The Design of a Two Level Code Generator

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Computer Science at the University of Waikato by Michael Byrne.

University of Waikato

1987

Acknowledgments

My thanks go to Keith Hopper for the considerable support and advice he has provided, and to my family who were patient during the time I spent on this thesis.

Abstract

The Rcode intermediate code used in the University of Waikato Portable Language Implementation Project (PLIP) compiler system has been designed to represent the source program independently of the source language and the target machine environment, with only sufficient structural information to ensure that efficient target machine code can be represented. Like many such intermediate codes, significant work is still required to produce target machine code. This study has investigated the design and use of a second intermediate code that divides the code generator into two phases, based on the observation that generation of target machine code will have many similarities for machines that are architecturally similar. This code is a Generic Action Set (GAS) code that represents common architectural features of a set of machines considered to form a family. The first phase is the generation of GAS code and its optimisation, and is common for all machines in the family. The second phase is the generation of target machine code from GAS code. It has been recognised that generation of target machine code for machines in the family will still involve many similarities, but machine idiosyncrasies make adequate abstraction difficult. However the development of "fluid" abstractions for machines in the family to assist with portability of compiler code has been studied, using the clear definition provided for the GAS family as a basis for the level of abstraction.

Producing a code generator for a new machine will often involves very little effort, if one already exists for a similar machine.

Table of Contents

Chapter 4

Chapter One

Introduction

This chapter discusses the portability concepts involved in the University of Waikato Portable Language Implementation Project (PLIP), and then describes the objectives of this thesis. The PLIP project involves the development of a portable compiler system. The aim is to develop a compiler system that can be easily modified to provide a compiler for any source language for any machine and operating system. Such a compiler system arose out of the need at the University of Waikato for a standard implementation of various programming languages on a range target machines and operating systems. Typically students are faced with implementations of programming languages on different machines that can differ in semantics and even syntax, and which offer a variety of directives. The development environment of editors, debuggers, subroutine libraries and linkers can also differ significantly. The solution to provide standard language implementations on all machines would only be practical if compilers could be easily ported. To allow a standard development environment, the project has been extended to the development of a portable linker and editor/debugger system. This editor project involves more than the development of a typical text editor; it involves a "generic" editor concept. Information on the portable linker and generic editor projects are provided in references [1] and [2]. For each language for which a compiler is developed, a standard runtime library will be developed. Currently a standard Modula-2 runtime library has been defined, with implementations produced for the Digital VAX/VMS [3] and UNIX [4] operating

systems.

The PLIP project requirements are also the requirements for development of portable software generally. Therefore the project has assumed wider significance in terms of an implementation study of portability. The combined tools will provide the ideal facilities for the development of software packages. The Modula II compiler implementation and runtime library provide a tool for the development of portable systems software such as data communications software, and the compiler system and generic editor themselves.

Portability of the compiler is achieved by:

a) Use of the Modula-2 language as implemented by the portable compiler system itself.

b) Use of the standard Modula-2 runtime library produced for the PLIP project for facilities such as file access and concurrency.

c) Use of a parser generator to develop a parser and semantic analyser for any given source language as easily as possible.

d) Division of the compiler into a front end which is target machine independent, and a back end code generator. The interface between the front and back ends is a target machine and source language independent intermediate code. This concept is central to the success of the project.

e) A module architecture that organizes machine and operating system dependence into a few modules.

Rcode intermediate code

The key element for portability is the intermediate code used to separate the front and back ends of the compiler. It provides the starting point for the subject of this thesis; the compiler backend. The intermediate code is called Rcode, is oriented towards the basic low level operations required such as adding two byte integers, but also allows the front end to communicate semantic structural information to the back end that will assist the back end to produce efficient machine code. The basic low level operations of Rcode are not machine specific. The objects manipulated are typical of target machine objects such as signed or unsigned integers, reals, booleans or bitstrings but are not specific to any target machine. The objects manipulated in Rcode are not source language objects such as array elements or record fields. Semantic structural information provided includes:

a) Flow control logic

b) Code blocks, procedures and associated lexical nesting information

c) Identification of independent global storage areas

d) Identification of local variables of procedures and code blocks

f) Identification of subroutine calling components such as parameters and results

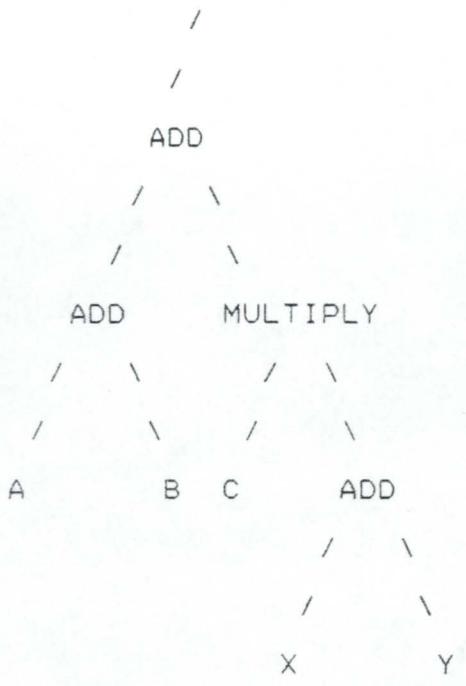
g) Expression structure via a tree format

These are all provided in a source language independent form. The equivalent structures for a given source language must be mapped by the front end into the intermediate code structures.

The Rcode basic low level operations are "tree" based. The operands of operations can be a subtrees of operations that return operand values. For example in figure 1, the top ADD has an operand which will be the sum of objects "A" and "B" and a second operand which is the product of "X+Y" and "C". This structure which is similar to a typical parse tree, provides structural information which could not be provided by a linear code. Consider the representation of A+B+(X+Y)*C in first linear form:

```
ADD  A,B,Temp_1
ADD  X,Y,Temp_2
MUL  C,Temp_2,Temp_3
ADD  Temp_1,Temp_2,Temp_4
```

then tree form:



The tree form represents directly the expression structure.

Efficient target machine code generation is assisted by availability of information on the semantic structure of a program as provided by Rcode.

Rcode represents semantic structures that are commonly available in a wide range of languages. It does not represent semantic information that is specific to very few languages, or which does not contribute to code generation. In particular, Rcode data types are simple machine level objects. Higher level source language objects, such as records or arrays are not identified. This is because these data types do not contribute much to code generation and also there are significant variations between source languages on rules governing their data types.

A discussion of intermediate languages is included in chapter two.

Thesis Objective

The central aim has been to design a portable back end for the compiler that produces efficient target machine code. In particular a prototype design has been undertaken to develop a backend for the Digital Corporation VAX. A more general mechanism for a portable back end will be developed at a later stage using the experience gained in the prototype design and development, but this is beyond the scope of this thesis.

The design for the backend prototype has been based on the following realisations and assumptions

a) That from Rcode to efficient target machine code is a large step.

b) There is much involved in this process that is not target machine dependent.

c) Many machines for which backends will be produced will have similar architectures.

The Rcode to efficient target machine code step will involve things such as of linearisation Rcode, introducing temporary variables as required, optimisation, calling convention strategies, operand addressing mode selection, and register allocation. Steps such as linearisation and temporary variable declaration, many optimizations and calling convention handling are not particularly target machine dependent, but depend more on

the general architecture of the machine. Machines with similar architectures will involve much that is similar in these steps. Similarity of architecture should therefore be a significant factor in the design of a portable backend.

Chapter Two

Survey of Portability Approaches

This chapter will discuss further the concept of portabililty of software via a portable language system, will survey previous techniques used to achieve portability of compilers and in particular, techniques used to develop portable code generator backends for compilers, and will contrast these to the approach taken in the PLIP project.

Portability of software depends heavily on the availability of a programming language for which compilers hopefully exist on all target machines. However several problems exist in finding such a language:

a) Semantics of the language and even the syntax accepted frequently vary from one compiler to another in annoying subtle ways.

b) Pragmas available are not standard

c) Access to system resources such as files and timers is not standard, or is provided in only a limited form.

Access to system resources should be provided in a machine independent conceptualised manner. Cobol provides many of the system resource access facilities required for commercial applications, but for applications such as writing compilers, very few of the required facilities are provided in the language.

This problem is often solved by the development of a standard runtime library that abstacts the system resources in a non machine-dependent manner. However, the library implementation must itself be portable. The main difficulty is that the implementation will at various points have to have access to information on the target machine, operating system and call system libraries. The solution adopted for the PLIP project is to ensure that all target machine and operating system information is kept in separate modules, and the compiler front end intercepts calls to the operating system and generates appropriate calling code for the host library involved. It is important to separate information on the target machine and operating system, because a compiler may be ported to run under more than one operating system on a given machine. A Modula-2 standard runtime library has been developed as part of the PLIP project that provides such things as file manipulation and access facilities, timers, delays and alarms, concurrency and process management, process synchronisation, interprocess communication, exception handling, and access to arguments passed to a program.

The first and second problems above can only be effectively solved by porting a standard compiler to all target machines. This must be carefully done to ensure that the semantics remain the same regardless of such things as different sized data objects of the host machine. Obviously, ease of porting of the compiler becomes critical.

Most attempts to develop portable compilers have centred on the use of an intermediate language. The compiler front end is designed to be target machine independent and produce an intermediate code which is target machine and source language

independent. The design of the intermediate code is critical to the success of this approach. It must allow efficient target code to be produced, as well as allowing portability.

The problem then becomes one of trying to find a way of easily developing code generators for a range of machines that accept the intermediate code as input and produce target machine code. One approach is simply to provide a code generator or even runtime interpreter for each target machine. An example of a runtime interpreter is the P-Code system [13]. A second approach is to represent the machine and operating system dependency of the back-end by representing relevant characteristics of the target machine and operating system by tables of data, or a mix of tables and procedure calls. One code generator algorithm for all machines uses this table of information. A third approach is to provide a code generator generator that requires a formal machine description and which will generate a code generator program. A survey of these approaches is provided by Ganapathi et al [12].

Intermediate Codes

Several intermediate codes have been developed in attempts to achieve portability of compilers. Often they are designed for different environments. Some are designed to allow a given language to be ported to a range of host machines, others to allow a range of languages to be ported to a given machine, and some to allow a range of languages to be ported to a range of hosts. The first environment would be typical of an organisation interested in a particular language. An example is the Diana [5] intermediate code. The second approach is typical of machine vendors. Many machine vendors have or have attempted to develop a

house standard intermediate code for use by several front ends and a single code generator. The third is typical of software houses that produce compilers and equipment vendors that sell a range of machines and operating systems. The Uncol[6] project intermediate code is an example of an intermediate code designed for such an environment. The PLIP project has shown that a single intermediate code can be successful for all three environments, except that the range of languages supported is restricted to a range or family of source languages rather than all languages. The Rcode language developed in the PLIP project is an implementation of such an intermediate code. The success of this language has resulted from a careful consideration of what needs to be in an intermediate code to allow efficient target code to be generated.

A major factor affecting intermediate code design is how close it should be to either the source level or target machine level. The closer to source language the easier it will be to develop a front end for a new language, but the greater the effort to develop a back end for a new target machine, and the reverse if the code is closer to the target machine.

If the aim is to develop compilers for one language on several machines it would appeal to place the intermediate code at a low level, so that much of what the compiler does is common. If the aim is to produce a compiler for several languages for one machine, it would appeal to place the intermediate language at a high level. If the aim were to produce compilers for several source languages on each of several machines, the level of the intermediate code is less clear. If an intermediate code optimiser is to be used and efficient code generated then the

decision on the form of the intermediate code becomes more complex.

A target machine independent code representation allows some target machine independent optimisations to be performed. More optimisation is possible the closer the representation is to the target machine. It is the efficient manipulation of target machine objects that provides a significant source of optimisation. Saving memory loads and stores of target machine objects is a prime source. The closer the representation is to target machine form, the more exactly target machine memory accesses will be represented. Many optimisations also depend on information on the semantic structure of a program, such as flow of control, what variables occupy a given storage area, and the structure of an offset computation. A low level intermediate code that does not provide this information will limit optimisation effectiveness. Optimisation therefore suggests an intermediate code involving target machine data objects, but higher level control structure information.

Diana - An intermediate code for Ada

To achieve efficient code generation, the intermediate code must effectively represent the semantic structures of the source language that are relevant to efficient code generation in the back end. If the intermediate code is designed to handle one language and several target machines then the intermediate code could be designed to represent all the information generated by the syntax and semantic analysis of the front end, that is the parse tree and symbol table. This is the approach taken with Diana, which is an intermediate code used for Ada. This language represents all the semantic structures of Ada, including data

type information in detail, Ada concepts of tasks, generics, modules, subroutines, exception handling, and control structures. It is an attributed tree-structured intermediate code. Tree nodes contain attributes that provide semantic information. Often this semantic information could be computed anyway, but is provided in a tree node because the front end semantic analyser would have computed the information and if represented in the intermediate code, it will not need to be recomputed. The result is that Diana is a complex intermediate code and complicates the entire compiler, and other software such as editors and interpreters that may use the intermediate code. This could perhaps be justified if more efficient code was produced, but the PLIP project intermediate code (Rcode) has shown that much of the information contained in Diana does not contribute significantly to target machine code. The complexity of Diana resulted because it is used for other requirements such as the need to generate source code from Diana.

If the concept of Diana were extended to an intermediate code for several languages and target machines, representing semantic structures of all the source languages would lead to a very complex intermediate language. The backend really becomes a translator for the union of all features of the source languages represented in the intermediate code. This would provide very little gain from the use of an intermediate code. This was one of the major problems in the Uncol project.

Rcode - The PLIP first level intermediate code

The solution adopted in Rcode is to select those semantic structures for a family of source languages that really are important to efficient code generation in the backend. In

developing Rcode it has been determined that for a related "family" of languages, only a few semantic structures are important for the generation of efficient target code, and Rcode is suitable for an environment where multiple source languages, target machines and operating systems are involved. The definition of such a family of languages is outside the scope of this thesis.

The PLIP project has shown the Rcode semantic structures useful in representing the family of languages that includes Pascal, Algol, Modula-2, Ada, PL1, C, Lisp, etc as:

a) Control structures

Loop
Counting loop ("for" loop)
Case
not concurrent (sequence)
Set (optionally concurrent)
Goto

b) Subroutine and function call mechanisms

c) Expression evaluation and structure

Note that assignment is really just a form of function call mechanism. Many of the additional semantic constructs of languages are not relevant to target code generation. A significant omission above is that source data types are not considered important. In reality source data types are only relevant to the user and the front end. The programmer is

concerned with manipulating source language data objects not target machine objects. Target code is concerned with manipulation of bits. Machine level instructions will often perform operations on bits interpreting them as being some form of representation such as two's complement binary integers. At the source level the concept of data types usually includes semantic rules such as preventing mixing of types in arithmetic expressions, or assigning a real value to an integer. At the machine level an operation may interpret bits as representing a certain sized two's complement value, no checks are made whether this piece of memory is "typed" for this data type. The bits may have been placed there by an operation involving a different interpretation of the bits. An exception may occur if the bit pattern does not represent a value within the machines capability (oversized). Thus the source datatype concept is almost totally unrelated to the manipulation of bits at the machine level. The Rcode "Basictype" parallels the machine level bit string interpretion for an operation rather than the source level "datatype" concept. At the machine level, major issues concern how to utilise registers efficiently and the addressing of objects given reasonably complex address computations which are usually related to accessing objects in arrays and records. However the concept of records or arrays is not important, only that an array or record represents that several target machine objects will have runtime addresses that are related to a common base address. In Rcode, computation of an address in a complex data structure is represented by a subtree. This address structure information can be used to generate efficient target machine addressing modes. The front end should map source operations on source data types to intermediate code operations that involve manipulation of target machine size bit objects.

This would seem to imply replacing the problem of representing all source types by representing all machine level bitstring interpretations available. This is not a real problem as target machine level interpretations are much less complex, and there is more standardisation than source language data types. Details such as actual size may differ but not the basic forms such as signed and unsigned integer, reals and decimal. The Rcode "basictype" can represent in a uniform way, basic interpretations and sizes across a wide range of machines. Source languages may have integers, reals, arrays, enumerated types, strings and records, and details of the rules governing such objects can vary greatly. Examples are rules governing semantics such as packing of arrays and records, and string variables.

The front end must be given access to information on bitstring interpretations supported by the target machine and will generate code that involves operations that use these interpretations. It will determine how objects of arrays and records will be packed based on alignment requirements for target machine objects, and the size of objects in memory. In Rcode the interpretations are represented in a coherent fashion using the "basictype" definition of types. This is a one byte value used to describe the type of manipulation such as cardinal, boolean, real and integer, together with size. It allows a standard method by which various parts of the compiler can access information on the interpretations available on the target machine.

In Rcode, expression evaluation is provided by the tree structure. The tree structure provides a clear indication of expression structure as opposed to linear code. Directed acyclic graphs would also of, course, allow common subexpressions to be

represented.

Rcode includes provision for expressing source type information. The intention for this is to provide for "symbol file" production for relevant languages and to provide a symbolic debugger with a template for interpreting storage in a form more suitable to the user. The types supported will not cover exactly all types available in any language, but provide sufficient for a debugger to provide a reasonable interpretation of the contents of memory.

Linear Intermediate Codes

An alternative approach to intermediate code is represented by the "P-code" system, which is a very low level unstructured linear intermediate code that has:

Memory which is word and byte oriented with word addresses and byte pointers supported

Zero address top of stack arithmetic instructions

Several pointer registers

Procedure calling mechanism that supports the lexical and scope structure of Pascal procedure, plus handling of result and local variables.

Block word and byte movement and compare instructions that assist with records, arrays and strings

Instructions are provided to allocate and

deallocate storage in a heap memory area

Instructions are provided for manipulating bitstrings that represent sets, to implement set operations

Conditional branch instruction are provided as the basic mechanism for providing flow control

Instructions for mainipulating boolean, integer, real, pointers, scalars (bytes) and strings are provided.

This language is designed as an intermediate code for implementing Pascal and therefore its design very much reflects the facilities of Pascal. However it could probably be reasonably used for a wider range of languages. As P-code is very low level, most of the Pascal compiler is target machine independent, the compiler would be easy to port. Only a P-code to target machine interpreter or translator would be required. However efficient code generation will not be easy because very little structural information is provided. This is a pity as Pascal was designed to allow efficient target code to be produced. Great effort will be required to extract structural information such as flow of control. The subroutine calling will allow effective calling target code to be generated. However the convention used limits P-code to source languages that have procedures involving statically sized variables, arguments and results. The low level intermediate code approach was never seriously considered as a viable alternative in the PLIP project.

An improved linear intermediate code that contains semantic structural information is often used. Typically these codes consist of three address instructions. Like Rcode, these instructions will involve operations on machine level data objects, and semantic structural information is also supplied. It may be provided in the form of instruction codes such as "parameter" or "call" which provide structural information for subroutine calling, and declaration pseudo instructions to declare procedures or storage areas, and For, Case and Loop codes. Expression structure may be retained in that an operand that is computed may actually be indicated in the three address instruction by a pointer to the instruction in which the operand was computed. Therefore a tree or even directed graph structure for instructions may be maintained. However control structures such as FOR loops and complex nesting of CASE and LOOP structures can only be provided if codes for marking beginning and end of loops and case options and for marking FOR index variables are included in the intermediate code. In this case the linear three address code carries much of the information provided by Rcode. However in most modern languages, directed graphs will be required for common subexpressions. An intermediate code cannot be written to an external device in post order form and still retain a directed graph structure. Linear codes solve this problem by the use of temporary variables. The next chapter will discuss how this problem can be solved by the introduction of a second intermediate language.

An example of a linear intermediate code that contains many semantic structure pseudo instructions is the intermediate code used by Anklam et al [7] for the development of a code generator back end for the Digital VAX. This intermediate code has

instructions that can have a variable number of operands rather than three address operands. As it is intended for a specific target machine it is reasonably target machine dependent, most notably that data types supported are designed to map easily to VAX data types, however the types are generally useful for a wide range of machines.

An intermediate code designed for implementing compilers for several languages on one machine may include specific target machine features. For example, the data manipulation operations can be specifically target machine codes, addressing modes specifically those of the target machine, and the registers of the target machine can be specifically included. An intermediate code for multiple target machines can't represent target machine features without becoming very large, or being modified for each target machine, and therefore the front end. An intermediate language designed for a single target machine would seem short sighted in that the compiler front end is likely to be ported to another machine at some stage, if it is any good.

A group in Denmark [14] have used the idea of two levels of intermediate code in the development of an ADA compiling environment.

Portable Target Code Generation

Table Driven Code Generators

The table driven approach of representing information on the machine instruction set, data types supported, and architecture generally, can become very complicated if the idiosyncrasies of the various target machines are to be accounted for, or

alternatively only a limited representation of target machines is used so that inefficient code is generated. The table is used to make decisions during such tasks as assignment of addresses and offsets, allocation of registers, selection of instruction and subroutine call construction. Information such as data object sizes supported for various operations, availability of two and three address instructions, the requirement of some instructions to use specific registers and the various sizes of offsets for branch instructions must be represented. As the code generator performs these tasks it will need to maintain a database on the location of objects in memory (variables, routines, constants, labels) and the allocation of registers. This resource database can be referred to as a machine resource allocation database. This database must be capable of representing the resources of a wide range of machines. Both the table and database must be able to handle the various memory structures and addressing modes of the intended target machines.

The major difficulty of the table driven approach is that the table which can represent a wide range of machines will be many times that required for one machine alone. This is because for any machine there can be factors important to generation of efficient machine code that are not present on other machines. If efficient code is to be produced it will be necessary for the table to represent the sum of factors important to all intended target machines. The wider the number of intended target machines, the larger and more complex the table would be. If the intended target machines can be limited to a family of similar machines then the table can obviously be kept significantly smaller.

For any given machine many of the characteristics represented in the table will not be relevant as they represent information relevant to other machines. However the code generator will still include the logic for processing this information in case it encounters a machine for which this information is important in code generation. The target code generator tables and code will be unwieldy in size.

Hand Coded Code Generators

The hand production of separate code generators for each target machine only seems a practical proposition if the intermediate code is low level so that the generator is small. The disadvantages of such an intermediate code have already been discussed. However this approach would be practical for higher level intermediate codes if each code generator was engineered in such a way that the code generator for one machine could be easily modified to produce a code generator for a similar machine.

Such engineering can be achieved by identifying the parts of the code generator logic that involve machine dependency and instead of accessing tables of information, insert target machine dependent compiler code. This approach is enhanced if the code generator is abstracted as much as possible and target machine dependencies kept to the level absolutely necessary. As the target machine dependent issues that are relevant will vary from one machine to another, the nature of target code generators for each machine could vary significantly. However if the generator structure is abstracted effectively, a code generator for one machine should be reasonably easy to modify to produce a code generator for any other machine that has similar characteristics.

This approach really comes down to a classical software engineering problem involving careful abstraction of functions and relevant databases so that portability and changeability are maximised. The resource database and any target machine table representing characteristics are implemented in a target machine dependent way but the structure is abstracted and access provided via procedures and enumerated data types. Code generator algorithms are carefully designed so that they only deal with machine specific details when absolutely required. Where target machine code is required it is clearly identified.

This approach has been taken throughout the PLIP project. At levels close to where target machine instructions are being generated, abstraction becomes more difficult. The issues involved will not be common to all machines, so that the abstractions required will differ. In this situation porting will require modification of the abstractions provided in the code generator logic and database, but this should never become an excuse to abandon abstraction attempts or a code generator developed for one machine will not be easy to port to even a very similar machine.

Code Generator Generators

This approach appears to offer the real possibility of easily generating a code generator for a new target machine. All that is required is definition of the machine characteristics in a formal grammar. This definition is processed by a code generator generator to produce a code generator for the target machine. This approach was seriously considered for the PLIP project but was considered to have the following problems:

a) The code generator produced would probably be table driven and would suffer the same problems as table driven compilers: the table and generator code would be too large, as the generalised table format would include more complexity than required for the actual target machine itself. An intelligent code generator generator would modify the table and code form as much as possible for each target machine, but this would make the generator generator very difficult to write.

b) As with most automatically generated software, the code generator will be slow.

c) Optimisation is difficult. It was envisaged that the intermediate code would probably have to be linear and optimisation limited to such options as peephole optimisation. Tree structured intermediate codes and more complex optimisations would require a much more complex generator generator.

d) The effort involved in developing the description for one machine would be heavy, nearly equivalent to hand coding the code generator. The definition would no doubt require extensive debugging equivalent to a hand coded code generator, but the relationship between errors in definition and invalid code produced would not be as obvious as the source of such errors in a hand coded generator.

e) Modification of the description for one machine to produce a description for another similar machine will not be easy because of the nature of many formal grammars.

f) Most automatic code generators have been unsuccessful,

although during 1986 two successful projects were reported. One by Schmidt and Voller [15] involved the development of a portable compiler system for "Pascal and Fortran like" languages. The Vienna Development Method and its specification language META IV are used to describe the formal specification of the target machine and source languages. A common intermediate language is derived from the definition of the source languages and a code generator generator processes the target machine specification to produce executable Pascal programs that implement the code generators. A second by Ganapathi and Fischer [16] uses affix grammars to describe the target machine instruction set and the code generator is obtained automatically using attributed parsing techniques. It is claimed a code generator such as this can perform most "popular" target machine optimisations.

Most attempts at producing retargetable code generators have mainly been based on code generator generators, even if the result of the generator generator is essentially a table representation of the target machine, or carefully engineered hand coded code generators.

Overall it was felt that with equivalent effort smaller, faster code generators that produced better quality code could be produced by hand coding as compared to table driven generators or automatic generator produced code generators. The decision was made to investigate ways of improving the engineering of target code generators so that code generators developed for one machine could more easily be modified to produce generators for other machines.

Other Literature

References to allied (more or less) work are:

a) Compiler tool kit, Tanenbaum et al [1]. They have produced a tool kit for making portable compilers. The "Amsterdam Compiler Kit" consists of a set of integrated programs to simplify the task of producing portable compilers. For each language a front-end must be written to produce intermediate code. A portable optimiser is provided for this intermediate code which is then translated or interpreted to the assembly language of the target.

b) Portability via virtual machines, Yankov and Bonev [18]. They describe the use of intermediate codes for virtual processors which are interpreted by emulators on the target machines.

c) Table driven code generation, Graham [19]. A Ph. D. study of table driven code generation.

d) Modula-2 optimising compiler, Powell [20]. This project demonstrates how a compiler that produces reasonable code can be constructed quickly if advantage is taken of existing software. The parser was generated using Yacc [21] and P-code intermediate code is produced, providing compatibility with many Pascal compilers. The procedure call convention conforms to that of Pascal and C compilers running under Berkeley Unix. The P-code of course allows existing P-code translators and interpreters to be used.

e) Portable C Compilers (PCC), Johnson [22]. The front-end

of the compiler is generated by Yacc. Its intermediate code consists of prefix notation for expressions and assembly code for the rest. Obviously such an intermediate code does not make portability easy. However three quarters of the compiler is independent of the target machine, indicating the degree to which effective software engineering should be able to reduce target machine dependency of compilers.

f) A portable C compiler, Snyder [23]. Developed a C portable compiler that is driven by a set of machine dependent information contained in a set of tables which are automatically constructed from a user provided machine description for a machine dependent but abstracted machine. The user defines translation to target machine code of abstract machine code produced by providing macro definitions.

g) Multiple front and back ends, Davison and Fraser [24]. Describes compiler organisations designed to allow combinations of various front and back ends. Defines the terms "union" and "intersection" machines in reference to intermediate codes. An intersection intermediate code has limited operators and addressing modes so that the front end has few choices. A union intermediate code has a wide range of operators and addressing modes so that the front end has a wide range of choices, but as not all target machines may support all these alternatives it may be necessary for the front end to have access to target machine information. These concepts are useful background for the design of any intermediate code form.

h) Code generator generator, Heyliger et al [25]. Produced a recommendation for a retargetable compiler using a compiler

compiler that utilised a machine description provided in a machine description language.

i) A re-usable code generator for Prime 50-series computers, T. Akin [26]. Involved the design and development of a target code generator that could be used with multiple front ends. This product is typical for compiler developers with one target machine in mind. The input is a tree structured intermediate form and the output is assembler.

Chapter Three

GAS code: A Second Intermediate Code

The objectives listed in chapter one for the design of the code generator required that account be taken of the fact that Rcode is a long way from target machine code, and that many of the target machines will have similar architectures. It is these two factors that led to the concept for a second intermediate code. This chapter outlines reasoning that led to the concept of a second intermediate code, and the role that such an intermediate code should play, including its relationship to Rcode. The next chapter deals in more detail with the concepts that arose during the develoment of a prototype GAS code developed for the prototype code generator for PLIP.

Justification for GAS code

Rcode has been designed to provide an intermediate code that the front end can effectively generate, and which contains semantic structural information essential for efficient code generation, but is not source language or target machine dependent. This provides the ideal vehicle for front and back end independence. It does involve operations on objects that are essentially target machine objects and not source language objects. However it is not close to target machine code in several ways:

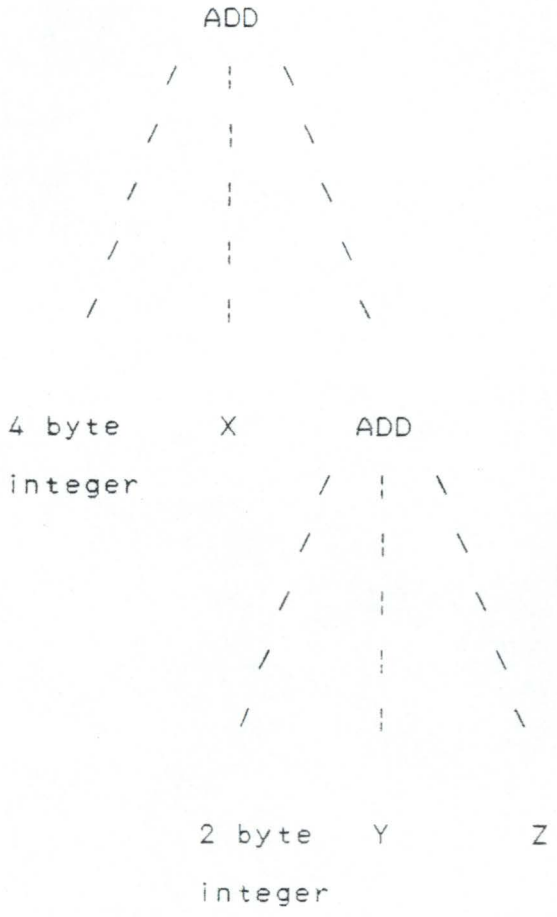
a) Computed operand values for an Rcode are specified by a subtree of Rcode instructions. In a register based machine, the value will be typically computed into a register. The operand computation is linearised. This linearisation is not performed in

Rcode as it loses useful structural information provided by the tree structuring and very much depends on the number of addresses allowed in target machine instructions. If zero address instructions are available, the tree's structure can be very easily converted to target code. Intermediate results are held on the stack. The only problems occur when attempting to handle common subexpressions. If the machine is three address, temporary storage locations must be created to store intermediate results. If only two address instructions are available, more temporaries and move instructions will be created. If the machine has registers, selection of locations for holding values will become even more complex.

b) Rcode includes arithmetic operations on block objects that may be both large and dynamically sized. For a real machine, these may have to be converted into operations on objects supported by the target machine. These breakdown operations are not performed in Rcode partly because breakdown requires some knowledge of machine architecture (the basic arithmetic instructions, flags, and branch instructions available to do the job) and because this breakdown would make the Rcode target machine dependent and defeat its purpose. The target machine may even support operations on very large and dynamically sized objects. Any breakdown should be left to the code generator so that full structural information is preserved, allowing the target code generator to decide how the breakdown is best performed.

