

Symbol table entries themselves are pointed to from the hash table entries described above and contain all the information connected with the identifier.

Type information is kept in the form of a tree structure as generated by the parsing of the declaration. As declaration of equivalent types, of the form 'type1 = type2' is permitted by the language, a procedure is called before almost all uses of type information in order to reduce the definition to the original declaration.

All procedure and function declarations have types associated with them in order to simplify assignment to variables and calls to constant or variable procedures. A special type class is introduced for those compiler defined procedures and functions which take arguments of varying types.

A point of note here is that the 'CAST' procedure returns its result as a reference parameter rather than as a function result because it would not always possible to determine the required return type of the function.

Instantiation of a generic type results in the building of a completely new structure by copying the generic type whilst making the appropriate substitutions. It was felt this offered a simpler and more

time efficient result than storing the instantiation information itself and connecting it with the generic type on each use.

Program Analysis.

As noted above this is controlled by a recursive descent parser using single symbol look-ahead. The lexical analyser is called as a function via a number of small utility procedures which check for given sets of legal tokens etc. Error recovery is through the use of sets of tokens which may legally follow the current construct. On an error, the input is skipped until a token within the following set is discovered. The method is similar to that used by Brinch-Hansen for Pascal [BRIN].

Within the lexical analyser, reserved words are stored in an ordered array and searched by the binary-chop method. Use of the hash table was considered and will be adopted in future versions but here it was thought that a separate structure could ease development by reducing dependencies between the functional parts. Single character look-ahead is used throughout the analyser, which is implemented in a standard manner using a 'Case' statement within a loop which terminates when a valid token is found.

Error reporting and the production of the listing file are handled from within the lexical analyser code. Input and output is buffered so that the simple I/O of this version can be quickly replaced by more direct and efficient calls to read disk sectors etc.

The majority of the semantic analysis procedures simply check that some condition is met by the information stored within the symbol table and associated data structures.

The Abstract Machine.

The earliest version of the compiler was designed to produce an abstract machine code for a number of reasons.

Firstly, the use of a suitable abstract machine would simplify code generation and hence allow greater concentration on the problems created by the source language definition. Secondly, the abstract machine could be used as a vehicle for transporting the language to a variety of processor systems in a similar manner to the Pascal P-Machine [PEMB]. An emulator for the abstract machine written in some common language such as C or Pascal would allow execution of *Primula* programs, albeit at reduced efficiency, on any systems that supported the language of the emulator.

In the *RASP* target compiler, the abstract machine forms an intermediate stage between source language and target code, hence providing a degree of independence between the analysis and synthesis stages of the full compiler.

Details of the abstract machine code and actions are given in the appendix. It is essentially a stack based architecture with a number of special operations which effectively implement the *Primula* run-time system ie. object synchronisation and communication.

The organisation of the stack, and in particular the stack frames, is as outlined in the chapter on the execution requirements of *Primula*. To briefly recap, each frame has a fixed part at the bottom containing fixed size elements and a dynamic part containing dynamic and generic structures, pointed to from the fixed part.

Target Code.

The initial aim was to produce simple, though not highly efficient code by a simplistic expansion of the abstract machine operations. Simple optimisations would then be considered for inclusion.

It could be performed by an independent program but this approach runs into difficulties when mapping the abstract stack frame onto the fixed size, directly addressable area of the target *RASP* architecture. Optimisation also becomes a problem as the information collected by the compiler is no longer available.

An approach to overcome this would be to perform the expansion as a second compiler pass, hence making all symbol table data available. The method chosen, however, was to perform the expansion within the single pass by expanding the abstract code as soon as it is produced.

To improve utilisation of the directly addressable area of the architecture the stack frame organisation is modified as again described in the 'Execution Requirements of *Primula*'. This puts all structures within the dynamic part of the frame hence limiting the fixed part to scalars and pointers. Mapping to the architecture is then a matter of placing the fixed part in the directly addressable area and the dynamic part on the main store stack.

In this version of the compiler an error is signalled if the size of the fixed part exceeds the size of the directly addressable area of the processor. A production compiler could signal this as a warning and incorporate an overflow strategy which places extra fixed part entries onto the main store stack, ie. effectively within the dynamic part of the stack frame.

No optimisations are included, although they could be added if required : the effectiveness of various techniques is discussed below.

Higher level optimisations, such as common-expression elimination, strength-reduction etc, are widely discussed in compiler literature [TREM] [AHOV] and therefore do not form a part of this project. Some of the actions of such optimisations can be simulated, however, by careful construction of the source program.

Run-time Kernel.

The current kernel is for a uni-processor system as provided by the emulator described below. It could, however, be easily extended for a homogeneous multi-processor by modification to the scheduler and dispatcher procedures.

Operation is similar to the actions described in the run-time requirements chapter. As the design is primarily for a uni-processor there is no contention disadvantage in providing all synchronisation services within the kernel and so this is the method adopted.

Service calls available to the user are to create a new object, call an object, fail a guard clause or end execution of a main body or access. As the return at the end of a public access could be either a normal procedure return or the end of an access the architectural facility to trap a return to address zero is used to detect the latter case and enter the kernel. If not available this would have to be replaced by an explicit test.

No protection is provided between user objects as the processor, as described so far, does not provide hardware support for, in particular, address checking. To allow a measure of protection, however, use is made of kernel mode with kernel being entered by the 'Extend' operation.

For simplicity, storage management is also included within the kernel with calls to allocate and free areas of memory. There is

currently no garbage collection within the kernel and so it relies on the user objects freeing store as and when they can. The problems and possible algorithms for dynamic storage management are widely known and discussed in [JONK].

The principal kernel data structure is a list of user objects each containing the necessary information on that object. This includes its synchronisation flag, base address, current status, processor state and waiting access queue etc, in addition to pointers used within the kernel to form the various queues of processes / objects.

Emulator.

There are again several purposes behind the construction of the emulator. Firstly, it is to allow the execution of *RASP* programs; secondly to monitor the activities of the processor during execution and allow experimental modifications to affect the performance of the processor; and thirdly to act as a software definition of the processor itself.

An initial simulator principally served the first purpose and therefore also the third in that it defined the user visible aspects of the processor. It was designed simply to execute *RASP* programs at a reasonable speed, its structure consisting of an outer loop which tests the *RASP* status register and exits only when the 'Stop' bit has been set. Within this loop is a 'Case' statement, each branch of which executes a particular instruction.

Input and output is performed by trapping read and write operations on a fixed memory location and then interacting with the user via screen and keyboard using Pascal I/O. There are no other debugging facilities.

The second emulator, also written in Pascal, is much more sophisticated and allows program monitoring and debugging as well as monitoring the actions of all parts of the processor and memory.

Each *RASP* clock pulse is simulated by a call to the main emulating procedure, which in turn calls all the procedures which simulate the action of a particular piece of the hardware. Every concurrently acting piece of hardware ie. every pipeline stage, memory etc, has its own procedure.

As with the hardware, each of these procedures acts on input buffers and produces output buffers,the validity of the buffers being indicated by flags. To allow procedures to 'wait' on a flag somewhere within their code the following structure is adopted.

Each procedure may only wait at a finite number of points, these being where it waits on a flag condition or, in effect, waits for the next clock pulse. The current point at which a procedure is waiting is indicated by a state variable associated with that procedure.

A label exists in the procedure code at each wait point and also one at the exit point of the procedure. On entry to the procedure, at each clock tick, a 'Case' statement tests the state variable and causes a jump to the appropriate label. The wait condition is then tested and if all is well execution can proceed normally. If the condition fails, however, the state variable is set to indicate this wait position and a jump made to the end point of the procedure.

When the end of the procedure code is reached normally, ie. at the point immediately before the end point label, the state variable is set to indicate the all work possible within this clock tick has been completed and hence the procedure is waiting for the next clock tick in order to recommence at the top of the code.

```
PROCEDURE PipeStage1;
BEGIN
  CASE Stage1.State OF
    WaitClock   : Goto 1;
    WaitInBuff  : Goto 2;
    WaitOutBuff : Goto 3;
  END;
1:
  ....
2:
  IF Stage1.InBuffEmpty THEN
    Stage1.State := WaitInBuff;
    Goto 9;
  END;
  ....
3:
  IF Stage1.OutBuffFull THEN
    State1.State := WaitOutBuff;
    Goto 9;
  END;
  ....                              (* OutBuff := f( InBuff ) *)
  Stage1.InBuffEmpty := true;
  Stage1.OutBuffFull := true;
  Stage1.State := WaitClock;
9:
END;
```

After the simulation procedures have all been called, the remaining action on each clock cycle is to examine the flags associated with the buffers and shift data along the pipeline from full output buffers to empty input buffers, updating the flags as appropriate.

Along with this quite detailed emulation, various performance information is collected within the procedures and a graphical display is used to monitor and interact with the processor.

This is constructed using the GEM system [BALM] on the Atari-ST. The user is presented with three windows - processor, memory and statistics - each of which may be moved, scrolled and sized as required. The processor window itself is split into sections for instruction cache, pipeline, directly addressable cache etc.

Values within the system may be modified by scrolling to find the appropriate location eg. scrolling through memory, and then selecting the item by clicking with the mouse. A GEM Form is then displayed,

prompting for the new value.

Pull-down menus allow the loading and storing of memory images, processor initialisation and selection of one of three execution modes. These allow the emulator to be stepped through with the mouse button acting as a clock signal, the emulator to run whilst continuously updating the display, or to run as fast as possible without updating the display. A mode can be halted at any time by selecting any other mode.

The later availability of Modula-2 was the major reason for the construction of a third emulator, closely based on the second. The major differences in the emulation are that the procedures simulating the 'active' units of the processor, ie. pipeline stages, memory etc, are constructed as Modula-2 coroutines and so the position at which a unit is waiting for some condition is held implicitly as the current PC of the coroutine.

The separate compilation features of Modula-2 also allows easy modification of processor parameters and addition of various features. This is important for the second purpose of building the emulator and was of great help in producing the performance figures given below.

Implementation Performance.

It is difficult and unreliable to assess the performance of any system without implementing a reasonably large sample of the type of tasks it is expected to perform. This is clearly impractical within the scope of this thesis.

The derivation of the architecture from program behaviour holds out the hope of good, if not optimal, performance. It is possible to test aspects of the design, and hence attempt to build up a picture of overall performance, by the use of benchmarks which exercise

particular aspects of the system.

As part of the design criteria, stemming from the program analysis, certain features were noted to be of greater importance to the performance and hence it is these that must be tested. In particular, procedure calls and local data access were noted as crucial.

Many popular benchmarks are based arpund numerical processing and lay particular emphasis on floating point operations. In the original design criteria it was established that this was a specialised area for which co-processors would be used and therefore floating point operations were not considered important. Hence there is no evaluation of them here.

Benchmarks.

Four small programs have been chosen to exercise various points of the system. The implementation listings are contained in the appendix.

The size of the data structures used are principally determined by the constraints imposed by the compiler or assembler. The use of larger structures would require special techniques which would not therefore correspond to common program activity.

A) Sieve- This is the well-known sieve to find prime numbers. It contains nested loops, simple arithmetic and array usage.
The sieve contains 8192 elements.

B) Gauss- Solving equations with n unknowns by the Gaussian elimination technique. This contains many procedure calls to arithmetic routines[13].
There are 6 unknowns in the tests.

C) QSort- Sorting numbers by the QuickSort technique. The method contains a number of very tight loops controlled by numeric comparison and a high degree of recursion.
There are 2192 thirty-two bit integers to sort.

D) Search- Searching for a pattern within a character string using the Boyer-Moore algorithm. Again tests loops and is highly character array intensive.
The search is for an 8 character pattern in a 910 character string.

To provide a measure of the effectiveness of the architecture the above algorithms have been written in Modula-2 and M68000 assembler. The Modula-2 was then compiled for M68000 and simplistically hand-compiled for *RASP*. The language was chosen as a recognised efficient high-level language and the micro-processor as a good, popular, design.

Primula was not used as this would have involved producing a run-time kernel for the M68000 and because the benchmarks are not of sufficient complexity to make general use of objects a sensible proposition. As the benchmarks do not test the more advanced facilities of *Primula* it is expected that there will be little

[13] Values are represented as integer numerator and denominator to provide real-number type handling using only integer arithmetic.

difference between performance using Modula-2 or *Primula* on either architecture.

Timings for Modula-2 are given for the standard version which includes range checks, stack checks etc., and for an optimised version which excludes such checks. The hand-compiled *RASP* versions also exclude such checks.

Note that the timings for M68000 will contain a small overhead incurred from operating system activities. This is confined to such items as updating the system clocks and so will be an almost negligible amount in the context of these general comparisons.

Where input data has to be copied in on each iteration, a run was made performing only the copy and not the benchmark itself and the time for this subtracted from the proper run-time to obtain the benchmark times given below. This was necessary for the Gauss (copy time 0.4s for 1000 iterations) and QSort (copy time 0.8s for 100 iterations) programs. Copy time differences between the three forms were negligible.

Benchmark	Iterations		Times in S		
			M2/68K Checks	M2/68K No Checks	ASM/68K
Sieve	*	1000	45	40	23
Gauss	*	1000	40	39	15
QSort	*	100	90	80	40
Search	*	1000	22	17	8

It is immediately apparent that on the M68000 processor the use of Modula-2 rather than assembler approximately doubles the execution time of these benchmarks. Analysis of the compiled code and the compilers register allocation strategy indicates that major reasons for this are the increase in the loading and storing of registers from

/ to the stack frame in memory and full recalculation of array element addresses in loops where an increment would have sufficed.

There are few differences in the actual instructions used to perform the operations, although the assembler version does use a much simpler procedure / subroutine call code sequence.

The increased Modula-2 / Assembler time ratio for the Gaussian benchmark is caused by the register allocation noted above in the arithmetic subroutines and by the more effective exploitation of the characteristics of the M68000 DIV and MUL instructions by the assembler code.

This is also borne out by the differences in Modula-2 timings with and without the checking code, which is dominated by the checking of array bounds. The Gaussian benchmark saves little by the elimination of checks as the majority of its time is spent in the number handling subroutines. Other benchmarks which use arrays more intensively save around just over 10%.

Basic *RASP* Performance.

This section begins a series of tests of the *RASP* processor, beginning with the minimum features necessary to provide the user-visible architecture and subsequently adding caches, pipelining etc, to show the effect these have on the performance. All M68000 figures are for Modula-2 benchmarks without range-checks and all *RASP* figures are from the hand compiled Modula-2.

In order to produce actual timings to compare with the M68000 figures above, it is necessary to assume speeds for memory accesses and processor cycles. It has been decided to use generally conservative figures in order to show that it is the processor design, rather than simply faster memory and/or processor speeds, which is

responsible for the results.

In addition, obtaining good performance with slow memory bodes well for multi-processor performance when using faster memory as the load each processor places on store will have been reduced and so there will be less contention.

To put the figures into perspective, the M68000 used in the tests has an 8MHz clock rate and can perform a simple operation such as the addition of two registers in no more cycles than it requires to fetch the instruction from memory. The memory itself is fast enough to require only two clock cycles for an access, ie. a memory access rate of 4MHz.

With this in mind, the instruction frequency for the initial, non-pipelined processor has been set to 4MHz implying that the total execution time for each instruction, excluding time for memory accesses, is 0.25µS. The memory access frequency is set to only 1MHz, for the reasons outlined above. An implication of this is that effective use of memory is vital to performance.

For comparison, the table below also gives execution times when a 4MHz main memory (access time of 0.25µS) is used and shows the extent to which the performance is memory limited.

The table also lists the number of instruction reads from main store, the number of data accesses to main store and the number of instructions executed. Times for the M68000 are taken from the table above (Modula-2 without checks), divided to produce a time for a single iteration.

Benchmark	I_{Read}	D_{Read}	D_{Write}	I_{Exec}	T_{1MHz}	T_{4MHz}	T_{M68000}
Sieve	18861	3742	3698	24173	32mS	13mS	40mS
Gauss	10152	3189	2455	14272	19mS	8mS	39mS
QSort	334610	143992	31910	535364	644mS	261mS	800mS
Search	8061	2524	313	11653	14mS	6mS	17mS

The difference between the *RASP* processor times for 1MHz and 4MHz memories clearly shows the extent to which the former are memory dominated. With the 4MHz memory, there is an approximate balance between memory and execution times and so further increases in memory speed would be of limited value and very costly.