On many target machines, operands of arithmetic instructions must involve operands of the same type. On other machines operands may be different in size, eliminating code needed to massage

different sized operands to the same size. Rcode arithmetic operations (block operations aside) seem to specify that all operands are the same size because a basictype operand is included. However consider:



A clever code generator would still generate a first instruction to add the two byte values in the first add, then an instruction to add a four byte value at X to the two byte result of the first add, rather than an instruction to massage the result of the first add to four bytes.

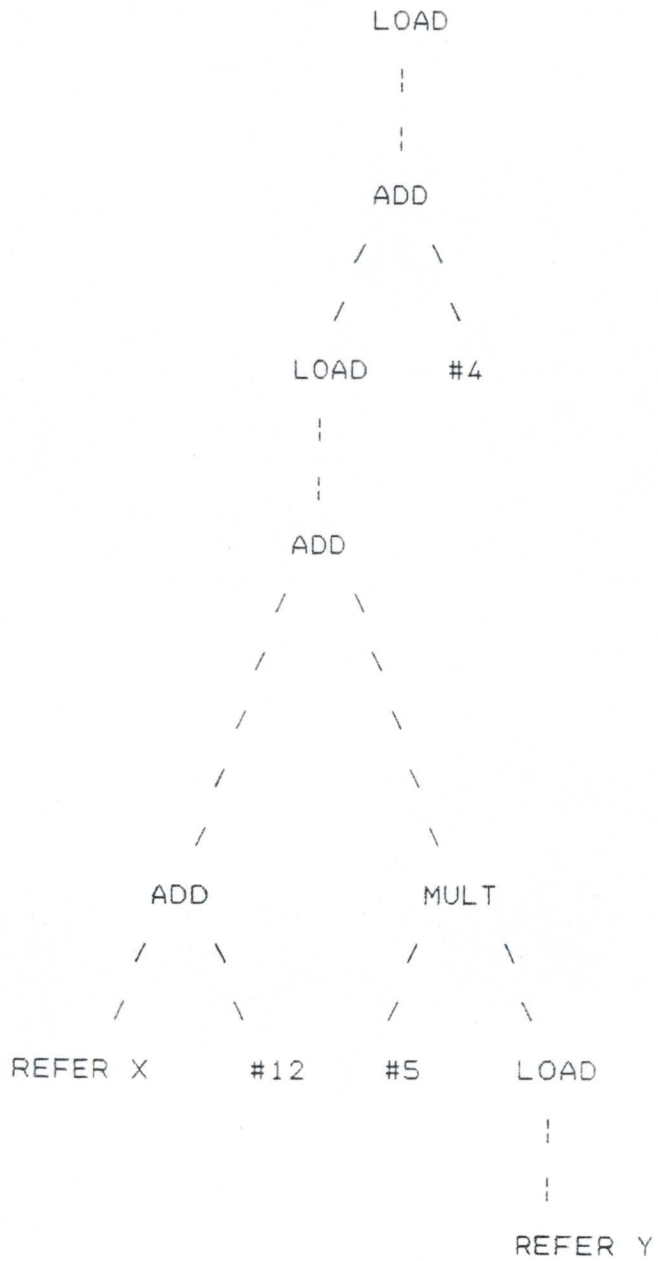
c) Bitstring manipulations are specified in Rcode. These involve bitstrings that could be both large and dynamically sized. The operations could be converted to operations on

bitstrings of size supported by the target machine, and using bitstring instructions supported by the target machine. This breakdown is not performed in Rcode for the same reasons described above for block arithmetic operations. Some target machines support some bit operations on dynamically sized bitstrings, so the breakdown may not be desired similar to the above in (b) for arithmetic operations.

d) Subroutine calling in Rcode does not involve any form of convention such as use of the stack for the result of function, passing parameters, or allocation of storage for local objects. Rcode will give a specification of the arguments, result and local objects, in terms of size and alignment, but not how this is to be implemented on the target machine. There will be significantly more involved at the target machine level to actuallly implement a subroutine call. The subroutine structure in Rcode is designed to adequately reflect the subroutine facilities and semantic structure of the source languages: lexical nesting and scope, arguments, result, and local objects of routines. Rcode does not attempt to represent how a subroutine calling structure is implemented as this is obviously target machine dependent. The implementation on a stack, single processor machine will probably differ significantly to the implementation for a transputer based machine.

e) Rcode does not reflect the way operands for instructions are defined in target machine instructions. Rcode operands can be subtrees, whereas target machine operands merely involve objects referenced via addressing modes such as indexing, static offsets and indirection. All operands in Rcode are referred to by specifying the identity of a storage area, and an offset into the

area. This corresponds to direct addressing if the offset is statically computable and indexed addressing if the offset is dynamically computed. Immediate addressing is supported, if the operand itself can be statically evaluated. It is possible to extract from the Rcode more information for almost unlimited complexity of addressing. The following example clearly shows the operand address computation structure and would allow complex addressing modes on the target machine to be used, if available.



Linearisation of Rcode expression trees will lose information on the structure of an operand address computation. This effect must be taken account of if efficient addressing modes are still to be generated in the target machine code. Various target machines provide fairly complex addressing modes such as computing an operand address as the contents of the memory location whose address is in the location whose address is given by the sum of the contents of a register and an immediate value. These allow complex Rcode operand address expressions to be implemented as a single complex target machine operand. Rcode contains the information on an operand address computation structure, but does not reflect that in target code this complexity is converted into a reasonably complex addressing modes, or more generally, rather than the address computation being simply linearised, it is converted to a fewer number of linear instructions using the complex addressing modes available. The effective use of complex addressing modes can significantly improve the quality of code produced. In reality most target machines will be limited to the extent of complexity for operand addressing modes. Additionally, in most source languages, there is a limit to the level of complexity of addressing modes required for the implementation of constructs in the language. Only when using an array of records which contain an array of records etc could very elaborate addressing be required. The limit on addressing mode complexity required in practice needs to kept in mind. It should also be noted that the generation of complex addressing modes is not really target machine dependent. Concept such as indexing, indirection are all reasonably common. If a machine does not have a given complex addressing mode, it is a fairly easy matter to

break down the complex mode into the simpler modes provided.

f) Many target machines will provide registers. Commonly used objects will be hopefully kept in registers. Therefore there is the question of where operands should be located for an instruction, and register addressing modes are introduced. When offsets or pointers to operands are computed, there is the question of whether and which index registers should be used. In the Rcode machine when a subexpression is required for an operation, the subtree for the operation is tacked on to the instruction.

g) For a machine that provides a stack, subroutine calling will typically use the stack. A major question in target code generation will be the effective use of stack pointer registers in the access of objects allocated on the stack.

h) In a zero address oriented machine, emphasis in target code generation will be on the effective locating of objects on the stack, and the addressing of these objects using various stack pointers. In fact the Rcode tree structure will suit a zero address architecture.

i) Rcode specifies control logic structures with high level facilities such as For loops and Case structures. In many machines these will be implemented at the target machine level using conditional and unconditional branches.

Rcode has been effectively designed to allow effective isolation between the front and back ends. However Rcode is obviously not close to target machine code. In the differences between Rcode

and target machine code discussed above, it is clear that several differences are dependent on the target machine such as use of target machine registers, and some differences such as linearising the tree structured code of Rcode, converting the high level control structures of Rcode into more typical branch instructions, breaking down arithmetic operations on oversized objects and linearisation of operand subtrees into instructions using complex addressing modes are not particularly target machine dependent, but are more dependent on the general architecture and instruction set. For example linearisation will depend very much on whether the machine is register or zero address oriented. Another example is the breaking of operations on block objects and bitstrings into operations on objects supported by the target machine. This will involve creating temporary objects as required, using hardware facilities for detecting overflow and carry, and using instructions supported by the target machine. For many machines, the instruction sets and status flags available are very similar for this purpose. If the machine is zero address oriented, a different approach will be used than for a machine with registers. If the target machine has a stack that can support subroutine calling, code will be generated to the use stack. In a transputer machine, a procedure call is notably different in the mechanism for passing of arguments. Though the details of this code will be specific to each target machine, the general form will be common to all machines with a stack.

Hence architecture and general instruction set will be a significant factor in the process of converting Rcode to target machine code.

Optimisation is another reason for the use of a second intermediate code. Optimisation could be undertaken at the Rcode level for such purely machine independent optimisations as common subexpression elimination, and removal of loop invariant code from within loops. However some of these optimisations require that a directed acyclic graph be created rather than a pure tree, but the convention for writing Rcode out in post order form will mean that the directed graph will be returned to a tree, effectively losing the optimisation. Optimisations would still have to be performed at a later stage, and many of these optimisations would use the same information generated for Rcode optimisation. The advantage of Rcode optimisation is that code does not have to be written for the optimisations performed for each target machine or family of machines encountered. However optimisers would still have to be written for each target machine code and these would require the regeneration of some of the same information used by the Rcode optimiser. The concept of writing an optimiser for each target machine does not really appeal. Additionally, at the target machine code level the program structure is no longer available as it is in Rcode, so optimisation effectiveness will be significantly diminished. A program representation that was closer to final machine code, but which contained structural information had several advantages. If this code linearised using temporaries for common subexpressions, common subexpressions could be represented by references to the same memory location rather than by a subtree. If the code were written to disk, the optimisation would not be lost. The code would also contain more of the objects, including temporaries, that would be directly manipulated in the final target machine code, so that information generated for optimisation, such as liveness analysis, would be more useful to the target code

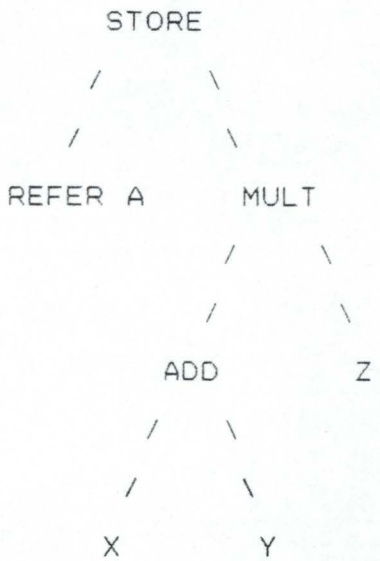
generator. The target code generator is particularly concerned with register allocation, and this benefits considerably from a liveness analysis performed across as many objects as possible. The retention of Rcode structural information would be useful right through to target code generation. It is also very useful for generating optimiser information such as for data flow analysis (see later chapter on GAS code optimisation), as basic blocks of code and loops are much easier to identify. At any time during target code generation, the context of the program structure is still known, so that possible uses of target machine instructions such as Loop can be identified.

A major source of optimisation is static evaluation of a program. This can eliminate significant blocks of code, allow initialised data area linker directives to replace code that effectively initialises areas, and indicate such things as whether an object size Rcode subtree produces a statically determinable value. An interpreter is an effective tool for static evaluation if available to the compiler. An Rcode interpreter would be difficult to produce because of the complexity of Rcode, but once produced would interpret any Rcode, independent of target machine. A machine code interpreter would merely involve the compiler executing the code the compiler has just produced. But why produce a lot of machine code that is found to be redundant? If the structure of the program provided by Rcode is not available, useful static evaluation would probably be limited anyway. A lower level representation of a program that was valid for a range of machines, and which was much simpler than Rcode would be a useful compromise. An interpreter would only have to be produced for each family.

GAS code Structure

The argument has therefore been made for converting Rcode to a form closer to target machine code. This second code is to reflect the significant architecture features of the target machine. The conversion of Rcode to GAS code will allow much that is common in producing target code for members of the family to be performed by a common GAS code generator. The GAS code will make common family optimisers and interpreters possible. The structural information provided by Rcode should be retained, except for expression structure provided by the tree structure of Rcode. The major role for the GAS code involves:

a) Linearisation of Rcode expressions. This will depend on the general machine architecture. Consider the following Rcode:



For a stack machine:

```
PUSH X
PUSH Y
ADD
PUSH Z
MULT
STORE A
```

For a register machine:

```
ADD  X,Y,Temp_1
MULT Z,Temp_1,A
```

Rcode expression subtrees will be replaced by GAS code sequences. GAS code should model the zero, accumulator, two, or three address orientation of the machines in the family.

b) As noted above, this linearisation will destroy information on the structure of operand address computation. Therefore the GAS machine should allow operand addressing, that are the equal of any of the modes in target machines in the family. This will allow the address computation to be converted to a series of instructions with complex addressing modes selected when advantageous. If simple addressing modes were provided by the GAS machine, the linearisation of addressing computation would lose structural information that would preempt use of more complex addressing modes on the target machine.

c) It may be necessary to breakdown block arithmetic and bitstring operations into operations on objects of types

supported by the target machine, including reflecting the need for all operands to be the same size, if this reflects the nature of all the machines in the family. This process will require access to the data types supported by the target machine. As noted above, the target machines in the family may support operations on dynamically sized objects, and if so, this breakdown will not be necessary.

d) The representation of the implementation of subroutine calls on the target machine. If the machine has a stack that can be used for subroutines, GAS code should allow subroutine calls to be converted into a series of GAS instructions that represent the PUSH and POP instructions that will be used, and the fact that objects on the stack will be accessed by stack pointers. In fact a standard general calling convention should be represented in the GAS code so that the GAS code represents as closely as possible the final target machine code structure. This is discussed below. Such a convention would bypass any host machine conventions unless these were compatible. However portability of software will be enhanced by a standardised calling convention. Instructions to support this convention must be included in this GAS code. If the target machine involves transputers, the subroutine environment will differ markedly form a single processor machine, and the GAS code must reflect the calling environment for this type of machine.

e) Control structures should be converted to typical machine code level instructions that will be used in the family such as conditional branches, and associated labels. However, as the program structural information is still important for optimisation and target code generation, the Rcodes such as CASE,

LOOP and FOR should be retained, though they will be redundant.

f) Special instructions should be provided for input and output, disabling/enabling interrupts, and processor register manipulation. These will allow GAS code to be implemented for these areas that reflects the typical form of target machine code. Also Rcodes such as "Add_In_Place" can be implemented in GAS code by disabling interrupts, if no single target machine instructions are available to do an indivisible add. The GAS generator will require information on whether Rcodes such as Add_In_Place can be handled directly in target machine code and if so, the Rcode will be left in place. The Rcode input and output and register manipulation codes seem adequate, but the subtrees for port address, size and value computations will be replaced by GAS code sequences, and as there is no way for GAS code to return a value to an Rcode node, a simple GAS input, output or target GAS instruction is placed on the end of the sequence that computes the size, address and value components. The Rcode for input, output and register access will be retained for structural information.

g) Calls to the operating system kernel should be modelled, for example for requesting saving or loading of process context. To support this GAS code should incude an instruction that represents a kernel call. Many machines have an equivalent instruction; CHMK on the VAX, INT on the Intel 8086.

The GAS code attempts to generically model the target code for a family of machines. The number of GAS codes will not need to be large to allow the above requirements for GAS code to be met. Instructions are required to generically model:

Memory moves

Stack operations (if stack present)

ALU operation (arithmetic and logical)

Subroutine call and return

Control instructions

Special instructions

The instruction set should reflect the instructions that will typically be used to implement various ALU and control structures on the target machines in the family. For example in many families arithmetic operations are provided on a few fixed sized objects. The arithmetic instructions in the GAS code should reflect this. Execution control is often provided by conditional and unconditional branch instructions. The GAS machine should reflect this. However for subroutine calls there is a wide variety of calling convention provided by machines which are often quite different. This problem is discussed separately below.

The instructions must allow an efficient choice of actual target machine instructions. This requires that no unnecessary constraints are placed on instructions. For example, requiring that each operand in an arithmetic code should be of the same size, or that certain arithmetic instructions should be two address. However constraints could be applied when it is certain

that all or almost all machines in a family will have the given constraint. An example is that many machines only provide arithmetic operations on operands of a few allowable sizes, and that dynamic sized operands are not catered for. The concept of constraints and family definitions is discussed more fully in the next chapter, and is very important for successful GAS machine and code design.

The lack of constraints on the form of GAS instructions makes it very likely that they will be orthogonal and three address. In many processors, the instruction count is considerably increased by non orthogonality, and by two address and three address variations of instructions. Additionally, the GAS code only attempts to model the major characteristics of a family. It contains no special purpose extra instructions. Therefore the GAS instruction count will always be small.

Subroutine Calling

Many machines and operating systems provide a standard calling convention. For GAS subroutine calls, the structure of the call should be clear, so that efficient target code can be generated on the target machine, perhaps using any efficient instructions available on the target machine. However a decision has been taken in the PLIP project to have a standard calling convention regardless of the target machine's own facilities. As a result the structure of the runtime stack will be fairly similar for all target machines in a family, only differing for reasons of alignment and perhaps a few objects that are required for implementation on the target machine. This will assist with portability of programs that manipulate the stack. An example of this is the implementation of exception handlers in the PLIP

Modula-2 portable runtime library. The target machine subroutine call convention and associated special instructions may not necessarily be used. The GAS machine references to local objects of the routine, arguments, the result area, and outer scope objects should be obvious given the GAS machine addressing facilities so that efficient use can be made of target machine facilities by the target code generator. The structure of subroutine calls will be made even clear because the Rcode "Call_" will be retained. This helps to clearly mark to the code generator the components of a call; creation of the argument block, computation of the result area, and the computation of the called routine descriptor.

Control instructions should reflect the significant control instructions available in the target machine. For example, conditional branch instructions may be the dominant mechanism. However in some machines, special control instructions such as loop instructions suited for implementing FOR type loops in certain situations may be available. A good code generator should recognise these situations and use the instruction. However if the GAS code has been generated to implement the loop using conditional branch instructions, the use of the loop instruction would seem preempted. As Rcode control codes are kept to assist target code generation, the detection of situations where special target machine codes can be used such as loop instructions is reasonably simple.

Declaration of data storage areas is necessary for any target code generation phase. Not only must code be emitted, but linker directives specifying global storage areas must also be emitted. The Rcode declarations for global storage areas will be retained

in the GAS code. Note that Rcode global storage area declarations contain area size specifications that are always statically determinable. Rcode variable declarations are for dynamically allocated storage, such as on a stack, and will not result in the generation of linker storage directives. These will generally be converted into GAS code to allocate this storage, for example a PUSH instruction. The size computation Rcode subtrees for variable declarations will have been converted to GAS code, before the PUSH is generated.

All Rcodes that mark the declaration of objects or routines will be retained. These Rcodes contain useful semantic information, such as "here is the declaration for a variable of the routine", or "here is the code for a routine", or "here is the code to compute the result size". This is all useful information to the target code generator.

It should now be clear that Rcode is not converted to GAS code, and the Rcode discarded, along with the wealth of semantic information it contains. An important concept of the second intermediate code is that it co-exits with much of the Rcode. The important semantic structural information of Rcode is retained by retaining Rcodes such as Call_, For, Case and Loop, and declarations, and only really replacing actual "real" Rcode operations such as Add, Load and Negate, so that "real" operations more closely resemble the "real" operations of the target machine. A move made closer to target code may make optimisation better because more of the real operations and data objects are represented, but this would be overshadowed if semantic information was lost. The only structural information that is lost is the structure of expressions, because of their

linearisation.

At the conceptual level, GAS code is generated for "real" Rcodes and replaces these Rcodes on the Rcode tree. "Real" Rcodes are action codes; those that actually do something. GAS codes are "tacked" onto the Rcode tree. At the representation level, GAS codes are coded in the data field of the two byte Reserved Rcode. This means that the routines developed for handling Rcode trees can equally be used for handling Rcode trees containing GAS code, a real plus for efficiency in the compiler.

The need to declare temporary storage objects as Rcode expressions are linearised, and the need to create labels to mark positions in the GAS code that are targets of GAS code branch instructions leads to the need for GAS pseudo codes to define these storage objects and labels. However there could potentially be many of these objects and labels, and the compiler databases could become clogged with information on them. One solution is to delete the database entries for the tenporary objects and labels for a routine when handling of a routine is complete. However, it may be possible a point is reached when it is known that a temporary object or label will not be referred to again. Therefore pseudo GAS instructions to indicate this would be useful from a practical view point.

The generation of GAS code from Rcode can be achieved by the standard techniques for generating machine code from a syntax tree, except that the GAS code is attached to and replaces some parts of the tree, rather than a machine code stream being produced. Techniques such as Syntax Directed Translation are quite applicable to generating GAS code. In most cases the

conversion of each Rcode essentially involves a template consisting of several alternative sets of possible GAS code sequences depending on the contents of the "holdup" stack of previously generated GAS codes and operands. Operands for instructions in the selected template are "filled" in on the basis of heldup operands. The is the standard procedure for syntax directed translation.

The GAS code will be spread through the Rcode tree, but it is important to realize that if the Rcode tree is walked in post order form, and

declaration Rcodes and GAS labels noted, and

GAS codes extracted and placed in one linear list

that the GAS code could then be executed. The GAS code executed, will or should, produce the required semantic effect of the program. The declarations are used to declare storage objects and pieces of code. Each of these objects can then be referenced. If GAS code were interpreted on a real machine, storage area declarations encountered would result in allocation of storage for the size required, a database entry would be created that points to the memory allocated to the storage area. When a routine declaration Rcode is encountered, a database entry would be created that pointed to the Rcode tree node that is the start of the routine code.

Rcode includes machine and system dependent codes (group 6). Except for 6a, these Rcodes will be passed untouched to the code generator as these are very target machine and operating system

dependent. Only the target code generator can handle these.

After GAS code has been generated, it can be optimised. The GAS code can be treated as any typical three address code, or zero address code, or whatever form, and optimised using any of the well known techniques available for optimising code in this form. A later chapter on GAS code optimisation describes the design for the GAS code optimisation in the PLIP project. The design takes significant advantage of the structural information provided by the retained Rcodes.

Chapter 4

Three Address/Stack GAS Machine

This thesis has involved considerable work on the refinement of the concept of a Generic Action Set of instructions for a generic machine. The generic machine and its instructions represent the architecture for a family of machines. A generic machine and code set has been developed for a family of machines that includes the VAX and many other common processors such as, for example, the Motorola 68000, Zilog Z8000 and Data General MV series. This GAS machine and code design has been incorporated in the code generator being developed for the VAX as part of PLIP. A Reference Manual and Users Guide Manual have been produced for the GAS design and are included in appendix A and B. The work on the VAX prototype code generator has provided an environment in which the two level concept has been refined. A fuller generalised development of the two level concept is planned but is beyond the scope of this thesis. This development will involve application to families such as zero address machines, and multiprocessor machines. In particular it is hoped to produce a design for transputer based machines in the near future.

This chapter describes the prototype design developed for register/stack machines of which the VAX is a member. The design of this GAS machine definition has consumed much of the time for this thesis. It has been found that the design of the GAS machine for a family is vital. If the definition does not accurately model the significant factors affecting machine code generation

for machines in the family, the code produced for all machines will be poor. The GAS machine concept offers much for portability, but if the definition is not precise, portability will be gained at the expense of code quality. One of the important goals of the PLIP project is that code produced must be of high quality. The prototype development has been essential to demonstrate that both portability and quality code can be produced by a two level code generator. If this were not possible, the GAS machine concept would just have become another interesting idea that did not live up to its promise. The prototype development has shown that portability and quality of target code can be achieved for register/stack based machines. Further prototype development will establish if this will also be true for zero address and multiprocessor machines.

GAS Machine and Interpretation

As stated in the previous chapter, it should be possible to interpret GAS code using the semantics defined for the generic machine. Interpretation would involve accepting a stream of Rcodes containing GAS code, in post order form, or an Rcode tree, and ignoring Rcodes except declaration Rcodes and Rcode reserved extensions containing GAS codes. The GAS codes are then "executed". The declaration Rcodes are equivalent to creating a new memory segment, either data or code. The machine therefore emulates a segmented virtual memory machine. If the GAS code is interpreted on a conventional machine, a database will be maintained which records the location in real memory of each object (storage area, routine, temporary etc) declared in the Rcode. This data base is similar to segment tables in a segmented virtual memory machine. References to routines, or objects in

memory will require access to this database.

Family definition for Prototype GAS Machine Design

The previous chapter introduced the concept that a generic action set for a generic machine represents a "family" of machines, argued the case for it's existence, described its relationship to Rcode, and discussed what should be achieved in the move from Rcode to the GAS code level.

The factors discerned as important for the second level code evolved during the design of a GAS machine prototype for the register/stack based family. These are not an exhaustive set of requirements for a generic machine. It is expected for example, that the design of a generic machine definition for transputer based machines will involve additional important concepts. A more complete generic machine conceptual model will no doubt be established during later more in-depth studies of the two level code generator concept. The issues that appeared during the development of the VAX family GAS machine prototype are now discussed and provide an insight into the major issues involved in defining a new family.

The model for a GAS machine requires an architectural definition and an instruction set. The architecture defines such things as stack facilities and any associated pointers, the structure of memory, addressing modes for operands, and types of operands handled. The instruction set describes GAS machine operations that are available, but will also include some "pseudo" operations, involved with declaration of objects. Additionally Rcode declaration codes are significant in the GAS machine model

for declaring code and memory objects.

The instruction set provides an important definition of the family, and generally reflects architectural features of the family. For a machine such as the VAX, instructions include:

ALU	arithmetic and logical
Memory movement	
Stack	push/pop
Control	mainly conditional branching
Subroutine	Call/Return

Arithmetic instructions essentially involve operations on statically sized objects, of size 8,16,32,64 bits. These objects are multiples of bytes, and if located in memory must be located on byte boundaries.

The types of arithmetic instructions will determine how block arithmetic instructions are broken down. The types of control instructions will detemine how the control structures of Rcode are represented.

How the Rcode tree structure and intermediate result values are handled on the target machine is significant. Rcode is tree based which means that storage for intermediate results is implied. An internal stack of intermediate results would be typical of any implementation of the Rcode machine. The Rcode machine is therefore very much aligned with a Stack (zero address) architecture. The handling of intermediate results is handled implicitly for the programmer. For a machine with registers and without zero address instructions, the code must be linearised

and the destination for all instructions explicitly specified. The programmer is therefore concerned with the handling of intermediate results. The essential advantage of registers is that values that are to be used earliest in following code can be kept in registers to avoid unnecessary memory references. In a zero address machine, many instructions do not require space for operand specification, but all temporary results are kept on the stack which is kept in memory. Hence all operands will require memory references. With accelerators such as cache memory, this loss of use of registers is compensated to some extent, and is balanced by many instructions not requiring memory references to obtain operand addresses. Conversely instruction caching and prefetching to some extent negates the advantage of zero address instructions not requiring operands. To handle common subexpressions requires that intermediate results be available at random, rather than in the regulated manner that a tree structure requires in which values will reappear automatically at the top of the stack when required again. To implement common subexpressions on a stack machine, common expression results required could be accessed using an offset into the stack. This is messy because the offset would have to be computed each time it is required, and when the value is no longer required it cannot be easily removed from the stack. A more practical approach is a second stack that would be used to allocate local variables for a routine and would also be used to allocate space for temporaries. When a common subexpression value is required, the value can be obtained from the variable stack and pushed on the computation stack. A register machine has the advantage that the common value can be stored in a register. Referencing the value later in the code is simple and fast. Therefore whether or not a machine has registers is obviously a significant factor in

defining a family.

An alternative to register based and zero address machines would be an accumulator based architecture. In a zero address architecture, GAS instructions for arithmetic and logical operations would not have any operands. For an accumulator based machine, the location of one operand is implicitly in the accumulator, and the GAS code would reflect this by arithmetic and logical instructions that only have one operand. If a machine has several registers, arithmetic and logical GAS instructions instructions will typically have three operands. It would be possible to create a family for register based machines that have arithmetic and logical instructions that are always two address. The three address form is more general, and can be used for two address target machines as two address instructions can always be derived from three address instructions. However if a GAS code is developed that is two address, then the GAS code will more closely model the final target machine code form if the target machine is two address.

Therefore whether a machine is register, zero address, accumulator, three or two address based will be very significant when defining a GAS machine. In fact whether a machine is register based is not as fundamental at the GAS level as whether the machine is two or three address. Registers are only important in deciding the location of operands. Register machines are likely to be two or three address machines, and therefore require temporaries for holding intermediate subexpression results. Therefore the GAS code for these machines is likely to be two address or three address, and include provisions to declare temporaries, but this would be no different from a memory

oriented two or three address machine that had no general registers.

The subroutine call structure of Rcode (lexical level structuring and scope, arguments, results, and local objects) must be coded efficiently in the target machine. The basic facilities provided by the family for implementing such subroutine calls will significantly affect much of the code. As described in the previous chapter, a decision has also been taken in the PLIP project to implement a general calling standard in the interest of portability of user code. Therefore the subroutine calling conventions inplemented in the target machine instruction set may not necessarily be used, unless it allows conformance to the family standard. Only simple CALL and RETURN instructions are required. However for the VAX family, machines, a stack is assumed available on which the result, arguments and local storage objects can be located and efficiently accessed. Efficient access really requires that pointers are available that can be used to access the result, arguments and local objects efficiently. It is also important that the GAS code retains information on the structure of subroutine calls. This is provided by the GAS machine calling convention and the retention of the Rcode routine declaration Rcode and the "Call" Rcode.

Some Rcodes such as "Add_In_Place" will be implemented in the family GAS code by specifying disabling of interrupts if the target machine does not have a suitable equivalent instruction. Therefore a machine belonging to the family must have instructions to disable and enable interrupts.

Other factors such as addressing modes and data types supported

for various instructions are also very important. However these two factors are not useful for defining a family. As discussed in the previous chapter, Rcode implies the addressing of objects in terms of offsets into storage areas, with indirect addressing allowed. Complex addressing modes allow information on the address computation structure to be retained to some extent. The level to which this information needs to be retained depends on the complexity of addressing modes of the target machines in the family. The GAS operand addressing modes should allow the more complex addressing modes commonly available on target machines in the family to be used. However it is not required that all machines in the family have a certain set of "typical" addressing modes, therefore addressing modes themselves are not part of a family definition. Complex GAS operand addresses will be broken down into a sequence of computations if the target machine does not support the level of complexity present in the GAS instruction address. In the GAS machine prototype developed, access to stack objects via pointers is modelled by GAS machine pointers. The data types supported by target machines and operations available on each data type often vary greatly from machine to machine. Requiring that machines in a family must have very similar datatype support would produce many more GAS families. In fact all that is important for defining the VAX family in reguard to data types support is as stated above; for machines in the VAX family, arithmetic operations must involve a small number of statically sized objects. Individual target machine data type support is instead taken into account during Rcode, GAS code and target machine code generation on a local level using tables of information, but machines in the family do not have to have support for certain datatypes. The constraint that operations are on fixed sized objects limits the range of

machines that will match the family, but allows GAS code to be closer to the target code for the machines in the family, making the second level code more useful. If operations on dynamic and block operands had been allowed, more machines would have been included in the family, but GAS code generated for machines which actually only allow operations on fixed sized objects would not reflect the break down of oversized and dynamic operand operations, which occur in Rcode via block exact arithmetic operations. Additionally, the constraint is applied that operands for arithmetic operations must begin on byte boundaries. The GAS code produced should therefore move operands of aithmetic operations not located on byte boundaries into temporaries.

The VAX family GAS machine prototype includes bitstring facilities. Bitstring operations include the logical operations NOT, OR, XOR and AND plus a SHIFT and an EXTRACT instruction. They operate on bitstrings that can be any size and begin at any bit location in memory. As an example of use, they will be generated to ensure any arithmetic operations that involve objects not located on byte boundries, are extracted into temporaries so that the arithmetic operation can be performed. This models what will be done to extract values from bit packed records and arrays into registers for computation. This does not necessarily imply that the target machine must have all these bitstring instructions. If the target machine does not have a certain bitstring instruction, the GAS bitstring instructions must be converted into a series of target machine instructions involving logical operations that are available such as OR, AND and shifts. Unlike the arithmetic operations, GAS bitstring operations are not limited to operands of fixed sizes. This is because the range of bitstring operations of machines is usually

quite wide, some degree of dynamically sized and large sized operands often being provided. If the target machine bitstring operations require operands to be sized in multiples of bytes, the massaging must be done by the target code generator. The only requirement for a machine to belong to the family is that it is capable somehow of emulating the bitstring operations in the GAS code, even if this only involves AND and OR of register contents.

Families and Constraints

The central issue of family definition should be coming clearer. Defining a family definition was stated as finding architectural features common to a group of machines. This is perhaps more accurately described as finding constraints that are mutually common to significant group of machines. All the constraints should be common to all machines in the family. The larger the number of constraints the greater the number of common code generation steps that can be represented in the Rcode to GAS code generation step, and the more of the final target code objects that will be represented, thus aiding the GAS code optimiser and code generator. An example of how more target machine data objects are represented by applying constraints is that extra temporaries used to massage operands to the same size will be created if a constraint is applied that operands must be of the same size. The architectural features such as two address versus three address can be seen as a constraint factor. Two address code is more constraining than three address code; it allows GAS code to represent the common step for all machines in such a family to convert three address Rcode operations to two address operations, creating the extra temporaries required. It is interesting to note that constraints really represent limitations in the target machine architecture. A machine with an

architecture with few constraints:

three, two, one and zero address instructions

operands can be of differing sizes

operands can be dynamically or statically sized

operands can begin on any bit

would be quite simple to generate target code for from Rcode. The more limited a machine architecture is, then the more work that is involved in generating target code from GAS code. The more that common constraints can be recognised amongst machines, the more the common work that can be done by a GAS code stage. Too many constraints will result in families that have only one member, so the advantage of the GAS code concept would be lost. Finding a balance is the key to a successful GAS code design. This balance for the GAS family took a long time to discern, but now that it has been done the concepts are much clearer and a GAS definition for say a transputer based family would be easier to develop.

For a factor to be important in describing a family, it must be necessary for all machines in the family to conform to the factor. Therefore if two address instructions is a requirement, then all machines must be two address. If arithmetic operations must involve fixed sized objects, this is a constraint. However addressing modes of the GAS machine is not a constraint. Machines in the family do not have to have these addressing modes to be

included in the family. Similarly data type support is not a factor.

GAS machine and code for family containing VAX

Characteristics of family

a) May have two or three address instructions
(not accumulator or zero address instructions
and probably has general purpose registers)

b) Has a stack with pointer register support for allocating and accessing arguments and local variables of routines

c) An instruction set:

- ALU arithmetic instructions
that operate on a small range of statically sized objects. All objects need not be of the same size, but must be of the same general type. All memory operands must be byte aligned

- Memory movement of bytes,
either
dynamically sized
or
statically sized

- Stack instructions to Push/Pop bytes

- Control instructions that essentially involve conditional and unconditional branching

- Simple subroutine call and return

- Interrupt enable and disable

- Instructions that allow logical operations OR, AND, XOR, NOT to be emulated

Temporaries

To allow linearisation of Rcode, temporary storage locations can be defined using the USEVAR GAS code. The basictype and unique id assigned to the temporary must be provided. As it may become obvious that a temporary is no longer required, a DELVAR GAS code is provided so that database entries related to the temporary can be deleted. The basictype size indicates the required size for the temporary, but smaller size values, or parts of the value stored may be used at a later stage. References may be made to values in temporaries by giving the temporary identification, and a byte offset and bit offset to the start of the value in the temporary. These offsets may be computed.

Addressing Modes

Operands will often require an address which marks the beginning of a location from which to start taking a source value or to which is to be written a value. For all objects except bitstrings, these locations are always byte boundaries. For bitstring operations, they can be any bit position. An object in

memory can be referred to in several ways:

a) specifying the identity for the area in which it is located in, a byte offset into the area, plus a bit offset. Both of these offsets can be computed

OR

b) a stack pointer register provides the memory address

OR

c) the memory locations provided by (a) and (b) could be considered to contain the address of the required memory location, i.e. indirection

Additionally, an "extra offset" can be added to the address provided by (a), (b) and (c). This extra offset allows modelling of source language concepts such as accessing the fields of a record that is referenced via a pointer.

The addressing modes effectively provided are:

immediate

value from which source value can be obtained is explicitly provided

storage

direct

direct reference to a memory location requires

a storage area id

and

static bit and byte offset into that area

and

static extra offset

indexed

a) reference provided by storage area id and any of the bit, byte or extra offsets are computed

or

b) stack pointer contents provides address (additionally any of the bit, byte and extra offsets may be computed to give more levels of indexing)

indirection

direct and indexed addresses above contain an address which is added to the extra offset

or

a value in a temporary holds an address which is added to the extra offset

Bit and byte offsets must be a basictype value specified by an immediate value, or computed into a temporary or a memory location whose address involves a static byte offset, and nil bit offset.

The identity of a memory area may need to include reference to the identity of the module from which it has been imported.

It could be argued that these addressing modes are not as complex

as are those on some target machines, the VAX included, but the modes provided are adequate to efficiently implement the constructs found in the intended source languages.

Data types supported

The datatypes supported are the same as for Rcode (Rcode Basictypes of signed/unsigned integer, real, decimal), blocks of Basictype objects and bitstrings. Block operands are only supported for memory move, compare and search instructions. Block arithmetic operations must be reduced to a series of operations on Basictype objects. This reflects that most machines do not provide arithmetic operations on blocks of memory, but do provide block memory move, compare and search instructions. The block structure of such operations is made clear in Rcode and should be retained in GAS code rather than be expressed as operations on a series of Basictype objects. GAS memory move instructions include LOAD, PUSH and POP.

A Basictype operand is specified in GAS instructions by a basictype specifier which is identical to the Rcode basictype specifier, and a reference indicating the operand value or location.

Basictype operands are taken from an immediate value or the contents of a temporary or a memory location. A Basictype coded byte must be given to indicate the size of the operand. If taken from an immediate value, the operand will be taken beginning at the least significant bit of the immediate value. If taken from memory, the operand will be taken from the bit whose address is given. If taken from a temporary, the value is taken from the bit at the the offset specified.

Block operands consist of a block of basictype objects. The size of the block in terms of number of basictype objects is specified by one basictype operand, and the start of the block and the basictype of each unit provided by another. Block operands always involve objects in memory, or constant, initialised storage areas. Therefore a bit address must be provided or a bit offset into a constant provided. If the block is in fact a single basictype object, the size operand will be an immediate value that specifies a length of one. This allows clear identification of memory move instructions that involve a basictype value and not a block. If the block only involves a series of bytes, the basictype specifier can indicate an unsigned type one byte long, and the size operand the number of bytes.

Bitstring operands may be taken from an immediate value, or located at any bit in memory, temporary, or constant area. They must also be provided with a size, which must be a basictype value which is immediate or located in a temporary, or at a location in memory that is specified with a static byte offset, and bit offset.

The data types supported are therefore quite extensive and should exceed most of those available on the target machine.

In practice, GAS operations will not necessarily be on objects supported by the target machine. Within one machine there is often a complex variation in the datatypes that will be supported by different instructions. Instead, the GAS generator only need chose a Basictype that is as big or perhaps slightly larger than is supported for the operation involved. The target code

generator then can easily generate code for the GAS code. However the closer the GAS code operand is to a supported size, the more effective the GAS code optimisation will be. An alternative approach would be to reduce operands to the same size or smaller than the largest size supported by the target. The GAS code optimiser would be effective on this code, but the code may be poor if many operands are smaller than necessary. The choice of approach depends on the ammount of information made available to the GAS code generator. Providing information on all the available data sizes for each data type supported by each GAS code operation could be significant. The approach taken has been to represent largest size of operand that can be handled by the target machine for a specified a given GAS code such as ADD. When ADD operations on large sized objects are encountered, they will be be broken down into operations on objects of this size. If the size shown is not actually supported by the target machine in the specific circumstances (e.g. register that allows ADD on objects of this size can't be used) a few extra target machine code instructions may have to be generated. This will not be difficult. Therefore accurate representation of target machine data types supported under all possible situations is not vital for GAS code generation.

Memory Structure

The GAS machine memory structure required much thought. The desire that GAS code reflect the nature of target machine code meant that code to reference local variables and parameters should reflect the fact that such objects would normally be stored on a stack. However it would require a lot of effort for the GAS code generator to accurately model the target machine stack as it will be at runtime. Additionally the contents of the

stack may change with optimisation and when the detailed target machine code is generated. Objects that appear dynamically sized may turn out to be statically sized. Objects not referenced may be omitted. Additional objects may appear on the stack. Alignment constraints make the possible effects of objects disappearing and appearing quite complex. The approach taken is to have a stack which consists of "objects". An object is one complete Rcode variable, GAS temporary, Rcode Basictype object (variable or immediate) or GAS pointer. Each stack location contains one object. When Rcode declarations for a local object of a routine are encountered, the object is assigned the next object location on the stack, and a PUSH instruction is generated. The PUSH instruction has three operands; alignment which is Rcode byte form used to specify required alignment, a basictype operand to specify size of block to push and a basictype operand that specifies the type of basictype object and the first basic object in the block. To allocate space for an Rcode variable, the operand that specifies the first basic object in the block will be "NIL". When target machine code is generated, the PUSH instruction will probably be converted into an increment of the stack pointer (probably for all local objects in one increment). Therefore the GAS machine will not involve the use of dope vectors for dynamically sized objects. When objects on the stack are referenced, they will be referenced by their "object offset" plus a byte and bit offset within the object. As stack pointers are maintained (see below) the offset for local variables of routines will be from a pointer LOCALSTORE. This implies that the GAS code generator must keep a data base of object offsets assigned to Rcode objects, so that later references to these objects in the Rcode can be converted to their object offsets. If the GAS code containing Rcode tree is

written out to disk, this database will be lost. However, the Rcode declarations should be retained, so the database can be rebuilt when the Rcode is rebuilt. Arguments and results will similarly be pushed onto the stack. A description of the GAS machine pointers and calling convention is given below.

Global storage areas will represent storage areas defined in the Rcode. The GAS machine will retain the same global storage structure as the Rcode. This involves the concept of several independent storage areas. The sizes of these areas will be defined in Rcode declare and append area codes and the import area code. These Rcodes will be retained and no GAS code will be directly generated for them. The target code generator will convert them into suitable portable linker directives. GAS operands that reference an object in a global storage area will do so by specifying the storage area id, module id if the area is imported, and a byte and bit offset into the area.

If a GAS instruction has an operand that would extend past the static storage area or stack object in which it is located, the behaviour of the program will be unpredictable. When an operand extends beyond a stack object, it may extend into the following object on the stack.

Code Labels and Addresses

The structure for memory containing compiler generated code is mainly important in terms of identifying routines and targets of branch instructions. The actual size of storage used and alignment requirements is not important to the front end or for the GAS code, as long as entry points for procedure and the target of branches can be declared and referred to.

In any computing machine, code path control structures are an essential component. The GAS machine must model the typical facilities provided by its family for code path control. Fundamental to this is the addressing methods used to specify the targets of subroutine calls and branches. In the GAS machine these cannot be specified as real addresses, or relative displacements because these require that target machine instructions have been determined so that the addresses of, or displacements to, the target addresses are known. GAS code operands must be generated for GAS CALL and branch instructions that specify the target in a way that doesn't involve an explicit target machine address displacement.

Many of the branches will be required to implement Rcode control structures such as loops and case structures. The GAS code pseudo instruction USEVAR can be inserted in the GAS code to mark points that will be used as targets of branch instructions used to implement these control structures. As will be discussed later in the chapter on GAS code optimisation, basic code block identification and flow analysis will primarily be involved with looking for USEVAR codes and branch instructions that refer to these labels. GAS branch instructions can use the label id as a target specifier. The target code generator will convert the uses of the label into the address allocated to the label. A GAS code interpreter will interpret uses of the labels as references to the GAS code following the USEVAR code that defined the label.

Some GAS branches will involve the direct implementation of Rcode jumps to targets that will involve labels that are defined in the Rcode by the Declare_Label Rcode. These jumps may also involve a

return to a different "environment" (see below under subroutine calling conventions). The aspects normally involved in change of environment on a target machine such as stack unwinding are assumed to be automatic in the GAS machine. The target of the GAS instruction will be specified by an operand that refers to the label by the Rcode defined label id and lexical level. The "Declare_Label" Rcodes are left on the Rcode tree by the GAS code generator to mark their location for following phases of the compiler. A GAS code interpreter would interpret the use of the label as a pointer to the Rcode subtree following the Rcode that declared the label. However it may also have to unwind the current "environment".

Subroutine Calls in GAS code will require a target address. The GAS CALL instruction will be produced in response to the Rcodes "Call" and "Fast_Call". The generation of the target addresses will ultimately be traced to a Refer_Routine Rcode which returns a routine descriptor, part of which contains the routine address. When the GAS code generator encounters a Refer_Routine Rcode, it should generate a GAS operand that refers to the routine by the Rcode assigned id and lexical level. For some GAS codes, this operand will be assumed to return a routine descriptor. If this operand is supplied as the target of a GAS CALL instruction the address part of the routine descriptor is assumed to be the target address. The environment pointer will be relevant for a "Call" Rcode, but not a "Fast_Call" Rcode. If the target of the call is computed the contents of the storage location specified in the GAS CALL instruction provides the target address of the call. Note that the storage location value will have been filled by a value that could be traced to a Refer_Routine Rcode. GAS code, as stated earlier, does not involve real target machine

code addresses. The "Refer_Routine" is the only way the address of a routine can be described in Rcode. The "Refer_Routine" Rcode and GAS operands that specify a routine id and lexical level are expected to return an address in the target machine. A Rcode or GAS code interpreter encountering these would have to generate a target machine pointer size value that uniquely identifies the relevant program points. The values wouldn't have to be true target machine code addresses. They could be identification codes for interpreter database records for labels or routines in the Rcode or GAS code.

Runtime Structures

This deals with the support the GAS machine will provide for maintaining runtime structures. These structures are related exclusively to structures associated with subroutine calls.

One of the more significant aspects of modern processor architectures is the facilities provided to support subroutine calls. Some architectures provide facilities required for Fortran type structures, with no concepts of lexical nesting and associated scope concepts. These processors provide call mechanisms that provide efficient allocation of local storage for the subroutine, and dynamic return from the subroutine, including the automatic deallocation of local storage, and space occupied by the arguments. Additionally the mechanism often provides facilities for the automatic saving and restoration of all or selected registers. Registers are provided that allow efficient access to arguments and local variables. Lexical nesting and scope must be handled by the programmer. Some processors provide facilities to allow the automatic maintainence of access paths to all objects in "scope" of the current procedure. Some processors

also provide for the inclusion of automatic facilities for exception handling associated with the procedure call structure. In the VAX, if an exception such as divide by zero or stack overflow occurs, the context of the current procedure may define the address of an exception handler which will be called. If the address defined in the current context is NIL, the exception handler definition in the caller environment will be examined, and if not NIL, this exception handler will be called. If it too is NIL the unwinding process continues down the dynamic environment call chain until a non-NIL exception handler is found. When a procedure is called, the default exception handler will be that of the caller. The programmer can then define a new handler at anytime within the procedure. The concept of exception handlers associated with procedures is not implemented in Pascal or Algol, but is present in ADA and the Portable Language Implementation Project runtime library.

Also associated with procedures is the handling of results. In many cases attempt will be made to return the result in a register. An alternative is to return the result on the stack.

The caller code and subroutine code must conform to the same convention as to how the subroutine arguments and results will be handled. Usually the assembler programmer or language compiler will adhere to a standard convention for all subroutine calls. In many machines these conventions are machine specific involving specific registers and expecting the use of specific instructions. Universal compliance with the convention enables the use of independently compiled subroutines. In the Portable Language Implementation Project, it is desired that the convention be as similar as possible on whatever target machine

is involved. This assists portability of any code that deliberately manipulates the stack, for example implememtation of the "Raise" exception procedure in the Modula-2 runtime library of PLIP manipulates the stack to force return to an environment that may be several layers below the current procedure level. Implementation of this on a new target will be easier if the stack structure is similar. The only differences in the stack structure of various target machines will be related to the alignment requirements and miscellaneous objects such as static and dynamic link pointers. This also implies that the result will always be passed on the stack, and that calls to subroutines may not involve the use of special instructions of the target machine, but will use any simple call instruction that merely involves saving the processor status word/register and return address, then jumping to the address specified. Special CALL instructions will only be used if they preserve the fundamental stack structure.

The additional advantage of using a common convention is that GAS code can be generated for the handling of arguments and results rather than leaving this to the target code generator. One of the major aims of GAS code is that it should represent as much as possible the common steps involved in generating target code. The alternative would be to have a high level GAS code instruction for subroutines that left the implementation details completely to the target machine generator.

With these concepts in mind, a calling convention and associated GAS codes and GAS machine registers have been designed. They allow the target code generator to recognise the full structure of the subroutine call so that special call instructions can be

used, and the call linkage mechanisms can be implemented, such as static links and register saving.

The GAS machine stack will be used for results and arguments. Several pointers are defined that point to relevant locations on the stack. These pointers are:

a) TOP_OF_STACK. Contains the offset of the object on the top of the stack.

b) ARGLIST. Contains the offset of the start of the argument for the current procedure.

c) LOCALSTORE. Contains the offset of the start of the local objects for the current routine.

d) STACKBASE. Contains the unique byte address of the base of the stack. This will not normally be referenced in GAS code.

e) STACKLIMIT. Contains the unique byte address of the limit to which the stack may grow.

f) ROUTINEDESCR. Contains the routine descriptor for the currently active routine. Consists of the current Program Counter and a STATICLINK pointer (see below).

g) PC. Contains pointer to current GAS code instruction.

h) STATICLINK. Contains pointer to the static environment for the current routine. This may be a pointer to an environment vector containing pointers to all the DISPLAY vectors that are

currently in scope, or a pointer to the DISPLAY vector of the lexically enclosing routine. In this it forms a static link chain mechanism. The static environment vector and static chain mechanisms are both used commonly.

i) PREVDISPLAY. Points to the DISPLAY vector for the previously active environment, which is for the caller routine. This represents the dynamic link mechanism.

j) RETURNMARK. Contains the offset of the stack location containing the return address to be used when returning from the curent location.

k) RESULT. Points to the result area for the routine.

l) EXCEPTHANDLER. Points to the routine to be executed if an exception is detected by some monitor. The routine takes one parameter which is pointed to by EXCEPTPARM. The exceptions which can be detected are obviously machine dependent. Typical examples are divide by zero, stack over or underflow, real arithmetic overflow or underflow. They require that the processor has hardware monitors to detect these exceptions. Different machines will have different mechanism for responding to such exceptions. They may have a VAX approach where all exceptions will invoke a common handler, or may have a separate handler for each exception type. If the latter is used, the target code generated should set all handlers to point to the same handler routine.

m) EXCEPTPARM. This contains the parameter that will be used by the exception handler routine. Its contents are machine specific, but should allow the exception to identify the

exception cause.

These registers allow implementation of subroutine calling, but also allow a way of expressing in a portable way, manipulation such as changing the return address so return will unwind to an outer lexical level.

Subroutine Calling Convention

The subroutine calling convention used (see later diagrams that demonstrate what happens at each step) is as follows. The full convention is based on the assumption that GAS code must be fully self-contained, without any requirement that Rcodes such as "Call" and "Declare_Routine" be present. The only Rcodes required will be the declarations of static areas and constants, and exported symbols. However the GAS code generator will normally leave on the Rcode tree Rcodes such as "Call" and "Declare_Routine" that will be useful during code generation.

a) Compute the size of the result area and allocate space on the stack using a PUSH instruction.

b) Execute the GAS PUSHMARK instruction. This will:

Create a new uninitialised "display" vector

Moves the contents of the TOP_OF_STACK to the RESULT pointer of the new display vector

Saves space for the return address and any necessary status information by performing a PUSH of an NIL value that is the size

necessary. This will require target machine information on the size required.

Moves contents of TOP_OF_STACK to the RETURNMARK of the new display vector

moves the contents of the TOP_OF_STACK incremented by one, to the ARGLIST of the new display vector

The PUSHMARK instruction alerts the target code generator that a subroutine call is under way. It assumes that the result area is the object on the top of the stack. The display vector may on the target machine exist as registers, or space may be allocated on the stack, or it may be a mixture of registers and stack objects. All parts of the vector do not have to be created at this point, but could be created in parts so that the components are spread through the stack. The implementation of the display is up to the target code generator.

Note that the PUSHMARK instruction is not necessary if the "Call" Rcode is present, as it makes it clear to the target code generator that a call involving the standard calling convention is under way. The subtrees of the "Call" Rcode will also make it clear when the result area size computation is under way that it is a part of a subroutine call. This information can be used to useful effect by the target code generator (see Chapter Six).

c) Push the arguments onto the stack.

d) The target routine descriptor is computed, and the environment pointer is pushed on the stack. If the routine

address is computed the descriptor may be in a temporary or allocated memory location. The routine descriptor contains the address of the routine entry point and a pointer to its environment. Computation of descriptors essentially occurs when the Refer_Routine Rcode is encountered. The GAS machine will assume routine descriptor computation is automatic. When a Refer_Routine Rcode is encountered, a GAS code operand containing the routine id and lexical level is generated. When this operand is used in a GAS code, it is assumed that the routine descriptor will be returned, unless the instruction is the CALL GAS code, in which case, just the address of the routine is returned. The CALL instruction is low level and is used for implementing the Rcode "Fast_Call". It is not involved in environments. If the Rcode subtree of the "Call" Rcode for computing the routine descriptor of the called routine merely contains a "Refer_Routine" Rcode, GAS code will be generated to PUSH the environment pointer of the descriptor, and the GAS CALL instruction will refer directly to the routine via its Rcode assigned id and lexical level. The target machine code generator will actually have to generate code to compute the environment pointer (and an environment vector if necessary). It will recognise this because the PUSH GAS code involves the environment pointer of the routine descriptor for a routine. If the routine descriptor Rcode subtree involves a computed descriptor, GAS code must be generated to perform the computation, PUSH the environment pointer of the resulting descriptor and the CALL generated will specify an indirect address through the address field of the descriptor, which may be stored in memory or a temporary. The contents of the environment pointer field will have been computed at some Refer_Routine Rcode in the descriptor computation subtree or even earlier. The environment structure chosen depends on the target machine. It

may be a static link pointer (the address of the activation record of the lexically enclosing routine) or an environment vector (which contains pointers to all activation records in scope for the procedure). As it is quite possible that procedure variables (hence a procedure call may involve a dynamically computed target address and associated environment) and procedure parameters may be used, the static link mechanism would probably be favoured reguardless of the host machine facilities. The backend is not designed for a specific language such as Pascal where procedure variables and parameters can be ruled out.

e) Call the routine using the CALL GAS instruction with the address determined in (d) as target. This will result in the current "address" being pushed onto the stack, any status automatically saved, and control being passed to the address specified. This instruction is a simple jump to subroutine instruction, and it is anticipated that the target machine code generator will usually generate such a simple instruction.

f) The first instruction in the subroutine will be the NEWDISPLAY GAS code. This instruction indicates to the target machine code generator that standard calling conventions are to be used when this routine is called. Note that this instruction is not needed in the presence of the "Declare_Routine" Rcode as it will be obvious that a subroutine is being entered. The NEWDISPLAY instruction will operate as follows:

> The most recently created display is made the current display, and will make its PREVDISPLAY field point to the previously active display. There could be several new displays that have

been created, but which have not yet been activated. This can occur when evaluation of arguments or the target routine descriptor involves routine calls, which themselves may involve further routine calls. The most recently created display is always the correct display to be activated when a NEWDISPLAY code is encountered.

The return address and status information is on the top of the stack and is "popped" into the memory location pointed to by the RETURNMARK field of the display. In target machine code this pop may not need to be done as the RETURNMARK may be implemented on the stack using the value pushed by the CALL instruction used on the target machine. There is

The static link is "popped" into the STATICLINK of the display vector. This operation may also not be performed in the target machine code as the STATICLINK may be implemented on the stack.

The TOP_OF_STACK is moved to the LOCALSTORE field of the display.

g) Allocate space for local objects by the use of PUSH instructions.

h) At the completion of the routine, the POPMARK GAS code will be used, followed by the RET code. This will place the

RETURNMARK value of the current display into the TOP_OF_STACK of the previous display, delete the current display, and make the previous display active. The RET instruction will return using the address pointed to by the TOP_OF_STACK and reinstate any status registers of the machine that had been implicitly saved during the subroutine call. RET is not involved with environments and is a simple return from subroutine.

The target machine code generator may be able to perform the steps of the call convention very easily using instructions and facilities such as frame pointers in the target machine. It may have to perform significant work to implement the call convention if the machine only has limited facilities for maintaining the display components. The GAS subroutine convention represents the essence of subroutine calls when lexical nesting and scope rules are involved, and the stack is used as the primary vehicile. However it does represent the conventions that the result and arguments will be passed on the stack, and that an environment pointer will be passed on the stack. It also reflects that registers will commonly be available to assist the process.

Non-standard calling conventions can be implemented by using only the CALL GAS code. The programmer or front end language processor will be responsible for handling arguments and results themselves. The Rcode "Construct" can be used to guarrantee that arguments are placed on the stack exactly as they should be. The language front-end could also decide to bypass the standard convention for short high-use routines, and generate Rcode to pass parameters in registers. In both cases the Rcode Fast_Call will be used instead of the Call_ Rcode this is, however not the concern of the back-end.

Input and Output

The Rcode input and output statements are supported by GAS code IN and OUT statements. The operands of the instruction specify the port address, size of block to transfer, buffer address, and the type of port (indicated by Rcode Basictype field of the buffer address operand) to indicate if byte, double byte etc port is involved). Whether the port address is valid or not will be target hardware dependent. The target machine code generator will have to generate appropriate code to check the port address. The target hardware may support the block port operation directly by a single instruction, or it may have to generate a loop of instructions to execute the operation. The GAS machine retains the block input output concept as it is much easier to break a block move down than to recognise a block move from a loop containing move instructions.

System Virtual Calls

These are used to make uses of services of the operating system kernel. What they are translated into will be determined by the target code generator. The operands of the SVC code is a value in the range [0..255] to represent the call code, an operand specifying the number of following operands, and zero or more following operands. The code value will be interpreted by the target code generator as a kernel function and will process it accordingly, using the operands as required. Currently two standard SVC codes have been defined:

254	Save Context
255	Load Context

These will be used by the front end or programmer to implement virtual concurrency facilities. Load Context will require one parameter to specify the identification of the process involved.

NOTE: Context is here specified to mean those elements of the state of the target machine needed to enable execution to continue after a save and load pair as if neither had been executed, and regardless of what occurred betwwen the save and load.

Interrupts

The GAS machine is provided with general interrupt facilities available on any machine. This consists of:

a) SETINT new_state. The new state operand can have values

DISABLE

ENABLE

b) RTI. Specifies a return from interrupt.

Both of these GAS code instructions will be converted to appropriate target machine code. As these facilities are available on most target machines they allow portable implementations for Rcodes such as Add_In_Place, that do not depend on the availability of any special indivisible instructions. Along with the SVC code for load and save context, these two instructions provide the essential portable tools for interrupt handlers. The register and input/output port instructions will also be very useful.

Machine State

The GAS machine status is specifically represented by five state variables. They are:

CARRY
OVERFLOW
EQUAL
POSITIVE
NEGATIVE

For any target machine there will be other additional machine status information. Manipulation of these status bits will therefore be non portable. Usually status bits are stored in a specific register. Manipulation of these registers is achieved by the two GAS instructions STATESET and STATEREAD for manipulating registers. Any target machine status registers, and the GAS status are preserved and reinstated on CALL/RET GAS instructions.

Any instructions that manipulate the machine status registers is obviously target machine dependent. However the bits related to the GAS status bits can be manipulated portably.

The GAS status flags can be used as source values for operations and therefore provide a portable means of implementing overflow handling. They are used to break Rcode operations on dynamically sized objects into GAS code operations on "Basictype" objects. When the condition of a flag represents is TRUE, the flag will have a value of one, else it will be zero.

The GAS flags are set by various GAS instructions depending on the instruction results. The target code generator must ensure

that target status bits used to represent the GAS status bits are set accordingly, either as a consequence of the target machine instructions, or by specific setting of the status bits if not set by target machine instructions. Reference to GAS status flags will require target code to access the target machine status bits used to represent the GAS flags.