Comparison with the M68000 timings shows that even with the 1MHz memory, the *RASP* shows a 20% advantage and so it is clear that it is the architecture of the processor which is producing that performance, rather than just clocking speed. It is also worthy of note that the 4MHz timings are quicker than those for the M68000 hand-coded assembler.

Instruction Cache.

The table above also shows that it is the instruction fetch part of the memory accesses which dominate the timings and hence it is logical and cost-effective to begin there to gain extra performance. In addition, of course, measures such as pipelining would have no effect because the instruction fetch would hold up the remainder of the pipe. As noted above, all the following timings refer to the 1MHz memory.

The initial design, as described earlier, proposed a 16 word cache comprised of four blocks and using a Least Recently Used replacement policy. In order to make a more informed choice of cache configuration, results were also obtained for a number of block sizes,

a larger (32 word) cache size, and a Random replacement policy.

Time constraints prevent further investigation of cache configurations and in particular exploration of the behaviour of other replacement strategies. These figures do, however, give an indication of the likely effects of changes to the cache.

The only figures to be modified from the above table are I_{Read} and the resultant timings. The percentages given in the table are cache hit rates found by subtracting the I_{Read} figure with the cache installed from the initial value above and then giving it as a percentage of this value.

The headings indicate the number of blocks followed by the block size.

	Least Recently Used					Random Replacement				
Benchmark	8/2	4/4	2/8	8/4	4/8	8/2	4/4	2/8	8/4	4/8
Sieve	75	72	72	78	80	76	74	69	78	81
Gauss	48	47	47	55	50	50	48	47	52	51
QSort	40	40	40	74	72	45	41	37	73	71
String	89	89	89	99	99	89	88	90	98	98

It is immediately evident that the String and Sieve benchmarks, with their predominating tight iterative loops, are much more susceptible to cacheing techniques than the larger QSort or Gauss with its many procedure calls to the small numerical routines.

The benefits of an altogether larger cache are demonstrated by the improvement in the QSort figures when using the 32 word cache instead of the original 16 words. Generally, however, successive increases in cache size will produce smaller improvements in overall performance and hence a drop in cost-effectiveness. This must be balanced against the cost of having the processor idle whilst

instructions are fetched from memory.

In practice, the amount of cache which can be cheaply provided may well be determined by chip size and layout rather than more theoretical performance considerations. As cache is user transparent, however, changes in cache size brought about through changes in chip technology or simply through use of larger or extra chips could form the basis of a range of processor configurations.

It is also important to note that as cache size increases it becomes increasingly difficult to obtain the required access times because of the physically longer paths involved.

A similar situation exists when considering block sizes for a given total cache size. Increasing the number of blocks generally gives an increase in performance but, depending on the replacement strategy, may require a significant amount of extra hardware. It is noticeable that the Random Replacement strategy benefits from a larger number of blocks, as the probability of the 'wrong' one being replaced is reduced.

The amount of hardware required per block suggests that the 4 blocks of 8 words cache may be only a fraction more costly than the smaller 8 blocks of 2 words variety. Indeed, with a Least Recently Used policy it may well be cheaper because of the extra counters and comparators needed for each block.

It appears that only smaller caches with few blocks will benefit from a LRU policy for replacement and that the hardware cost of such a policy is such that for larger numbers of blocks, adopting a pseudo random replacement strategy and using the released resources to increase cache size will provide better results.

To provide a base-line for performance probably the most cost-effective choice is the 4 blocks of 4 words with Random Replacement

configuration and this is used in all subsequent tests. Execution times for this form and for the more expensive 8 blocks of 4 words, also with Random Replacement, are given in the table below. Timings assume that if the instruction is in cache then it can be fetched and executed within the same 0.25μS processor cycle time.

Benchmark	$I_{A/4}$	$I_{B/4}$	$D_{B/W}$	I_{Even}	T_{MC}	$T_{A/4}$	$T_{B/4}$	T_{HARDD}
Sieve	4983	2179	7440	24173	32mS	18mS	16mS	40mS
Gauss	5244	4840	5644	14272	19mS	14mS	14mS	39mS
QSort	197832	88677	175902	535364	644mS	508mS	398mS	800mS
Search	990	134	2837	11653	14mS	8mS	6mS	17mS

The four block cache has reduced execution times by about a quarter in the worst case, with every benchmark except QSort executing in under half the time required by the M68000. The larger cache allows QSort to achieve this landmark whilst giving smaller increases for the others.

It is evident from these timings that all the benchmarks, except perhaps QSort, now have a factor other than instruction fetch as the major constituent of execution time. Hence the point noted above with regard to the value of larger caches can be made more strongly in that it is evident that larger instruction caches will not, for the most part, produce significant performance increases without first giving attention to other aspects of the design.

A further measure of cache effectiveness is the amount by which timings approach those obtained using the 4MHz memory in the previous table.

Pipelining and Pipeline Implementation.

The next architectural feature to be included in the emulation is the four stage pipeline consisting of Instruction Fetch, Decode, Execute and Store Result stages. It is synchronously organised such that each stage has input and output buffers, with associated flags, which it uses during the operational part of each clock cycle. At the end of each cycle, the flags are examined and where a stage has completed and the next is free then the information is forwarded.

In order to coordinate accesses to memory, there is a also a memory access unit which can be utilised by any of the four in-line stages. It uses a similar system of buffers and flags to coordinate its use and is hence also synchronous. This implies that a unit wishing to use memory must first check that the buffers are available and then deposit information required for the access in them.

At the end of the clock cycle, and when the memory unit itself is free, this data will be transferred to the input buffers of the memory unit. A similar mechanism operates on the completion of a memory access, a unit requiring data having to wait until the beginning of the next clock cycle after the memory access unit has completed its activities.

One result of this is that, for example, an instruction fetch from main memory effectively requires a extra cycle in order to complete. The remainder of the cycle after the fetch unit requires the access, and the remainder of the cycle when it completes processing having received the data, are wasted.

This is a part of the more general problem of resource reservation. Within the *RASP* this issue also arises where an instruction requires the results of a previous instruction before it can commence. Hence a gap appears in the pipeline and performance is

reduced.

As the *RASP* is a simple processor, the implementation of reservations is also simple and consists of a flag for each register which is tested whenever an instruction requires that register for its execution. Note that this includes implicit use of registers such as the stack pointer in 'PUSH' instructions.

If the flag is set, the register is reserved and the instruction must wait. Otherwise it may proceed and sets the reservation flag on any registers that it may change during its execution. This is all performed within the Decode stage. Reservation flags are reset when an instruction has written its result to them, either in the Execute or Store stages.

Another commonly analysed problem with pipelining is that the pipe must be flushed when a change of control flow occurs and hence the smooth flow of execution is interrupted. The effects of this can be reduced by techniques such as Branch Prediction as outlined previously in the architecture design.

For these initial figures, however, no branch prediction was used at all, even for calls and unconditional branches. The fetch unit simply carries on fetching instructions sequentially until the pipeline is flushed. This is signalled by the execution unit and affects both fetch and decode stages.

Two methods of flushing were tried and the comparative results are shown below. The first method is that the stages only check for the flush signal from the execute unit at the completion of each operation. Hence any memory accesses or other actions within the units will complete, even if not started at the moment the flush signal is made.

The second, and intuitively more effective method, is to allow

the flush signal to cause an immediate reset of the fetch and decode units, hence saving the execution of any actions not yet performed. Note though that the memory access unit always fully completes even though the data may no longer be required. This simplifies the interface to main memory and so ensures that no special memory characteristics are required.

The results include the use of the instruction cache as outlined above. Note that the figures for D_{write} and I_{Exec} are unchanged from the non-pipelined version. As there are now four stages to the execution of a single instruction, it follows that to maintain the same execution time for a single instruction the clock period must be divided by 4 to become 0.0625µS, a frequency of 16MHz. This is the value used to work out all times for the pipelined processor.

The times quoted on the right of the following table are for the Immediate flush version, but the differences are insignificant.

	Flush at end.			Immediate flush.			
Benchmark	I_{Read}	D_{Read}	Clocks	I_{Read}	D_{Read}	Clocks	Time
Sieve	6783	2899	271256	6685	3035	269921	17mS
Gauss	6262	2394	211560	6310	2394	212283	13mS
QSort	227256	127594	7605661	227535	127594	7610274	476mS
String	1219	2496	96336	1152	2496	95306	6mS

As the results show, there is little end difference in the clock cycles required by each of the flushing methods.

Examining processor activity indicates that the Immediate Flush method sometimes allows the processor to proceed to fetch the correct instruction more quickly. A side effect of this, however, is that the fetch tends to 'keep ahead' of the processor resulting in an increase in fetches which are then discarded. This is noticeable in the

table above where the benchmarks with the lowest cache hit rate have a greater number of instruction fetches with Immediate Flushing than with flushing at the end of the cycle. This 'keeping ahead' also produced an increase in the number of data reads issued by the decode unit[14].

The small reduction in overall execution time suggests that the desired overlap of instruction execution is not happening either for the reasons listed above in connection with pipelining or because of memory contention.

Leaving aside the use of extra cache memory or faster main store, memory contention and pipeline flushing can be reduced by the use of the branch prediction mechanism as outlined in the architecture design.

Branch Prediction.

The architecture specifies a bit in all branch instructions to allow the compiler to indicate the expected direction of the branch to the processor. This is used to produce the figures below, again for both pipeline flushing techniques.

Prediction of the 'Call' and 'Goto' changes of control flow is also performed as the address is specified in the instruction. Other instructions affecting control flow, such as 'Return' and 'CallR' always cause the pipeline to be flushed.

The prediction accuracy percentage quoted is for all branch instructions where the compiler has the option of which way to direct the instruction fetching. Hence it includes the unconditional branch but excludes the other instructions noted above.

---

[14] Data reads are also issued by the execute unit but these are not affected by pipelining or flushing.

Bench	Pred%	Flush at end I_{Read}	Flush at end D_{Read}	Flush at end Clocks	Immediate flush I_{Read}	Immediate flush D_{Read}	Immediate flush Clocks	Time
Sieve	78	5376	2888	244926	5319	3039	244322	15mS
Gauss	63	5544	2358	198139	5550	2377	198265	12mS
QSort	75	203067	127594	7196023	201506	127594	7199595	450mS
String	88	1061	2493	92221	1120	2493	93111	6mS

It is evident that the branch prediction achieves its highest accuracy in the more iterative benchmarks and performs worse where there are a greater proportion of conditional statements. This is to be expected from the program analysis, but could be improved by the use of one of two techniques.

Compiler pragmas or pseudo-comments could be utilised to allow the programmer to indicate the likely direction of a conditional. Alternatively, the compiler could always choose a particular option and the programmer made aware of this so as to write the program appropriately.

Leaving this issue aside, again there is little to immediately choose between the flushing methods but despite the quite high percentages of accurate branch prediction the improvement in terms of clock cycles is limited to the area of 5%. Instruction reads have been reduced but data reads are very little changed.

Memory contention still appears to be a blockage to higher performance. The memory unit is occupied for around two-thirds of the clock cycles of each benchmark and performance is slowed by in-line, and therefore not cached, code and by data read and write.

Without a multi-word width store there is little that can be done about the first problem except to increase instruction cache size in an effort to include more large loops within the cache boundaries. The

second problem will be addressed later by the use of a data cache.

Chaining.

It is noticeable that some otherwise efficient code sequences are being hampered by the register reservation problem in that sequences of code repeatedly use the same register and hence do not exploit the potential pipelining.

A popular solution to this is to include knowledge of the pipeline within the compiler and re-order the instructions so as to minimise delays due to reservations. The following is a typical case of what a clever compiler could achieve at the expense of extra instructions and register usage.

```
          Simple Code                          Clever Code

                                               MOVE @start,@t
          MOVE @start,@loop                    MOVE @Start,@loop
Loop:     <use_@loop>                Loop:     ADD  #1,@t
          ADD  #1,@loop                        <use_@loop>
          BRLT @end,@loop,>Loop                MOVE @t,@loop
                                               BRLT @end,@t,>Loop
```

A similar principle can be applied to 'FOR' loops where the test is at the top. The difficulty comes in loops which are conditional on more than a simple count as it then becomes almost impossible to use the extra register technique and the scope for instruction shifting is limited, particularly in the important tight loops.

One way around this problem is to use a technique which here is called 'Chaining'. As each instruction passes through the Execution unit in the pipeline, any result of that instruction is not only passed on to the Store unit but also kept in a buffer within the Execute unit.

The Decode unit can detect where the result of the previous instruction is required as an operand of the current instruction and

so instead of waiting for the reservation flag to be lowered it passes a signal to the Execute unit indicating that it should use the contents of its 'Last Result' buffer.

As the Execute unit is a single pipeline stage there is no performance penalty in the provision of this facility which gives a greater performance potential than the compiler optimisation alone.

The principal increase in hardware complexity required for this system is the changing of the reservation flags into two bit counters. This is necessary as chaining allows a register to be reserved by more than one instruction at once and the register must be kept reserved until the last reserving instruction has written its result.

Note that chaining was not discussed in the original *RASP* design but has been included here as a result of the analysis of processor behaviour.

	Flush at end			Immediate flush.			
Benchmark	I_{Read}	D_{Read}	Clocks	I_{Read}	D_{Read}	Clocks	Time
Sieve	5363	2888	235256	5356	3043	235299	15mS
Gauss	5563	2358	195467	5550	2377	195247	12mS
QSort	205096	63599	5661490	202069	127594	6689895	418mS
String	1073	2493	79290	1119	2493	80059	5mS

Rather as with Branch Prediction, the timings have not improved as much as might be expected because of the effects of memory contention. In this case it is noticeable that the QSort and String benchmarks have done much better than the other two and that, as noted earlier, decreasing the time spent in the decode stage tends to allow the fetch unit to 'keep ahead' and perform more fetches which prove unnecessary.

An analysis of the number of clock cycles in which each pipeline

unit is active indicates that whilst the time taken by the Decode stage has been reduced by over 10% on average, the time spent by the Execute stage has increased by almost the same amount. As the number of instructions executed is constant, this can only be due to store contention.

It is noticeable that the number of data reads required by the QSort program, and hence the number of clock cycles, has fallen significantly when using the Flush At End technique. This anomaly may be explained by the chaining allowing an incorrect branch prediction to be flushed earlier before any data reads commence.

Unless memory contention can be reduced, and hence pipeline overlap increased, the extra hardware required by chaining may not be cost-effective and better used elsewhere.

Data Cache.

The two catagories of access to main store are instruction read and data read / write. The effects of an instruction cache on the basic processor have already been examined and it is reasonable to assume that the general characteristics will remain the same on the more complex machine, although numbers will differ.

Results above indicate that when using the current type of instruction cache there are more data accesses than instruction reads and hence it is worth investigating if a small data cache could substantially reduce this number. The goal is to achieve the most cost-effective balance between memory contention and cache resources and as the effects of a larger instruction cache are known the effects of a data cache should be investigated.

Unfortunately there was no time to investigate the results using a joint cache but it is expected that for the small cache sizes used

here the ability to provide differing block sizes and replacement strategies for the access types may well outweigh the extra flexibility of a joint system.

In addition to the choices of size, block size and replacement policy, the data cache also has a choice of write-back policy. Data can either be written back to main store when a write to cache is performed or alternatively can wait until the block is replaced when all words with the 'Dirty' bit set are written back. The latter option will tend to cause a substantial delay when replacement is necessary whereas the former will cause smaller but more frequent delays. The former will also tend to result in greater overall main memory usage as some words may be written back many times before replacement.

The caches are, in general, smaller than the instruction cache as it was felt that data access was less iterative and so without a multi-word width store there would be little benefit from the use of larger sets. This is borne out by the results.