Logical Shifts

Most processors provide some form of instruction to rotate the contents of memory or a register. The shift will be by a certain number of bits, and could be left (the most significant bit direction) or right (the least significant bit direction). The shift could be arithmetic, logical, or rotation. If arithmetic, the most significant bit will be considered a sign bit. If the shift is left, the shift will simply not involve the sign bit. If the shift is right, on each bit shift, the sign bit will be replicated into the bit immediately to its right. If the shift is binary, all bits will be involved. If the shift is rotation, bits shifted off the end will be placed back in the bit position on the other end of the bitstring. Target machine code must be generated to produce the exact effect specified by the GAS shift instruction. The bitstring specified as the target of the shift in the GAS shift instruction can be any size, so a series of target machine instructions may be required. The GAS code generator should try to produce shift operations on objects directly supported for shift operations.

GAS Instructions

The format of GAS instructions is an operation code followed by several operands. All instructions except the SVC have a fixed

number of operands. The target machine instruction set must have an instruction set that can sensibly emulate the GAS instructions. As a miminum, emulation of each GAS code should only require a standard template of target machine code instructions. Additional target machine code will be required, particularly for register dumping and restoration.

GAS Operands

Gas operands specify the values to be used for an instruction. The values can be basictype objects, blocks of bytes, bitstrings, a branch label, condition value or shift mode. A special case exists when an uninitialised object is pushed on the stack. This is done when space is allocated for the result of a routine and for local variables of a routine. In this case the source value to push will be specified as a "nil" address value(see above) .

Compiler Architecture

The PLIP compiler structure has been organised to allow separate phases to operate as independent processes. From the backend viewpoint, the phases are:

Front end

Gas code generator

Gas code optimiser

Target Code generator and optimiser

As GAS code will be attached to the Rcode tree, all phases are designed to operate from Rcode queues. An Rcode queue effectively

provides a pipeline between phases. The common connection mechanism between these phases is one of the benefits of encoding GAS codes as part of the Rcode tree. A plug and socket approach can also be supported as each phase does not know how Rcodes are placed on its input queue, or what happens to Rcodes it places on its output queue. For example, Rcodes on the input queue to the GAS code generator may be placed there by an Rcode file reader process, or directly by the front end process. The Rcodes from the output queue may be absorbed by a process that writes Rcodes to a disk file, or directly by the GAS code optimiser. When the decision was made to attach GAS codes to the Rcode tree it became clear that a common connection mechanism between phases became possible, and some time was spent developing the concurrent, queue connected architecture. Additionally the Rcode file readers and writers were extended to call routines that encoded and decoded GAS codes from the Rcode reserved extension, thus allowing Rcode file input/output routines to be used for any intermediate files between the GAS code generator, GAS code optimiser, and the target code generator.

The Generation of GAS code from Rcode

The generation of GAScode from Rcode will be done on a post-order flow of Rcode using what is essentially a syntax directed translation mechanism. Code holdup for one procedure will not be used. Therefore the generator operates on a very local level. The main code improvement mechanisms used during this phase are to collapse as many address computations as possible into one GAS addressing mode so as to preserve as much information on an address structure as possible, and to try and eliminate unnecessary store instructions. The unnecessary store instructions will commonly be generated for Rcode sequences such

as:

It would be easy to generate GAS code of the form:

```
USEVAR   temp1
ADD      X,#5,temp1
LOAD     temp1,Y
```

However the USEVAR and LOAD instruction can be eliminated

```
ADD      X,#5,Y
```

A holdup stack is maintained to allow these code improvements to be achieved. Each node of the stack contains a GAS operand and a GAS instruction. The node at the top of the stack will contain the GAS operand that contains the result of the previous operation, and also the instruction itself. If no instruction was generated for the previous Rcode the instruction code field of the stack node will be NIL. This will occur for Rcodes such as "REFER_VARIABLE", which are involved in address computations that can be represented directly in GAS addressing modes. Each

instruction of the address computation will result in a more complex addressing mode being constructed. If the previous Rcode does not return a result, the operand field will be NIL. This will occur for Rcodes such as "STORE". To illustrate how this stack is used consider the generation of GAS code for the following Rcode. The tree structure of the Rcode is represented by indentation.

```
STORE
     data_type_1
     REFER_VARIABLE Z
     ADD
          datatype_2
          LOAD
               datatype_3
               REFER_VARIABLE X
          LOAD
               datatype_4
               ADD
                    datatype_5
                    REFER_VARIABLE  Y
                    LITERAL #6
```

The REFER_VARIABLE Z will be encountered first and converted to a "source" GAS operand that will return the address of "Z". The "basictype" field for the operand will be set to that required for a target machine pointer value. This operand will be pushed onto the stack with a NIL instruction field. The "REFER_VARIABLE X" Rcode is encountered next and a "source" GAS operand is generated that returns the address of "X". The "basictype" field value is also set for a target machine pointer type. This operand

is also pushed on the stack along with a NIL instruction field. The first LOAD instruction is then encountered. The holdup stack is popped and the operand which returns the address of X is obtained. This is changed to an operand which specifies an object of basictype datatype_2 that is loacated in memory at X. This operand is pushed back onto the stack with a NIL instruction. The REFER_VARIABLE Y Rcode is then encountered and an operand that specifies the address of Y is pushed along with a NIL instruction. The "LITERAL #6" is encountered next and a "source" GAS operand returning an immediate value #6 is generated. The "basictype" field of the operand is set for the size and type of the immediate value. This operand is also pushed on the stack along with a NIL instruction field. The second "ADD" Rcode is then encountered. The two operands on the stack are popped, and it is determined, one specifies the address of Y and one is an immediate value #6. As a result, a new operand is created that specifies the address of the location at an offset of 6 bytes from the start of "Y". This operand is pushed onto the holdup stack with a NIL instruction. This illustrates how address computation structure provided by Rcode trees is represented in the addressing mode facilities of GAS code. The LOAD Rcode is then encountered. The holdup stack is popped and the operand that specifies an address offset 6 bytes from the start of Y is found. This is changed to an operand that specifies the contents of this location, and the Basictype specifier for the operand will be set to datatype_4. This operand is pushed on the stack with a NIL instruction. The first ADD Rcode is then encountered. The holdup stack is popped twice. This will provide one operand that specifies a basictype object of type datatype_3 and located in memory at X, and an operand which specifies a basictype object of type datatype_4 located in memory at offset 6 bytes from Y. A

GAS temporary is defined (but no USEVAR GAS code is generated yet), and an operand generated that specifies a basictype object of type datatype_2 located in this temporary. A GAS ADD instruction is then generated which has as source operands the two operands popped, a target operand the operand that specifies the temporary. Note that the datatype of each operand may be different. The operand referring to the temporary and the new GAS ADD instruction are pushed on the stack. The STORE Rcode is then encountered. The holdup stack is popped twice and it is noted that the operand popped first is a temporary whose contents are generated in the instruction popped with it. The temporary is discarded. The second operand popped defines the destination address of the store. The operand specifies the address of location Z. This operand is changed to specify a basictype object of type datatype_1 located in memory at Z and the GAS ADD instruction is altered so that this operand is the target. This ADD instruction is then emitted.

Note that if any of the operands for this ADD instructions did not begin on byte boundaries, temporaries would have to be used and the value moved to the temporary for the ADD operation. This reflects the family restriction that arithmetic operands must begin on byte boundaries.

This process is essentially a standard bottom up translation. The techniques for converting such things as Rcode arithmetic and control operations to linear GAS code involve standard well established code translation methods for converting a parse tree to target code. Code has been produced for this translation process but has not figured dominantly in the design effort for the two level concept as no new issues are involved.

Dynamic sized and oversized objects

When arithmetic operations on dynamically sized objects or objects too large for basictypes, are encountered, they must be broken down into operations on objects of size supported by the target machine. In the VAX family GAS code, this breakdown will be performed at the GAS level. The following notes decribe how this problem can be approached for operations involving equal size source objects and illustrates how the GAS resources defined for this GAS machine can be used. The handling of overflow as specified by the Rcode arithmetic operations is also discussed. These multiple arithmetic algorithms themselves are well known.

Dynamically sized or oversized objects can be encountered in Rcode extended exact arithmetic codes, decimal arithmetic and bitstring operations. Extended arithmetic codes and decimal arithmetic involve objects that are a multiple of bytes with the operands specified by Block_Load, Block_Update and Select Rcodes or as the result of a preceding block operation. Bitstring objects start on byte boundaries and are a multiple of bits with operands specified by the Bit_select, Bit_Block_Load and Bit_Block_Update Rcodes or as the result of a previous operation. Bitstring operations are not broken down since target machine bitstring facilities are usually quite varied such as dynamic bitstring operations up to 32 bits. It will be the task of the target code generator to break down bitstring operations into the target machine instructions required.

The breakdown for dynamically sized objects must be done at run time. Statically sized objects can be broken down at compile time. The algorithm for the breakdown depends on the type of

operation involved. In the following algorithms it is assummed that the largest source operand the target machine will support for the arithmetic operation involved, is called a basic unit and that data is packed with the least significant byte at the lower memory address.

Addition/Subtraction of (signed/unsigned) oversized integers

This description assumes both operands will be of the same size. For subtraction it is assumed that the second source operand is the subtractor (see GAS code reference manual). When two integers of a size n bits are added the result will be an integer of size n but with the possibility of an overflow bit. The Rcode datatype for the operation will specify how overflow is to be handled. A byte may have to be prepended to the result which is a boolean indicating whether or not overflow has occurred, or an overflow error may have to be signalled. Appropriate GAS code should be generated. The signalling of overflow error is left to a default target machine handler. The Rcode only specifies that an overflow error message should be signalled, not what the message should be. The GAS code SVC call instruction with a value indicating integer overflow will be generated.

The algorithm for breaking down dynamically sized integer addition and subtraction is as follows:

> When the size of the source operand is greater than the size of the largest object that the target machine allows for the operation involved then perform the following:
>
> Create a binary byte sized temporary initialised to

zero, and two temporaries of size equal to the largest operand that the target machine can handle directly for the GAS code instruction involved. These temporaries will be called basic unit temporaries "A" and "B".

For each basic unit of the source operands taken in turn from the least significant end, until the remaining most significant part of operands are of size equal or less than the size the machine can support:

(i) Except for the least significant basic unit, perform an unsigned operation on the basic unit of the second source operand and the byte temporary and store the result in the "A" basic unit sized temporary. If overflow occurs, move "1" to the byte temporary, else move zero to it.

(ii) Except for the least significant basic unit, perform an unsigned operation on the first source basic unit and the "A" basic unit temporary, storing the result in the corresponding basic unit of the target operand. If overflow occurs add "one" to the byte temporary. For the least significant basic unit, add the two source operand basic units, and store the result in the corresponding target basic unit. If overflow move one to the byte temporary.

Generate code to load the remainder of the first source

operand into the "A" basic unit temporary and the remainder of the second source operand into the "B" basic unit temporary. This is done because the remainder may be any size less than or equal to the size the target machine can handle. Hence the compiler must use a temporary of basic unit size.

Generate code to perform the operation on the byte temporary and the remainder of the "B" basic unit temporary storing the result in the "B" basic unit temporary. If overflow, set the byte temporary to one, else zero. Perform the operation on the remainder of the "A" and "B" basic unit temporaries placing the result in the "A" basic unit temporary. Test if the result in the "A" basic unit temporary is too large for the remainder of the result operand or that overflow has occurred; if so set the byte temporary to one (else do nothing). If the byte temporary is still zero copy the least significant bytes of the "A" basic unit temporary to remaining bytes of the result operand. If the Rcode operation datatype specifies, move zero to the least significant byte of result operand to indicate no overflow has occurred. If the temporary byte is not zero, and the Rcode datatype requires a byte to indicate overflow, do the copy as above, but move one to the least significant byte of result. If the Rcode datatype of the operation specifies signalling a runtime error on overflow, generate GAS code SVC instruction for overflow instead.

Generating code when the the compiler can determine if the

operation involves operands that are not larger than supported by the target machine for the operation is as follows:

This requires that the size be statically determinable. Check whether it matches a datasize supported by the target machine. This would require detailed information available on the data sizes the target machine directly supports for each GAS arithmetic operation and for each datatype such as signed integer. This is not a large volume of information as there are not many GAS arithmetic instructions. If the operand sizes are supported by the target machine generate an instruction to perform the operation. If not then generate an instruction to load the remaining bytes of each source operand into temporaries sized to the next largest size supported by the target machine and then perform the operation on the temporary. Code is generated to check whether the result in the temporary can fit back into the remaining bytes of the result, or that overflow has not occured. If no overflow occurred the result temporary which corresponds to the remainder of the result is moved to the result operand. A zero value is moved to the least significant byte of the result if the operation datatype specifies prepending a byte indicating if overflow has occurred (see Rcode reference manual). If overflow has occurred, then the action taken depends on the datatype specified for the operation. If the least significant byte must be set to indicate overflow, move a value of one to the most least significant byte of the target operand, after doing the copy from result temporary as above. If a

runtime error must be signalled, generate an SVC Gascode instead.

If the operands are oversized, by statically sized, the breakdown will follow that described for dynamically sized operands, however the compiler could take advantage of knowing the exact size of the operands and generate a simple iterative counted loop through the basic units of the operands.

Multiplication of oversized integers

This assumes both operands are of the same size and that the result of multipication is double this size and that division produces a result which consists of a dividend and remainder each of the same size as the divisor. However each iteration of basic units will be more complex, but the algorithm will involve the usual multiple precision algorithms that can be implemented using temporaries, and information on target machine sizes for arithmetic operations.

Implementing Control Structures

Control structures such as FOR loops, CASE structures and loops are implemented using conditional and unconditional branches. However unlike code generation of code for a specific target machine in which branching addressing modes of the target machine will be used, typically involving relative addressing modes, the GAS branch instruction will involve a branch to a specific code location marked by a GAS label. As the chapter on target code generation will describe, the target code generator will convert any branch into an addressing mode suitable for the target machine.

The GAS generator can take any approach in generating GAS code that, given the semantics of the GAS machine, will lead to the production of correct target machine code. This leaves room for such decisions as allocating indexes of FOR loops into temporaries or onto the stack. It is envisaged that the GAS generator will be significantly modified to improve target code produced, based on tests of code produced for various pieces of source code. The GAS code generator algorithms are complicated by the fact that not only is the efficiency of the GAS code produced important, but that some seemingly efficient GAS code may produce inefficient target machine code. It is also important that the writter of a GAS code generator does not have a specific target machine in mind. The GAS code may produce efficient code for this machine but very poor code for others.

Chapter Five

GAS code Optimisation

This chapter discusses the reasons for optimisation of GAS code and then discusses the requirements and design of the optimiser. The design uses many of the optimisations based on data flow analysis as described for example by Aho et al [9]. They describe these concepts in terms of linear three address code and do not provide a rigorous description of implementation. This chapter presents a design using data flow analysis in the context of GAS code attached to an Rcode tree. It includes a description of many of the required implementation aspects, such as how to represent basic code blocks (defined below), the contents and structure of a database that stores information generated during optimisation and problems such as identifying "variables" relevant to optimisation. Many of the optimisations are well described in the literature on compilers and hence are not described in this chapter. This chapter aims to present the design for a complete optimiser; what optimisations should be performed, what data structures should be generated to enable efficient implementation of these optimisations, and elaboration of handling problems such as identifying variables and handling pointers in the Rcode and GAS code environment.

In earlier chapters it was suggested that an Rcode optimiser could be produced but that certain problems existed:

a) Significant target code optimisation would still be required

b) Interpretation is very useful for optimisation, but construction of an Rcode interpreter will not be very easy

c) Optimisations such as common subexpression elimination will be represented by a directed graph but that if Rcode is written out to disk, the optimisation will be lost since the postorder external representation does not allow for directed graphs to be represented

d) Much of the information generated for optimisation is also useful for target code generation. An example is information on which definitions of variable values reach a given point in a program. Another example is whether at a given point, a variable with its current value could be used at some later stage. This is specifically useful for register allocation. This information is more useful if computed for all objects that will be involved in target machine code. The information generated for Rcode optimisation will often involve operations on non target machine objects (block operations and bitstrings), and will not involve all objects required at the machine level, for example temporaries used to break down block arithmetic or bitstring operations.

A second intermediate code offers reasonable compromises in solving these problems. All the optimisations that can be achieved in Rcode can be achieved in GAS code, plus additional optimisation can be achieved as such as copy propagation involving the extra temporaries that are created in GAS code, and elimination of branches to instructions that are themselves unconditional jumps. GAS code will be easier to interpret than

Rcode as it involves fewer operations. The information generated for optimisation will be more useful to the target code generator as it involves more of the actual target code objects involved, both because temporaries are created, and because arithmetic operations are broken down. However bitstrings are not broken down so in this respect not all target machine temporaries will be represented. Common subexpressions can be represented in external form as they are stored in temporaries. Though GAS code is a linear code, retaining the Rcode information improves the ease of optimisation. Data flow analysis and recognition of loops are much easier than having to discern this information from a purely linear code.

The price of optimisation at the GAS code level is that an optimiser will have to be written for each family of machines for which a GAS machine is defined. This is, of course, still much less effort than writing one for each target!

Optimisation will, however, still be required at the target machine code level as GAS code still will not represent all temporary data objects required in target code, and extra branches will be generated in target machine code that may themselves branch to instructions that involve unconditional branches. A description of additional target code optimisation needed is given in the chapter on target code generation.

GAS Code Optimisations

The GAS code optimizations consist of various global (performed across basic code blocks using data flow analysis and interpreter results) and local code block optimisations. Basic blocks are "straight line" pieces of code with one entry and one exit point.

Within these blocks it is known that the instructions will be performed. Some instructions could actually be performed in parallel via pipelining and instruction caches. Note that operand holdup has already been used in initial GAS code generation to allow for efficient addressing mode selection and limited instruction holdup is used to avoid some memory operations. The optimisations that are recommended to be performed on GAS code are:

Global

- Static evaluation
- Loop invariant code removal
- Induction variable elimination from loop
- Constant folding

Local Code Block

- Common subexpression elimination
- Instruction rearrangement
- Peep hole optimisation

All optimisations are performed on a procedure by procedure basis. This is limiting in that inter-procedural optimisation is ignored. The worst case is assumed that all "out" and outer scope variables are alive on exit from the procedure and must be copied to memory from registers at the end of the procedure. Local variables are obviuosly not live on exit. The "out" parameters are parameters such as "pass by reference". At the start of the procedure, no variables are assumed resident in registers. When a procedure call is encountered, all variables are copied from registers back into memory. Any temporaries in registers must be saved on the stack, but how this is done is explained later in

the target machine code generator notes. In languages such as Pascal and Modula, use of procedures is heavy hence inter-procedural motion analysis could conceivably produce useful code improvements, particularly for heavily used short procedures such as a "Get_Character" procedure. This could easily be added as an additional phase. However time limitations currently preclude implementation of inter-procedural flow analysis but its incorporation should be seriously considered in the near future.

Static evaluation

Whenever an assignment is performed (LOAD, ADD, etc), then an interpreter will be called to attempt to evaluate the code that produces the operands to be added. The interpreter will be given the Rcode tree pointer of the Rcode tree containing the GAS code that produces the source used in the assignment.

Whenever a conditional expression is used for a case structure, the interpreter will be called to attempt to evaluate the expression. The possibility of eliminating unreachable case options can then be detected and the code for the options eliminated.

The interpreter will be expected to return the computed value if static evaluation is possible. If a variable is assigned a static value a STORE instruction assigning the static value to the target variable is generated, and the subtree for the source is discarded. If the lexical level is global, and the variable has not been assigned a value previously, the gascode INITVAR is generated which will be used to generate a linker directive to initialise the storage allocated for the global variable to the static value computed.

Other Global Optimisations

The remaining global optimisations all depend on the results of code motion and data flow analysis. Code motion analysis identifies basic blocks that consist of code sequences that have one entry point and one exit point. Additionally the basic blocks that can immediately precede and succeed each block are identified. Within a code block, common subexpression and register allocation can be effectively performed but these are usually done after the global optimisations. The following descriptions of code motion and data flow analysis are based on ideas and algorithms by Aho and Ullman, but extended to account for the specific environment provided by Rcode with GAS code and to allow for the effects of type coercion in languages such as Modula-2.

An algorithm for detecting basic code blocks based on one by Aho and Ullman is as follows:

i) Identify leader statements, statements which start each basic block.

- The first statement lexically in a program unit or procedure is a leader statement
- Any statement which is the target of a conditional or unconditional jump, (or any labeled statement), is a leader.
- Any statement which follows a conditional jump or call is a leader.

ii) For each leader construct its basic block, which consists of the leader and all statements up to but not including the next leader, a RETURN or the lexical end of the program or routine. Any statement not placed be removed.

A flow graph can then be constructed which shows which basic code blocks can be predecessors and successors of a given block. A block B2 follows block B1 if:

i) There is a conditional or unconditional jump from the last statement of B1 to the first statement in B2.

ii) B2 immediately follows B1 lexically, and B1 ends in a call or conditional jump.

Also the lexically first block in the routine has no predecessors, and the basic blocks which terminate with a RETURN have no successors.

During the identification of basic blocks several useful extra functions can be performed. Set the "in" and "out" lists to NIL to indicate that variables have not yet been identified (after basic blocks have been identified, target machine generation may be performed which would try to use these "in" and "out" lists. The NIL value will be an indication that the lists do not exist. The identification of "variables" and "pointers" of interest can be performed.

After the identification of all basic blocks, collapsing of blocks with only one predecessor, into the predecessor can be performed. GAS codes that do not belong to any basic block can be deleted. Another pass through the basic blocks to perform common expression elimination can then be performed so that extra "useful variables" can be identified. Note that common subexpressions elimination cannot be performed until "variables" of interest (as opposed to Rcode storage areas and variables) have been identified.

The major task of data flow analysis can then begin. This analysis takes several forms. The major forms are du-chaining, ud-chaining and live variable analysis.

Reaching analysis (ud-chains) involves determining for each basic block for each variable, what definitions of variables are still valid on entry to a block, and which definitions are valid on exit from the block. Definitions are any statements which assign a value to a variable. This information is useful for such things as constant folding where a variable (A) has its value used at a point but it is noted from ud-analysis that the definition A := 2 is the only live definition of A at the instruction, then the value "2" can be used in place of the reference to A. The ud-chains also assist in detecting loop invariant code, particularly subexpression computation, copy propagation (e.g x=y z=x if x has no further uses, it is possible to generate z=y), and detection of induction variables.

Live variable analysis involves identifying for each basic code block, variables that are alive on entry and variables that are alive on exit. "Aliveness" of a variable means that the variable

may be used with its current value at some time latter in the code. A use can be made of this information in register allocation. If a register is required but all registers are currently in use, choose for a register that contains a variable value that is no longer alive. Live variables analysis can be achieved using the results of ud-chain analysis.

The du-chaining analysis involves computing for each basic block, for each variable, uses of the variable that could use the value of the variable as it is on entry to the block, and similarly on exit from the block. "Uses of a variable" are those GAS code instructions that have the variable as an operand. Hence for each variable there will be a list of identifiers of instructions that will use the variable's current value. The du chains can be used for assisting with the detection of loop invariant code, particularly subexpressions, from the body of a loop. It can also be used to assist with register allocation by providing information on which variables will be used soonest after a given point in a program.

Global common subexpression analysis would require that available expression data flow analysis be performed. This would require that each expression be assigned an id and expressions alive on entry and exit from each basic block computed. Available expressions analysis is similar to reaching analysis, except that for an expression to be available at the start of a block, it must be available at the end of each of its predecessors. An expression is killed by a basic block if any operands of its operands are defined. An expression is defined in a basic block only if it is computed in the block.

Identification of Variables

A major question is what constitutes a "variable" for this analysis. The Rcode will only contain definitions of storage areas which are variable or static areas. These storage areas may contain only a simple Rcode basictype object which are the basic operand type used in GAS code instructions (except for block operands used in memory move instructions and bitstring operands used in bitstring operations) or could contain an array or record structure. If these storage areas were the "variables" identified, the optimisations achievable would be limited, and register allocation would be severely limited in the information that it could use from data flow analysis. As Basictypes will correspond very closely to objects that will be stored in registers and will be the objects used in computations used in subexpression evaluations and loop control variables, these are the variables most advantageous to specify. In the following these are what are referred to as "variables". Rcode storage areas and variables are referred to together as "storage areas". Storage areas may contain several basictype variables. The problem is then how to identify basictype variables that may be located anywhere in a storage area.

The Rcode has provision to convey symbol table information, which would be very useful for the identification of variables. However the Rcode reference manual states that the backend will not expect to see these Rcodes. If Rcode is handed directly to the backend by the front end, the symbol table could be provided also. However, if the Rcode is written to disk the symbol table would not be available to the backend. For this reason the backend has been designed assuming no access to the symbol table. It has also been assumed that Rcodes for symbol table

definitions will not, as the Rcode manual states, normally be present in the Rcode. If Rcode symbol table definitions were contained in the Rcode, then the Rcode could be very long if many modules had been imported and symbol definition Rcodes for imported modules would be duplicated in any programs that also import those modules. An alternative would be to assume that an Rcode file only contains the symbol table definition Rcodes for the program excluding those from imported modules, and then the symbol files for all imported modules could be read to reconstruct the full symbol table. The front end will generate type definition Rcodes for debugging purposes, but only for current module. Additionally, all definitions will not necessarily be included. The pragma involved will specify which type definitions are to be included. Whether for optimisation the extra overhead of reading symbol files for all imported modules and handling the symbol table is worthwhile is not obvious. Perhaps at a later stage of this project an investigation can be made of the benefits obtainable. As a compromise, the specific inclusion of symbol table information for pointers would be of most value (see notes on pointers).

It is not necessary to have the full symbol table available to the backend. In fact the optimiser database structure (see notes below) is a useful form for optimisation. If symbol table information were present in the Rcode, it could be used purely to help build the variable and pointer database of the optimiser. A full symbol table is not necessary for optimisation. Full symbol table information can be useful. For example in Pascal pointers are guarrenteed to point only to objects of specified types. When an object pointed to by a pointer is changed, only those objects of the type the pointer can point to are considered likely to

change.

In the absence of symbol table information in the Rcode, basictype objects and pointers must be identified by an examination of the GAS code. When new Basictype objects are encountered, an entry should be made in the optimiser database defining the variable address and its datatype (obtained from the GAS code instruction in which the new variable is encountered).

Only those variables that can be clearly identified are useful in dataflow analysis. Such variables are temporary variables and those with static offset into a storage area. Those that are identified by a computed offset are not explicitly identifiable by the compiler, and the compiler will not be able to determine what actual piece of storage is involved although it may be possible to determine when the same variable is later referred to- with the emphasis on may. The computed offset is always computed into a temporary. If two variables have a computed offset in the same the temporary, and the datatypes are the same, then the two variables are actually the same (because in GAS code, temporaries are only ever defined once but may be used several times). Note that the offsets for two variables will only be computed in the same temporary if common subexpression elimination has been performed. This is because if Rcode refers to the same variable at a computed offset in two locations, the same subtree will be duplicated in both locations to compute the offset. In GAS code these subtrees will be converted to producing a computed offset in two different temporaries. If common subexpression analysis is performed, the same temporary will be used in each location.

When a variable identified by a dynamic offset is encountered during dataflow analysis, all variables in the storage area involved must then be considered potentially used or defined as appropriate. This is further complicated in languages that allow static type coercion, such as Modula. Static type coercion will allow variables to overlap!!! so that when one variable is altered, other variables may also be altered in a way that depends on the degree of overlap. Modula allows address manipulation and this provides another source of possible overlap of objects. As this address manipulation may involve dynamic computation, the effect of overlap would be impossible for the compiler to analyse. Static overlaps could however be analysed and the effects determined.

The definition of a variable in a storage area is an address and a datatype. In a language such as Modula 2 variables could start at the same address, but are of different datatype and possibly also of different size.

In data flow analysis and register allocation, these effects must be considered. Any consideration must be conservative, so that incorrect code will not be generated.

For definition reaching analysis, it is desired to compute all definitions of variables that may reach a given basic block. As stated above, this information is used for such optimisations as constant folding. If it is known that only one definition such as y:=10 can reach an instruction x:=y, then folding can be used to generate x:=10. If any other definition can reach the instruction then y:=x must be generated. It wouldn't matter if the exact form of the definition is unknown, all that is important is

whether another definition could reach the y:=x instruction. When a variable at a computed offset is used, a definition of all variables in the storage area should be considered generated but to an unknown value because the variable involved may be overlapped partially by some unknown amount. All current definitions of variables in the area should be considered killed. For example if x:=10 reaches a code block, but x:=20 occurs in the block before y:=x then the x:=10 can be considered "killed", but constant folding of y:=x to y:=20 can still be used. However if "x" is a variable in storage area two, and an assignment definition setting a variable to 20 is encountered for a variable in storage area two but at a dynamically computed offset, it is not known if "x" is affected. We therefore assume x:=10 is killed and a new but undefined definition of "x" is generated. This will prevent constant folding as the value of "x" cannot be computed by the compiler. The concept of an undefined definition is a useful way of handling the effects of uncertain overlap. The existence of one of these definitions will prevent erroneous code being generated. That is, they will stop optimisations such as constant folding from being performed when there is uncertainty of the validity of the optimisation.

For "Liveness" analysis, if a variable at a computed offset is defined, then consider "live" variables in the storage area, to remain "live". In liveness analysis it is hoped to find when a variable is clearly defined at some point because the variable is dead between the previous use and this definition, hence any old value in a register between the previous use and this definition is dead hence the register can be reused without saving the value in memory. In the case where a variable defined by a computed offset is defined, none of the variables in the storage area may

be defined for sure, so all live variables must be considered still live (hence the register contents will be saved if the register is reused). Additionally, when target machine code is being generated for the GAS code involved, any registers currently holding variables in the area will have to first be saved back to memory, and the register marked to indicate it no longer holds the variable value, because of possible partial overlap. This would ensure any reference to any variable in the storage area in a later instruction will use the value in memory which will be the correct value. If any values were left in registers, these values would no longer be correct if there were any overlap at all. If the computed offset for a variable is stored in the same temporary as the temporary used to compute the offset for the variable stored in the register, and the datatype is the same, then the register will not have to be dumped (this will allow useful register use for source code such as a[i+1]:=a[i+1]+1. If the variable is used, (as opposed to defined) consider all objects in the storage area become live. This is conservative in that all variables in registers will be saved if the register is reused, because the value involved may be touched later by this use. Again any registers containing values in the storage area should be first dumped to memory but the register is still marked to indicate it holds the value, in case of possible overlap. This will ensure that the value "used" will be correct.

For available expression analysis, an expression is considered no longer available when it is possible that one of its operands is changed. Therefore when a varible is defined at a computed offset all variables in the area could change so that any expressions in which any of these variables are used must be

considered no longer available.

In the presence of type coercion when any variable is defined, any registers containing variables that overlap should be dumped first, and marked as no longer containing the variable. When any of the overlapping variables are latter referenced, the memory values will be up to date and will have to be used as no register copy for any of them will exist. When a variable is used, any variables overlapping should be dumped first, but the registers still marked as containing the variables. This is true even if the variable involved is identified with a static offset.

Coercion makes it necessary to know which variables overlap each other. This can be achieved by a bitstring (see below) for each variable to represent any variable that it overlaps. For variables identified by computed offset, it would have to be assumed all other variables in the storage area are overlapped. When a variable is used or defined, all variables it overlaps must be treated as being referenced, or defined with an unknown value. The same approach could be taken for variables identified with a static offset. However this is not as accurate as possible. To improve on this approach, the exact effect on the variables overlapped could be computed. These computations would involve determining which variables are overlapped and dumping only these variables if in registers. A bitstring representing variables overlapped (see below) associated with each statically identified variable would be an efficient method of implementation. In initial implementations of this compiler, the simpler approach will be used.

As variables are only identified as code is processed, it makes

sense to identify useful variables during the pass of the code used for Basic Code Block analysis.

Bitstrings in the optimiser

During dataflow analysis, it is often necessary at the start and end of basic code blocks to identify the state of all variables, definition instructions and use instructions important to the optimiser at that point in relation to a particular property that is boolean in nature (i.e. the liveness of each variable). An economical way to achieve this is to use a bitstring where each bit represents the state of the property for each variable, definition or use important to the optimiser. Each variable, definition or use important to the optimiser has a bit position allocated in these "optimizer bitstrings". For example a bitstring is used to represent liveness of variables at the start of basic blocks of code typically may be:

101110011

i.e variable one is live, variable two isn't etc

The bitstring will be used in logical OR and AND operations many times, hence it must be efficiently implemented. Bitstrings will consist of a linked list of bitstring pieces. Each piece is of size of the most convenient object size for manipulating bitstrings on the machine the compiler is running on. Various bitstrings recorded for a basic code block will be pointed to from the basic code block optimiser database records and additionally variable bitstrings will be maintained for each storage area of variables identified that are located in that

storage area. An array of such pointers will be kept for each of variable, global, and imported storage areas for each lexical level.

The optimiser database record for an identified variable should include the following:

a) Optimiser assigned variable id
   (also serves as optimiser variable
   bitstring bit offset)
b) Rcode storage area id
   (area id or
   variable id or
   temporary id)
c) Offset (must be static value or temporary)
d) Type of variable (Rcode Basictype)
e) Pointer to overlap bitstring
f) Link to next variable in storage area
g) Used as pointer (see later)
h) Reference count in routine
i) Liveness
j) Next use GAS code instruction

The (i) and (j) fields are used in next-use analysis during target machine code generation. See the chapter on target machine code generation.

Access to variables records is required as follows:

a) Given a storage area id, offset and basictype,
   locate the variable record

b) Given one variable in a storage area locate variables records for all other variables in the area

c) Given variable id locate the record for the variable

The first access requirement is obtained by accessing the GAS code entity database given storage area id, obtaining the variable id of the the first variable in the storage area, then using the link to the next variable in the storage area, find the variable record that matches the offset and basictype.

The link to the next variable in a storage area is used for the construction of a linked list of variables belonging to an area. The link list will be maintained as a ring. The start of the ring will be pointed to from the GAS code entity database record for the storage area which will contain the variable id of the start of the ring. The ring allows the second access requirement to be provided.

The pointers to all currently active variable definitions will be kept in an array indexed by the variables id. This allows the third access requirement to be provided.

GAS code statements that define and use variables must also be identified. Bitstrings will be used but statements of interest to the optimiser must be assigned an id by the optimiser. An array of pointers to instructions of interest will be maintained. The array will be indexed by the statement ids.

If available expression analysis is performed, bitstrings will be maintained to represent whether expressions are alive or not at the start and end of basic blocks.

Pointers and their effect on data flow analysis

Use of pointers complicates dataflow analysis. When an object that is pointed to by a pointer is altered, all variables could be assumed to change. This is on the conservative side. Analysis of the effects of pointers on reaching analysis and live variable analysis is very useful for optimisation in languages such as Pascal and Modula because they are heavy in the use of pointers. Also any languages that allow call by reference parameters will benefit from analysis of pointers. However such an analysis can be very difficult paticularly in a language such as Modula that allows coercion, which would allow overlap of variables and arithmetic of any desired form on pointers. The problems covered above involved in handling objects at computed offsets, and in the presence of possible overlapping of variables is made significantly worse in the presence of pointers. Pointer analysis also has a high overhead because for each basic block, for each pointer, a list of variables that may be pointed to on entry, and exit, must be maintained. The analysis of pointers as discussed below is based on the ideas of Aho and Ullman [24], but has been extended to include identification of pointers, and coercion and arbitary arithmetic on pointer contents. In particular, the concept of a pointer "touching" a variable as opposed to pointing to a variable is introduced. When a pointer "touches" a variable, the pointer will not point explicitly at the variable, but may point to an object that overlaps the variable. This concept is important in reaching analysis, in that

a definition statement may touch a variable, hence altering it's value, but it can't be inferred exactly how the value of the variable will alter, which we could have done if the variable specifically pointed to the variable. This would be important, for example, where two definitions reached a point, but they both set the variable to 2. It would seem appropriate to perform constant folding, however, if one of the definitions only touched rather than pointed explicitly to the variable, this cannot be done. Again the concept of definition to an undefined value becomes important.

Pointer analysis is made even more difficult in Rcode or GAS code because pointer type variables are not identified types. In fact the only objects declared are storage areas for which size and alignment is given. Aho, Sethi and Ullman [9] express the opinion that pointer analysis should only depend on use of objects which are clearly defined as pointers, and for source languages in which assignments to pointers are guarranteed to only provide valid pointer values. In a language such as Modula, this is in no way guarranteed because of coercion. Also because of coercion, when an object pointed to is used or defined, all variables in the storage area containing the object pointed to should be considered possibly used or defined indirectly. This is unless the compiler performs accurate variable overlap computation as mentioned above.

The above problems in a language such as Modula make pointer analysis of debatable value, but it is planned to be implemented. The user must however have to explicitly request pointer analysis, and this must be clear in the project documentation.

In the absence of overlap calculations, pointers are only worth handling on a storage area basis. For a pointer, it is only required to know which storage areas the pointer may point to. When the object pointed to is defined or used, all variables in any storage areas that may be pointed to will be considered used or defined, because the variable pointed to may overlap any other variables in the area. Note that if one of the variables in a storage area so used is a pointer, this pointer will then be considered to possibly point to all variables. This will in fact also be the case whenever any variable in a storage area is altered; all pointers in that area must be considered to be able to point to all variables, from that point on.

Pointers can be identified whenever a part of a storage area or temporary is used to store an address value. That part of the storage area involved must be identified by a static offset into the variable or by a computed offset into a storage area. If the pointer value is stored at a computed offset into a variable area (which occurs when an array exists whose elements contains pointers), it will not be possible to clearly identify when this pointer value is used later. However, as the the behaviour of such pointers cannot be analysed by the compiler because it can never always be certain which pointer is being referenced. Whenever a pointer value located at a computed offset into a variable is used to reference a variable, all variables (and pointers are also variables) will have to be considered possibly used or defined because it will not be known what actual object is being referenced because it cannot be computed by the compiler when the contents of this pointer was set. It would be impossible to determine even the Rcode storage area that is pointed to. The approach to pointers must be conservative, and the conservative

approach when unsure of a pointer's exact contents is to assume that all variables may be potentially defined or used. Consider the pointer to possibly "touch" all variables, touch in the sense that all variables may not be pointed to exactly, but they may overlap with the variable pointed to. If certain of the variable pointed to, all variables in the storage area of the object pointed to must be considered "touched" because of possible overlap with the object pointed to. The conservative treatment in data flow analysis of pointers that may point to several objects is described below.

The term "touched" is used in the descriptions below to define all variables that the compiler considers may be pointed to or possibly overlapped by the objects that may currently be pointed to.

In reaching analysis when the object pointed to is defined, all variables that may be touched are considered defined by an unknown value (unknown because the overlap involved is unknown because of the possibility of coercion), unless it is known by the compiler that a specific variable is pointed to directly in which case the variable will be defined to the value generated in the instruction. Any current definitions of the variables touched should be considered killed. If the pointer explicitly points to any variables then this variable will be considered defined by the value generated in the instruction.

In liveness analysis, consider first the case where the variable pointed to is defined. Then consider no change; all live variables remain live. Now consider when the variable pointed to is used. Consider all variables in storage areas that may be

pointed to to become live. As any variable in any storage area that can be pointed to can be involved this is conservative and is equivalent to saying "immediately preceding this instruction, save the contents of any register that contains the contents of a variable in any storage area that may be pointed to because this statement may refer to a part of that variable".

If an address is loaded into a location that is not clearly identified, then consider all pointers in the storage areas that may contain the pointer to now be able to point to all variables. The variable whose address is involved cannot just be added to the list of variables that the pointers can point to, because possible overlap of pointers means that the effect on pointers that have been identified is unclear. This produces the same effect as loading any value, not just an address value, into a variable that can't be clearly identified.

The next problem in pointer analysis involves specifying what is pointed to by the pointer. The contents may be unknown or undefined. If unknown it means that the pointer may have been assigned a value but nothing is known of the value. In this case, the pointer must be assumed to possibly point to all variables, or storage areas. Undefined means that the pointer has not been assigned a value. A use of a pointer with unknown value will produce a compiler warning message. The contents of a pointer will be represented as follows:

> a bitstring indicating variables that may be "touched", in that the object pointed to may overlap an object that is touched, but is not pointed to directly

AND

a bitstring indicating variables that may be explicitly pointed to

The AND represents the fact that on some paths into a basic block, the contents of a pointer may have been defined to point explicitly to variables and touch any other variables in the same storage areas as these variables, whereas other paths may only have determined that the pointer may touch certain variables.

The following is how pointers will be analysed to determine what pointers may point at, and what they may touch. The analysis is based on the the algorithm of Aho and Ullman [24].

a) If there is an assignment to a pointer where the source GAS code operand is "addressof x" then the pointer will point to "x" explicitly and may touch any other variable in the same storage area as "x".

b) If a pointer "a" is assigned a value which is the contents of another pointer "b", then pointer "a" can only point to the variables that "b" can point to and can only touch the variables that "b" can touch.

c) If a pointer "a" is assigned a value which results from a computation which involves adding a constant value to the address for the start of a storage area, and the result will not go outside the area, then assume the pointer may point to any variable in the area. If the result would go outside the area, assume all variables may be indicated.

d) If a variable, including any pointer, in the same area as a pointer is changed, then the pointer must be considered able to touch to any variable, but not point to any. This is on the conservative side.

e) If the pointer is assigned a computed value other than as specified in (d), assume the pointer can then point to any variable.

When analysing pointers across code blocks, any pointer on entry to a block should be considered to possibly point to the union of what the pointer may point to on exit from all predecessor blocks.

Loop Optimisation

Code loop optimisation involves global code motion analysis on the code comprising the loop, using the information on reaching analysis. However most loop optimisations will be invalid using these techniques if the code motion flow graph (the graph representing the possible motion of basic blocks) for the procedure is not reducible. To be reducible essentially means that any loop must have only one entry point. Languages such as Modula-2 are inherently reducible, but languages such as Fortran and Pascal with GOTO's can allow programs with loops that can be entered in more than one place. However structured programming also results in reducible flow graphs. Additionally loops with multiple entry points will not be common because the logic associated with them is tricky and usually avoided. However, the programmer should have the ability to suppress such loop optimisations during compilation, and there should be clear warning in the documentation about the need for this suppression

when loops with more than one entry point are used.

If loop analysis is to be performed, the Rcode structure makes loops easy to identify. The "Loop" Rcode clearly identifies loops (except those implemented using GOTOs but if the graph is reducible, this just means that some loops are missed). This greatly simplifies the process needed to find loops as described by Aho and Ullman for three address intermediate code. When loop analysis is performed, for each loop identified, a database record will be produced that contains:

Loopid
Contained loops
Contained basic code blocks

The loopid is assigned by the optimiser. The contained loops are loops contained within the loop (sub-loops). These are easily detected in the Rcode by the nested structure clearly represented by the Rcode. Note that a reducible flow graph cannot have overlapping loops. Contained basic code blocks are those code basic blocks in the loop that are not contained inside sub-loops.

Loop identification will be useful in register allocation. Emphasis will be placed on allocating to registers, variables heavily used within an innermost loop, for the duration of that loop. The level of use of each variable is obtained by computing usage counts for variables in the basic blocks of the loop (see under target machine code generation).

If loop analysis is not to be performed, loops will not be identified. Latter stages that may use loop analysis results will

operate quite happily as they will merely find that no loops have been identified.

Loop optimisations include moving loop invariant code out to the start of the loop and creating a preamble to the loop which is only executed once on entry to the loop. There will not be the problem that can occur in some intermediate languages in that instructions are removed from the loop that are the target of jumps, because jumps refer to points defined by the USEVAR Gascode instruction, which will not be removed from the loop. A second optimisation will involve identification of induction variables and replacememt of their computation by addition and subtraction rather than multiplication and division (strength reduction) or even complete elimination of the variable if only involved in tests. The algorithms for loop optimisations are well known and available from various sources (see bibliography).

Local Code Block Optimisation

Local code block optimisation is performed after global data flow based optimisations. Typical local code block optimisations include common subexpression elimination, re-ordering of instructions and peephole optimisation such as elimination of unnecessary assignment statements.

Common subexpression analysis can be achieved easily within a local code block by keeping an "event number" which is incremented each time an assignment is encountered. At the start of the basic block, this event number is set to zero. The symbol table descriptor of each variable records the event number of the last assignment that altered its value. The "event number" of the expression is the largest event number of the variables within

it, when the expression is computed. When the expression is needed again, it's event number is recomputed and compared to its event number when previously computed. If changed, code to recompute the expression value will be generated. Several alternative well known algorithms can be used. Note that common subexpression elimination is performed on basic blocks during the identification of basic blocks, because it is easy, and this will hopefully allow more variables to be identified as being referred to in more places by allowing recognition of common computed offsets for variables.

Peephole optimization involves the following:

* Redundant assignment elimination

```
a=b c=a a=z --> c=b a=z
a=b b=a     --> a=b
```

* Unreachable code elimination

  If first statement in basic code block
  is unlabelled and follows immediately
  an unconditional branch statement
  then it can be eliminated

* Elimination of ADD A,0,A MULT A,1,A

It has been decided that initially common expression elimination will not be performed globally because empirical studies show that little benefit is gained as out-lined by Anklam et al [7]. Because of this decision, temporaries will not survive beyond a basic block.

Only local code block common expression elimination produces useful benefits. A global common expression elimination phase could be added to the optimiser. Temporaries would then exist beyond basic block boundaries.

Each of the steps of the optimiser consists of a subroutine. Code motion and data flow analysis and the various optimisations are implemented as subroutines that operate on the Rcode tree for the routine and use the entity database and optimiser database. This makes it easy for any particular optimisation to be omitted or for the optimisation phases to be omitted entirely.

The code motion and data flow analysis is the necessary first stage of the GAS code optimiser and generates the optimiser database which contains information on basic code blocks and for each basic code block, and the results of du, ud and live variable analysis. Loop information may also be generated. The database information is used by the target machine generator for register allocation and assignment. However if the generator detects that no optimisation and flow analysis has been done, it will allocate registers on a very local basis. This gives the option of producing code very quickly, but the code will be terribly bad!!!!

Representing Optimiser Database Information

The optimiser database is the fundumental basis of the optimisation operation and may be used by the target machine code generator. A vital property of the nature of the database is that it should allow as much independence between various parts of the optimiser and target code generator as possible. For example, if

loop optimisations result in moving instructions around, this should not affect liveness analysis or reaching analysis. Whether loops have been identified or not, or live variable analysis done should not upset the target machine generator; only the effectiveness of code generation should be involved. Design of the optimiser database is therefore vital.

Basic blocks, variables and GAS code database entries will all be assigned individual "ids". This is for use in bitstrings as defined elsewhere, but also allows that one of the objects may easily be deleted. If a GAS code is deleted, a later direct reference to it by a pointer will cause an error. If any reference is via an "id", the GAS code can only be accessed via an optimiser database. This will reveal that the GAS code for the "id" has been deleted.

There will be a separate database structure for each basic block. A list of all basic blocks in a routine, which will include a pointer to each basic block's database will be kept in an array in lexical order of blocks. The offset of a pointer in this array is the id of the basic block pointed to. Each basic block database will include the id of each GAS code instruction in the block kept in motion sequence, live variables on entry and exit (bitstring linked list pointer), live variable definitions on entry and exit, live uses on entry and exit (bitstring) and all predecessor and successor basic block ids, and the lexically successor basic block id. Bitstrings are also pointed to that represent definition instructions that are killed, definition instructions that are generated in the block, variables that are used before they are defined in the block, and variables which are defined before they are used. The first two bitstrings are

useful in reaching analysis, and the second two are useful in liveness analysis. When processing of the current routine is complete, all information relating to basic blocks in the routine will be removed. No allowance has been made in the database for recording information on available expressions, as common subexpression is not being performed across basic code blocks, as described earlier. It is quite likely this will be added later.

A database of all variables identified by the optimiser will also be maintained, as described earlier. An array of pointers to all variable structures will be maintained. This array will be indexed by variable ids. When processing of the routine is complete, all entries for variables will be removed. Therefore a new variable database is created for each routine. Code for procedures will be encountered in post-order form so that inner procedures will be encountered first. Variables for outer procedures will not be identified first and therefore will not fill the variable index first. Variables from various lexical levels will be intermingled in the variable array. Hence merely removing variables for the current lexical level at the end of a routine will not be straight forward. In the version of the backend being developed this will not be done. All variable entries are deleted, and a variable database is built for each routine. This approach will not be sufficient if interprocedural data flow analysis is performed in later implementations. However an advantage is that the variable database will only be as large as the number of variables used by a routine and its lexically enclosing routines. This will save significant space in the data flow analysis.

An array of pointers to all GAS code instructions in the routine

being processed, that involve a definition of a variable will be maintained. This array will be indexed by the instruction id assigned to GAS code involved. Note that GAS code instructions that do not involve definitions or uses of interest will not be assigned an id, so that bitstrings are kept as small as possible. When the processing of the current routine is complete, the array is purged.

A database structure will be maintained for all loops identified as described earlier. An array of pointers to all loop structures will be maintained, with the offset for each loop pointer being the id for that loop. When processing of the current routine is complete, the loop database is purged.

It must be remembered also that the GAS code entity database will also be present. When the current lexical level is left (the processing of the routine has finished), the GAS code entity database will only be purged of entries relating to the current lexical level.

GAS code and Basic Blocks

When basic blocks are formed, an optimiser database is formed which contains for each basic block, a pointer to the GAS code instructions for the basic block. The instructions in a basic block, except for the last instruction, will be any GAS code except a conditional branch, return from subroutine, or call. This will allow USEVAR and USELABEL Rcodes to appear in basic blocks.

During optimisation, GAS codes may be moved around or deleted, but after optimisation it must be possible to write the modified

Rcode back to disk in post-order form. This would seem to involve patching the Rcode tree as changes are made to the GAS code. However, a GAS code BLOCK has been defined, which is only used by the optimiser, and target machine code generator, and which has the field block-id. When a basic block is identified, the GAS codes involved are removed from the Rcode tree, and the GAS code Rcode BLOCK is grafted to the tree in their place. When the Rcode tree is written to disk, and the BLOCK GAS code is encountered, the GAS codes for the block will be accessed via the optimiser database and written with appropriate Link Rcodes to bind them together. Before code generation can be performed, the Rcode file will need to be read, an Rcode tree and entity database regenerated, basic code blocks identified and the optimiser database created. If improved target machine code is to generated, data flow analysis will have to be performed to generate the live variable "in" and "out" lists again. Writing optimised GAS code to disk will therefore have a significant overhead. However it may be necessary because of restrictions on memory size, or to allow later interpretation of GAS code, or to allow target machine code generators to use the same GAS code.

To conserve space, target operands will be stripped off GAS code instructions. This "original" target operand will represent a "variable" which is loaded into the database, unless this has already been done. The database entry will point to the target operand stripped from the GAS code. The GAS code will be modified to point to a target operand of a special "Target_Id" variety which contains the fields:

target_type

indirection

targetid

the targetid field contains the unique id assigned to the variable by the optimiser. This approach conserves space, and also allows fast efficient access to the variable database when processing instructions later to produce generate and kill lists (see above under reaching analysis), use and definition sets (see above under liveness analysis), and next use analysis (see chapter on target code generation). Having the "id" of variables in GAS code instructions will be most useful. If optimised GAS code is written to disk, operands must be written in full, so the "target id" of targets must be used to access the full operand in the database so that the operand can be written in full.

Chapter Six

Target Machine Code Generator

Emphasis in the code generator design has been placed on producing a code generator that is as independent of the target machine as possible, but which must produce efficient target code. For the prototype, the functions required of the code generator include:

a) Converting GAS code addressing modes into target machine addressing modes. This involves manipulation of pointer and index registers and will often involve general register allocation and assignment strategies.

b) Converting bitstring operations into operations of the target machine.

c) Mapping operations on basictype objects in GAS code into operations of size supported by the target machine. For GAS code, objects are chosen to be as close as possible to target machine types, but as already stated, the GAS code generator may not necessarily be able to exactly specify target machine datatypes.

d) Convert arithmetic operations which are represented three address operations into appropriate target machine operations, which are perhaps two or three address operations.

e) Allocate and assign variables to registers. Allocation

involves deciding which variables should be assigned to registers, and assignment involves assigning actual registers to these variables. How independent these two operations are depends to a large extent on the orthogonality of the target machine.

f) Map GAS branch instructions to target machine instructions. At this stage use may be made of Rcode control codes still present to recognise when special target machine instructions such as LOOP can be used. Fixup of target addresses will be required. This requires information on the size of target machine code instructions so that offset of branch target instructions is known.

g) Map the GAS subroutine calling convention to target machine facilities. This will involve manipulation of target machine stack pointer registers, and possible use of target machine special subroutine calling instructions, but only, of course, if these are compatible with the standard GAS calling convention. The use of dope vectors to handle dynamically sized local variables and results will also be required. Strategies for saving register contents at call time are also needed.

h) Any GAS SVC instructions must be implemented by appropriate generation of thunks or calls to the kernel.

i) Machine and operating system specific Rcodes which still remain must be implemented, again either by generation of thunks or calls to target machine libraries.

The target machine code generator is a subroutine that receives as input an Rcode tree for a routine, a GAS entity database and

the optimiser database. This generator will expect as a minimum that Basic blocks have been identified, "variables" have been identified and that live variable, use and definition, and "in" and "out" lists for basic code blocks have been set to initial NIL values (as before any data flow analysis is performed). Hopefully full dataflow analysis information is available as well.

Target code generation is still being performed with the Rcode structure available as GAS code is attached to the Rcode tree. The Rcode tree structure provides useful program structure information. For example it is clear when code is being generated to allocate space for local variables of a routine, or when the conditional expression for a choice structure is being generated.

Target machine code will be generated directly from some Rcodes. This specifically involves group six machine specific Rcodes, which are passed untouched through to the target code generator.

The generator is itself very important in terms of code optimisation. The GAS code optimiser performed general logical optimisations. The target machine generator is involved in code improvement based on efficient register usage, effective instruction selection and operand addressing mode selection. These the code improvement strategies are target machine specific.

It is obvious that there is still significant work to be achieved by the target machine code generator. To enhance portability, the aim has been to make the algorithms involved as target machine independent as possible, with target machine dependency limited

to as few as possible clearly defined constants and procedures. In fact it has been found that such a clean solution has not been practical, and instead "fluid" abstracted procedures, constants and associated database have been developed that are ported to each new target machine in a generic family. Hopefully the porting only involves changes in implementation but in practice some changes to the definition of procedures and the database will be expected, hence the term "fluid". The more effective the abstraction, the less the number of definition changes required. The concept of generic families allows that "fluid" procedures, constants and database will be developed with abstraction generalised with a family in mind. This has been found to be much more practical that abstracting for as many machines as possible as the the abstractions become so general they are not very useful. The GAS family definition provides a clear definition for a group of machines, and provides a clear basis for the abstraction.

In some code generators, the approach that has been taken of representing relevant target machine and operating system characteristics by tables of information. A totally machine independent program performs target code generation using the table of information when requiring information on target machine and operating system characteristics (see chapter two). Tables are adequate when there are several independent target machine characteristics for which the target code generator will require information. However the characteristics involved are usually related. It is difficult to satisfactorally represent such relationships in a table. How, for example, can the following be represented; not only can a certain register be used to store 32 bit values for integer addition, but the register is also the

only one that can be used for storing a bitstring size value? a two address addition instruction may be provided, but only if one operand is located in a register, perhaps a specific register. These pieces of information may only be relevant for a single machine. The table structure becomes too complex, and must be designed to represent any information relevant in code generation on any of the target machines in the family. The code generator must take account of these characteristics; it cannot ignore them. The table and code generator become a superset of all the target machines. The family concept represented by the GAS machine helps to reduce considerably the complexity required as opposed to a table structure and code generator for all target machines, and makes a table based approach more practical. The characteristics relevant for machines in one family are likely to be very similar.

The approach which has been attempted in this design, is to produce a target code generator whose structure is based on an orthogonal machine with the characteristics of the generic family, and modify the structure for the non-orthogonal irregularities of the particular target machine. This approach seems to be successful because machines in a family have similar architectures and differences between them tend to be localised. Significant architectural features have much greater effect on the code generator form. Within the family there can be many minor differences which modify the general algorithms in a straight-forward way such as providing only two address instructions, and the result of these instructions must be a register, or restricting registers for operands of a division instruction to using specific registers. However the task of selecting registers is still the main issue, it is only the local

details that vary. The target machine family concept therefore has benefit through into the target code generation phase. A generalised target code generator for all machines is not practical, but one for a family of machines does seem practical. A general target code generator for all machines would involve variations that included major architectural features. The database that characterises the machine and represents the state of allocation of machine resources could be quite different for many machines. A single database that allowed representation of all these factors would be huge. Within a family, variations only relate to variations on the central architectural features that characterise the family. A single database that can be used to characterise all the machines in one family and the state of allocation of resources is of a more practical size.

The prototype code generator for the VAX machine is based on producing a basic code generator for a machine which is orthogonal in the sense that any operands of any given instruction can be of different sizes and can be located in memory or any one of a set of registers, and for any three address GAS instructions, two and three address target instructions are available. Additionally any registers can be used as pointer or index registers. Any further non-orthogonality will have to be catered for by target machine specific code included where required. An example is where a two address instruction would be the most efficient approach, but the target machine only provides two address instructions where the operand that is overwritten must be located in a register, it cannot be in memory. An example of such an instruction would be a=a+y, where "a" is not used before any value currently located in a register and therefore "a" will not win a register. When the

code generator for the orthogonal machine will call for a two address instruction to be generated with both operands in memory, instead a decision will have to be made to generate a three address instruction with operands in memory if one is available or forcing the dumping of a register or a decision may be made to dump a register anyway. These changes in the code generator should be clearly marked. It should be possible to easily generate a target code generator for one machine using an existing target code generator for a machine with similar non orthogonalities. For example, many machines will have a non-orthogonality in which the result of the two address instruction must be a register. A modification to the prototype generator to handle this will produce a target code generator that will be easy to modify for all target machines with this non-orthogonality. For machines that are reasonably orthogonal, the prototype code generator for a family of machines will easily provide the target code generator. Their code generators will also not have the overhead of many of the tests required to test for the presence of non-orthogonalities that would be required if a single general table driven target code generator was used.

The elements of the target code generators for all machines in the VAX family that will essentially be common include:

a) Register allocation, which involves use of live variable analysis and next use information, plus accessing whether variables are currently located in registers. Register assignment will be target machine specific.

b) Building the subroutine calls

c) Implementation of control structures primarily by conditional branch instructions.

c) Branch fixup, the general machanism is not target machine dependent.

Elements which could vary significantly from machine to machine include:

a) Bitstring manipulation

b) Addressing mode selection

c) Register dumping on subroutine calling

d) Register assignment

e) Building instruction bit patterns

The VAX prototype target code generator will be non-orthogonal in that all operands of arithmetic instructions must be of the same size, except for immediate values. Additionally, registers 12 to 15 will be reserved as pointers as described in the VAX Architecture Handbook. A few instructions require use of specified registers, or register pairs. These will be handled on the local level at which they occur. For many VAX instructions, there are restrictions on the allowable formats for some or all operands. The datatype of an operation is usually reflected in the instruction code.

As discussed above, abstraction of the target code generation

process is the key to portability. Consider in more detail what this abstractions involves.

As mentioned, database will be maintained which represents the characteristics and current state of allocation of the machine resources. To allow the code of the compiler to be as modifiable for other machines as possible, it is desirable that the database be abstracted as much as possible. The actual database structure is therefore hidden, and procedures provide access. For example, for register allocation it will be necessary to know if a variable is currently stored in a register. This is not achieved by directly accessing the database records as this requires specific knowledge of the record structures which will vary from machine to machine. Instead a procedure such as

```
In_Register(Variable_Id)
```

is provided which returns the register (or None) used. Other examples include:

```
Allocate_Specific_Register(Reg_Id,Variable_Id)

Allocate_Register(Variable_Id)

Deallocate_Register(Register_Id)

Has_Two_Address(Gas_Code,Datatype)
```

Abstraction requires that careful consideration be given to what procedures are required generally across a range of machines. These procedures may be changed from one machine to another

because of non-orthogonality. However the required set of procedures will be similar for machines with similar non-orthogonalities. These abstracted procedures for accessing the database will therefore be important in achieving portability of the target code generators. They are discussed in more detail later in this chapter.

Such fuctions as handling of operands, requests for generation of target code, and the algorithms involved in such things as register allocation, handling of declarations and subroutine call handling should also be handled in an abstracted manner. This is discussed later but will lead to code such as:

```
IF In_Register(X) <> No_Register AND
   Used_Before(Y,X)                    THEN
     Set_Location(Y,Location(X))
     Deallocate_Register(X)
END
```

In this abstracted code, the procedures In_Register, Set_Location and Deallocate_Register will access a target machine resources database, but the details are hidden from the code above. The Used_Before procedure will access the optimiser database. Note that "X" and "Y" are variables relevant to the target code generator and are not Rcode defined variables.

Other information can be made available through procedures that access the optimiser database, and information about the target machine is contained in the PLIP "Machines" module.

If the functions of the target code generator are to be

abstracted and the abstracted approach used for developing a target code generator for one machine ported to many other machines in the family, it is necessary the the approaches to generator functions be well thought out, but that the choice of approach will be significantly influenced by the ease of porting the approach to other machines in the family. If a differing approach to a function is considered more effective on a specific machine, someone could implement this approach at some later time. The following sections discuss most of the central functions of target code generators and suggest approaches that will be taken to produce efficient code but still remain easily portable to a range of machines.