The values in the table are the total data reads and writes for the particular configuration. Hit rate percentages have not been used as differing cache configurations cause differing numbers of requests to be issued by the instruction pipeline and hence the hit-rate would not give a true indication of the effectiveness of the cache in reducing main store activity.

For compactness, no I_{Read} figures have been given. As a general rule they are slightly higher than those without data cache, due again to the 'keep ahead' effect. A random replacement policy for the data cache also appears to slightly increase the values of I_{Read}.

Clock counts are also omitted from the table, but execution times are given to the nearest 0.1mS in order to give an idea of the effect of the cache on overall performance. The Immediate Flush type of

pipeline was used to obtain the figures.

Write-On-Replace

	Least Recently Used				Random Replacement			
	4/1		4/2		4/1		4/2	
Benchmark	$D_{R/W}$	T_{Exec}	$D_{R/W}$	T_{Exec}	$D_{R/W}$	T_{Exec}	$D_{R/W}$	T_{Exec}
Sieve	4995	13.1	4731	12.8	5003	13.3	4569	12.8
Gauss	4039	10.3	3369	9.4	4045	11.4	3966	11.2
QSort	79689	336.3	77680	334.2	104476	363.0	94768	351.8
String	1387	3.5	1335	3.5	1512	3.7	1356	3.4

Write-Through

	Least Recently Used				Random Replacement			
	4/1		4/2		4/1		4/2	
Benchmark	$D_{R/W}$	T_{Exec}	$D_{R/W}$	T_{Exec}	$D_{R/W}$	T_{Exec}	$D_{R/W}$	T_{Exec}
Sieve	6086	14.2	5973	14.0	6011	14.4	5825	14.1
Gauss	4456	11.9	4106	11.5	4298	11.6	4151	11.7
QSort	78822	335.3	77150	333.5	104465	363.4	95824	353.3
String	1345	3.4	1342	3.4	1471	3.6	1382	3.4

It is immediately evident that with the block sizes tested the pauses in execution caused by the Write-On-Replace policy do not have as great an effect as the multiple individual writes of the Write-Through configuration. It is also apparent that for the number of blocks tested, an LRU replacement policy will give a better performance than Random replacement.

From the differences between the 4 / 1 and 4 / 2 cache sizes it appears likely that there is little to be gained in increasing the block size. This is almost certainly the case in array processing where the general pattern is to process an element and then move on to the next one. Hence there is generally little point in holding on to large numbers of previously processed entries. This does not apply to record structures where it is likely that many fields will be accessed

in a more random pattern whilst the record is in use. A larger block size to hold such records would be applicable in this case but it is probably better to provide more blocks and hence increase overall flexibility.

When the best figures, generally those for the Write-On-Replace, LRU, 4 / 2 configuration are compared with the number of data reads and writes given in the table for the non-pipelined version of the processor, hit-rate percentages vary from 39 to 56%. Actual hit-rates will be slightly better due to the spurious extra requests generated by the pipeline which are subsequently discarded.

These figures seem low compared to the claims of many cache systems but note that it is a data cache only and that without the benefit of a multi-word width store no values are brought into the cache before they are actually demanded.

Improvements in execution times by the use of the cache vary from 13% to 30%, the latter with the intensive character processing string search benchmark and it is interesting to note that they now begin to approach, or in the case of String better, the times set by the simple processor using the 4MHz memory.

Memory Speeds.

The figures obtained so far have assumed a memory rate of 1MHz and a clock speed of 4MHz for the non-pipelined processor, 16MHz for the pipelined. As indicated earlier, the memory speed in particular is very conservative and so some evaluations have also been performed using a 4MHz memory - the same as that used by the M68000 timings.

Note that for the pipelined processor it is the ratio between memory and processor speed that determines the number of clock cycles required. Therefore all the results above would be valid for a 2MHz

memory, 32MHz processor combination with the exception that final execution times would be halved.

The table gives the results for processors using the standard instruction cache and the four single word blocks with Write-On-Replace and LRU replacement policies as the data cache.

The Gain figure is the percentage improvement in performance resulting from the use of the faster memory. Figures in brackets are the percentage performance improvement of the pipelined version over the non-pipelined processor.

All times are in mS.

	No Pipeline			Pipelined		
Benchmark	1MHz	4MHz	%Gain	1MHz	4MHz	%Gain
Sieve	16.8	8.7	48	13.1 (22)	5.3 (39)	60
Gauss	13.9	6.1	56	10.3 (26)	3.7 (39)	64
QSort	430.1	207.9	52	336.3 (22)	123.1 (41)	63
String	5.3	3.5	34	3.5 (34)	1.6 (54)	54

The more detailed results obtained with the above figures also show some interesting effects. The memory utilisation of the pipelined processor, measured by the percentage of cycles for which the memory unit is active, falls from around 80% with the 1MHz memory to around 50% with 4MHz.

From the table it is evident that the use of faster memory and pipelining go hand in hand, as greater gains are made from the faster memory in a pipelined system and a pipelined system is better able to take advantage of faster memory.

Summary.

The *RASP* execution times above are between 20% and over 50% quicker than for hand-written M68000 assembler whilst still using the 1µS memory access time with 16MHz processor clock frequency. These times can be better than halved when using the 4MHz memory access frequency as used by the M68000 timings.

Comparing the Modula-2/M68000 results against the *RASP* with 4MHz memory shows that the *RASP* executes the benchmark programs some seven to ten times more rapidly than the M68000 with the same memory speed.

It is possible to keep on enlarging and modifying the cache configurations to gain better and better performance, but the specification arrived at here is around the most cost-effective in terms of performance for cache resources. It is also important to ensure that the processor is not simply tuned for the effective execution of these benchmarks! As specified, the processor and caches should be easily small and simple enough to fit onto a M68000 sized chip.

Indeed the configuration as it stands could well be small enough to be fabricated in Gallium Arsenide, thereby allowing a potential order of magnitude increase in on-chip clock speed. To take real advantage of this, however, a faster supply of instructions and data would be required. This would best be provided by another, larger, cache system placed between main memory and the on-chip caches themselves.

With modern silicon fabrication techniques it is likely that space would remain on a chip for the enlargement of the cache system at relatively little cost. The likely outcome of this can be determined from the cache performance figures above.

With the 4MHz memory, the utilisation of the various pipeline

stages shows that the processor is quite well balanced in that it is not highly memory limited but that the memory is not unnecessarily quick. Memory utilisation has been reduced to 50% of cycles by the use of a 16 word instruction cache and 4 word data cache - not at all large by the standards of established mini-computers.

This efficient use of memory is a valuable asset when considering a multi-processor configuration. Obviously to avoid excessive store contention in such a system, larger caches or local memories must be employed, but these need not necessarily be quicker than main memory as their purpose is to allow multiple accesses rather than to increase access speed.

Such local memories could best be organised with a multi-word width bus to main memory in order to produce sufficient bandwidth without resorting to faster access speeds. The possibility of multi-word width connections to the processor caches has been avoided because of the likely limit on the number of pins available to connect to the processor chip.

Strengths and Weaknesses.

Although the *RASP* code in general appears to be efficient and compact, there are some areas which may benefit from further consideration and some which appear somewhat cumbersome because of the architecture design.

- Procedure entry and exit involves upwards of eight in-line instructions and is particularly long where space for data structures must be allocated in the dynamic part of the frame.

* Although dynamically not very common, constants above 31 involve use of the stack and hence an often slow sequence of 4 instructions.

* The need to convert between byte and word addresses, despite the ASL2 instruction, is still an overhead.

* Sequences of instructions to perform common operations are a feature of RISC architecture and so although at first sight a disadvantage must be considered in the context of the overall system.

On the positive side:

* Executable statements and expressions compile easily into highly efficient code.

* Many of the statement sequences noted above can be reduced by compiler optimisation, eg. multiplication replaced by shifts and range checks removed by compile time analysis.

* The limited number of instructions and formats has eased rather than hindered compilation.

The implications of this for the execution of languages such as Pascal are that they too should perform well. Earlier languages such as Fortran will have to resort to compiler register allocation techniques when the number of variable names in the program exceeds the number of register locations available. This is, however, a problem with which compiler writers for such languages are familiar.

Effects of Compiler Optimisations.

As discussed earlier, compiler optimisations can be broadly divided into those which are independent of the target machine such as common sub-expression elimination, and those which seek to deal with the characteristics of the particular machine.

We are concerned here with the latter catagory, into which such techniques as pipeline optimisation, branch prediction, store and/or register allocation and peephole methods can be grouped.

The potential of pipeline optimisation has been much reduced by the provision within the *RASP* processor of the chaining mechanism, allowing an immediately following instruction to use the result of the previous instruction without penalty. Benefits can still be obtained, however, by the exploitation of this feature in conjunction with the pipeline.

Branch prediction is more likely to have a significant effect on overall performance. Prediction of loops is generally straightforward, but the case of conditionals is much less clear cut. Two methods of improving this situation were outlined above in the section of branch prediction, both of which use the programmers knowledge of anticipated program behaviour. Using compiler pragmas has the advantage of not interfering with the way in which the program is laid out and appears to be a more flexible mechanism.

The problems of conventional register allocation have been effectively removed from the *RASP* processor, but an optimisation worth considering is that of placing small, statically sized structures within the fixed part of the stack frame, where space is available. With a heavily used structure this could significantly reduce the number of store accesses required.

Peephole style optimisation normally consists of such things as

the removal of redundant store and load operations. The *RASP* only uses such operations in accessing data structures and so, when combined with common sub-expression elimination, the scope for such improvements is limited.

An important optimisation for the *RASP* is the elimination or strength reduction of multiply and divide type operations. These operations are slow on the basic machine as they are implemented by software. The most common use for multiplication is in scaling array indices and hence is multiplication by a constant. Hence it is quite simple for a compiler to substitute a small series of shifts and adds to obtain the required result. The code used in all of the tests above assumes that the compiler has performed this task.

Primula for Software Engineering.

The situation when evaluating a language in terms of its 'usefulness' to the programmer is similar to that encountered when analysing the performance of a system in that without much practical experience there is a limit to what can reliably be stated.

As with the architecture, the derivation of the language from the ideas which it is designed to support gives good reason to assume it will go at least some way to providing the properties which programmers desire.

One method of evaluating a language is to compare it against others in order to balance their relative merits and problems. This is the approach adopted here, comparing *Primula* to a number of common languages on a variety of grounds including the properties necessary to support the basic ideas of abstraction etc. The basic information used in the comparison is detailed earlier in the discussion of existing languages.

Two comparison tables are presented. The first covers aspects not directly related to the object nature of the language and so compares against common conventional languages which are based on the procedural model of programming. The second table deals with the object oriented aspect and so compares against languages which claim to provide some form of object programming capability.

Property	C	Pascal	Modula-2	ADA	Primula
Strong types	-	*	*	*	*
Nesting	-	*	*	*	*
Sep. Compile	*[15]	-	*	*	[16]
Generic types	-	-	-[17]	*	*
Concurrency	-	-	*[18]	*	*
Low level	*	-	*	*	*

Property	Simula-67	C++	ADA	SmallTalk	Primula
Attrib Hiding	-	*	*	-	*
Inheritance	*	*	-	*	*
Mult Inherit	-	-	-	-	-
Garbage Colln	*	-	-	*	[19]
Obj Conc/y	-	-	*	-	*
Nested defns	*	-[20]	*	-	*

The first table clearly illustrates that Modula-2, ADA and *Primula* have been designed with software engineering in mind, rather than as a systems programming language (C) or a teaching language (Pascal).

The second table highlights the disagreement in what constitutes an object oriented language. It seems, however, that at least single inheritance is required. The inclusion of attribute hiding and the nesting of definitions depends on the character of the base language

[15] But limited link-time checking.
[16] Not yet implemented.
[17] Does have 'Open Array' parameters.
[18] Albeit in restricted form.
[19] Not yet implemented.
[20] May be textually nested with no effect on meaning.

in the cases of C++ (C) and Simula-67 (Algol).

Other features, however, which would seem to be implied by the use of objects for software engineering are not available within the languages. This must inevitably lead to more program complexity and a reduction in the use of the object model approach.

The Design of an Object-Oriented Operating System.

Operating systems are notoriously complex pieces of software which must safely share the resources of the machine and provide services to a dynamic number of simultaneous users. The following is an outline design of such a system based around the object paradigm.

Following the real-world modelling idea of object-oriented programming each user is represented by an instance of the *User* object type. This logically contains the resources or capability of the user - available store, permitted I/O devices, current terminal etc - and is roughly equivalent to the process control block of many other systems. The resources of each user are, of course, set by the instantiation parameters of the object.

User objects then instantiate a *SystemService* object which provides operating system services to user programs by way of its public procedures. It uses a reference to its creating *User* object where necessary to check that an operation is legal for the particular user.

The *User* objects also instantiate the initial object which the user requires to run - typically this will be a shell or command processor type program which will be passed, as an instantiation parameter, a reference to the *SystemService* object.

It may then call this object to obtain references to the objects controlling the I/O devices by which it is to communicate with the

user etc. It can request memory into which it can load user programs and switch execution to the start of that memory. User programs would also generally be passed a reference to the *SystemService* object although the caller could always substitute another suitable object.

An important function of *SystemService* is to allow the creation of new objects by existing user objects. This may be within object-oriented user programs or at command shell level in the form of a Unix-like fork operation. Depending on the system policy, such objects may be granted independent status with their own *User* and *SystemService* objects or they may be treated as dependent on the parent and hence must share these objects and hence the resources they contain.

Hardware protection will be necessary to ensures objects do not deliberately or accidentally overstep their address limits etc. The information required for this is held within the *User* object. One measure would be to force users to access the *SystemService* object through trap instructions.

Having examined the organisation from the user object point of view we now look at the objects that *SystemService* requires in order to provide its services. These correspond essentially to the parts of a conventional system ie. *StoreManager*, *FileSystem*, *Communications* etc.

Depending on the action involved, the *SystemService* object may return a result directly, or in the case of more complex actions return a reference to another object. An example of the latter would be that requesting the opening of a file returns a reference to a file type object.

Objects such as *FileSystem* could run under the *User* and *SystemService* objects like any other user, with the exception that the

User object may allow them special capabilities.

At the lowest level there is the *ProcessorManager* object which performs the function of Scheduler and Dispatcher in a conventional system. In general it maintains queues of waiting objects (processes) or processors and dispatches objects onto processors by taking the required information from that objects *User* object and then executing a low-level context switch.

Objects are placed in the *ProcessorManager* ready queue structure when first created or when accessed whilst not already busy. The relevant calls are made by the object requesting the action via the *SystemService* object.

This simple outline illustrates how the object model can be used to build large systems out of smaller components whilst maintaining the protection and integrity of those components.

Conclusions.

In order to summarise and assess the above work, it is first necessary to briefly restate the aims and content of the project.

* An examination of the needs of software engineering and how well those needs are met by a variety of programming languages.

* A theoretical look at object oriented programming with a view to establishing what characterises this particular form.

* Design and development of a programming language to support the needs of software engineering based around the theoretical notions of the object oriented style.

* Design of a processing architecture to efficiently execute programs written using the above language.

* A performance evaluation of this architecture in comparison with established machines of similar anticipated cost.

Each of these is covered by a distinct part of the thesis which is summarised below. Ideas are also included for future work in both the research and product development areas. The latter is of interest because both *Primula* and *RASP* are not far from the stage at which they could be commercially marketed.

Software Engineering.

There is a great deal of specialised work being done in this area and so this section consists basically of a review of the fundamental ideas relevant to the design of programming languages. These are based around the notion of abstraction, divided into the constituent parts of classification, aggregation and generalisation.

A second and orthogonal division of abstraction is into definition and implementation parts, leading to the ideas of modularity, data and implementation hiding. The definition of an abstraction is its *type*, which leads to the practical idea of strongly typed languages.

Abstractions may be nested and since they form independent units and may act concurrently with the provision of suitable synchronisation and communication mechanisms.

An examination of current popular and research languages shows that a variety of models are used, with varying degrees of success, to provide the abstractions out of which a program is built. Four common language classifications are Procedural, Functional, Logic and Object-Oriented.

The latter is chosen here for further investigation as appears to provide a natural method of abstracting from the real-world and to some extent it subsumes the others as they may be individually employed within the objects themselves.

Object Oriented Programming.

Definitions of what constitutes object oriented programming vary considerably so it was necessary to develop a coherent working definition based around the ideas of abstraction. This includes an outline of the associated terminology.

An object is an abstraction consisting of a state (optionally null) and actions which may be performed on that state. Such actions are executed in response to a request from some other object in the system, sent via an underlying communications mechanism.

A Class is a collection of similar objects and may therefore be viewed as the data type of those objects. Inheritance allows some characteristics of a parent class to become a part of a new child class. The characteristics involved may be definitions or implementations of the abstraction and may be fixed or modifiable by the child depending on the form of inheritance employed. The differences between Type Theory, External Interface, Code Sharing and Polymorphism were discussed within the thesis.

Where a child redefines some characteristic or attribute of the parent, there must be a policy to indicate when each definition will be used. Static binding decides using the class of the reference through which the attribute is being accessed whereas dynamic binding uses the class of the object itself.

Multiple inheritance, i.e. inheritance from more than one parent, and Delegation, a form of run-time inheritance, were also reviewed, although with no definite conclusions. There is scope for future research here in establishing more of what multiple inheritance is about and perhaps producing a model which brings together the existing, varied, ideas.

From the ideas of abstraction it follows that objects may execute concurrently and therefore that some synchronisation mechanism is required. To maintain the data integrity of objects, only one action may be executed at any one time - objects can be viewed as single tasking machines - and this can also be used to synchronise the actions of other objects.

The communications system linking the objects can, therefore, be asynchronous although it is more common to provide synchronisation through this system such that an object making some request of another will wait until the request is completed. Provision of suitable primitives allows the construction of Monitor type constructs.

Garbage collection is also commonly a feature of object oriented systems and also one of its major problems. Programmer controlled object disposal is prone to all the errors associated with disposal of heap storage in conventional languages such as Pascal, whereas automatic garbage collection imposes what may be an unacceptable overhead on the system. Much more work must be done in this area if a generally satisfactory solution is to be found.

The *Primula* Programming Language.

The initial requirements of the language were to support the object oriented programming model whilst maintaining the prospect of execution efficiency comparable with that of the traditional procedural languages.

This suggested that the code of an object be of a procedural nature, with variable declarations to represent the object state. Object definitions consisting of variable (state) and procedure (method) declarations can then be bundled as data types within the procedural framework.

Accesses to other objects appear as remote procedure calls which may then be implemented by techniques such as message passing to suit the underlying architecture. A Guard construct is provided to conditionally prevent the execution of particular methods which then allows the construction of Monitor style objects. Objects also form active, independent entities and so permit full multi-processing with

a suitable architecture.

Inheritance is restricted to a single parent, partly for efficiency, but also because of the confusion as to what multiple inheritance should be and to its general application. Full control is given over the visibility of methods both from inheriting children and from external, public, users.

Two distinct forms of inheritance are provided, the first being called 'Refinement' which is the most general in allowing sub-classes and a free choice of inheritance styles in terms of those theoretically outlined. The disadvantage of this choice is the lack of compile-time checking for correctness and completeness of a particular form and a worthwhile task would be to establish whether syntactically distinguishing such forms would reduce errors by allowing compile-time checks or whether it would impair the overall usefulness of the concept.

The second form of inheritance, 'Uses', has the distinct function of permitting code-sharing, but to do this flexibly whilst maintaining the efficiency of compile-time checking has meant that a Class created by this mechanism is not a true sub-class of the parent. Further investigation may reveal if this restriction should be lifted or the construct dropped altogether, perhaps replaced by a single, more powerful, inheritance construct.

Block structure and generic types are introduced from consideration of the abstraction properties outlined within the discussion on software engineering. Imperative statements are similar to those of Pascal / Modula-2 although with some generalisations including the use of default parameters.

Whilst *Primula* provides many of the features of both object and procedural languages, without using it for a suitably large project it

is impossible to say whether it has fulfilled its objectives. Hence any future development should include a substantial piece of programming, perhaps based on the outline operating system suggested in the previous chapter, in order to take account of this experience in subsequent language changes.

A following task would be the development of a production quality compiler producing efficient *RASP* target code to be used alongside a full run-time system including tracing and debugging facilities. The object model of programming lends itself well to a windowing debug system where each window represents a single object.

Separate compilation of objects with full checking at link time was an aim of the language which was not realised in the implementation. It is, however, an important part of the overall system and therefore should be part of the improved language and support software.

Together with the language design was the design of an execution model detailing information such as the layout of stack frames, communications primitives etc. This plays a vital part in the production of compilers and the design of the *RASP* processor.

RASP Architecture.

The design process proceeded with the idea of making best use of a given amount of resources, implying that the resources must be shared between the various actions that the processor must perform in order to yield best overall performance. Traditionally this has been achieved by the use of a micro-code engine, shared by all the processors actions, but an alternative is the RISC approach which provides a few simple instructions to make effective use of a small amount of hardware, the remainder of the processor containing operand storage.

To maximise the sharing of resources requires a knowledge of dynamic program behaviour, in addition to the execution model already devised. The similarities between *Primula* and Pascal were exploited to obtain this information from an analysis of a wide range of Pascal programs.

In addition, mention was made of the ability to support multiple languages and systems requirements. There was also a brief examination of the hardware technologies likely to be used in implementation of the design.

The resources were allocated to actions in proportion to their frequency in order to obtain best results. An action whose frequency was to low, relative to its complexity, to permit it to be implemented was excluded and its effect simulated by the use of other actions. Hence the frequencies were distributed and actions modified in an iterative process.

It must also be remembered that operand storage is a resource which most instructions make use of, and it was this factor, along with achieving a balanced instruction set of only a few simple actions, which made the *RASP* into a RISC style processor.

Memory speed considerations suggested a small instruction format which then also suggested that the operand area be directly addressable. The use of the directly addressable cache avoids the need for explicit register load and store operations and also reduces the number of implicit store accesses to a minimum. The size of the cache is limited not only by the instruction format but by consideration of the amount of information which must be saved and restored when a process / object switch occurs.

Using only simple instructions does have limitations, however, particularly for arithmetic operations, and future work should give careful consideration as to whether support for, or inclusion of, an integer multiply would be justifiable and to the provision of a co-processor interface for integer and floating point arithmetic.

The limited number of instructions and formats requires that effective use is made of them and so consideration of the less frequent instructions and alternatives would be worthwhile. This also includes the problem of large constants and the shifting to convert from word to byte addresses required principally because the Stack Pointer and Frame Base registers are kept as word addresses to avoid potential addressing errors.

A further area for examination is that of exception handling and operating system support. The current design performs the minimum necessary actions in keeping with the simple design philosophy of the machine, but it may be that there are benefits in changing or adding to these actions.

Finally, of course, is the production of a working machine based on the design outlined in the thesis. The software emulator will form a useful specification and cross-checking facility as it is detailed down to the register level. The overall simplicity of the design and

the number of repeated small units should allow quite rapid design, production and testing.

Performance Evaluation.

Having designed the user visible architecture in a theoretical manner beginning with the anticipated execution characteristics of *Primula* programs, it was again noted that for the purposes of evaluation these characteristics would be similar to those of the Pascal / Modula-2 family of programming languages, with the exception of the use of the object concept.

To give an indication of the performance of the *RASP* architecture, therefore, it was compared against the popular Motorola M68000 processor with both machines executing benchmark programs written in Modula-2. The features of the M68000 were outlined along with those of some other popular machines in the introduction to the thesis.

The benchmark programs were also used to evaluate the effects of the non-visible parts of the architecture such as the caches and pipeline and allow the tuning of these to the optimum configuration. Such tuning was angled towards obtaining cost-effective rather than ultimate performance, ensuring that the final chip design would be relatively small, simple and therefore cheap.

As a result of analysing processor activity, a further feature, Chaining, was introduced into the non-visible architecture. This prevents breaks in pipeline flow by allowing the ALU stage to act as a kind of accumulator in the case where an instruction uses the result of the preceding one.

It is expected that the final design, containing a four stage pipeline with operand chaining facility, sixteen word instruction

cache and four word data cache will occupy no more silicon space than the M68000 it is compared against. Although it is generally unreliable to make performance claims from benchmark results, execution speeds on the *RASP* machine were between seven and ten times as fast as the M68000.

A major part of the work remaining in this section is the design and evaluation of a multi-processor version of the *RASP* architecture. Much of the work will involve the effects of various cacheing or local memory systems on the overall performance, trying to obtain speeds which increase as linearly as possible with the number of processing elements. This should be aided by the efficient use that the *RASP* makes of memory.

Other work consists of attempting to isolate the benefits to performance that the various parts of the architecture make, in order that the relationships between them be better understood so that the design may be further fine tuned.

The Overall Picture.

The increasing interest in both object oriented programming and RISC machines was mentioned in the introduction and so here we briefly indicate the position of *Primula* and *RASP* within the spectrum of commercial products and research ideas.

Primula is best classed as a third generation language as it is neither an applications generator (fourth generation) nor is it declarative (fifth generation). As such it is alongside the common languages in use today and so the questions must be asked - why develop another third generation language and what does it do that such languages do not already do?

The answer to the first part is that fourth generation languages

are not general purpose, in that they tend to be aimed towards a particular type of problem, and the fifth generation will not become a practical possibility within the immediate future.

Much of the interest in object programming has been brought about by an increasing awareness of the problems of existing procedural languages and the lack of any real progress in practical programming methodology for a number of years. Hence the answer to the second part of the question is that *Primula* offers a fuller and more efficient implementation of the object concept than otherwise available and has the potential for genuine multi-processor implementation. A further practical consideration is that it will be quick to learn for programmers already familiar with the Pascal language family.

A similar question can be posed in respect of the *RASP* processor and this will be dealt with in three different areas - personal computers, workstations, and minis and mainframes.

In view of the domination of the micro-processor market by the Intel 80x86 and Motorola M680x0 processor families and associated software, there is little scope for the *RASP* except as an add-on coprocessor for specific tasks.

Workstations, however, are much less tied and several manufacturers have produced new RISC processor designs aimed at this area, such as the SUN SPARC, Digital DECStation and IBM RT6150. As with *RASP* the designers began with a clean sheet rather than the need to be compatible. Fast performance and multi-processor capability make the *RASP* a suitable candidate in this area, both for general purpose and specialist, eg. CAD, applications.

To move into the area of multi-user workstations and mini-computers requires the addition of protection, virtual addressing facilities etc. The simplicity of the processor is such, however, that

little of the existing design would have to be changed in order to add in the extra functionality. Hence the addition of such features can be done quickly and relatively cheaply, the costs being almost entirely due to the feature itself rather than processor modifications.

With these additions, and noting that 'C' is a stack-based language and therefore will run efficiently on the *RASP*, the processor becomes a genuine multi-user machine capable of running Unix or any other system. It then, of course, becomes competitive in the mini-computer market and upwards to larger machines.

Note also that, in general terms, *RASP* is strong in symbol manipulation and weak for arithmetic which matches the workload in many office / commercial situations. Many larger computers are used almost exclusively as data storage and selective retrieval devices, with little processing other than the selection process. A restriction, however, of the current processor in terms of large applications is the thirty-two bit address bus; many mainframes now exceed this.

A benefit of RISC technology is that improvements in chip design or manufacture can be passed on more quickly and cheaply than for a more complex design. Set against this, however, is the issue of compatibility.

Micro-coded machines allow different internal architectures to provide the same user-visible architecture, thus creating a range of compatible machine with widely varying performance characteristics. There is no scope within a RISC design for this type of approach, but it is possible to create a wide range of processors by different methods.

The principal variable factor is the hardware construction of the chip itself - changing fabrication techniques and materials would

allow changes in cost and performance. A top end machine could be constructed from several smaller scale chips thus allowing a faster clocking rate although the problems of inter-chip communication mean that the functionality of each must be carefully chosen. A second possibility of changing the construction, or accepting lower yields and so increasing chip cost, is to increase the sizes of cache units and so gain performance by that method.

Finally, then, the work undertaken has produced an interesting programming language and a high performance processor design. I hope it is as informative for the reader as it has been for me.

Primula Syntax.

This is the syntax supported by the current compiler and so differs in a number of details from the language description - these are listed below. Note also that suggested extensions or modifications are also not included in the syntax.

* Parameters are passed to the main body of the parent by the new procedure and not within the header of the child.

* Strongly typed pointers are provided by the current compiler, instead of the more general ADDRESS type.

* Default parameters are not permitted by the current compiler.

The syntax is described by BNF where :

<x> represents the non-terminal x.
x represents the terminal symbol x.
[x] represents optional presence of x.
{x} represents zero to n repetitions of x.
x|y represents selection of x or y.

```
<Object_Spec> = <Object_Header> <Object_Body>
<Object_Header>= <Class_Spec> <Formal_Parms> <ParentSpec> ;
<Class_Spec>  = [ SYSTEM ] OBJECT
<Formal_Parms>= ( { <Formal_Parm> ; } )
<Formal_Parm> = [ REF | CONST ] <Ident_List> : <Type>
<Parent_Spec> = [ REFINEMENT OF <Desig> ]
<Object_Body> = <Block>

<Block>       = BEGIN [ <Declarations> EXECUTE ] <Stat_List>
                  [ <Error_Handler> ] END
<Declarations>= [ CONST <Const_Decls> ] [ TYPE <Type_Decls> ]
                  [ VAR <Var_Decls> ] [ CODE <Code_Decls> ]
<Const_Decls> = { <Use_Spec> <Ident> = <Const_Expression> ; }
<Use_Spec>    = [ PUBLIC ] [ BEQUEATH | DELEGATE ]
<Const_Expression> = <Expression>
<Type_Decls>  = { <Use_Spec> <Type_Ident> = <Type> ; }
<Type_Ident>  = <Ident> [ ( <Generic_List> ) ]
<Generic_List>= [ INDEX ] <Ident> { , [ INDEX ] <Ident> }
<Var_Decls>   = { <Use_Spec> <Ident_List> : <Type> ; }
<Code_Decls>  = { <Use_Spec> <Code_Spec> ) ; }
<Code_Spec>   = ( PROCEDURE | FUNCTION ) <Ident> <Formal_Parms>
                  [ : <Type> ] ; <Guard> <Block>
```

```
<Guard>         = [ GUARD <Expression> ]
<Stat_List>     = { <Statement> ; }
<Error_Handler>= ON ERROR DO <Statement>

<Statement>     = <Block> | <Desig_Stat> | < Return_Stat > |
                    <If_Stat> | <Case_Stat> | <While_Stat> |
                    <Loop_Stat> | <For_Stat> | <With_Stat> |
                    <Inner_Stat> | <Exit_Stat>
<Desig_Stat>    = <Designator> [ := <Expression> ]
<Return_Stat>   = RETURN <Expression>
<If_Stat>       = IF <Expression> THEN <Statement>
                    [ ELSE <Statement> ]
<Case_Stat>     = CASE <Expression> OF <Cases_List>
                    [ OTHERWISE <Statement> ] END
<Cases_List>    = { == <Case_List> : <Statement> ; }
<Case_List>     = <Case_Spec> { , <Case_Spec> }
<Case_Spec>     = <Expression> [ .. <Expression> ]
<While_Stat>    = WHILE <Expression> DO <Statement>
<Loop_Stat>     = LOOP <Statement>
<For_Stat>      = FOR <Designator> := <Expression> TO <Expression>
                    [ STEP <Expression> ] DO <Statement>
<With_Stat>     = WITH <Designator> DO <Statement>
<Inner_Stat>    = INNER
<Exit_Stat>     = EXIT

<Expression>    = <Simple_Expression> [ <Relation>
                    <Simple_Expression> ]
<Relation>      = = | <> | < | <= | > | >= | IN
<Simple_Expression> = [ + | - | ~ ] <Term> { <AddOp><Term> }
<AddOp>         = + | - | OR | |
<Term>          = <Factor> { <MulOp> <Factor> }
<MulOp>         = * | / | DIV | MOD | AND | &
<Factor>        = <Value> | <Designator> | ( <Expression> )
                    | "NOT" <Factor>
<Value>         = <Number> | <String> | <Structure> | <Based_Number>
<Structure>     = # <Type> ( <Set> | <Record> | <Array> )
<Set>           = { <Element> [ , <Element> ] }
<Record>        = RECORD { <Expression> ; } END
<Array>         = [ [ <Exp_List> ] ]
<Based_Number>= $ <Binary_Integer> | # <Hex_Integer>
<Designator>    = <Ident> <Spec_List>
<Spec_List>     = { . <Ident> | [ <Exp_List> ] |
                    ( <Actual_Parm_List> ) }
<Actual_Parm_List> = [ <Actual_Parm> { , <Actual_Parm> } ]
<Actual_Parm> = <Expression> | REF <Designator>

<Type>          = <Simple_Type> | <Structured_Type> | <Code_Type>
                    | <Object_Type> | < Pointer_Type>
<Simple_Type> = <Type_Indicator> | <Enumeration>
<Type_Indicator> = <Designator> [ <Generic> | <Subrange> ]
<Generic>       = ( <Type_List> )
<Subrange>      = RANGE <Const_Expression> .. <Const_Expression>
<Enumeration> = ( <Ident_List> )
<Structured_Type> = <Array_Type> | <Record_Type> | <Set_Type>
<Array_Type>    = ARRAY [ <Type_List> ] OF <Type>
<Record_Type> = RECORD <Field_List> END
<Field_List>    = { <Ident_List> : <Type> ; }
<Set_Type>      = SET OF <Type>
```

```
<Code_Type>    = ( PROCEDURE <Formal_Type_List> ) |
                 ( FUNCTION <Formal_Type_List> : <Type> )
<Formal_Type_List> = ( { [ REF | CONST ] <Formal_Type> ; } )
<Formal_Type> = <Type>
<Object_Type> = <Object_Spec>
<Pointer_Type> = POINTER TO <Type>

<Type_List>    = <Type> { , <Type> }
<Exp_List>     = <Expression> { , <Expression> }
<Desig_List>   = <Designator> { , <Designator> }
<Ident_List>   = <Ident> { , <Ident> }

<Ident>    = <Letter> { <Letter> | <Digit> | _ }
<Number>   = <Integer> | <Real>
<Integer>  = <Digit> { <Digit> }
<Real>     = <Integer> [ . { <Digit> } ] [ <Scale_Factor> ]
<Scale_Factor>    = E [ + | - ] <Integer>
<Binary_Integer> = ( 0 | 1 ) { 0 | 1 }
<Hex_Integer> = <Hex_Digit> { <Hex_Digit> }
<String>   = " { <Character> } " | ' { <Character> } '

<Digit>    = 0 .. 9
<Hex_Digit> = <Digit> | A .. F
<Letter>   = A .. Z | a .. z
<Special_Char> = ( | ) | [ | ] | { | } | + | - | * | / | = | <
                 | > | , | ; | : | . | " | ' | ! | #
<Character> = <Letter> | <Digit> | <Special_Char> |
              <Any_Other_Char>

<Comment>       = <Line_Comment> | <Full_Comment>
<Line_Comment> = ! { <Character> } <End_Of_Line>
<Full_Comment> = << { <Character> } >>
```