Storage Allocation

The first step will involve calling the storage allocator routine for any declarations encountered in the routine (higher lexical level declarations will have already been encountered and processed). If the current lexical level is the outer most level, then the storage involved is static and linker directives must be generated. As a portable linker is used, the generation of linker directives is independent of target machine. Handling of Rcode global storage area and "variable" declarations is not significantly different, except for the generation of linker directives for global storage areas. Note that Rcode variable areas may contain several variables significant to target code generation (see also optimiser chapter). GAS Variables in a global storage area will be coded with the address field set to the offset of the object into the storage area. A linker directive will be generated to relocate this operand by the address the linker assigns to the storage area. Storage allocation for static areas is therefore not performed by the

target code generator.

The allocator will update the database entries for Rcode variable storage areas, to record the byte offset of the area (if staticly sized) or its dope vector (if dynamically sized) from the the byte on the stack pointed to by the stack pointer that will be used to access local variables. Whether a variable storage area is statically sized is determined by static evaluation by the GAS code interpreter of the subtree that determines the size of the object. If the value returned is static, then the size subtree is deleted and the variable is allocated space at a static offset into the routine activation record. The entity record for the object is updated to indicate no dope vector. If not statically sized, target machine code will be generated for the size subtree (this code will not be emitted yet), with the result to be placed in a dope vector. The entity record for the object will be updated to record that a dope vector is used. The machine module will contain information on the size and required alignment for dope vectors. Note that space for these storage areas may not necessarily be allocated in the order declarations are encountered as alignment requirements may make it possible to use less total space by using a different ordering. Database entries for GAS code variables located in variable storage areas will be updated to record the byte offsets of the variables into the area.

The Rcode tree that specifies the size of the variable now contain GAS code that terminates with a PUSH GAS code. The operand of this push can be used to determine clearly if the variable is statically or dynamically sized. This PUSH GAS code will normally be converted into an instruction to increment the

stack pointer.

After all declarations for a routine have been processed, the space consumed by all statically sized dynamic objects and dope vectors will be known and code to extend the stack pointer by this amount is generated. The code generated earlier to compute the size of each dynamically sized object is emitted, and code is generated to advance the stack pointer by this computed size, and to store the computed byte offset of the object into the routine invocation record in the dope vector for the object. References to objects in the area will require generation of code that accesses the object via the dope vector for the object, and at the byte offset of the object into the area.

For global storage areas, all storage areas are statically sized. A linker directive will be emitted for each area which specifies the alignment of area, size (which is always static), area protection, explanatory text, and the area id. Whenever references are made to a global object, the address will be an offset into the area, and a linker directive specifying the relocation of this address by the start address of the area will be emitted. For constants the expression subtree of the Declare_Constant Rcode that specifies the constant value is passed to the GAS code interpreter which should return a static value. Code to push an object of this size and alignment on the stack will be generated. The entity database entry for the constant will be updated to include the address assigned to the the constant, and its size. It is assumed that constants of a word or less will be handled as immediate values directly in the code, although this can vary depending on the level of use of the value and the effectiveness of the immediate addressing mode on

the target machine. Constants declared in a routine will still be allocated storage globally. Linker directives will be emitted for constants specifying a read only initialised storage area. Global initialised storage areas are defined by using the Append_Area Rcode. This Rcode specifies statically sized extensions to global storage areas with initial literal values specified. When encountered the value subtree of the Rcode is handed to the GAS code interpreter to evaluate the initial value and then a linker directive is emitted that specifies an extension to a storage area, and an initial value. Additionally, if an INITVAR GAS code is encountered, a linker directive to initialise part of a storage area with the specified statically computed value is emitted. This code is generated when the GAS interpreter determines by static evaluation that a part of a global storage area has an initial value.

The output of the target machine code generator will be suitable for input to the portable linker. This will mean that any instructions that refer to global objects or constants will have linker relocation directives appended to specify an offset from the start address of the relevant storage area. For all global storage areas and routines, linker directives defining them are generated which can be used for resolving external references to them in other modules. For references to global storage areas and routines in other modules, linker directives are generated for such references specifying relocation of references by the addresses assigned to the areas/routines identified by module id and area-id/routine-id.

Handling operating system interface and intrinsics

Calls to the operating system libraries, and kernel calls will

need to be generated. These may be required to implement arithmetic functions such as real number operations, to implement SVC GAS codes, and direct references to target system dependent relocatable object language symbols. In each case linker directives must be generated that identify the target symbol. GAS code SVC calls such as saving and loading context may be implemented directly by generation of thunks or by calls to operating system a software interrupt instruction of some form, or call to target system dependent relocatable object. The requirements for operating system calls and use of host intrinsics will be known to the target code generator and will be handled accordingly.

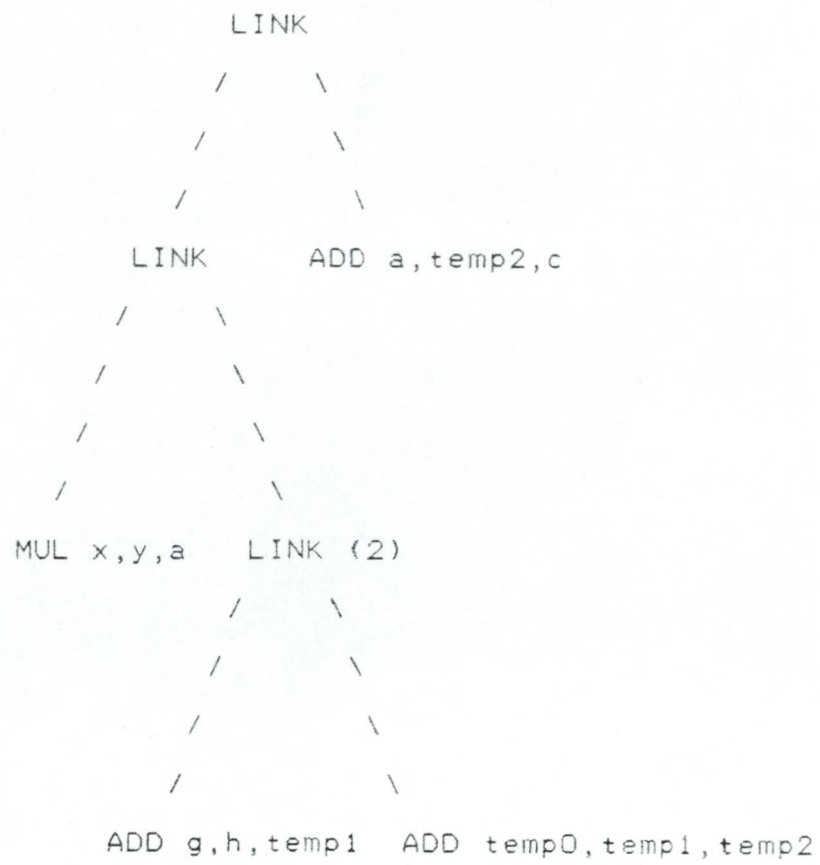
Register Allocation and Assignment, and Instruction Selection

These algorithms will use the results of basic block identification, variable identification and liveness analysis. If loops have been identified, this information will be used to provide global register allocation to registers of heavily used variables in the loop. This will be based on a usage count analysis across the basic blocks comprising the loop. Additionally for each basic block, "next use analysis" will be performed. This analysis is as described by Aho and Ullman [9]. It uses information developed during GAS code optimisation. It is not itself performed during optimisation as this information is not required for optimisations performed. A backward pass is made over each basic block at the stage when target machine code is being generated for the block. For each instruction encountered, record (in a separate list that consists of one entry for each GAS code instruction in the block and has a forward and backward link) the id of the GAS code and the "liveness" and "next use" status for each of its operands (as maintained in the variable

database established. If an operand is altered by the instruction, the record for the variable entry for the operand is set to indicate it is not alive and has no next use (NIL value for next use field of database entry). The variable database entries for operands not altered are set to indicate that they are live and that the next uses of them are in the instruction being processed (store the id of the current GAS code instruction in next use field). Note that the effects of operations on objects at computed offsets, operations on objects referenced via pointers, and the effects of possible type coercion, will be handled as described in the optimiser chapter. Any object in the same storage area as an object altered, or any objects in storage areas that could be pointed to by a pointer to an object to be changed, will be left with the same liveness and next use status. If the object is used, these other objects will be marked as live, but the next use field will be left unchanged, indicating that these objects have low priority for registers because of they are not directly involved. The live status ensures that any changes to these objects will not be thrown away during register reallocation. Additionally, the current location(s) (register/memory) of each operand used in a basic block is maintained in the database. At the start of the backward pass, the liveness of each variable will be set to the liveness of each object as computed by the global liveness analysis on exit from the code block, and the next use value will be set to NIL. If the RET instruction is the last instruction in a basic block, all variables except those at the current lexical level are considered to be live but the next use field in their symbol table entries will be NIL. This is because no interprocedural data flow analysis is currently done. Little can be done in a modular compilation system. If this is added at a later date,

then not all variables will be live on exit. If no global optimisation has been done, the "out" live variable list will be NIL, hence assume all variables are live on exit.

The variation in order of computation within a basic code block has not been considered at this stage in project development. The reason is that common subexpression elimination on basic code blocks makes tree labeling and reversal of order of operators less than optimal, because temporaries may live beyond the subtree in which they are defined. However there are techniques for producing sub-optimal trees in terms of registers used that could be implemented in later versions. Note that tree weighting of the GAS code is possible because the following is the form of Rcode tree that will be generated for binary operations:



Note that temp0 is an "outside" temporary, representing a common expression result. To allow tree reversal, the GAS codes for a basic block will have to be put back into tree form as above but a special form of tree would have to be allowed that allowed tree weighting values to be recorded on nodes. Also note that next use analysis within a basic code block would have to be done after tree reversal, and after the GAS code instructions for the block have again been linearised.

Consider now how registers will be allocated in code generation and how this interacts with selection of target code instructions addressing modes.

Code will be generated in a walk of the Rcode tree for a procedure, and when each basic block is encountered code for it will be generated. As the walk follows the Rcode tree, procedure structural information is available whenever code is being generated for a basic block. GAS code instructions for each basic block will be accessed using the optimiser database pointers to GAS code instructions for the block. Code will also be generated for group six Rcodes.

To illustrate the basic target code generation process consider the GAS binary operation involving Rcode basictypes, ADD B,C,A where A:=B+C. Assume a multi-register machine, allow for the effects of coercion, and to allow the possibility of deferred storage of elements in arrays (objects identified by computed offsets). The following is a general description of the algorithm that will be used, and discusses some of the relevant factors. The allocation and assignment of registers is always a

statistical problem, but the aim is to minimise memory references by trying to keep in registers objects with earliest next use. The uncertainty of what is being manipulated that is introduced by coercion and pointers makes effective allocation of registers very difficult. Several steps are involved in the target code generation.

Register stores before instruction

If any registers currently contain values from the same storage area as A (but not including A) or contain values for locations identified by a pointer that could point to an object in the same storage area as A (if no pointer analysis has been done, then any register containing such a value will be involved) then generate code to dump these values back into memory, and mark the registers as no longer storing these variables. They may be changed by the instruction so these variables must be considered no longer live. A similar approach is taken for B and C to dump register contents to memory, but leaves the registers still marked as pointing to the variables. If efforts are made to more exactly identify variables with static offsets that overlap, less register dumping would be required (see notes on GAS code optimisations). If indirect addressing is used to specify A, then generate code to dump the contents of any registers that contain variables in any of the storage areas that may be pointed to by the pointer value providing the address of A, and consider these registers then free. The possible areas that could be pointed to would be identified by pointer analysis (see GAS code optimisation). In the absence of any global pointer analysis, all registers will have to be dumped and considered free. Note that even if pointer analysis has not been done by the GAS code optimiser, limited pointer analysis could be done within the

local basic code block. If indirect addressing is used to identify B or C then do the same as for A, but leave the registers marked as still pointing to variables dumped.

Selection of Operand Locations

The next factor to consider is where the operands will be accessed for the instruction. This depends on many factors; whether the machine supports three address and two address instructions, and any limitations on where the operands of these instructions can be located. For example, two address instructions such as ADD X,Y may be limited to the first operand in a register, the second either in memory or a register, with the first operand receiving the result.

If only two address instructions are available, several factors will be significant. In a two address instruction, the result overwrites one of the source operands, typically the first. The following cases arise:

If the target operand doesn't win a register from a source operand that can be the source operand replaced by the result in a two address instruction (using a reverse instruction if necessary, and available, such as RSUB, RDIV) it will be necessary to generate a move instruction to move the first source operand to the location chosen for the target, before the two address instruction is generated. However the generation of this move delayed until after the location of the source operands has been determined. The "win" of a register from a source operand is based on the current location of source operands, current register usage, next use of objects in the basic code block, and liveness analysis. The target may not win because neither source

operand is in a register, or if any source operand is in a register, the target won't win the source register (because the source has a next use before the result or before the contents of other registers).

If the target operand wins a register from a source operand that can be replaced by the result in a two address instruction available on the target machine, then if the source operand is still live, an instruction must be generated to dump the contents of the register to memory first. Two address instructions are most useful in a situation where the target wins a register from a source operand that is not live after the instruction.

If a source operand is in memory but has a next use and wins a register, then code must be generated to load the register. The operand value will be loaded into the register allocated, then a copy will be moved into the location reserved for the result if necessary. The operand is then available in a register for the next use, saving a memory reference.

The location of the source operands and of the result have to be considered together. First consider the case of a source operand currently in a register, that has no next use, and does not contain the value for any other object, and can be replaced by the result in a two address instruction available on the target machine (considering reverse instructions if necessary). This register will be chosen for the target. Then the operands (minus the target if the previous step succeeded in allocating the result a register) are considered for register allocation, the operand with the earliest next use first. Note that if a source operand has no next use, it will not be considered for a

register. An operand will win a register if it has a next use and any of the following are true:

- there are free registers that can be used for the specified operand in the given instruction

- there are registers with objects that are live but have no next use in the basic block (these will be globally allocated objects) and these registers can be used for the operand

- the operand will be used before the object currently stored in any register that can be used for the operand, unless the register has been earmarked to receive an operand already. The register it will be allocated is one that contains a value used latest in the code block.

the selection criteria are applied in this order.

If the source object that is overwritten by the result in the two address instruction must be in a register (required by the target machine instruction forms), then the target operand must be allocated a register, and the source operand that is overwritten moved to this register. The above algorithm will be applied, except that the target operand will be considered for a register before the source operand reguardless of next use. If the target fails to obtain a register after the next use criteria has been applied, it will be allocated the register containing the object with latest use in the block.

It is after the location of the target and source operands have been determined that instructions are generated to move a copy of the source operand to the location of the result and to dump any registers reallocated or allocated for receiving the result.

If a three address instruction is available it would be used in several situations:

-When it is not desirable to place the result in a register. This occurs when registers are fully used, and the result has a next use after any of the register contents or the result has no next use in the basic block (but is still live of course or the instruction is redundant), and is not globally allocated a register.

-Either of the two source operands may be in registers or "win" register locations because they have an earlier next use than other objects in registers. Source operands may "win" registers specifically over the target because of next use considerations.

A two address instruction will be chosen in preference to a three address instruction if:

-either of the source operands has no next use after the instruction but is currently stored in a register and the result has a next use in the basic block

-either of the source operands is stored in a register, but has a next use after the target operand, and after all other values currently stored in registers

-the target operand is the same object as one of the source operands (may need to use reverse operation for sub, div etc instructions

The above preferences can be modified when the timings of various target machine instructions are taken into account.

Generate Binary

After deciding the target instruction form to use, and the locations selected for the operands, generate the relevant binary target bit pattern for the instruction that implements the actual GAS operands. Also generate any linker directives required for relocation of operand addresses.

Dump Registers

If any of the operands are in registers but have no next uses in the basic block and are not allocated globally to registers, generate code to dump the registers to memory and mark the registers as free. If any source operands are no longer live, mark the registers as free.

At the end of the basic block, the contents of registers that hold values must be considered. In a simple approach, at the end of the basic block, use the register descriptors to determine if any registers hold any variables that are live on exit from the basic block and for these variables check if a valid memory copy still exists. For any live variables that do not have current live memory copies, generate MOV instructions to save the register copies in memory. In a more complex approach, a global decision to hold commonly used variables in registers may be

taken. For example for all basic blocks of a loop, a decision may be made to hold variables "x" and "y" in registers "R1" and "R2". If these variables have been dumped (for example due to register shortage), code must be generated to reload them, as all basic code blocks will assume on entry that these variables are in the registers globally allocated to them. If these variables are in registers, they won't be dumped at the end of this basic block. Their variable database entries will indicate the variable is to be kept in a specified register. Another useful extension for the global allocation of registers is to compute disjoint lifetimes for variables with high usage, and globally allocate registers for these variables using graph colouring algorithms.

Index registers and register allocation

Index registers are involved with the implementation of GAS addressing modes. GAS addressing modes (see the GAS machine chapter) allow indentification of several possible levels of indexing. The structure addressing modes selected during GAS code generation and optimisation should allow as much address structure as possible to be retained so that maximum advantage can be taken of target machine addressing modes. In particular where an array address involves computation that includes a loop index variable, the loop index variable should be retained as clearly identified in the address computation for the array access.

Index registers will be used in several situations. One use is when an operand is specified as a computed offset into a global area and the offset loaded into an index register, with the base the address of the start of the storage area. Another use occurs when a GAS code operand is identified by an indirect address

contained in a temporary, plus an immediate offset. An example of the advantage of this form of addressing is that successive references to various fields of a record pointed to by a pointer would be efficiently implemented. The base address of the record would be loaded into an index register and the various fields would be accessed using different immediate value offsets. If the GAS code for computing the temporary involves adding two computed objects, and the machine supports two level indexing (two index registers added to a static base address) then generate code to load two index registers with these values. Hopefully one of the register values will be used again for another reference. This situation could occur when there is a two dimensional array or an array of records. In each case the static base value would be zero. Another use of indexing occurs when accessing stack objects. This will use a stack pointer and either an immediate and/or computed offset. If the stack object is accessed via a dope vector, a combined indexed/indirect addressing mode will be used. If the offset is computed, two level indexing can be used if available on the target machine (e.g. Intel 8086 has it) or the contents of the stack pointer register must be added to the computed offset, and single indexing used.

The use of index registers can be considered part of normal register allocation if the machines general registers can be used as index registers. In many machines, a limited number of general registers can be used as index registers, or index registers are quite distinct from general registers. This will not allow index register allocation to be considered just part of normal register allocation. In the VAX machine, any general register can be used as an index register, although it is normal to reserve register 12 to 15 as pointer registers, and registers 0 and 1 are used to

return results from any VMS library routines called. Therefore the VAX prototype code generator will mainly consider registers 2 to 11 as available for general register and index register allocation. For target machines with restricted use of general registers for indexing, additional code will be required so that only allowable registers are used when index registers are allocated. This code will typically have to be hand coded on a machine by machine basis. However as described earlier in this chapter, the VAX prototype is fairly orthogonal in the use of registers for general use and indexing and should provide the structure for machines with constraints.

This completes the factors involved generating code for an instruction such as ADD. Consider now how to handle registers that are dumped to memory.

Allocating space on stack to dump registers

When code is being generated and register contents have to be saved in memory, the problem is "where". For variables already allocated space in memory, the location is obviously in this allocated storage location. Temporaries have not however, been allocated any memory. Space will be allocated on the stack for storing dumped temporary values. The space for these temporaries is allocated as if it were local storage. However, sufficient space should only be allocated to hold the maximum number of temporaries saved at any one time. When a temporary of a specific datatype is to be dumped, a check is made for an allocated temporary storage object on stack that contains a variable that is no longer live. If one is found, the temporary is dumped to it. If one is not found, extra space is allocated on the stack. With this approach, more space may be allocated than absolutely

necessary, because space may have been allocated for a temporary that is now dead, but is of a smaller size than the temporary currently being dumped. Therefore there may be unusable gaps on the stack. However garbage collection to reclaim these gaps would not be worth the overhead. The code generator must keep track of temporary space allocated on the stack.

Saving resources used by a routine

When a routine is called, any variables the routine uses must be dumped to memory if the current memory value is not up to date. Additionally, any registers used by the routine that are currently being used by the caller, must be temporarily saved. A bitstring indicating which registers are used at the call point, could be pushed onto the stack by the caller. The called routine will use this bitstring to save the registers indicated. At the end of the routine, the bitstring is used to restore registers. Hence the first and last instructions in any routine will be standard register dump and restore code. A more complex dump/restore algorithm would involve only saving those registers that are in use by the caller and which are also used by the routine. This would be done by the compiler performing an AND operation on the bitstring representing registers in use in the caller at the call point and a bitstring representing the registers used by the routine, with the resulting bitstring used to decide registers to be saved/restored. The bitstring representing the registers used by a routine would hopefully be available in the GAS code entity database record for the routine at the time the compiler is generating code for the call so that the compiler, knowing what registers the caller has in use, can evaluate common registers, and hence generate code to push a bitstring that indicates these common registers. If the entity

database record for the routine has no bitstring for registers used (code has not yet been generated for the routine; only the definition has been encountered), or a record doesn't exist (the routine has not been encountered at all for example an imported routine) then generate code to push a bitstring that indicates saving of all registers currently used by the caller. This all, of course, depends on being able to clearly identify the routine being called. If this is not possible, then again push a bitstring to indicate all regsiters in use by the caller must be saved.

Variables that need to have their memory copy updated from register can be identified in a similar manner. The GAS code entity database record for the routine will contain a bitstring indicating which variables it MAY use. The caller will require code to dump registers that contain any of these variables used. If no information is available on the routine, all registers except those containing temporary values will have to be dumped. The MAY for the use of a variable refer indicates that a variable may be referenced. Uncertainty due to coercion, computed offsets and pointers could mean that many variables MAY be touched. Note however that handling the variables used bitstring only involves the compiler, it does not affect target machine code.

If the target machine has few registers, and efficient instructions are available to dump them, the simplst approach may be to dump all registers anyway.

When a routine uses very few registers, then it may be faster for the routine to dump these registers rather than have code to

process the register dump bitstring on the stack. Hence it will ignore the bitstring on the stack. This decision also depends on how efficient the code to dump registers based on the bitstring can be made. The target machine may have a register save instruction that can utilise the bitstring directly, or it may require a series of bit tests and push instructions. Hence the decision at which point to dump registers used by the routine (rather than using the registers indicated in the bitstring) will depend very much on the target machine instructions for saving registers.

Parameters of subroutine

Parameter passing conventions (call by reference, value etc) are handled by the Rcode. All the backend sees is the need to build an argument block.

Handling of subroutine GAS calling convention

The handling of the result, argument and local variable storage areas have been discussed. They require computation of size and the allocation of space on the stack (possibly using dope vectors). The GAS calling convention indicates the form that the stack should have during a subroutine:

result
return block (address and status)
arguments
local variable storage

Priority is placed on implementing this structure in the interest of portability as discussed in chapter four, rather than varying the format to suit special target machine calling conventions and

instructions. Associated with the GAS subroutine convention is the Display Vector. This contains pointers to the result, arguments and local variables, dynamic and static link pointers, and exception handler information. It is suggested that an effective way to implement the stack of display vectors that will be required at runtime is to maintain a separate stack consisting of the display vectors. This stack is referred to as the display stack to distinguish it from the "main" stack. The base of the main stack will contain four pointers:

Display Vector stack base

Display Vector stack limit

Currently active Display Vector

Top of display stack (Most recently created Display Vector)

Assume that the Call and Declare_Routine Rcodes are left on the Rcode tree by the GAS code generator. The PUSHMARK, NEWDISPLAY and POPMARK GAS codes become unimportant, as the structure of the call is provided by the Rcodes. These GAS codes are provided though (see Chapter Four) as these subroutine related Rcodes may not be present. In this case they are used to indicate the difference between a fast call and a full GAS subroutine calling mechanism.

When a CALL is encountered during target code generation, the structure of the call (result, arguments, local variables and routine descriptor handling) is made clear by the "Call" Rcode. On encountering a Call Rcode the target code generator will:

a) Allocate space for a new display on the display stack, and an instruction generated to increase the top of display stack pointer by the size of a display vector.

b) An instruction is generated to move the "main" stack pointer to the RESULT field of the new display vector.

c) The GAS code to compute the size of the result is then processed. This is located on the result computation sub-tree of the Call Rcode.

d) The GAS code to generate the arguments, also on a subtree of the Call Rcode, is processed. The first GAS code will be a PUSHMARK instruction. At this point the code generator will generate instructions to:

-increment the stack pointer by the size of a pointer (return address) and the size of any status information required for the target machine return instruction.

-move the address of this return block to the RETURNMARK field of the new display vector.

-move the stack pointer to the ARGLIST of the new the display vector.

The target machine code to generate and place the arguments on the stack is then generated.

e) The sub-tree of the Call Rcode that contains the GAS

code to compute the routine descriptor and push it on the main stack is then processed. The generator will use information on the target machine to decide how to form the static environment. The descriptor will not in fact be pushed on the stack. It is suggested that static link pointer be moved to the STATICLINK field of the new display, and the address of the target routine be retained by the code generator.

f) The CALL GAS code is encountered as the last instruction in the routine descriptor computation sub-tree. A target machine call instruction will be generated using the address computed in (e), ignoring the GAS operand for the CALL instruction.

When processing a routine declaration, the target code generator will respond as follows:

a) The first GAS instruction in the subroutine, before any declarations for local variables, will be the NEWDISPLAY instruction. Again this instruction is present in case the "Declare_Routine" Rcode is not retained by the GAS code generator. It clearly marks the start of a subroutine. When this instruction is encountered several instructions are generated:

-the current display vector pointer is moved to the PREVDISPLAY field of the new vector on top of the display vector stack.

-the pointer to the new display vector is moved to the current display vector pointer (in the main stack base)

-the return address and any status information

currently located on the top of the main stack is popped and moved to the location pointed to by the RETURNMARK field of the current display vector

-the stack pointer contents are moved to the LOCALSTORE field of the current display vector.

b) The local variable declarations are encountered and space allocated as suggested.

c) The end of the routine will consist of two GAS codes: POPMARK and RET. The code generator will generate the following instructions:

-The RETURNMARK field of the current display vector is copied into the TOP_OF_STACK field of the display vector pointed to by the PREVDISPLAY field of the current vector, and to the stack pointer register.

-The current display vector pointer is set to point to the display vector pointed to by PREVDISPLAY.

-The pointer to the top of the display stack is decremented by the size of a display vector. This deallocates the display vector for the routine that is terminating. At termination of a routine its display vector will always be the most recently created display vector.

-a return machine code instruction is generated

The static link method is left up to the target code generator based on the resources of the machine. The static link may be a pointer to a list of pointers to display vectors that provide the static environment, or simply a pointer to the display vector for the previous static level implementing a linked list for the environment.

The above describes how the calling mechanism can be implemented with no help from any target machine facilities, except for a stack pointer and simple call and return instructions. Improvements in the speed of the mechanism could be achieved by reserving registers to hold the pointers to the current and new display vectors, and the fields of the current display vector such as PREVDISPLAY, LOCALSTORE, ARGLIST and RESULT, and by utilising instructions of the target machine, particularly call instructions, that improve the overhead of the process.

It would be possible to utilise the VAX CALLS instruction to automate more of the call process. Registers could be used as follows (note that these registers may not point directly to the objects specified, but can be used to easily access the desired objects by the addition of a small static offset):

-The Argument Pointer register (AP register 12) provides the ARGLIST for the current display vector.

-The Stack Pointer register (SP register 14) provides the TOP_OF_STACK for the current display vector.

-The Frame Pointer register (FP register 13) provides the LOCALSTORE for the current display.

-The Program Counter (PC register 15) provides the PC for the current display vector.

-The Frame Pointer register also effectively provides the RETURNMARK for the current display vector. It does not point directly to the return block, but does provide the effective return address for the VAX RET instruction.

-Register 10 provides the RESULT field of the current display.

-The Argument Pointer also is used for the STATICLINK field. After the arguments have been pushed, the static environment will be computed, and a pointer to it is pushed onto the stack. Therefore the AP provides access to both the static environment and the argument list, although it does not actually point to either directly.

-The EXCEPTHANDLER for the current display vector is provided on the stack along with the saved register contents pointed to by the frame pointer. The exception handler mechanism provided by the VAX can then be utilised.

-The EXCEPTPARM for the current display vector will be provided on the stack after the exception has occurred (see VAX Architecture Handbook).

-The Frame Pointer register also effectively provides

the PREVDISPLAY as the AP, FP, and SP registers described here will be saved on the stack when a subroutine is called and together they effectively provide the previous display vector for the routine which called the current subroutine.

The STACKLIMIT and STACKBASE fields of the current display are not currently implemented because the VAX does not provide registers for these, but provides read only pages on either end of the memory allocated for the stack.

To begin a subroutine call, the result area is computed and stack space is allocated.

When the PUSHMARK GAS code is encountered, register 10 is PUSHED and the stack pointer is then moved to register 10. Register 10 then will point to this saved value and at an offset of 4 bytes from register 10, is the space allocated for the result area of the subroutine to be called. The arguments are then pushed on the stack. The routine descriptor is computed. The address of the routine, if computed will be computed into a register. The static link will be pushed on the stack. During computation of arguments and the routine descriptor, the code generator must note that the current result area pointer is now on the stack, but can be accessed via register 10. The CALL GAS code will then be encountered and a VAX CALLS instruction is generated with the number of arguments operand set to zero. This CALLS instruction will automatically:

PUSH the number of arguments operand

save stack pointer in a temporary internal register

PUSH registers specified in bit mask at the start of the called routine

PUSH PC (return address), FP, AP

PUSH a longword (32 bits) (not relevant to discussion)

PUSH zero longword (exception handler)

FP is replaced by SP

AP is set to the saved SP

PC is set to the first instruction in called subroutine

This stack structure provides what is termed the return frame for the called routine.

The PC, AP, FP, SP, Register 10, the pushed static link, and the pushed exception handler together provide the display display vector for the new routine. At plus 4 bytes from the AP is the static link, and at plus 8 bytes is the first argument. The FP points directly to EXCEPTHANDLER. At plus 12 bytes from the FP is the previous FP which can be used to access the previous display vector EXCEPTHANDLER field (see above).

The previous display vector is provided as follows:

PC	pushed PC
LOCALSTORE	pushed FP
ARGLIST	pushed AP
RESULT	the previously pushed R10 contents
STATICLINK	pointed to by pushed via AP
TOP_OF_STACK	register 10 contains the value the stack pointer will be set to on return to the caller; this will point to the pushed static link for the caller.
RETURNMARK	effectively provided by the pushed FP
EXCEPTHANDLER	provided on the stack in the return frame of the caller (used saved FP to access this frame)

When the VAX RET instruction is encountered, the following is performed by the VAX:

-PC, AP,FP replaced by values in return frame

-any saved registers restored

The first instruction in the caller after the CALLS should be a POP into register 10 to restore the pointer to the RESULT area of the caller routine. The RESULT produced by the called routine will then be available on the top of the stack.