Modifications introduced by the language definition are as follows :

```
<Parent_Spec>   = [ REFINEMENT OF <Desig> <Formal_Parms> ]
<Formal_Parm>   = [ REF | CONST ] <Ident_List> : <Type>
                  [ DEFAULT = <Expression> ]
<Pointer_Type> = ADDRESS [ OF <Type> ]
```

Source Code Analysis Details.

Results from New Survey.

This is the result file of the Pascal source code analysis program suite. Annotations have been added in italics where explanation of headings was thought necessary.

SUMMARY OF RESULTS

```
Total errors detected :-       3603
  Mostly caused by non-standard extensions
Labels
  Total count       :-         20
  Number of entries :-       4532    ie. number of blocks
  Average count     :-          0.00
  Maximum count     :-          2

Names of all classes
  Total count       :-      15250
  Number of entries :-       4532
  Average count     :-          3.36
  Maximum count     :-        184

Variable names
  Total count       :-       7171
  Number of entries :-       4532
  Average count     :-          1.58
  Maximum count     :-        110

Scalar variables          :-        662 enumerations
Subranges < 256           :-        524 max value < 256
Other Subranges           :-        145
Integer variables         :-       1861 excl subranges
Real variables            :-        330
Character variables       :-        262

Pointers to arrays        :-         21 variables, not types
Pointers to records       :-       1557
Pointers to others        :-         11

Files of text             :-        286
Files of arrays           :-          1
Files of records          :-         42
Files of other types      :-          0
Arrays of structures      :-        127 unpacked
Arrays of simples         :-        281    "
Pack Arrays of Char       :-        285
Other packed arrays       :-          6
```

```
Set size
  Total count       :-        955     number of elements
  Number of entries :-         72     number of sets
  Average count     :-         13.26
  Maximum count     :-         56

Array index size
  Total count       :-       7610
  Number of entries :-        462
  Average count     :-         16.47
  Maximum count     :-      81199

Array number of dimensions
  Total count       :-        462
  Number of entries :-        408     number of arrays
  Average count     :-          1.13
  Maximum count     :-          3

Fixed fields in record
  Total count       :-       2808
  Number of entries :-        608     number of records
  Average count     :-          4.62
  Maximum count     :-         21

Number of alternative variants in record
  Total count       :-        551
  Number of entries :-         50     no. of recs with vars
  Average count     :-         11.02
  Maximum count     :-         55

Maximum fields in record
  Total count       :-       2902
  Number of entries :-        608
  Average count     :-          4.77
  Maximum count     :-         21

Value parameters                       declared
  Total count       :-       4190
  Number of entries :-       4815     no. of procs / funcs
  Average count     :-          0.87
  Maximum count     :-          7

Reference parameters
  Total count       :-       2330
  Number of entries :-       4815
  Average count     :-          0.48
  Maximum count     :-          8

Code parameters
  Total count       :-          3
  Number of entries :-       4815
  Average count     :-          0.00
  Maximum count     :-          1

Structured value parms   :-        964
Simple reference parms   :-       1150
```

```
Procedure definitions
  Total count        :-       3014
  Number of entries  :-       4532    number of blocks
  Average count      :-          0.67
  Maximum count      :-         76

Function definitions
  Total count        :-       1801
  Number of entries  :-       4532
  Average count      :-          0.40
  Maximum count      :-        115

Total code block definitions
  Total count        :-       4815
  Number of entries  :-       4532
  Average count      :-          1.06
  Maximum count      :-        156

Declared constants        :-        959
In-Line constants         :-      22355
Constants > 255           :-        293  integer type only
Constants > 15            :-       1600     "     "    "
Use of decl consts        :-       6207

String constant lengths
  Total count        :-     124578    number of chars
  Number of entries  :-       7316    number of strings
  Average count      :-         17.03
  Maximum count      :-        116

Local variable use        :-      19142
Scope variable use        :-       9591
Total variable use        :-      21073 no dereferences

Levels of dereferencing               use of '.', '[', '^'
  Total count        :-      12816    dereference operators
  Number of entries  :-      28733    variables used
  Average count      :-          0.45
  Maximum count      :-          9

Number of factors in expression
  Total count        :-      49695
  Number of entries  :-      43288    number of expressions
  Average count      :-          1.15
  Maximum count      :-          9
```

```
Counts of expression forms
  Copy                          :-       8574 one var factor
  Use Constant                  :-      17956 one const factor
  A := A + B form               :-        136 excl A:=A+Const
  A + B form                    :-        446 excl A:=A+B
  A := A + Const form           :-        969
  A + Const form                :-        594 excl A:=A+Const
  Other general forms           :-       7468
  A relop B form                :-       1655 A,B are vars
  A rel Const form              :-       4884 or Const relop A
  Const relop Any form          :-        166 excl Const relop A
  A relop Any form              :-        118
  Any relop Any                 :-        824 excl above
  Sets in expressions           :-        281
  Consts in expressions         :-      22355 excl declared
  Variables in expressions      :-      13165
  Functions in expressions      :-       4049

Goto statements                 :-         32
Procedure calls                 :-      16349
Assign statements               :-      18622
With statements                 :-        899
IfThen statements               :-       2285
IfThenElse statements           :-       3850
Case statements                 :-        525
For statements                  :-        969
While statements                :-       1025
Repeat statements               :-        258
```

Results from Schmidz' Survey.

These are taken from [SCHM] and are based on an analysis of systems type programs executing on the Imperial College Computing Department IBM370 under VM/CMS, using the Pascal P4 compiler.

The first set of figures are static counts of the number of occurrences of the grammar construct within the text of the programs. The second set are counts of the various code fragments output by the compiler to implement the source operations. Fragments are counted statically as output by the compiler, and dynamically as executed. All values are taken directly from the text of the thesis.

Grammatical Construct	Count
<program>	38
<block>	1577
LABEL <label>[,<label>]	39
<id> = <constant>	611
<unsigned_number>	1650

```
     <signed_number>                          24
     <unsigned_identifier>                  1453
     <signed_identifier>                       3
     <string>                                373
<unsigned_integer>                          1673
<unsigned_real>                                1

<id> = <type>                                644
<type>                                      4633
<scalar_type>                                94
<subrange_type>                             560
<type_identifier>                           3298
<packed_structure>                           206
<unpacked_array>                             339
<unpacked_record>                            194
<unpacked_set>                                20
<unpacked_file>                               27
<field_list> ::= <fixed_part>                287
     ::= <fixed_part> ; <variant_part>        23
     ::= <variant_part>                       35
<fixed_part>                                987
<record_section>                            1075
<pointer_type>                               101

<formal_parameter_section>
     ::= <parameter_group>                   977
     ::= VAR <parameter_group>               900
     ::= FUNCTION <parameter_group>            0
     ::= PROCEDURE <id>{,<id>}                 5
<parameter_group>                           2021
<function_declaration>                       214
<procedure_declaration>                     1363

<statement_part>                            1577
<label> : <unlabelled_statement>              42
<unlabelled_statement>                     27808
<assignment>                                8505
<assignment_to_variable>                    8165
<assignment_to_function_name>                340
<procedure_call>                            8109
<goto>                                        58
<empty>                                     2260
<compound>                                  3381
<if>                                        3419
<case>                                       213
<while>                                      556
<repeat>                                     183
<for>                                        331
<with>                                       793

<variable> ::= <entire_var>                24304
<indexed_var>                               3476
<field_designator>                          3842
<file_buffer>                                296
<referenced>                                2247

<expression> ::= <simple_expression>       30175
<relational_operator>                       3753
```

```
=                                          1718
<>                                          990
<                                           259
<=                                          149
>=                                          102
>                                           259
IN                                          276
<simple_expression> ::= <term>            35947
     ::= <sign> <term>                       51
<add_op>                                   1683
+                                          1054
-                                           484
OR                                          338
<term> ::= <factor>                       38605
<mul_op>                                    902
*                                           165
/                                            50
div                                         137
mod                                          57
AND                                         587
<factor> ::= <variable>                   19900
     ::= <unsigned_constant>               5683
     ::= ( <expression> )                  1636
     ::= <function_designator>             1684
     ::= <set>                              384
     ::= NOT <factor>                       744
<unsigned_constant> ::= <number>          5683
     ::= <string>                          5365
     ::= <constant_id>                     4783
     ::= NIL                               1068
<user_function_designator>                  848
<standard_function_designator>              836
<element_list>                              340
<element> ::= <expression>                  668
     ::= <expression>..<expression>         129

<user_procedure_call>                      4387
<standard_procedure_call>                  3722
<actual_parameter> ::= <expression>        7611
     ::= <variable>                        3015
     ::= <procedure_id>                      17
     ::= <function_id>                        0
<case_lists>                               1236
<case_labels>                              1554
<record_var_list>                           962
```

Code Fragments.

Fragment	Static	Dynamic
program	38	23
external files	122	73
user call with parameters	3395	85416
user call without parameters	992	110650
standard procedure call	3722	217104
procedure entry	1577	260514
assignment	8505	687068
if..then	1507	158953

if..then successes	1507	53769
if..then..else	1912	343435
if..then..else successes	1912	85661
case	213	10847
while start	556	26164
while iterate	556	145058
repeat start	183	7332
repeat iterate	183	59497
for start	331	6838
for iterate	331	70119
with	793	49644
goto	48	2247
inter-level goto	10	92

The following data is a summary of code fragment counts.

Arithmetic and Logical operators (not sets ops) :		
const op var	1174	131284
const op expr	197	951
var op var	360	14040
var op expr	269	14665
expr op expr	822	183287
set operations	16	8278
of which the most significant entries are :		
add (const op var)	695	118792
or (expr op expr)	261	75212
and (expr op expr)	482	105772
Relational Operators :		
const op var	2852	548656
const op expr	70	1102
var op var	727	171797
var op expr	92	3994
expr op expr	12	46
Dereferencing :		
record access	3842	674652
array access	3584	422085
pointer access	2247	304930

file buffer

of which the most common accessed types were :		
record access (array)	1088	322723
array access (subrange)	1024	272025
pointer access (record)	2060	304924

Summary and Comparisons.

An initial motive for the survey conducted for this thesis was to sample a wider variety of software than Schmidz and ascertain whether his results could be treated as representative of the general use of Pascal. As the new survey only collected static information, that is the basis on which the comparison is made.

Differences in the results could be due to the nature of the software samples used, noting that Schmidz used systems type software whilst the new survey took a larger, more general sample. It must also be noted, however, that the samples are seven years apart and that this sample contains much student code, the style of which is influenced by the advocation of functional programming.

The results of both surveys are summarised, and a compromise figure produced to approximate the character of Pascal programs. Where appropriate, dynamic figures from the code fragment analysis are included for comparison.

Constants.

Schmidz analysed constants found in declarations or in code by type and produced the following results. Our survey only segregated string constants, including single characters, and found they constituted 32.7%[1] of all constants, comparing well with the 23.9 + 10.7 = 34.6% from Schmidz.

---

[1] 7316 string constants were found out of 22355 in-line and 959 declared constants.

Type	Static%	Dynamic%
Scalar	13.4	10.5
Integer	38.4	40.2
String	23.9	4.2
Char	10.7	28.6
Boolean	6.5	3.0
Real	0.1	0.0
Pointer (Nil)	6.3	13.5

Of the non-string constants, this survey found 293 requiring more than 8 bits (2%) and 1600 more than 4 and less than 8 bits (10%). Similar information is not available from Schmidz, as his data is based only on declared constants which tend to have a higher percentage of larger values.

He does give data on the use of implicit constants, almost all integers, used for run-time checking and so not included in the text.

Use	Static%	Dynamic%
Array bounds	69.2	63.5
Assign	21.5	36.0
Value Param	9.6	0.7

Variables.

Both surveys analyse static variable declarations by type and produce the following results.

Type	Schmidz%	This%
Integer	13.8	26.0
Real	0.9	4.6
Char	1.8	3.7
Boolean	10.7	1.3
Scalar	3.7	9.2
Subrange	19.9	9.3
Pointer	20.5	22.1
File	3.4	4.6
Array	16.8	9.7
Record	6.9	8.5
Set	1.8	1.0
Ptr to Record	20.4	21.7
File of Char	2.6	4.0
Array of Char	14.1	4.0

These last groups are also included in the main figures, but indicate a predominant sub-group. The Schmidz survey found 10% more subrange types, but used 12% less integers. This could be explained by the systems nature of the software requiring subranges that would be stored in 8 or 16 bit units. The difference in array percentages is due to far fewer character arrays found in this survey. Other than these points, the results seem consistent.

In the use of variables, this survey found an average 0.45 dereferences as opposed to the 0.29 found by Schmidz. Some of this may be explained by the reduction in use of the 'With' statement. The dynamic figure was 0.57.

Schmidz found that the entire variable was used in 71.14% of cases, and this survey gives 21073 out of 28733 ie. 73.3%. Where dereferencing occurs, it can be broken down as follows :

Access type	Static%	Dynamic%
Field	38.96	40.13
Array	35.24	25.1
Reference	22.78	18.13
File Buff	3.08	16.61

Procedures and Functions.

Our survey found 63% procedure definitions and 37% functions, contrasting with the 87% / 13% balance found by Schmidz. This may reflect the function style of programming advocated in the teaching course.

This survey found 0.87 value parameters. 0.48 reference parameters per definition, totalling 1.35 and approximately agreeing with the 1.25 found by Schmidz. Value parameters make up 64% of parameters in this survey and 60% in Schmidz. Code parameters are insignificant in both cases.

Schmidz analysed formal parameters, other than code parameters, by type, with the following results.

Type	Value	Var
Array	264	124
Pointer	232	110
Subrange	199	44
Integer	134	61
Record	79	259
Scalar	66	31
Boolean	61	105
Set	31	2
Char	23	7
Real	6	3
File	na	87

Note that structured value parameters account for 55% of Schmidz' value parameters, as opposed to 23% in this survey, reflecting the use of character arrays. Simple reference parameters are 30% in Schmidz and 49% in this survey.

Using call figures from statement and expression analysis this survey has 20% of calls to functions; Schmidz has 17%, of which 50% static and 80% dynamic are to standard functions. The high number of standard dynamic calls arises mainly from the use of the 'EOF' function for controlling loops.

Statements.

Schmidz' percentages have been modified to exclude the compound and empty statements which occurred in his analysis. 0.19% of Schmidz statements were labelled, as opposed to 0.04% in this survey. The distributions are -

Statement	Schmidz%	This%	Dynamic%
Assignment	38.3	41.6	40.3
Call	36.6	36.5	24.3
IfThen	6.8	5.1	9.3
IfThenElse	8.6	8.6	20.1
Case	1.0	1.2	0.6
While	2.5	2.3	1.5
Repeat	0.8	0.6	0.4
For	1.5	2.2	0.4
With	3.6	2.0	2.9
Goto	0.3	0.1	0.1

The degree of agreement here is good, with this surveys fewer 'With' statements helping to account for its higher average number of variable dereferences.

The dynamic figures are based on the number of iterations of a loop to reflect the number of tests required. Average iteration counts were While - 5.6; Repeat - 8.1; For - 10.25. There was an average 7.3 selectors per Case. 33.8% of IfThen conditions were successful, 24.9% of IfThenElse.

Within assignments, Schmidz found 66% static, 57% dynamic had the right hand side as a single constant or variable, 14% static, 37% dynamic needed a range check, and 14% static, 3% dynamic were the assignment of structured types.

For later use, a comparison of the number of procedure and function blocks declared relative to statements, is required. This survey finds 10.7%, Schmidz 7.3%.

Expressions.

Schmidz found an average of 1.21 factors per expression compared with the 1.15 found in this survey. He also found 0.21 operators per operand statically and 0.69 dynamically.

This survey, as noted earlier, found more function calls, but additionally has considerably more constants and fewer variables. Note that bracketed sub-expressions and 'NOT' operators are not counted as

factors for these percentages.

Factor Class	Schmidz%	This%	Dynamic%
Variable	51.25	33.0	52.00
Constant	43.41	56.1	37.30
Func Call	4.33	10.2	10.32
Set Expr	0.29	0.7	0.38

The class of operators found by Schmidz is :

Class	Static%	Dynamic%
Relational	52.99	61.77
Add type	23.76	19.10
Mul type	12.74	11.02
Not	10.51	8.28

Schmidz' data suggests that the static proportion of relational to other operators is 53 / 47, whilst this survey has a division of 44 / 56. It also found that 60% of expressions had no operator, whilst Schmidz notes that 66% of assignment expressions are of this type. The proportions of 20 (arithmetic) / 20 (relational) / 60 (no operator) are used in the static results quoted in the main text.

Relative Numbers.

These are used to establish the relative frequencies of the principal language components.

Class	Static		Dynamic
	Schmidz	This	Schmidz
Statements	22167	44814	1941168
Normalised	100	100	100
Operators	7280	-	1175235
Normalised	33		61
Designators[2]	19900	28733	3525705[3]
Normalised	90	64	181
Dereferences	*9861*	*12816*	*1681166*
	44	*29*	*87*
Expressions	33928	43288	1700000[4]
Normalised	153	97	88

Range sizes.

Schmidz collected on the size of various items. The summary here indicates the percentage of the item named in the row which will fit in the number of bits determined by the column.

Item	4 bits	8 bits	16 bits	Average
Variables				
Upper array bnds	82.58	97.82	100	4.02
Upper subr bnds	21.68	50.38	70.15	13.96
Scalar size	84.74	89.65	95.61	4.25
Value Parameters				
Upper array bnds	90.15	99.00	100	3.56
Upper subr bnds	13.64	43.94	61.62	16.28
Scalar size	67.72	75.00	89.76	7.26
Reference Parameters				
Upper array bnds	66.39	95.08	100	4.46
Upper subr bnds	38.64	72.73	80.00	10.39
Scalar size	93.38	97.00	100	2.98

Average number of fields in a record were 6.12 for variables, 6.77 for value parameters and 8.26 for reference parameters.

---

[2] Variables used in expressions. Reference parameters and left hand side of assignments not included.

[3] Utilising 0.33 operators per operand.

[4] Calculated from the number of statements where expressions occur, and tallies with the ratio of operators to expressions.

Conclusion.

The two surveys appear to agree on many aspects of the usage of Pascal. Given the approximate nature of the values, and the use to which they are to be put, an arithmetic mean of the two surveys is used in the main report, except in the few cases where a difference is accountable and a compromise figure has been used. Dynamic figures are modified in accordance with any difference between Schmidz' static figure and the static figure decided upon.

The *Primula* Abstract Machine.

This was designed as an intermediate code form for the first *Primula* compiler, and will also be suitable as the basis of a highly portable *Primula* system. Such a system requires a software emulator, written in some commonly available language such as C or Pascal, and a *Primula* to Abstract Machine code compiler, written in either the common language or in *Primula* and compiled on another system to produce abstract machine code for the new machine. It follows then that any system supporting the common language will then be able to execute *Primula* programs.

The machine as defined here is stack-based, follows the execution model given in the body of the thesis, and provides run time system actions as machine primitives. There are currently no actions to support program tracing or debug activities although the inclusion of such facilities would be desirable from a programming point of view.

What follows is a list of machine instructions and a brief description of the operation. Operands not explicitly given with the instruction are implicitly on the stack unless otherwise stated.

Many instructions contain a type specifier, indicated in the instruction list as (T). This is set by the compiler to indicate Integer (32 bit), Real (32 bit) or LongReal (64 bit operands).

Relational Operators : Take two arguments of given type from stack and place integer value zero or one onto stack. Structural operators have length word on TOS with structures beneath.

EQUALS(T), NOTEQUALS(T), LESS(T), LESSEQUAL(T), GREATER(T), GREATEREQUAL(T), EQUALSTRUCT, NOTEQUALSTRUCT.
(For set relations see Set Operators.)

Unary Operators : Take argument of given type from stack and place result onto stack.

NEGATE(T), BITNEGATE, LOGICALNOT, CONVERT(T)[5].

Binary Operators : Take pair of arguments from stack and place result onto stack.

ADD(T), SUB(T), MUL(T), DIV(T), MOD(T), BITOR, BITAND.

Set Operators : Top of stack is taken as length in words of the set operands beneath. Operators remove sets and place result on stack, with length word on top if result if of set type.

UNION, DIFFERENCE, INTERSECTION, IN, SUBSET, SUBSETEQUAL, SUPERSET, SUPERSETEQUAL.

CONSTRUCT — Adds word element on top of stack to set underneath (length is at TOS - 1).

CONSTRUCTRANGE — As above but two TOS words specify range of elements.

EMPTYSET — Creates empty set of given length on stack and places length on TOS.

Load / Store Operators: Operate on store addresses. Structures have length specified on TOS.

LOAD(T), STORE(T), LOADSTRUCT, STORESTRUCT.

Stack Manipulation : Operate on TOS.

DUPLICATE(T), SWAP(T), REMOVE(T).

RESERVE — Reserve n locations on stack where n is value on TOS.

COLLAPSE — Remove n locations from TOS.

Special purpose :

FORSTART — Stack contains address of loop variable, final value and increment. Instruction pushes boolean indicating if to exit loop.

FOREND — Stack as above. Instruction adds increment to loop variable.

GUARDTEST — Tests boolean word on stack and if zero performs run time system action for guard condition failure.

REFERENCE [LexLevel] [LexOffset] Put address of given item onto TOS.

NEWOBJECT — Stack contains code address of object, parameters and parameter length on TOS. Removes these and returns reference to object on TOS.

OBJECTACCESS [Entry]Object reference on TOS. Instruction only completes when access finished and results returned onto TOS.

---

[5] Converts integer to real or real to longreal according to T. There is no longreal variant.

Constants : The instruction contains a constant of the appropriate type. The string constant is null terminated and the instruction leaves the length in words on TOS.

CONST(T) [Value], STRINGCONST [String]

Control Flow : Conditionals use boolean word on stack, addresses are absolute and specified by instruction. Call and Return adjust display, set frame pointers etc. Return to address zero signals end of access to run-time support.

CONDITIONALTRUE [Address], CONDITIONALFALSE [Address],
BRANCH [Address], CALL [Address], RETURN

As specified, this machine will not be particularly efficient because of the generalised nature of the operations available. For example, an instruction which loaded to the top of stack the value at a given lexical offset in the current frame would save the frequent REFERENCE / LOAD sequence.

Similar optimisations can be made in the handling of structures by providing operations which have the structure size as an immediate operand rather than requiring it to be loaded onto the stack.

Evaluation Benchmark Programs.

The following contains the Modula-2 code of the four benchmarks used for evaluation purposes. To reduce size, M68000 and *RASP* assembler is only included for the QuickSort benchmark, to give a flavour of the code quality used.

Also to save space, the untimed parts of the benchmarks, such as the data input routines, are omitted.

Sieve : Modula-2.

```
PROCEDURE Sieve() : CARDINAL;
VAR i, k, Prime, Count : CARDINAL;
    Flags : ARRAY [ 0 .. Size ] OF BOOLEAN;
BEGIN
  Count := 0;
  FOR i := 0 TO Size DO
    Flags [ i ] := TRUE;
  END;
  FOR i := 0 TO Size DO
    IF Flags [ i ] THEN (* Found prime. *)
      Prime := 2 * i + 3;
      k := ( Prime * Prime - 3 ) DIV 2;(* or i + Prime *)
      WHILE k <= Size DO   (* multiples are non-prime. *)
        Flags [ k ] := FALSE;
        k := k + Prime;
      END;
      INC( Count );
    END;
  END;
  RETURN Count;
END Sieve;
```

Gauss : Modula-2.

```
TYPE
  Index  = [ 0 .. Size ];
  Vector = ARRAY Index OF Value;
  Matrix = ARRAY Index OF Vector;

PROCEDURE Triangularise();
VAR
  i, j : Index;
  Multiplier, Temp : Value;
BEGIN
  FOR i := 1 TO Size DO
    IF NonZero( Coeffs [ i, i-1 ] ) THEN
      Divide( Coeffs [ i-1, i-1 ], Coeffs [ i, i-1 ], Multiplier );

      FOR j := i TO Size DO
        Mult( Coeffs [ i, j ], Multiplier, Temp );
        Sub( Coeffs [ i-1, j ], Temp, Coeffs [ i, j ] );
      END;
      Mult( Sums [ i ], Multiplier, Sums [ i ] );
      Sub( Sums [ i-1 ], Sums [ i ], Sums [ i ] );

    END;
  END;
END Triangularise;

PROCEDURE SolveBack() : BOOLEAN;
VAR
  i, j : Index;
  Temp, Total : Value;
BEGIN
  FOR i := Size TO 0 BY -1 DO
    Set( 0, Total );
    FOR j := i + 1 TO Size DO
      Mult( Coeffs [ i, j ], Unknowns [ j ], Temp );
      Add( Total, Temp, Total );
    END;

    IF NonZero( Coeffs [ i, i ] ) THEN
      Sub( Sums [ i ], Total, Temp );
      Divide( Temp, Coeffs [ i, i ], Unknowns [ i ] );
    ELSE
      RETURN FALSE (* Error exit. *)
    END;
  END;
  RETURN TRUE;
END SolveBack;
```

```
(* Numbers module to support Gauss. *)

TYPE
  Value = RECORD
            Num, Denom : INTEGER;
          END;

PROCEDURE Tidy( n, d : LONGINT; VAR v : Value );
VAR
  x, y : LONGCARD;
BEGIN
  IF ( n <> 0 ) AND ( d <> 0 ) THEN
    IF ( n < 0 ) AND ( d < 0 ) THEN
      n := - n;
      d := - d;
    END;
    IF n < 0 THEN
      x := LONGCARD( - n )
    ELSE
      x := LONGCARD( n );
    END;
    IF d < 0 THEN
      y := LONGCARD( - d )
    ELSE
      y := LONGCARD( d )
    END;
    WHILE x <> y DO
      IF x > y THEN
        x := x - y
      ELSE
        y := y - x
      END;
    END;
    n := n DIV LONGINT( x );
    d := d DIV LONGINT( x );
    WHILE ( n > 65535 ) AND ( d > 65535 ) DO
      n := n DIV 2;
      d := d DIV 2;
    END;
    v.Num := INTEGER( n );
    v.Denom := INTEGER( d );
  ELSE
    v.Num := 0;
    v.Denom := 1;
  END;
END Tidy;

PROCEDURE Set( i : INTEGER; VAR v : Value );
BEGIN
  Tidy( LONGINT( i ), 1, v );
END Set;

PROCEDURE Add( i, j : Value; VAR v : Value );
BEGIN
  Tidy( LONGINT( ( i.Num * j.Denom ) + ( j.Num * i.Denom ) ),
        LONGINT( i.Denom * j.Denom ),
        v );
END Add;
```

```
PROCEDURE Sub( i, j : Value; VAR v : Value );
BEGIN
  Tidy( LONGINT( ( i.Num * j.Denom ) - ( j.Num * i.Denom ) ),
        LONGINT( i.Denom * j.Denom ),
        v );
END Sub;

PROCEDURE Mult( i, j : Value; VAR v : Value );
BEGIN
  Tidy( LONGINT( i.Num * j.Num ), LONGINT( i.Denom * j.Denom ), v );
END Mult;

PROCEDURE Divide( i, j : Value; VAR v : Value );
BEGIN
  Tidy( LONGINT( i.Num * j.Denom ), LONGINT( i.Denom * j.Num ), v );
END Divide;

PROCEDURE NonZero( i : Value ) : BOOLEAN;
BEGIN
  RETURN i.Num <> 0;
END NonZero;

PROCEDURE WriteNumber( i : Value );
BEGIN
  WriteInt( i.Num, 1 );
  WriteString( " / " );
  WriteInt( i.Denom, 1 );
  WriteLn;
END WriteNumber;

BEGIN
END Numbers.
```

QuickSort : Modula-2.

```
PROCEDURE QSort( LowBound, HighBound : SortIndex );
VAR
  Partition, Temp : LONGCARD;
  Low, High : SortIndex;
BEGIN
  Partition := TheData [ ( LowBound + HighBound ) DIV 2 ];
  Low := LowBound;
  High := HighBound;
  WHILE Low < High DO
    WHILE TheData [ Low ] < Partition DO
      INC( Low );
    END;
    WHILE TheData [ High ] > Partition DO
      DEC( High );
    END;
    IF Low < High THEN
      Temp := TheData [ Low ];
      TheData [ Low ] := TheData [ High ];
      TheData [ High ] := Temp;
      INC( Low );  (* Avoid loop with many values = Partition. *)
      DEC( High );
    END;
  END;
  IF Low = High THEN (* Decide where to place this value. *)
    IF TheData [ Low ] < Partition THEN
      INC( Low )                          (* Place in lower. *)
    ELSIF TheData [ Low ] > Partition THEN
      DEC( High )                         (* Place in upper. *)
    ELSE                                  (* Eliminate from sort. *)
      IF Low < HighBound THEN (* Can't guarantee this condition. *)
        INC( Low )
      END;
      IF High > LowBound THEN
        DEC( High )
      END;
    END;
  END; (* If Low = High ... *)
  IF LowBound < High THEN
    QSort( LowBound, High );
  END;
  IF Low < HighBound THEN
    QSort( Low, HighBound );
  END;
END QSort;
```