In this approach to implementing the GAS call mechanism, the current display is contained in a mixture of registers and stack locations. Previous displays are spread through the stack. Though display vector fields are spread, it is reasonably easy to obtain access to any of them. Extra items appear on the stack than are

specified for the GAS call convention, but these items are simply located. Any programmer that manipulates the stack to implement some function and hopes the standard stack structure of the GAS convention is available on each target machine so that their code is easily ported, will find the extra items on the VAX stack easy to account for. The display vector is however provided, though the programmer must take account that the registers used for display vector fields do not all point directly at the objects involved. It is important to note that the "number of arguments" operand of the CALLS instruction has not been used to deallocate the space used by arguments. This VAX mechanism limits the size of the argument block to 1024 bytes. Additionally code is required to compute the size of the arguments so that the "number of arguments" operand can be provided. It is felt the use of register 10 for stack storage deallocation is more effective on both counts. The allocation of the result to the stack rather than returning through register zero as the VAX convention specifies also gives wider portability. The major defect of the use of the VAX CALLS instruction as described here, is the excessive use of registers. Additionally the real gain in performance of subroutines is limited.

An elimination of the need for register 10 can be achieved by using a stack of ARGLIST pointers as follows:

a) Allocate space for the result, with any descriptor required for a dynamically sized result allocated last.

b) Push the current stack pointer onto the ARGLIST stack.

c) Generate the arguments onto the stack, statically sized

arguments and descriptors for dynamically sized arguments first.

d) Compute the routine descriptor and push onto stack.

e) The compiler selects the registers to be saved.

f) Allocate space for saving registers, a display vector and a dummy VAX return frame by decrementing the stack pointer. The space is allocated assumming it will occupy the space currently occupied by the routine descriptor static link (but not the PC field of the descriptor). Set the return frame condition handler, register mask and the "number of arguments" fields to zero.

g) Move the computed static link to the STATICLINK field of new display.

h) Save selected registers in the allocated space.

i) Call the subroutine using a JSB instruction that uses the address in the computed routine descriptor on the stack, if the address is dynamically computed, else use the statically evaluated address.

j) The first instructions of the subroutine should be as follows:

-The return address on the top of the stack is popped into the "saved" PC field of the return block.

-The current FP and AP registers are moved to the

"saved" FP and AP fields respectively of the return block.

-The value on top of the ARGLIST stack is copied into the AP register.

-The current SP contents are moved into the FP.

The subroutine is not finally considered entered until the last step which makes the display vector for the new routine finally active. Between the JSB instruction and the move of the SP to the FP, the caller is still considered active.

The current display vector is effectively pointed to by the FP and is comprised as follows:

-EXCEPTHANDLER is located in the Condition Handler field of the return block. The VAX Condition Handler mechanism will therefore still operate correctly. The FP points to the previous return frame, which will contain the next level condition handler.

-RESULT and ARGLIST fields are effectively provided by the AP register.

-LOCALSTORE is effectively provided by the FP.

-PREVDISPLAY is provided by the "saved" FP.

-PC is provided by the machine PC.

-TOP_OF_STACK is provided by the VAX SP register.

k) Space is allocated for the local variables of the subroutine. Statically sized objects and dope vectors first, then space for dynamically sized objects.

l) Return from subroutine is achieved by the VAX RET instruction. This will automatically load AP, FP and PC with those of the caller (saved in the return block), and will leave the SP pointing to the display vector of the subroutine that has just completed. The following steps are then performed:

-Restore saved registers.

-Pop the value on the top of the ARGLIST stack into the SP. This value effectively points to the result and arguments of the subroutine just returned from.

This completes the subroutine call process. Its main advantages over the previous two methods are:

a) Does require exclusive use of any more registers than the VAX convention (AP, FP, SP).

b) Recognises that in physical reality, a separate stack of created display vectors is not required, but that merely a separate stack of ARGLIST values for created display vectors is required. Space is only allocated for display vectors when its routine actually becomes active. This is useful when generating arguments requires subroutine calls.

The problem for this method (and the first method involving a separate display vector stack) is how to allocate space for the ARGLIST stack. Space can be allocated in the same the storage area allocated for the main program stack. It "grows" in from the opposite end to the main stack. The base of the main stack contains a pointer to the top of the ARGLIST stack. The difficulty is that hardware stack overflow protection would not now operate on the VAX. The main stack could overflow into the ARGLIST stack. In a machine with stack base and limit registers, the limit register could be moved to always point to the top of the ARGLIST stack. Alternatively space could be allocated in heap, but this has the disadvantage with multiple processes that space would be wasted if there were many of these stacks allocated and only partially full. An alternative is for ARGLIST stacks to grow in segments, but management of these structures would add extra overhead to subroutine calling.

Jump/Call displacement Fixup

Jumps to labels are considered in two categories, jumps to labels in the routine, and jumps to labels outside the routine. Jumps inside the routine will be optimised to minimise required offset sizes. Jumps to labels outside are more complex and will require a return to the environment of the label. This will require manipulation of the stack. The routine containing the target label will not have been coded, hence so the offset to the label will not be computable. The branch instruction will be encoded with a displacement large enough to address anywhere in the program address space. The target label will not have been encountered because Rcode presents the program in Post-order form. The code for outer enclosing lexical levels will be encountered last, enclosed lex levels first. In the languages

that will be handled the target of a jump must be within the lexical level or to an outer enclosing lexical level, in the current environment. The code for enclosing lexical levels will not have been encountered. A linker directive will be generated to indicate that the contents of the displacement field must be set to the address of the label. When this label is later encountered during target code generation, a linker directive is generated to identify the label to the linker.

Branch instructions will be encoded using PC relative addressing modes whenever possible.

Call instructions can be encoded to minimise the displacement field if the code for the procedure has been generated which will be true if the routine called is lexically contained. If the code for the called routine has not been generated, then full-length displacement must be generated. However the required displacement length can be computed as the Call instruction is generated. The actual value of the displacement cannot be computed until the size of displacement fields of any branch instructions preceding the call instruction in the routine are computed.

As code is generated for the routine, branch instructions are coded assuming long displacement fields in the instruction. Labels are given byte offsets from the start of the routine based on branch instructions with long offsets. A list of pointers to branch and call instructions is kept, in physical order. When the coding of the routine is complete, the relative displacement for each branch instruction is computed in terms of initially allocated offsets for labels, and the minimum length displacement

fields necessary for each branch is computed. Given the new computed size of displacement fields, byte offsets for labels are recomputed. Branch and call instruction displacement operands are then computed and stored in the instruction.

References to global data objects

References to global objects will involve operands whose size can be computed at the time the instruction is generated. The displacements involved will not depend on the size required for instructions, as do branch instruction displacements. References to global objects will usually involve an offset into a data segment area. The address for the start of this area may be in a register, or a global addressing mode may be used. In the latter case, the address field of the instruction will be loaded with the offset of the global object into the data segment involved, and a linker directive will be generated to specify relocation of the offset by the start address assigned to the data segment involved.

VAX Abstracted Resource Database and Code Generation Procedures

The discussion of target code functions above suggests abstracted approaches to target code generation. The following discusses in more detail what is involved in the mechanics of abstraction. It attempts to clarify how abstraction is being manifested in the PLIP project VAX prototype target code generator, hopefully demonstrates how abstraction can contribute to portability of target code generators.

As described earlier in this chapter, portability is enhanced if the access to information on the resources available on a machine, and the state of allocation of those resources is

provided by an abstracted database that is accessed by standard abstracted procedures and constants. Additionally the abstraction of the target code generator itself whenever possible will also improve the portability of code generators. It was also suggested that a single abstracted target code generator for all machines in the family should not be attempted as the complexity of some machines would make the abstaction large and complex. Therefore the aim is a set of fairly fluid procedures and constants that will hopefully be easily modified to produce code generators for new machines in the family. In some respects this appears like a new level of families within the target code generators, a third level of the code generator. However machines at this level are not easily categorised into useful families, as many subtle differences often exist between any two machines which makes a complete generic handling of the two machines not as straight forward as the generic treatment of families at the GAS level. What is really being aimed at is a pragmatic treatment of the mechanics of code generator development that will maximise the potential for easily modifying a code generator for one target machine to produce the code generator for another machine. The abstracted procedures and constants will greatly assist this aim. These procedures and constants for machines in a GAS family will tend to differ more in implementation rather than definition.

It has been suggested that a good starting point for a target code generator is to base the structure on the structure that would be developed for an orthogonal machine that would be a member of the same generic family as the target machine. The VAX provides a reasonably orthogonal machine with registers, and therefore the VAX prototype provides a good basis for target code generators for other machines in the family. Its main non

orthogonal features have already been discussed.

The following is a description of some of the abstracted procedures and constants developed for the VAX prototype that will illustrate the typical form of these procedures. Note that most constants required by the target code generator will be provided in the "Machine" module and will be information related to the size of objects suported. These procedures and constants are based on the approaches to issues such as register allocation and two, three address instructions, as discussed earlier in the chapter. They are therefore abstractions of the approach taken in relation to these issues. For example the procedure "In_Register" is important regardless of the machine. Similarly when an instruction ADD X,Y,Z (Z=X+Y) is encountered, it will be necessary to copy back to memory the contents of any registers that contain values that share the same storage area of X,Y or Z. These can be requested by procedure calls:

```
Dump_All_In_Area(X,Source)

Dump_All_In_Area(Y,Source)

Dump_All_In_Area(Z,Target)
```

and these procedures will be implemented in terms of the target machine. The procedures below are an example of typical procedures that are being developed for the VAX prototype. However many of these procedures will appear in the target code generators for a wide range of machines.

Several procedures will return information required for decision

making by the target code generator or are used to update the target code generator database to reflect changes in say register contents or locations of variables:

In_Register(variable_id)

If the variable is currently located in a register returns the register id, else returns no register.

Used_First(var_1, var_2)

Returns TRUE if var_1 has a next use before var_2

Is_Live(variable)

Returns TRUE if variable is live- i.e. could be used agian before it is defined again

Has_Two_Address_Form(Code, Datatype)

Returns TRUE if the target machine supports a two address instruction for the GAS code specified and the Rcode Basictype specified

Has_Three_Address_Form(Code, Datatype)

Returns TRUE if the target machine supports a three address instruction for the GAS code specified and Rcode Basictype specified

Can_Be_In_Registers(Code,Form,Datatype,Operand)

Returns the set of registers that can be used for the location of the specified operand, for the target machine instruction of specified two or three address form, that would be used to implement the GAS code instruction specified in

"Code" for the the Rcode Basictype specified in Datatype. The "Operand" specification can be: Target_Op, First_Op, Second_Op, Third_Op. Note that this procedure should be implemeted without an array to represent the information because it would be a large array, and usually a standard set of registers will be allowable, except in few instructions.

No_Registers

Returns an empty register set. Useful in context:

IF Can_Be_In_Registers = No_Registers

Can_Be_In_Memory(Code,Form,Datatype,Operand)

Returns true if the specified operand can be in memory (see Can_Be_In_Registers)

Location(Varid)

Returns a location specifier of the variable specified. If the variable has several current copies, for example in two registers and memory, will return a register by preference. The represention of the location is target machine dependent.

Set_Location(Varid,Location_Specifier)

Will modify the target code generator database to reflect the fact that there is now an additional copy of the variable at the location specified by "Location_Specifier".

Several procedures are needed that specifically request generation of target machine code to implement standard functions.

Allocate_Register(Variableid)

Will allocate a register to the variable purely on the basis of next use criterea. Will generate a load instruction and will update the target code generator data base to reflect that the variable now has a copy in a register, if the variable does win a register. Will also generate an instruction to dump the contents of a register won form another variable, but only if there is no other current copy of the variable.

Dump_All_In_Area(Variable,Source_or_Target)

This procedure will generate target machine code to dump any registers that contain variables that share the same area as the specified variable. If "Source_or_Target" is "Source", the registers will still be considered to contain the variables. If "Source_or_Target" is "Target", the registers will be considered free.

Dump_All

Will generate target machine code to dump all registers that are not globally allocated and contain live variables.

Dump_Variable(Variableid)

Will generate target machine code to dump the specified variable, if currently in a register.

Get_Register(Variable, Allowed_Registers, Operand)

Will return in Operand a pointer to the target machine code operand that specifies the location to use for the variable. If a register is won, this will be a register operand. The "Allowed Registers" are the registers that can be considered for use. Will not generate any code to spill any register allocated that contained a value for a variable.

Dump_Register(Registerid)

Will generate code to dump register indicated in the target machine operand.

Leave_Basic_Block

Will load into registers any globally allocated variables that are not currently in their globally allocated registers, then will call Dump_All.

Generate_Add(type,locid,locid,target_locid)

Will generate target machine code to add objects of the type specified using values at the locations specified by the first two locations, and placing the result in the third location. Note that on some machines this procedure may generate a sequence of instructions or a call to an instrinsic subroutine library.

The following illustrates how these procedures are used. This is not a comprehensive algorithm for generation of target code for an addition operation, but is intended to illustrate the use of the procedures without being too complex. This does not of course, handle all the possible combinations of locations of operands that could occur.

```
Dump_All(X,Source)
Dump_All(Y,Source)
Dump_All(Z,Target)

Allocate_Register(X)
Allocate_Register(Y)
IF Has_Two_Address_Form(ADD_,Type) THEN
     IF In_Register(X) AND
        Used_First(X,Z) AND
        Used_First(X,Y) THEN
            Set_Location(Z,Location(X))
     ELSIF In_Register(Y) AND
            Used_First(Y,Z) AND
            Has_Two_Address_Form(ADD_,Type) THEN
               Set_Location(Z,Location(Y))
     ELSE
            Allocate_Register(Z)
            IF In_Register(Z) THEN
               IF NOT (In_Register(X)) THEN
                 Generate_Move(Type,Location(X),Location(Z))
                 Set_Location(X,Location(Z))
               ELSIF NOT (In_Register(Y)) THEN
                 Generate_Move(Type,Location(Y),Location(Z))
                 Set_Location(Y,Location(Z))
```

```
                    END
                END
            END
        ELSE
            Allocate_Register(Z)
        END
        Generate_Add(Type,Location(X),Location(Y),Location(Z))
```

The "Dump_All" procedures will cause "STORE" instructions to be generated to dump registers containing values that may overlap any of the operands. The "Source" and "Target" values specify whether the operand involved may be changed by the operation involved. If a "Source" operand, the registers whose contents are copied back to memory can still be considered to hold the variablesafter the instruction. If a "Target" operand the registers should no longer be considered to hold the values involved as the variables may be changed. For more information on possible overlapping of variables in the same storage area or via use of pointers, see notes earlier in this chapter, and the GAS code optimisation chapter.

The "Allocate_Register(X)" and "Allocate_Register(Y)" will result in generation of register "LOAD" instructions if the variables are not currently in registers, but can win a register. If a register is won from another variable, a register "STORE" instruction will be generated to copy the register back to memory. Registers are won on a "next use" basis. Register allocation is discussed earlier in the chapter. Many approaches have been developed to the problem of register allocation. In this thesis a primary aim has been to demonstrate that any register allocation scheme can be effectively abstracted in a set

of procedures, constants and database.

The next step in the algorithm above is to consider the possible locations for the operands and the possible use two address instructions. This logic represents an abstraction of the approach outlined earlier in this chapter. The procedures and algorithms reflect that the VAX usually offers the choice of two and three address instructions for most operations.

The "Generate_Add" procedure will note the locations chosen for the operands and the datatype and will generate appropriate target machine instructions. It will detect when two address instructions are to be used when the location of the target operand is the same as one of the source operands.

Another example of the abstraction of target code generation logic is the actual generation of target machine code bit patterns. For many machines there are often just a few basic instruction formats. Consider a fictional machine that has instructions with formats:

no operands (e.g. return form subroutine etc)

one register operand (e.g. increment register etc)

two register operands (e.g. register to register move)

register then memory reference operands

two memory reference operands

Additionally assume that the instruction format depends entirely on the operation code. This means that each target instruction has a fixed format, obviously a fairly unrealistic assumption, but simplifies the discussion.

The above instruction formats are quite common. Note though that no immediate operands are allowed. This can be represented by the following data structure and procedure abstractions:

```
Code_Form = (
              No_Ops,
              One_Reg,
              Two_Reg,
              Reg_Memory,
              Memory_Memory
            )

PROCEDURE Encode_Operation(
                           Code,
                           Op_Location_1,
                           Op_Location_2
                          )

TYPE
     Byte = [0..255]

VAR
     Address_Length  :  Byte
     Code_Bytes      :  ARRAY[0..7] OF Byte

BEGIN
```

```
Code_Bytes[0] := Code_Bits[Code]
CASE Operation_Form[Code] OF

One_Reg:
   Code_Bytes[1] :=
       Register_Bits[
                    Register_Id(Op_Location_1),
                    First_Register_Operand
                   ]

Two_Reg:
   Code_Bytes[1] :=
       Register_Bits[
                    Register_Id(Op_Location_1),
                    First_Register_Operand
                   ]
                                  &
       Register_Bits[
                    Register_Id(Op_Location_2),
                    Second_Register_Operand
                   ]

Reg_Memory:
   Code_Bytes[1] :=
       Register_Bits[
                    Register_Id(Op_Location_1),
                    First_Register_Operand
                   ]
       Address_Length :=
          Encode_Address(
                        Op_Location_2,
```

```
                                    Code_Bytes,
                                    2
                                   )

        Memory_Memory:
            Address_Length :=
                Encode_Address(Op_Location_1,Code_Bytes,1)
            Address_Length :=
                Encode_Address(
                                Op_Location_2,
                                Code_Bytes,
                                1 + Address_Length
                               )
        END
        Send_Linker(Code_Bytes)
    END
```

If the instruction has no operands or only one operand, the extra operand specifiers provided in the procedure call would be ignored.

The array "Operation_Form", returns the operation format in terms of the enumerated data type "Operands_Form". This reflects that in many machines the number and allowable forms of operands are a function of the operation code. The operands themselves may be specified independently as occurs in the VAX, or operands may be coded jointly or with the operation code bit pattern. In this ficticous machine, operands are coded independently, unless there are two register operands, in which case the registers are coded into the same byte. The algorithm above could be extended to check the supplied location conformed to that required (register

or memory location).

The additional procedure "Encode_Address" is required which will take a location specifier that represents a memory location and will encode this into bits necessary for the target machine to specify the given location. This procedure itself may have a similar form to the Algorithm above, with a "CASE" structure based on an "Operands_Form" enumerated data type that represents the target machines form of memory reference operands. Additionally this procedure will emit linker directives for relocation if necessary. Two arrays are required that store bit sequences needed for register identification and for each code operation. The register array is two dimensional, with one dimension the register identification, and the second an indication of whether the register operand is the first or second operand. This assumes the first register is coded into the first four bits of the second byte of the target code instruction, and if the instruction is a register to register instruction, the second register is coded into the second four bits of the second byte. It could be argued in keeping with the philosophy that table representation is less effective for representation of machine information, that these arrays should be replaced with procedure calls. Procedures provide more felexibility, particularly when it comes to porting, but they have a higher overhead than the time required to access an array. This decision will always be important where low level pieces of information are required.

The VAX instructions have operands that are dependent on the operation code, but only in respect of the number of operands and restrictions on the formats of each operand. Operands are coded

independently, and not into the same bytes as the operation code or other operands. This means that bit pattern generation could be abstracted as generating bits independently for the operation code and each operand. This approach is valid for a machine where there are no restrictions nature of any operands. There is no need for the concept of instruction "forms". Each operand can be coded into bits independently. All that is required for each instruction is the number of operands. However, some VAX instructions naturally involve restrictions on the nature of some or all of its operands. An example is the "MOVC" instruction which involves moving of a block of memory, therefore the operands that specify the source and destination blocks must obviously be memory addresses given explicitly or via an address in a register or a memory location. The source and destination cannot be registers. The independent operands approach could produce invalid target machine code by invalid operands to be produced. Therefore a more rigorous abstraction would allow checks to be made on operand types based on the operation code. Therefore the concept of instruction "forms" could be used. For the VAX there will be many forms to accomodate the full instruction set. Each form would consist of an operation code field and several operands, each one of which has a set of allowable forms.

Very few machines have a fully orthogonal instruction format where for any instruction all operands can have any of the forms supported by the machine. Most machines will be similar to the VAX in which the instruction "forms" concept is an effective basis for abstracting the process of target code bit generation.Therefore the general algorithm above will be applicable to most machines. Modifications may have to be made in

each CASE option, but hopefully many machines will have such similar instruction formats that often very few changes will be required.

Hopefully this discussion of abstraction of target code bit pattern generation illustrates how portability of the target code generator can be enhanced by abstraction of this function and various other functions of the target code generator. As the abstraction is not performed with all possible target machines in mind but rather a generic family, the abstraction can be more effective.

Full implementation of the prototype VAX code generator based on the approach of a target code generator database and code generation algorithms that utilise the procedures, is still no where near complete. This task involves significant work and is beyond the scope of this thesis. What is significant from the viewpoint of this thesis is that the two level approach offers the chance to develop target code generators that are generalised as much as possible for the machines in the same generic family. This is much more practical than developing a generalised target code generator for all machines. Generalisation is not achieved via tables of information but via a fluid abstracted set of procedures and constants that utilise an abstracted database. This development of such target code generators, and in particular identifying and developing the most effective abstracted procedures and target code generator database will require significant further study. During the researh work for this thesis the concepts have been developed to a stage where they have been demonstrated to be practical and to contribute to the goal of portability, and that the two level code generator

approach enhances their practicality and effectiveness.

Target Machine Code Optimisation

Any target machine optimisation will obviously require such an optimiser be produced for each target machine. Therefore it will not be desired to perform much optimisation at this level. Use of the GAS code optimiser requires only one optimiser will be developed for each family, and that hopefully provides much of the optimisation relevant for the machines in the family, so that little optimisation is required of target machine code. Additionally, optimisation at the machine code level is usually difficult at the machine code level as program structural information is usually not available. However in the PLIP compilers, the code generation is performed with Rcode structural information still available, and basic blocks and code motion information usually available. This would allow more reasonably complex optimisations to be performed, but GAS code optimisation would already have achieved most of these. The only practical and useful optimisation at the target machine level are therefore peephole optimisations, in particular branches to unconditional branches, copy propagation, and elimination of unnecessary loads and stores that may be introduced by the target code generation process.

Chapter Seven

Conclusions

Introduction

The aim of this study has been to develop a design for a compiler back-end code generator for the VAX/VMS environment for the PLIP project. This project required that portability be a major consideration in the design. GAS code has been developed as an effective way to improve portability primarily by representing the common architectural features of a family of machines. which for the VAX family machines include:

Linear code using three address instructions

Control logic implemented by branch instructions

Subroutine calls implemented via stack

All arithmetic on fixed sized objects

The GAS code is designed to contain as few codes as possible by effectively removing all object "typing" and addressing modes to the GAS operands. Future target code generation for RISC hardware will find this useful. The addressing modes are designed to allow address computation structure to be clearly represented, but only to the level necessary to efficiently implement source language structures rather than all target machine addressing modes.

GAS code Generation

Generation of GAS code will involve many of the common steps of generating target machine code for machines in the family. GAS code more closely resembles target machine code than Rcode. It is attached to the Rcode tree in the Reserved two byte extension Rcode, but program structural information represented by Rcodes such as "Loop", "Case" and "Call" are retained, as well as declaration Rcodes. These are useful to a GAS code optimiser and interpreter, and during target code generation. This has the added advantage that the procedures for handling Rcode will handle GAS code since it is only a form of Rcode.

Optimisation at the GAS code level has advantages over optimisation at either the Rcode or target machine level. At the Rcode level, available optimisations are not as comprehensive as possible at the GAS code or target machine level. Optimisation at the target code level would require an optimiser be written for each target machine. Optimisation at the GAS code level offers a useful compromise. Because GAS code consists of few codes the optimiser is simpler than either an Rcode or target machine code optimiser, and a GAS interpreter is a more practical proposition than an Rcode interpreter. The fact that GAS code is attached to the Rcode tree allows program structural information to still be available to the GAS optimiser; this has been found to assist both the mechanics and effectiveness of the optimisation.

Extending portability into the generation of target machine code has been achieved by abstracting the process of generating code as much as possible with the aim that the code for a target code generator for one machine will be easy to port to another machine. How well the abstractions for one machine will apply to

another machine will obviously depend on similarities between the machines. Decisions as to the optimum context in which to extract functionality are subjective, dependent upon the viewpoint of the individual designer. The more general the abstraction the wider the range of machines to which it will apply, but probably the less useful the abstraction. It has been noticed during development work for the first code generator that it is the major features of a machine's architecture that dominate the functions that must be performed to generate target code. The target code generators for the machines in a GAS family will therefore have strong similarities. The abstractions developed for target code generation should therefore aim to be general in terms of the architectural features that define the GAS family to which the machine belongs. It is not expected that one single set of abstractions will apply to all machines in a family, but that a fluid set of abstracts will be developed with modifications in the abstractions for each machine to allow for idiosyncrasies of each target machine. Abstraction will be manifested in a set of procedures and a database that represents the characteristics and resources of the target machine and the current allocation of these resources by the compiler. A table driven approach to target code generation was rejected as being incapable of adequately representing the machine characteristics necessary for quality target code generation across even a small range of machines. Table driven code generators are based on a single code generator that is supplied relevant target machine information in table form. The code generator tends be the superset of characteristics for all the intended target machines. As there are many idiosyncrasies and local differences the code generator and table have to be large. The concept of a set of "fluid" abstracted procedures and associated database allows advantage to

be taken of what is common between code generators, but hand modifications are made to incorporate idiosyncrasies of the target machine. Code generator generators were mainly rejected on the grounds that to produce a code generator for a new machine, the two level code generator approach would, given the same effort required for code generator generators, allow development of a smaller and faster code generator that produced better quality target code. This would be more so if the new machine is similar to a machine for which a code generator already exits.

To assist with abstraction and maximise commonality between target code generators, common approaches have been taken to target code generation issues such as handling storage and code declarations, subroutine calling conventions, register saving during subroutine calling and allocating space for temporary variables.

Future Research and Development

The further development and study of Rcode as the main basis of portability via front-end and back-end separation should be pursued. This will occur as compilers are developed for a range of source languages, target machines and operating systems.

The development of GAS code for other families such as zero address and transputer based machines is required. This work will also allow the "family" concept to be refined. The work done for the prototype GAS code in this study should provide the basis for other families as much of the GAS codes will be the same for any two families.

The preliminary GAS code optimiser designed in this study

requires refinement.

The VAX target code generator that is under development requires further work, with abstractions of functions and extension and refinement of the associated database. The abstractions most useful for portability will always be subjective and will be improved with further study and experience.

A set of source programs is required for testing purposes. These would allow the correctness and quality of target code produced to be assessed. Additionally the value of improvements in the code generator could then be assessed in an objective manner. It is important that these programs provide a wide and balanced representation of typical programs for which the compiler would be used. It is hoped eventually to use the test suite being developed at the University of Tasmania for this purpose.

Appendix A

Machine family No 1 GAS code Reference Manual

Introduction

The Generic Action Set (GAS) Code described in this Reference Manual is for machines with the following general architectural characteristics

a. Has two and/or three address instructions. Does *not* provide zero address or accumulator instructions for binary operations.

b. Provides a stack with pointer register support suitable for allocating and accessing arguments and local variables of routines.

c. Offers an instruction set which can directly implement the GAS Code instructions described below.

If a target machine cannot implement most of these instructions with one or a few of its own instructions, the target machine is not compatible with this family. The characteristics of this set are :-

a. ALU arithmetic instructions operate on a small range of statically sized objects which must be byte aligned. All operands need not be of the same size.

NOTE This requires that Rcode block exact arithmetic operations be broken down into series of operations on basic objects that are supported by the target machine.

b. Memory movement instructions can involve movement of dynamically or statically sized numbers of bytes.

c. Stack instructions equivalent to PUSH and POP are available.

d. Control instructions essentially involve conditional and unconditional branching.

e. Both simple and complex subroutine Call and Return facilities are offered.

f. Facilities are available to enable and disable interrupts or, at the least, an uninterruptible test-and-set style instruction is available.

g. Provides at least instructions for logical operations OR, AND, XOR and NOT.

The GAS Machine Stack

The GAS Machine has a stack which is used for allocating storage for results, local variables, arguments and return address blocks for subroutines. It is considered to consist of a stack of objects, each of which can consist of a different

number of bytes. Objects are placed on the stack by the PUSH GAS Code and removed by the POP GAS Code.

Display Vector

Associated with each routine, whether active or not, is a "Display Vector" which contains several pointers to objects on the stack plus a few other values. This vector consists of the following "fields" :-

a. **TOP_OF_STACK**. This is a counter of objects on the stack. When an object is pushed onto the stack, it is incremented by one. When a new stack is created it is set to zero. Most references to an object on the stack will involve a display pointer plus an object offset (object number), and a byte and bit offset within the object, rather than a conventional stack pointer which points to bytes plus a byte offset and bit offset. This overcomes the problem of handling dynamically sized objects which are usually handled on many machines by descriptors of some form. The GAS machine has facilities to automatically locate an object given an object offset. When the address of a stack object is required as data for an instruction, it is assumed that a unique byte address is generated. Reference to this object can then be made via this unique address. This address is a runtime value and the mapping of objects, given their object offsets and their unique byte addresses is handled automatically by the GAS Machine.

NOTE When the GAS Code is being interpreted in a real machine, some mechanism will be required to achieve this mapping.

b. **RESULT**. This contains the object number (offset) of the current result object.

c. **ARGLIST**. This contains the object number (offset) of the start of the current argument list.

d. **LOCALSTORE**. This contains the object number (offset) of the start of the local objects of the current routine.

e. **STACKBASE**. This is the unique byte address of the base of the stack.

f. **STACKLIMIT**. This contains the unique byte address of the limit to which the stack may grow.

NOTE Stack underflow corresponds to an attempt to pop an object when the TOP_OF_STACK is zero. Stack overflow occurs when a push would result in the stack growing past the address given by the contents of STACKLIMIT.

g. **ROUTINEDESCR**. This contains the routine descriptor for the currently active routine. It consists of the current PC and the STATICLINK.

(1) **PC**. This contains a pointer to the current GAS Code when the machine is running. Such a pointer indicates where in the Rcode-style tree structure this is to be found.