QuickSort : M68000 Assembler.

```
QSort:
* Recursive procedure. Bound parameters in D1,D2.

LowBound  equr D1
HighBound equr D2
Low       equr D3
High      equr D4
Partition equr D5

     MOVEM.L   D1-D7/A3,-(SP)
     MOVE      38(SP),LowBound
     MOVE      36(SP),HighBound

* Find partition
     MOVE      LowBound,D6
     ADD       HighBound,D6
     ASR       #1,D6                Div 2
     ASL       #2,D6                LongWord addressing
     MOVE.L    0(A2,D6),Partition

     MOVE      LowBound,Low
     MOVE      HighBound,High

OuterLoop:
* While Low < High Do
     CMP       Low,High
     BLE       CheckMiddle

LowLoop:
* While TheData [ Low ] < Partition DO INC( Low )
     MOVE      Low,D6
     ASL       #2,D6
     CMP.L     0(A2,D6),Partition
     BLE       HighLoop
     ADDQ      #1,Low
     BRA       LowLoop

HighLoop:
* While TheData [ High ] > Partition DO DEC( High )
     MOVE      High,D6
     ASL       #2,D6
     CMP.L     0(A2,D6),Partition
     BGE       TrySwap
     SUBQ      #1,High
     BRA       HighLoop

TrySwap:
* If Low < High Then Swap( TheData [ Low ], TheData [ High ] ) and
*    Inc( Low ), Dec( High )
     CMP       Low,High
     BLE       EndOuter

     MOVE      Low,D6
     ASL       #2,D6
     MOVE.L    0(A2,D6),A3  use A3 as temporary
     MOVE      High,D7
```

```
        ASL         #2,D7
        MOVE.L      0(A2,D7),0(A2,D6)
        MOVE.L      A3,0(A2,D7)
        ADDQ        #1,Low
        SUBQ        #1,High

EndOuter:
        BRA         OuterLoop

CheckMiddle:
* If Low = High Then
        CMP         Low,High
        BNE         EndCheckMiddle

* If TheData [ Low ] < Partition Then Inc( Low )
        MOVE        Low,D6
        ASL         #2,D6
        CMP.L       0(A2,D6),Partition
        BLE         TestHigh
        ADDQ        #1,Low
        BRA         EndCheckMiddle

TestHigh:
* Elsif TheData [ Low ] > Partition Then Dec( High )
        BGE         TestBounds
        SUBQ        #1,High
        BRA         EndCheckMiddle

TestBounds:
* Else
* If Low < HighBound Then Inc( Low )
        CMP         Low,HighBound
        BLE         TestHighBound
        ADDQ        #1,Low
TestHighBound:
* If High > LowBound Then Dec( High )
        CMP         High,LowBound
        BGE         EndCheckMiddle
        SUBQ        #1,High

EndCheckMiddle:
* If LowBound < High Then QSort( LowBound, High )
        CMP         LowBound,High
        BLE         TryNextCall

        MOVE        LowBound,-(SP)
        MOVE        High,-(SP)
        JSR         QSort
        ADDQ.L      #4,SP

TryNextCall:
* If Low < HighBound Then QSort( Low, HighBound )
        CMP         Low,HighBound
        BLE         EndQSort

        MOVE        Low,-(SP)
        MOVE        HighBound,-(SP)
        JSR         QSort
```

```
        ADDQ.L      #4,SP

EndQSort:
        MOVEM.L     (SP)+,D1-D7/A3
        RTS
```

QuickSort : *RASP* Assembler.

```
        align
Multiply
* Uses simple shift and add mechanism.
* Entry code
        PUSH    @FrameBaseReg,@DummyReg      Save dynamic ptr
        MOVE    @StackPointer,@FrameBaseReg HW dumps old frame
        SUB     #3,@FrameBaseReg             Set frame base
        MOVE    @DisplayReg,@5
        ADD     #4,@5                        new lex level * 4
        LOAD    @5,@4                        old static entry
        STORE   @FrameBaseReg,@5             set new display
        ADD     #2,@StackPointer             space for fixed frame

* Multiply body
        MOVE    #0,@5                        Used as accumulator
MultLoop
        MOVE    #0,@CarryReg                 Ensure zeros at left of @1
        ROR     @1,@1
        BREQ    #0,@CarryReg,>NoCarry
        ADD     @0,@5
*       BRIN    #1,@StatusReg,<MultOverflow Overflow trap
NoCarry
        ASL     @0,@0
*       BRIN    #1,@StatusReg,<MultOverflow Overflow trap
        BRNE    #0,@1,>MultLoop              Done if multiplier = 0
EndMultLoop
        MOVE    @5,@0

* Exit code
        MOVE    @DisplayReg,@5
        ADD     #4,@5
        STORE   @4,@5
        MWSF    #1                          Return result in R0
        MOVE    @3,@FrameBaseReg
        SUB     #3,@StackPointer
        RET     @DummyReg,@DummyReg

MultOverflow
        EXTEND  #16,@DummyReg               Exception and halt
* Never return here

        align
QSort
* Entry code
        PUSH    @FrameBaseReg,@DummyReg      Save dynamic ptr
        MOVE    @StackPointer,@FrameBaseReg HW dumps old frame
        SUB     #3,@FrameBaseReg             Set frame base
        MOVE    @DisplayReg,@5
        ADD     #8,@5                        new lex level * 4
        LOAD    @5,@4                        old static entry
        STORE   @FrameBaseReg,@5             set new display
        ADD     #13,@StackPointer            space for fixed frame

* Main body
        MOVE    @DisplayReg,@0
        ADD     #4,@9
```

```
        LOAD    @9,@9                           Address of LL1
        ADD     #3,@9                           Address of TheData
        ASL2    @9,@9                           Convert to byte address
        LOAD    @9,@9                           Pointer to array
        MOVE    @0,@10
        ADD     @1,@10
        ASR     @10,@10
        ASL2    @10,@10                         Convert to byte form
        SUB     #4,@10                          Offset
        ADD     @10,@9
        LOAD    @9,@5                           Partition := TheData [ x ]

        MOVE    @0,@7
        MOVE    @1,@8
QOuterLoop
        BRGE    @8,@7,<EndQOuterLoop

QLowScanLoop
        MOVE    @DisplayReg,@9
        ADD     #4,@9
        LOAD    @0,@9
        ADD     #3,@9
        ASL2    @9,@9
        LOAD    @9,@9                           Pointer to array
        ASL2    @7,@10
        SUB     #4,@10
        ADD     @10,@9                          Address of TheData [ Low ]
        LOAD    @9,@9
        BRGE    @5,@9,<EndQLowScanLoop
        BRA     #1,@7,>QLowScanLoop
EndQLowScanLoop

QHighScanLoop
        MOVE    @DisplayReg,@9
        ADD     #4,@9
        LOAD    @9,@9
        ADD     #3,@9
        ASL2    @9,@9
        LOAD    @9,@9                           Pointer to array
        ASL2    @8,@10
        SUB     #4,@10
        ADD     @10,@9
        LOAD    @9,@9                           TheData [ High ]
        BRLE    @5,@9,<EndQHighScanLoop
        SUB     #1,@8
        BRA     @DummyReg,@DummyReg,>QHighScanLoop
EndQHighScanLoop

        BRGE    @8,@7,<QEndSwap
        MOVE    @DisplayReg,@9
        ADD     #4,@9
        LOAD    @9,@9
        ADD     #3,@9
        ASL2    @9,@9
        LOAD    @9,@9                           Pointer to array
        ASL2    @7,@10
        SUB     #4,@10
        ADD     @9,@10
```

```
* Assumes compiler can maintain pointer to array in a temporary
* whilst no branches possible.
        LOAD    @10,@6                          Temp := TheData [ Low ]
        ASL2    @8,@11
        SUB     #4,@11
        ADD     @9,@11
        LOAD    @11,@12
        STORE   @12,@10
        STORE   @6,@11
        ADD     #1,@7
        SUB     #1,@8
QEndSwop
EndQOuterLoop

        BRNE    @7,@8,>EndQMiddleValue
        MOVE    @DisplayReg,@9
        ADD     #4,@9
        LOAD    @9,@9
        ADD     #3,@9
        ASL2    @9,@9
        LOAD    @9,@9                                   Pointer to array
        ASL2    @7,@10
        SUB     #4,@10
        ADD     @10,@9
        LOAD    @9,@9
        BRGE    @5,@9,>QNextMiddleTest
        BRA     #1,@7,>EndQMiddleValue
QNextMiddleTest
        MOVE    @DisplayReg,@9
        ADD     #4,@9
        LOAD    @9,@9
        ADD     #3,@9
        ASL2    @9,@9
        LOAD    @9,@9                                   Pointer to array
        ASL2    @7,@10
        SUB     #4,@10
        ADD     @10,@9
        LOAD    @9,@9
        BRLE    @5,@9,>QMiddleEquals
        SUB     #1,@8
        BRA     @DummyReg,@DummyReg,>EndQMiddleValue
QMiddleEquals

        BRGE    @1,@7,>QMiddleEquals2
        ADD     #1,@7
QMiddleEquals2
        BRLE    @0,@8,>EndQMiddleValue
        SUB     #1,@8

EndQMiddleValue

        BRGE    @8,@0,>QSecondCall
        PUSH    @0,@DummyReg
        PUSH    @8,@DummyReg
        CALL    #QSort
        SUB     #2,@StackPointer
QSecondCall
        BRGE    @1,@7,>EndQSecondCall
```

```
        PUSH    @7,@DummyReg
        PUSH    @1,@DummyReg
        CALL    #QSort
        SUB     #2,@StackPointer
End@SecondCall

* Exit code
        MOVE    @DisplayReg,@5
        ADD     #8,@5
        STORE   @4,@5                       restore display
        MUBF    #0                          Mask write backs
        MOVE    @3,@FrameBaseReg            restore dynamic
        SUB     #14,@StackPointer           adjust SP
        RET     @DummyReg,@DummyReg         Dummy operands
```

StringSearch : Module=2.

```
PROCEDURE Match( VAR Text, Pattern : ARRAY OF CHAR ) : CARDINAL;
VAR
  t, t2, p : CARDINAL;
  TextLen, PatternLen : CARDINAL;

  PROCEDURE GetLength( VAR Str : ARRAY OF CHAR ) : CARDINAL;
  (* Return string length, excluding terminator char. *)
  VAR i : CARDINAL;
  BEGIN
    i := 0;
    WHILE Str [ i ] <> Terminator DO
      INC( i );
    END;
    RETURN i;
  END GetLength;

  PROCEDURE SetRightMost( VAR Pattern : ARRAY OF CHAR;
                          PatternLen : CARDINAL );
  (* Set array to indicate distance of rightmost position of character
     index from end of the pattern. *)
  VAR c : CHAR;
      i : CARDINAL;
  BEGIN
    FOR c := 0C TO 255C DO
      RightMost [ c ] := PatternLen;
    END;
    i := 0;
    WHILE i < PatternLen DO
      RightMost [ Pattern [ i ] ] := PatternLen - 1 - i;
      INC( i );
    END;
  END SetRightMost;

  PROCEDURE SetSentinel( VAR Text, Pattern : ARRAY OF CHAR;
                         TextLen, PatternLen : CARDINAL );
  VAR i : CARDINAL;
  BEGIN
    FOR i := 0 TO PatternLen - 1 DO
      Text [ TextLen ] := Pattern [ i ];
      INC( TextLen );
    END;
  END SetSentinel;

BEGIN
  PatternLen := GetLength( Pattern );
  IF PatternLen = 0 THEN
    RETURN 0;
  END;
  TextLen := GetLength( Text );
  SetRightMost( Pattern, PatternLen );
  SetSentinel( Text, Pattern, TextLen, PatternLen );
  t := PatternLen - 1;

  REPEAT
    WHILE Pattern [ PatternLen - 1 ] <> Text [ t ] DO
      t := t + RightMost [ Text [ t ] ];
```

```
      END;
      p := 0;
      t2 := t - PatternLen + 1;
      WHILE ( p < PatternLen ) AND ( Pattern [ p ] = Text [ t2 ] ) DO
        INC( p );
        INC( t2 );
      END;
      INC( t );
    UNTIL p = PatternLen;

    Text [ TextLen ] := Terminator; (* Remove sentinal. *)
    RETURN t - PatternLen;          (* t incremented at end of loop. *)
  END Match;
```

StringSearch : M68000 Assembler.

```
GetLength:
* Returns result in D0

      MOVE.L    A0,-(SP)
      MOVE.L    8(SP),A0            Get argument
      CLR       D0
GetLenLoop:
      CMP.B     #Terminator,(A0)+
      BEQ       EndGetLenLoop
      ADDQ      #1,D0
      BRA       GetLenLoop
EndGetLenLoop:

      MOVE.L    (SP)+,A0
      RTS

SetRightMost:
      MOVEM.L   D0-D3/A0-A2,-(SP)
      MOVE.L    34(SP),A0           Pattern
      MOVE      32(SP),D0           PatternLen

      MOVE.L    #RightMost,A1
      MOVE      #255,D1
InitLoop:
      MOVE      D0,(A1)+
      DBRA      D1,InitLoop

      CLR       D1                  Loop count
      MOVE.L    #RightMost,A1
      MOVE.L    A0,A2               Use to move through Pattern
      CLR       D2                  Used as indexing register
SetLoop:
      CMP       D0,D1
      BGE       EndSetLoop

      MOVE.B    (A2)+,D2            Access Pattern
      ASL       #1,D2               Array elements are 2 bytes
      MOVE      D0,D3
      SUBQ      #1,D3
      SUB       D1,D3
      MOVE      D3,0(A1,D2.W)       Store in RightMost
      ADDQ      #1,D1
      BRA       SetLoop
EndSetLoop:

      MOVEM.L   (SP)+,D0-D3/A0-A2
      RTS

SetSentinel:
      MOVEM.L   D0-D1/A0-A1,-(SP)
      MOVE.L    28(SP),A0           Text
      MOVE.L    24(SP),A1           Pattern
      CLR.L     D0                  Used in address arithmetic
      MOVE      22(SP),D0           TextLen
      MOVE      20(SP),D1           PatternLen
```

```
        SUBQ        #1,D1
        ADD.L       D0,A0                   End of text
SentinelLoop:
        MOVE.B      (A1)+,(A0)+
        DBRA        D1,SentinelLoop

        MOVEM.L     (SP)+,D0-D1/A0-A1
        RTS

Match:
        MOVEM.L     D1-D4/A0-A3,-(SP)
        MOVE.L      40(SP),A0               Text
        MOVE.L      36(SP),A1               Pattern

        MOVE.L      A1,-(SP)
        JSR         GetLength
        ADDQ.L      #4,SP
        MOVE        D0,D1                   PatternLen
        TST         D1
        BEQ         EndMatch                Returns D0=0 if null pattern
        MOVE.L      A0,-(SP)
        JSR         GetLength
        ADDQ.L      #4,SP                   TextLen in D0

        MOVE.L      A1,-(SP)
        MOVE        D1,-(SP)
        JSR         SetRightMost
        ADDQ.L      #6,SP
        MOVE.L      A0,-(SP)
        MOVE.L      A1,-(SP)
        MOVE        D0,-(SP)
        MOVE        D1,-(SP)
        JSR         SetSentinel
        ADD.L       #12,SP

        MOVE        D1,D2
        SUBQ        #1,D2               t := PatternLen - 1;
OuterMatchLoop:

        MOVE        D1,D3
        SUBQ        #1,D3
        MOVE.B      0(A1,D3.W),D3   D3 := Pattern [ PatternLen - 1 ]
MainSkippingLoop:
        MOVE.B      0(A0,D2.W),D4   D4 := Text [ t ]
        CMP.B       D3,D4
        BEQ         EndMainSkippingLoop
        MOVE.L      #RightMost,A2
        CLR         D4                  Indexing register into R/Most
        MOVE.B      0(A0,D2.W),D4   D4 := Text [ t ]
        ASL         #1,D4               Array elements are 2 bytes.
        MOVE        0(A2,D4.W),D4   D4 := RightMost [ Text [ t ] ]
        ADD         D4,D2
        BRA         MainSkippingLoop
EndMainSkippingLoop:

        CLR         D3                  p := 0;
        MOVE.L      A1,A2               Pattern address
        CLR.L       D4                  Used in address arithmetic
```

```
        MOVE        D2,D4
        SUB         D1,D4
        ADDQ        #1,D4           t2 := t - PatternLen + 1
        MOVE.L      A0,A3           Text address
        ADD.L       D4,A3           Text [ t2 ] address

CharMatchLoop:
        CMP         D1,D3
        BGE         EndCharMatchLoop
        CMP.B       (A2)+,(A3)+
        BNE         EndCharMatchLoop
        ADDQ        #1,D3
        BRA         CharMatchLoop
EndCharMatchLoop:

        ADDQ        #1,D2
        CMP         D1,D3
        BNE         OuterMatchLoop

        MOVE.B      #Terminator,0(A0,D0.W)   Remove sentinal
        MOVE        D2,D0
        SUB         D1,D0
EndMatch:
        MOVEM.L     (SP)+,D1-D4/A0-A3
        RTS
```

StringSearch : *BASP Assembler.*

```
        align
Match
* Entry code
        PUSH    @FrameBaseReg,@DummyReg      Save dynamic ptr
        MOVE    @StackPointer,@FrameBaseReg HW dumps old frame
        SUB     #4,@FrameBaseReg             Set frame base
        MOVE    @DisplayReg,@6
        ADD     #8,@6                        new lex level * 4
        LOAD    @6,@5                        old static entry
        STORE   @FrameBaseReg,@6             set new display
        ADD     #8,@StackPointer             space for fixed frame
        BRA     @DummyReg,@DummyReg,>MatchBody

        align
GetLength
* Entry code
        PUSH    @FrameBaseReg,@DummyReg      Save dynamic ptr
        MOVE    @StackPointer,@FrameBaseReg HW dumps old frame
        SUB     #3,@FrameBaseReg             Set frame base
        MOVE    @DisplayReg,@5
        ADD     #12,@5                       new lex level * 4
        LOAD    @5,@4                        old static entry
        STORE   @FrameBaseReg,@5             set new display
        ADD     #3,@StackPointer             space for fixed frame

* Main body
        MOVE    #0,@5
GetLengthLoop
        MOVE    @1,@6
        ADD     @5,@6
        LOADB   @6,@6
        BREQ    #0,@6,<EndGetLengthLoop
        BRA     #1,@5,>GetLengthLoop
EndGetLengthLoop
        MOVE    @5,@0

* Exit code
        MOVE    @DisplayReg,@5
        ADD     #12,@5
        STORE   @4,@5                        restore display
        MWBF    #1                           Mask write backs
        MOVE    @3,@FrameBaseReg             restore dynamic
        SUB     #4,@StackPointer             adjust SP
        RET     @DummyReg,@DummyReg          Dummy operands

        align
SetRightMost
* Entry code
        PUSH    @FrameBaseReg,@DummyReg      Save dynamic ptr
        MOVE    @StackPointer,@FrameBaseReg HW dumps old frame
        SUB     #3,@FrameBaseReg             Set frame base
        MOVE    @DisplayReg,@5
        ADD     #12,@5                       new lex level * 4
        LOAD    @5,@4                        old static entry
        STORE   @FrameBaseReg,@5             set new display
        ADD     #6.@StackPointer             space for fixed frame
```

```
* Main body
        MOVE    #0,@5
        CONST   #255
        ASL2    @StackPointer,@7
        LOAD    @7,@7
        SUB     #1,@StackPointer
SRMInitLoop
        BRGT    @7,@5,>EndSRMInitLoop
        MOVE    @DisplayReg,@8
        ADD     #4,@8
        LOAD    @8,@8
        ADD     #5,@8
        ASL2    @8,@8
        LOAD    @8,@8                   pointer to RightMost

        ASL2    @5,@9                   convert to byte address
        ADD     @9,@8
        STORE   @1,@8

        BRA     #1,@5,>SRMInitLoop
EndSRMInitLoop

        MOVE    #0,@6
SRMMainLoop
        BRGE    @1,@6,>EndSRMMainLoop

        MOVE    @DisplayReg,@7
        ADD     #4,@7
        LOAD    @7,@7
        ADD     #5,@7
        ASL2    @7,@7
        LOAD    @7,@7                   pointer to RightMost

        MOVE    @0,@8
        ADD     @6,@8
        LOADB   @8,@8                   Pattern [ i ]

        ASL2    @8,@8
        ADD     @8,@7                   indexed

        MOVE    @1,@8
        SUB     #1,@8
        SUB     @6,@8
        STORE   @8,@7                   store into RightMost

        BRA     #1,@6,>SRMMainLoop
EndSRMMainLoop

* Exit code
        MOVE    @DisplayReg,@5
        ADD     #12,@5
        ADD     #1,@2
        BRA     #1,@7,>SetSentinelLoop
EndSetSentinelLoop

* Exit code
        MOVE    @DisplayReg,@7
```

```
        ADD     #12,@7
        STORE   @6,@7                   restore display
        MWBF    #0                      Mask write backs
        MOVE    @5,@FrameBaseReg        restore dynamic
        SUB     #6,@StackPointer        adjust SP
        RET     @DummyReg,@DummyReg     Dummy operands

MatchBody
        ADD     #1,@StackPointer
        PUSH    @2,@DummyReg
        CALL    #GetLength
        SUB     #1,@StackPointer
        ASL2    @StackPointer,@11
        LOAD    @11,@10
        SUB     #1,@StackPointer
        BRNE    #0,@10,>NotZero
        MOVE    #0,@0
        BRA     @DummyReg,@DummyReg,>Exit
NotZero
        ADD     #1,@StackPointer
        PUSH    @1,@DummyReg
        CALL    #GetLength
        SUB     #1,@StackPointer
        ASL2    @StackPointer,@11
        LOAD    @11,@9
        SUB     #1,@StackPointer

        PUSH    @2,@DummyReg
        PUSH    @10,@DummyReg
        CALL    #SetRightMost
        SUB     #2,@StackPointer
        PUSH    @1,@DummyReg
        PUSH    @2,@DummyReg
        PUSH    @9,@DummyReg
        PUSH    @10,@DummyReg
        CALL    #SetSentinel
        SUB     #4,@StackPointer
        MOVE    @10,@6
        SUB     #1,@6

MainMatchLoop
SkippingLoop
        MOVE    @2,@11
        ADD     @10,@11
        SUB     #1,@11
        LOADB   @11,@11
        MOVE    @1,@12
        ADD     @6,@12
        LOADB   @12,@12
        BREQ    @11,@12,<EndSkippingLoop

        MOVE    @DisplayReg,@11
        ADD     #4,@11
        LOAD    @11,@11
        ADD     #5,@11
        ASL2    @11,@11
        LOAD    @11,@11                 pointer to RightMost
```

```
        MOVE    @1,@12
        ADD     @6,@12
        LOADB   @12,@12
        ASL2    @12,@12                 convert to index R'Most
        ADD     @12,@11
        LOAD    @11,@11
        ADD     @11,@6                  t := t + RM [ Text [ t ] ]

        BRA     @DummyReg,@DummyReg,>SkippingLoop
EndSkippingLoop

        MOVE    #0,@8
        MOVE    @6,@7
        SUB     @10,@7
        ADD     #1,@7

MatchingLoop
        BRGE    @10,@8,<EndMatchingLoop
        MOVE    @2,@11
        ADD     @8,@11
        LOADB   @11,@11
        MOVE    @1,@12
        ADD     @7,@12
        LOADB   @12,@12
        BRNE    @11,@12,<EndMatchingLoop
        ADD     #1,@8
        BRA     #1,@7,>MatchingLoop
EndMatchingLoop

        ADD     #1,@8
        BRNE    @10,@8,>MainMatchLoop

        MOVE    @1,@11
        ADD     @9,@11
        STOREB  #0,@11
        MOVE    @6,@0
        SUB     @10,@0

Exit
x Exit code
        MOVE    @DisplayReg,@5
        ADD     #8,@5
        STORE   @5,@6                   restore display
        MWBF    #1                      Mask write backs
        MOVE    @4,@FrameBaseReg        restore dynamic
        SUB     #9,@StackPointer        adjust SP
        RET     @DummyReg,@DummyReg     Dummy operands
```

References.

[AGHA] "An Overview of Actor Languages". G. Agha, Object-Oriented Programming Workshop, SIGPLAN 10/1986.

[AHOV] "Principles of Compiler Design". V. Aho and J. Ullman, Addison-Wesley, 1979.

[ANS1] "ANS Fortran", ANSI 1966 and 1977.

[ANS2] "ANS Programming Language COBOL", ANSI 1974 and 1985.

[AT&T] "Unix System V - Release 2", AT&T, 1983.

[BALM] "Programmers Guide to GEM". P. Balma and W. Fitter, Sybex, 1986.

[BER1] "The Case Against Multiple Inheritance". J. Bergin, Dept of Computing Science and Mathematics, Marist College, Poughkeepsie, NY 12601.

[BER2] "Object Modula : An Extension to Modula-2". J. Bergin, Dept of Computer Science and Mathematics, Marist College, Poughkeepsie, NY 12601.

[BRIN] "Brinch Hansen On Pascal Compilers". P. Brinch Hansen, Prentice-Hall, 1985.

[BUCK] "The ICL 2900 Series", J.K. Buckle, Macmillan, 1979.

[BURS] "HOPE : An experimental applicative language". R.M. Burstall et al, Technical Report, University of Edinburgh 1980.

[CARD] "Modula-3 Report". L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson, Digital Systems Research Center, 1988.

[CHAR] "Software Engineering Environments". R. Charette, McGraw-Hill Inc, 1986.

[CHUR] "The Calculi of Lambda Conversion". A. Church, Princeton University Press, 1941.

[CLAR] "PARLOG : Parallel Programming in Logic". K.L. Clark and S. Gregory, Research Report DOC 84/4, Imperial College, London 1984.

[DAHL] "SIMULA 67 Common Base Language". O.J. Dahl, B. Myrhaug, K. Nygaard, Norwegian Computing Centre, Oslo, 1968.

[DAR1] "A Functional Programming Environment Supporting Execution, Partial Execution and Transformation". J. Darlington et al, Imperial College Internal Report, 1988.

[DAR2] "ALICE A Multiprocessor Reduction Machine for Parallel Evaluation of Applicative Languages". J. Darlington, M. Reeve, ACM Comp. Arch. 1981.

[DIGI] "VAX Architecture Reference Manual". Digital Press, 1987.

[DJIK] "Goto Statement Considered Harmful". Djikstra, Comm. of the ACM, Vol 11/3, 1968.

[GOLD] "SMALLTALK-80 : The language and its implementation". A. Goldberg, D. Robson, Addison-Wesley, 1983.

[HAIL] "A Model for Object Based Inheritance". B. Hailpern, V. Nguyen, from "Research Directions in Object-Oriented Programming", MIT Press, 1987.

[HENN] "Organisation and VLSI Implementation of MIPS". J. Hennessy et al, paper contained in [IEE2].

[HINZ] "Control Data STAR-100". R. Hinz and D. Tate, CompCon Digest, 1972.

[HOA1] "Communicating Sequential Processes". C.A.R. Hoare, CACM Vol 21, No.8. 1978.

[HOA2] "Monitors : An Operating System Structuring Concept". C.A.R. Hoare, CACM Vol 17/10, 1974.

[IBMC] "IBM System 370 Principles of Operation". IBM Corp. 1978.

[ICHB] "Reference Manual for the ADA Programming Language". Ichbiah et al, ANSI, 1983.

[IEE1] "Floating point formats", 1982.

[IEE2] "Tutorial on Advanced Microprocessors and High Level Language Computer Architecture". IEEE, 1986.

[IEE3] "VLSI In Computers", IEEE, 1985.

[INMO] "Occam Programming Manual". INMOS Ltd, Prentice-Hall.

[JONK] "Abstraction, Specification and Implementation techniques with an application to Garbage Collection". H.Jonkers, Mathematish Centrum, 1983.

[KATE] "Reduced Instruction Set Architecture for VLSI". M. Katevenis, MIT Press, 1985.

[KERN] "The C Programming Language". B. Kernighan, D. Ritchie, Prentice-Hall, 1978 and 1988.

[KHAN] "Vulcan: Logical Concurrent Objects". K. Kahn, E. Tribble, M. Miller, D. Brobow, from"Research Directions in Object-Oriented Programming", MIT Press, 1987.

[KRIS] "The BETA Programming Language". B. Kristensen, O. Madsen, B. Møller-Petersen, K. Nygaard, from "Research Directions in Object-Oriented Programming", MIT Press, 1987.

[LEHM] "Program Evolution". M. Lehman and L. Belady, Academic

Press, 1985.

[LOVE] "Extensions to Modula-2". A. Lovejoy, Journal of Pascal, Ada and Modula-2, Vol 6, No. 2, pp20-44. 1987.

[MCFA] "Reducing the Cost of Branches". S. McFarling, J. Hennessy, SIGARCH 13th Conference, 1988.

[MOTO] "MC68000 16-bit Microprocessor Users Manual". Motorola Inc., Prentice-Hall, 1982.

[MYER] "Advances in Computer Architecture". G. Myers, Willey & Sons, 1982.

[NYGA] "Basic Concepts in Object-Oriented Programming". K. Nygaard, University of Oslo, Norway.

[ORGA] "Computer System Organisation - The B5700/6700 Series". E.I. Organick, Academic Press.

[PARN] "On the Criteria to be Used in Decomposing Systems into Modules". D. Parnas, Communications of the ACM, Vol 15, pp1053-1058, 1972.

[PATT] "Architecture of SOAR : SmallTalk On A RISC". D. Patteson et al, IEEE 11th Int. Conf. on Computer Architecture, 1984.

[PEMB] "Pascal Implementations : Compiler and Assembler / Interpreter". S. Pemberton, Ellis Horwood, 1982.

[POHM] "High Speed Memory Systems". A. Pohm, O. Agrawad, Preston Publishing Co., Prentice-Hall, 1983.

[RAND] "VLSI Architecture". Randall and Treleaven, Prentice-Hall, 1982.

[REDD] "A DAP Computer Architecture". Reddeway, 1st Annual Symposium on Computer Architecture Vol2, 1973.

[RUSS] "The Cray-I Computer System". R. Russell, Comm. ACM Vol 21, 1978.

[SCHA] "An Introduction to Trellis/Owl". C. Schaffert, T. Cooper et al, OOPSLA/86 proceedings, ACM 9/1986.

[SCHM] "Pascal Oriented Computer Design". E. Schmidz, PhD Thesis, Dept of Computing, Imperial College. 1980.

[SIG1] Proc. of 9th Annual Conference, SIGARCH Vol 10/3.

[SIG2] Stack based statement structure machine. SIGARCH Vol 12/3 6/1984.

[STEF] "Object-Oriented Programming : Themes and Variations". D.G. Brobow, M. Stefik, Xerox Palo Alto Research Centre. AI Magazine Vol 6, No.4, p40. 1986.

[STRO] "The C++ Programming Language". B. Stroustrup, Addison-

Wesley, 1986.

[TREM] "The Theory and Practice of Compiler Writing". Trembling & Sorenson.

[TURI] "On Computable Numbers with an application to the Entscheidung Problem". A. Turing, Proc. London Math Society Vol42 1936.

[WIJN] "Report on the Algorithmic Language Algol-68". IFIP, Wijngaarden (ed), 1968.

[WIR1] "Pascal User Manual and Report". N. Wirth, K. Jensen, Springer-Verlag, 197?.

[WIR2] "Programming in Modula-2". N. Wirth, Springer-Verlag, 1985.