(2) **STATICLINK**. This contains a pointer to the static environment for the current routine. It may be an environment vector or a link to the display for the lexically enclosing routine.

h. **PREVDISPLAY**. This field points to the previously active display. This is not actually required by the GAS machine but is present to reflect the fact that the target machine will require it. It represents the dynamic link mechanism.

i. **RETURNMARK**. This contains the object number (offset) of the return address to be used when returning from the current routine.

j. **EXCEPTHANDLER**. This points to the routine to be executed if an exception is detected - by some "monitor", for example by the GAS Machine "hardware". This routine takes one parameter which is contained in the EXCEPTPARM pointer.

k. **EXCEPTPARM**. This contains a pointer to the parameter to be used by the exception handler routine.

Run-time Subroutine Structure

A number of run-time data structures provide the basic GAS Machine mechanism for handling the environments of routines. Each time a new procedure is entered, a display vector is created.

Note that the CALL GAS Code is equivalent to the FAST_ CALL Rcode. That is, it only involves pushing a return address and branching to code at the address given. To handle full routines the PUSHMARK, NEWDISPLAY and POPMARK GAS Codes are provided. The GAS Machine implements the CALL Rcode in the following way

a. Compute the size of the result area, then push a block object of this size onto the stack, giving it the alignment required for the result, and a NIL value.

b. Execute the PUSHMARK GAS Code. This results in the creation of a new display vector, and allocates space on the stack for a subroutine return block (return address and any status information required in a return block) by

pushing an object of the size required for the return block, and with a NIL value. See the details of this instruction later in this manual.

c. The arguments are then pushed onto the stack.

d. The routine descriptor is computed and also pushed on the stack.

NOTE The GAS machine never needs code to establish the static link field. It is assumed that this is built into it. The generation of target machine code or GAS Code interpretation will, however, require some suitable mechanism.

e. Use the CALL GAS Code using the address field of the descriptor (which is currently the object on the top of the stack). The byte offset of the address field into this object will be *target* machine dependent, and the GAS Code generator must therefore have access to this information. The address of the instruction after the CALL and any status information that is also pushed by the target machine calling instruction will be pushed on the stack; control then passes to the instruction pointed to by the routine address. The return address and status information will be considered by the GAS Machine to occupy one object location and to be of the size required for such a return block on the target machine. The GAS Code for calling may thus be written as

CALL Always, Offset_ of_ RetAddr + (TOP_ OF_ STACK)

f. In the routine code, use the NEWDISPLAY GAS Code as the first instruction. This performs several actions

(1) The most recently created Display becomes the current (active) display with its PREVDISPLAY field pointing to the display that was active. The TOP_ OF_ STACK pointer of this new display will be made equal to the TOP_ OF_ STACK of the old display.

NOTE 1. There could be several new displays yet to be activated. This would occur if the generation of the arguments, or the routine descriptor involved calls to routines and that these routines themselves involved calls to routines (etc!). The most recently created display is the one to make active because these calls are nested. However, note that if when computing and pushing arguments a subroutine call is encountered, the computation of the environment of the called procedure will always be correct as the new display created by the PUSHMARK is still not the active display at the time of the new call.

2. One of these calls may involve a FAST_ CALL Rcode, in which

case a new display is not generated and used in the routine. However, this is why the activation of the new display cannot be done with the CALL GAS Code, because even though there may be new inactive displays, on a fast call, the CALL GAS Code does not involve a new display becoming active.

(2) The return block is on the top of the stack and must be moved to the stack location pointed to by RETURNMARK.

(3) The static link contained in the routine decriptor which is now on the stack top is moved into the STATICLINK field of the display.

(4) The routine descriptor on the top of the stack is now not needed, so the TOP_ OF_ STACK is decremented by 1.

(5) Move the TOP_ OF_ STACK to the LOCALSTORE field of the display

g. Allocate space on the stack for local objects.

To return from the subroutine use the POPMARK instruction followed by the RET instruction. The POPMARK instruction will place the RETURNMARK value in the TOP_ OF_ STACK of the previous display, effectively deleting the current display and making the previous display active. The RET instruction will find the correct return address and status information (required for the target machine return instruction) at the top of stack.

GAS Machine Flags

Several flags are provided by the GAS machine. Two of them, Overflow and Carry are explicitly set by executing various instructions in the GAS Machine. These flags will have a value of "One" (TRUE) OR "Zero" (FALSE). They can be treated as providing a source value (1 or 0) or a condition (TRUE or FALSE) depending on the context. The flags are :-

OVERFLOW
CARRY
POSITIVE
NEGATIVE
EQUAL

Temporary Objects

As GAS Code is generated, it will become necessary to provide temporary storage objects to hold temporary results. It will also be necessary to provide temporary labels to implement such things as Case structures and loops. To allow for GAS Code to identify, use and dispose of these objects the following pseudo-GAS

Codes have been identified

a. USEVAR id

b. DELVAR id

c. USELABEL id

d. DELLABEL id

Encountering a USEVAR GAS Code means that storage is to be allocated for a data object of the Rcode Basictype. This temporary object can then be referred to using the id. When a DELVAR is encountered use of the id will no longer be valid. The same identifier may be used later for other temporary objects if needed. Encountering a USELABEL GAS Code means that the current Rcode node pointer should be noted and associated with the temporary label id given. Reference to the id will then be associated with the Rcode tree pointer. When the DELLABEL is encountered, the temporary label id becomes invalid.

GAS Data Types

Basictypes. It is assumed that the target machine can handle directly some of the "Basictypes" defined in Rcode. It is also assumed that the front-end of a compiler will generate only Basictypes which the target machine can handle. When generating GAS Code, extended arithmetic operations that appear in the Rcode should be broken into a set of operations involving the Rcode basic datatypes supported by the target machine.

Basictype operands are required for all arithmetic operations, and are specified by an operand that consists of two parts, the basictype and the object reference. The object reference identifies the value involved and may be an immediate value or identification of the memory location containing the value, or the address of a memory location (code or data). Object referencing is discussed below.

Blocks. Memory move GAS Codes (LOAD, PUSH, POP, IN and OUT) are operations on blocks of basictype objects as are TEST and SCAN. Blocks are specified by one operand that specifies the number of basictype objects in the block, and another operand that specifies the first basictype object in the block. If the block is considered to be just a series of bytes, the length operand should specify the number of bytes and the second operand should specify a basictype object that is unsigned and one byte in size.

If a block is specified in such a way that the size results in running out of real target machine data space, the program is *erroneous*.

NOTE In practice the exact effects are target machine dependent.

c. **Shift Mode**. Used to indicate the type shift in the SHIFT instruction. It is one byte coded with a value Arithmetic, Logical or Rotate.

d. **Alignment**. Used in PUSH instruction to specify the alignment required for an object to be pushed. It is a byte coded using the format specified in the Rcode Reference Manual.

e. **Byte Cardinal**. This is a single byte providing an unsigned binary exact value. It is used in the SVC instruction to specify both the SVC code and how many arguments follow.

f. **General**. Provides identification of a Basictype value or bitstring to be used in a GAS instruction.

General Operand Forms

These operands consist of a byte coded according to the Rcode "Basictype" definition, a byte coded to indicate the variant form involved, and a number of bytes that depend on the variant. Before describing the variant forms, various sub-fields of these operand descriptors must be described.

a. **Immediate Sub-field**. This provides an immediate value. It consists of one byte that is a binary unsigned value in the range [0..7] that specified how many bytes follow, then this number of bytes follow. The first byte is the least significant byte.

b. **Display Field**. This is a one byte value coded to identify one of the components of the GAS Display vector.

c. **Temporary_ Id**. This is a two byte value coded as an unsigned binary exact value indicating the GAS generator assigned identification for a GAS temporary label or storage location.

d. **Direct Basictype**. This is a sub-field used to specify a value that indicates the size of a bitstring or an offset of an object in a storage location (object, byte and bit offsets). These values may be static or computed. If computed, they must be stored at a location whose offsets are definitely given as static values. This sub-field consists of a byte coded according to the Rcode "basictype" definition, a byte coded to indicate the variant form of the sub-field, followed by a number of bytes dependent on the variant. The variants are coded as follows -

(1) **Immediate**. Coded as Immediate sub-field.

(2) **Dynamic**.

- Rcode Lexical Level (one byte)
- Display component (coded as display sub-field)
- Object offset, Byte offset, Bit offset
  all coded as Immediate sub-field

(3) **Static**.

- Rcode Areanumber (one byte)
- Byte, Bit offsets (coded as immediate subfields)

(4) **Temporary**.

- Temporary id (coded as Temporary_ Id sub-field)
- byte, bit offsets (coded as immediate sub-fields)

(5) **Imported**.

- Rcode Moduleid (one byte)
- Rcode Areaid (one byte)
- byte, bit offset (coded as immediate sub-fields)

e. **Code Object** This identifies a code object. The value returned is the address of the code object. Consists of one byte coded to indicate variant form of this sub-field, followed by a number of bytes depending on the variant

(1) **Routine**.

- Rcode routineid (one byte)
- Rcode Lexical level (one byte)

(2) **Code**.

- Rcode Labelid (one byte)
- Rcode Lexical level (one byte)

(3) **Temp**.

- Gas assigned Temporary label id (coded as Temporary_ Id sub-field).

(4) **Imported**.

- Rcode Moduleid (one byte)
- Rcode Routineid (one byte)

Tha variant forms of a General operand are :-

a. **In**. This variant provides a value from which a basictype object is to be taken. If the value is an immediate series of bytes or an address, the basictype value will be taken starting at the least significant bit. If the value is specified

by a reference to a bit in memory, the basictype value will be taken starting at this bit. Whether this bit is assumed the most or least significant bit is machine dependent and depends on how objects are packed. If the value provided is an immediate value or an address and is longer than required for the basictype, only the least significant part is used. If the value is not long enough, missing bits are assumed to be sign extended, based on the basictype type involved. The variant part of the In form is coded :-

(1) **Immediate**. Coded as an Immediate sub-field.

(2) **Special**. Coded as a Special sub-field.

(3) **Target**.

- is_ address. This is a one byte BOOLEAN value indicating if the value to be used is the address of the memory location identified (TRUE) or the contents (FALSE). If TRUE the bit identified must be located on a byte boundary in memory (not a temporary storage location or display pointer).
- Target location specifier coded as specified for the "Out" variant (see below).

(4) **Code**. Value is the address of a code object identified. Coded as Code Object sub-field.

(5) **Constant**.

- Rcode Constantid (one byte)
- Rcode Lexical level (one byte)
- location (coded as Direct Static sub-field)

b. **Out**. Provides identification of a storage location for a basictype object. The identification is given by identificying the bit that marks the beginning of storage used for the object. The contents are required or will be changed, or its address is required (excludes display pointers and temporary locations). It consists of an extra offset field coded as a Direct Basictype field, a byte indicating "indirection" and which is coded as a binary value, a byte coded to indicate variant form, and a number of following bytes that depend on the operand form and which identify a bit in storage. The "indirection" byte is coded TRUE if this bit marks the beginning of a stored address that identifies the actual storage bit required, else it is FALSE. The extra offset is a byte offset that identifies the offset from the bit identified (after the possibility of indirection) of the actual bit required. The variant part identifies a bit in storage coded as follows :-

(1) **Target**. Bit located in display pointer

- Rcode lexical level (one byte)
- Display pointer identity (coded as Display Field sub-field)

(2) **Dynamic**. Bit located in stack object

- Rcode lexical level
- Display pointer identity (coded as Display Field sub-field)
- Object, byte, bit (coded as Direct Basictype sub-field)

(3) **Static**. Bit located in a static storage area in current module

- Rcode Areaid
- Byte, bit offset (coded as Direct Basictype sub-field)

(4) **Temp**. Bit located in a GAS temporary storage location

- Temporary identity (coded as Temporary_ Id sub-field)
- Byte, bit offset (coded as Direct Basictype sub-field)

(5) **Imported**. Bit is located in static storage area of another module

- Rcode Moduleid
- Rcode Areaid
- Byte, bit offset (coded as Direct Basictype sub-field)

c. **Fixed**. This is an immediate fixed value, coded as an Immediate sub-field.

d. **Ident**. Basictype value is taken from or stored starting at the least significant bit of the GAS temporary storage location specified. Coded as a Temporary_ Id sub-field.

e. **Bit_ In**. This refers to a bitstring used for a GAS Code source operand information. It consists of two parts. The first part provides a value which specifies the length of the bitstring in bits, coded as a Direct Basictype sub-field. The second part provides the reference to the value from which the bitstring will be taken, coded in the same way that the "In" variant is coded.

NOTE If the value is an immediate or memory address value, and is longer than required, the bitstring will be taken starting at the least significant bit. If the value is too short the "missing" bits will be taken as zero.

f. **Bit_ Out**. Identifies a bit in storage to which a bitstring will be written. It is of the same form as the Bit_ In variant.

g. **NIL**. Provides NIL reference when no object is required (pushing a NIL

value). Has no further components.

Gas Instruction Descriptions

Arithmetic Instructions

All "source" operands should be Basictype objects specified by General operands of the "In", "Fixed" or "Temp" variant form or Condition operands. The destination must be of the "Out" or NIL variant of General operand. Therefore all operands will be basictype objects.

All operands must begin on byte boudaries, ie the bit offset must be zero.

For all arithmetic operations if the operation results in a positive value then the GREATER condition will be set to TRUE. If the result is negative the LESS condition will be true. If the result is equal to zero the EQUAL condition will be set.

If the size of the result produced exceeds that of the data type specified for the destination operand, the overflow condition will be set TRUE, and the result will have the most significant part truncated. The datatype for the destination operand will be set by the GAS Code generator to the Basictype specified in the Rcode instruction form which the GAS Code arithmetic instruction is derived.

NOTE Note that the Rcode Basictype bit 6 specifies information on how to handle overflow on integers and bit 7 specifies whether to truncate or round for real arithmetic. The GAS code generator should generate appropriate code.

ADD_

source_ 1 source_ 2 destination

Adds source_ 1 and source_ 2 and places result in destination.

SUB

source_ 1 source_ 2 destination

Subtracts source_ 1 from source_ 2 giving destination.

MULT

source_ 1 source_ 2 destination

This multiplies source_ 1 by source_ 2 giving the result in destination.

DIV_

source_ 1 source_ 2 destination

This operation divides source_ 1 by source_ 2 giving the destination.

NEG

source destination

Takes source operand and negates it (based on the data type) and places the result in destination.

Logical Instructions
AND_

source_ 1 source_ 2 destination

Performs a logical AND on the data in source_ 1 and source_ 2 and places result in the destination. The source_ 1 and source_ 2 operands must be of the General form. The destination must be NIL, "Out" or "Bit_ Out" variants only of the General operand form. If the destination is a Basictype the Positive, Negative or Equal conditions will be set according to the result placed in the destination. If the result is too large for the destination, the least significant bits will be placed in the destination, and the Overflow flag will be set to true. If either of the source operands are basictypes, they will be treated simply as a bitstring for the purposes of the operation.

XOR

source_ 1 source_ 2 destination

As described for AND but performs logical XOR operation.

NOT_

source destination

Inverts all the bits in source and places result in destination. The source operand must be of the General form. The destination must be of either the "Out" or "Bit_ Out" variant of the General form of operand. If the destination is specified as a Basictype operand, will set the Positive, Negative, and Equal conditions based on the value placed in the destination. It the result is too large for the destination, only the least significant bits will be placed in the destination and Overflow will be set TRUE.

TEST

blocksize operand_ 1 operand_ 2

Compares the block of Basictype objects starting at operand_ 1 and the block of Basictype objects starting at operand_ 2. The size, operand_ 1 and operand_ 2 must be of the General operand form. Both operand_ 1 and operand_ 2 must be

refer either to bitstrings of the same size, or Basictype objects of the same type and size. If the blocksize value is "one", and the operand_ 1 and operand_ 2 are Basictype objects, the values will be compared and the Positive condition set TRUE if operand_ 1 > operand_ 2; the Equal state set TRUE if they are equal; the Negative state set TRUE if operand_ 1 < operand_ 2. If the size is greater than one, the block comprising a series of Basictypes of length as specified by the blocksize operand will be tested only for equality and the Equal condition set accordingly. If the operands are bitstrings, they will also be only be tested for equality.

Special Instructions
HLT

With no operands this operation terminates program execution.

TEST_ AND_ SET

operand

Tests the bit at the address specified by operand, and sets to one if zero. This is an *indivisible* operation. If the bit was zero then the EQUAL condition will be TRUE, otherwise it will be FALSE. The "operand" must be a General operand of "Bit_ Out" variant. The bit involved will be the first bit in the Bitstring specified.

RTI

With no operands this operation returns from an interrupt.

IN_

blocksize, port, location

This will move a block of Basictype data objects from the port specified to the storage location starting at the Basictype object specified by the location operand. The port operand must provide a Basictype value. Whether this value is a valid port address for the target machine is for the user to ensure. The location operand must be an "Out", "Temp" or NIL variant of the General operand form. The blocksize operand must provide a Basictype value, and specifies how many Basictype values are to be transferred.

OUT

blockize, port, location

Moves data from the location specified to the port specified. Similar to the IN operation.

STATESET

regno, source

Moves the Basictype object specified by source, into the special register specified by regno. The source must supply a Basictype value. Regno must be a "Fixed" variant" (immediate) of the General operand form.

NOTE Encodings of possible register codes are contained in the Reg_ Kind enumerated data type contained in the portable project runtime library MACHINE module.

STATEREAD

regno, destination

Moves the Basictype object from the special register specified to the storage location specified by destination, otherwise similar to STATEREAD.

SETINT

new_ state

new_ state is an "Interrupt State" operand form.

SETINT Disable

is used, for the following GAS Code all maskable interrupts are masked, until

SETINT Enable

when the previous masking state will be restored

SVC

code, no_ ops, operand, ...

Used to make use of services provided by an operating system kernel. These are calls that cause a "wakeup" of the kernel, effectively a software interrupt of the kernel. The following two codes have been reserved

254 Save Context
255 Load Context

The SVC code is a "Byte Cardinal" operand form and provides an immediate value, in the range [0 .. 255].

The no_ ops is a "Byte Cardinal" operand that specifies how many operands are following.

The following operands can be any operand of the General form.

Jump Instructions
BRC

condition address

This instruction branches to the address specified if and only if the condition specified is TRUE. The condition will be Positive, Equal, Negative, Overflow or Carry. The address operand may be any General operand that returns a Basictype value. Whether the address is valid is a problem for the user.

CALL

condition address

This instruction calls the code at the address specified provided that the condition is TRUE. The condition will be Always, Positive, Equal or Negative, Overflowed or Carry_ Set. It is equivalent to the Rcode FAST_ CALL. A return block consisting of the return address and any status bits pushed by the target machine call instruction, is pushed on the stack before the routine is called. The address operand may be any General operand that returns a Basictype value. Whether the address is valid is a problem for the user.

RET

This instruction is equivalent to the Rcode Fast_ Return. It returns through the target machine return block on the top of stack.

PUSHMARK

This instruction will

a. Create a new Display vector

b. Current TOP_ OF_ STACK → new RESULT

This assumes that an object of the size/alignment of the result area is on the top of stack.

c. Push a NIL object of the size and alignment required for a return block used by the return from subroutine target machine instruction that will be used for actual subroutine return.

d. TOP_ OF_ STACK → new RETURNMARK

e. Current TOP_ OF_ STACK+1 → new ARGLIST

NEWDISPLAY

This code sets up a new display, and must be the first instruction in a subroutine. It performs the following

a. Make most recently created display active.

b. Previous TOP_ OF_ STACK → new TOP_ OF_ STACK

c. Make PREVDISPLAY of the new display point to the previous display.

d. POP return block on top of stack to location pointed to by RETURNMARK

e. POP the return block now on top of stack into ROUTINEDESCR of current display.

f. Move TOP_ OF_ STACK contents to LOCALSTORE of current display

POPMARK

This performs the action of

a. Current RETURNMARK → previous TOP_ OF_ STACK.

b. Delete current display.

c. Make previous display active.

Multiple Instructions
SCAN

blocksize target search_ for result

This instruction expects target to specify an Unsigned Basictype object of one byte in length. Together the blocksize and target identify a block of memory to be searched. The search_ for operand identifies a Basictype object provides a mask value to be searched for in the target block. The search begins at the byte specified by target and continues until a match is found, or the number of bytes left in the block is less than the size of the search mask value. If the mask is found, its start address is stored in result. If not found, result will be 0. The result should be a Basictype large enough to receive a pointer value, or truncation of the most significant bits will occur.

XLATE

source destination

Converts the Basictype value provided by source to the basictype specified for

destination, and places the result in destination. Note that this instruction can be used for sign extending operands. If the result is too large for the destination, the Overflow condition is set and the most significant bits are truncated. Such programs will be unpredictable.

Operand Movement Instructions
PUSH

size source

Pushes the object, whatever its size, into the next object position on the top of the stack, but allowing for the required alignment of the new object. The TOP_ OF_ STACK pointer of the current display vector will be increased by ONE. Note that source could provide a NIL reference.

POP

operand

Will Pop the object on the top of the stack into the location specified. The operand must be an "Out" or "Temp" variant of the General Operand form. The TOP_ OF_ STACK pointer of the current display is decremented by ONE. The object popped may consist of a block of objects of the Basictype specified by operand. There the POP will result in a block of objects being placed in the location specified by operand.

LOAD

blocksize source destination

Moves the block of Basictype objects specified by blocksize and location to the location specified by destination. The destination must be an "Out" or "Temp" variant of the General type operand. Note that the source operand could provide a NIL reference.

SHIFT

shiftmode shiftsize operand

Shifts the data in the operand, by the number of bits specified in the Basictype value provided by the shiftsize operand, and using a shift mode as specified in the shiftmode operand. If the shiftsize value is negative, the shift is left, if positive the shift is right. The allowable shift modes are Arithmetic_ Shift, Logical_ Shift, and Rotate. The operand must be a "Bit_ Out" variant of the General operand form.

EXTRACT

source destination

Moves the bits of source and places in destination. The source operand must be a "Bit_ In" variant of the General operand form and the destination operand must be a "Bit_ Out" variant of the General operand form. If the size of the source exceeds that of the target, only a bitstring equal to the length of the target will be placed in the target. The remaining bits of the source will be ignored.

Pseudo Instructions

USEVAR

id

The "id" operand is an immediate value that contains the identity of a new temporary data object.

DELVAR

id

Specifies that the temporary object specified by the immediate value in "id" no longer exists.

USELABEL

id

The "id" operand is an immediate value containing the identity of a new temporary code label.

DELLABEL

id

Specifies that the temporary label specified by the immediate value in "id" no longer exists.

Appendix B

Machine Family No 1 GAS code Users Guide

1 Introduction

This manual provides background information describing the concept and use of Generic Action Set Code for the Machine Family No 1. The Reference manual provides detailed specifications of the GAS Code and GAS Machine for the family of target machines involved.

The Portable Language Implementation Project which led to the production of this manual, involves the development of a software production system that is easily portable to a wide range of target machines. This is achieved by the use of a compiler system that can be easily adapted to compile a range of languages and to produce code for a range of target machines. Parts of the compiler which are specific to a language, machine or operating system are kept to the minimum absolutely required and any such dependency is confined to one kind per module (eg only machine or only language). Careful design has ensured that each form of dependency only involves a few modules.

To assist in this, the Modula-2 language pseudo-module SYSTEM has been slightly modified to include low level mechanisms for handling machine registers, input/output and execution context. The runtime library which forms an essential part of the portable system includes facilities for concurrency, message and event-based synchronisation and exception handling in addition to the more traditional input/output facilities.

For all languages, the front end of the compiler will produce a common intermediate code, Rcode (see the Rcode User's Guide and the Rcode Reference Manual). This is a tree-structured language. The front end considers storage element sizes in the production of Rcode, as the only target machine related parameters.

The back-end of the compiler has to produce target machine code. For machines which are architecturally very similar, the compiler back-end code involved is very similar. To take advantage of this fact, the concept of a Generic

Action Set code has been developed. This Generic Action Set code (or GAS Code) is a *second* intermediate code. It is a linear machine-code language which is hung on the Rcode tree to replace relevant Rcode segments. The "instructions" of the GAS Code and the actual GAS Code "stream" produced from the Rcode will reflect the general nature of code generated for any target machine that has an architecture similar to that of the GAS Machine.

Consider, as an example, the production of GAS Code for a high-level language assignment statement such as

```
K := (X + Y) * Z
```

For a real target machine with registers the GAS Code generated would be of the form

```
ADD   X Y TEMP1
MULT  TEMP1 Z K
```

During generation of machine code for a typical family member, this GAS Code fragment could be converted to

```
LOAD R1, X
ADD R1,  Y
MULT R1, Z
STORE R1, K
```

For a zero address real machine on the other hand, the GAS Code generated would be

```
PUSH  X
PUSH  Y
ADD
PUSH  Z
MULT
POP   K
```

Consider, as a second example, the following

```
Z = ROUT (X,Y)
```

The GAS Code produced in a machine with registers and a hardware stack facility could then be

```
PUSH  RESULT
PUSH  X
PUSH  Y
```

```
CALL  ROUT
STORE RESULT Z
POP   RESULT
```

where RESULT is an "area" reserved for results of procedures that return a result. This could be converted almost directly to target machine code as

```
PUSH  R0
PUSH  X
PUSH  Y
CALL  ROUT
STORE R0,Z
POP   R0
```

As can be seen from these simple examples, the conversion to GAS Code from Rcode is common to all target machines in a "family". The GAS instructions represent what is typical of the instructions of the machines in the family, and the GAS Code generated generally reflects the kind of target machine code which will eventually be generated.

The GAS instructions to represent a family will have the form of a reduced instruction set machine. This is because :-

a. Individual machines in the family will have special instructions for efficiently implementing certain functions. There is no point including "specials" in the GAS instruction set, only the general instruction categories need be represented. When converting GAS to target machine code, these "special" instructions in the target machines may be used, although various workers have argued in favour of adhering to a limited set of instructions even when such "specials" are available.

b. The GAS instructions are orthogonal in the sense that operations such as ADD for all data types are implemented by one generic GAS instruction, with the type of data involved indicated within each extra operand. In many machines, the number of instructions is greatly increased by a separate instruction for each data type (and size!).

c. The number of objects for an instruction such as ADD is fixed. For example ADD assumes four operands -

```
ADD source_ 1, source_ 2, result
```

In many machines there are several instructions for ADD, one for two operands one for three, etc.

d. The GAS Code operations in this family are at most binary. This rules out instructions such as

```
ADD  X, Y, Z, K
```

ie Add X, Y, Z giving K.

2 Machine Family Definition

The concept of family in relation to GAS Code generation, refers to the classification of machines according to the general architectural features of target machines in that family. Machines in the same family should give machine code for a program that is very similar in nature. The major factors that influence the form of machine code produced are :-

a. Does the machine support zero (stack based), one, two, or three address operations or a mixture?

b. Does the machine hardware support PUSH, POP and stack pointer relative addressing?

c. What is the general range of instructions provided - ie, ADD, DIV, MOVE, BNE?

d. Does the machine only support arithmetic on statically sized objects, and do the objects have to be byte aligned (not bit)?

Other factors can affect the code produced. The most important of these factors is the data types supported by the machine. However, this is an area of great diversity among machines, and is not therefore an appropriate factor to be used for differentiating machine families. For a factor to be useful in discerning families, it must be clearly present or not present on machines. Many machines either have registers or don't, have zero-address operations or don't, do or don't have PUSH and POP instructions. However, very few machines have identical or even very close data structure support. Data structure support is so complex and important, that even the Rcode generator has to know what data sizes are available in the hardware in order to allocate space reasonably. Rcode generates instructions involving what it calls Basictypes, which are basic exact, (binary coded) decimal and real types of various sizes. Simple Rcode will only be generated for datatypes which are directly supported by the target machine.

The front-end may also generate (block exact arithmetic) Rcodes which use operands indicating a multiple hardware-sized object. The GAS Code generator

must in this case perform the operation in a series of stages, using the Basictypes supported by the target machine. This is because the machines in the family are only assumed to provide arithmetic operations on Basictype objects. This is a significant machine factor that can be used to discern families. It is present or it is not.

NOTE A machine is considered to directly support an operation on a given data type, either by having a direct machine hardware instruction, or by providing a software emulating system call.

Another significant factor for machine code generation is whether indexed addressing is supported. However this has more of a local minor effect on the code generated and is not included as a factor for family classification of machines. The use of index registers and allowable offset sizes is very machine dependent and is, therefore, best left to the stage of converting GAS-code into to target machine code. The GAS Code will have the form :-

```
ADD ... refer_ variable+offset, ...
```

If a machine supports indexed addressing this will convert to

```
LOAD    IReg, variable address
LOAD    Reg, (IReg+offset)
ADD     Reg, ...
```

whereas if the machine only supports indirect addressing, the machine code will take the form :-

```
LOAD    Reg, variable address
ADD     IReg, offset
LOAD    R, (Ireg)
ADD     R, ...
```

The GAS Code produced does not preclude either alternative above. However if the GAS Code generated was of the form

```
PUSH    A
PUSH    B
PUSH    C
ADD
MULT
```

and the target machine did not in fact support zero address operations, but had registers, conversion of GAS Code to target machine code would be very difficult. The GAS Code addressing modes should, however, provide sufficient information

on addressing modes so that address computation structure can be easily identified, allowing the target machine to select efficient addressing modes of the target machine.

The above example clearly demonstrates that factors used to differentiate families of machines must be such that they significantly affect the form of machine code produced; the presence or absence of such a factor on a machine must be clearly discernible.

3 Block Structure and Scope

Many high-level programming languages have what is referred to as block structuring. The implication of this is that blocks of code are defined which are lexically contained within other code blocks. A code block can declare it's own objects. Some form of "scope" rules are applied - typically such that the code in a block can access it's own local objects and objects in blocks within which it is lexically contained. Objects declared inside blocks that are lexically contained within a block are "invisible" to the containing block.

This block structuring is also implemented in conjunction with an extended subroutine structure where each code block is named and can be invoked by the use of this name. Arguments can be passed at any such invocation. Implicit in a call is that space for local objects of the block called is automatically allocated.

When the execution of a code block terminates, space allocated for local objects of the block and any arguments are assumed to be released, control returns to the instruction in the outer calling environment after the one which invoked the code block. Those code blocks that can be called from any point in a program follow the same scope rules as defined for access to objects. That is, code blocks themselves are considered to be objects. A code block which is lexically enclosed in a code block is considered to be a local object of the enclosing block. As a code block is assumed to be able to access local data objects in all code blocks which enclose it lexically, and as the local data objects are only assumed to exist for the time of activation of a code block, then it is necessary that when a code block is active, all it's lexically enclosing code blocks are also currently active even though possibly suspended. When recursion is allowed there can be many current activations of a given code block. In this case the interpretation of "in scope" data objects is dynamic. When a given code block wants to access a data object in some enclosing block, but there are several current activations of that code block, the data object accessed will be the one in the most recent activation of the code block.

Also related to code blocks is the question of error traps provided by many real

target machines, such as "divide by zero". To enable programs to recover themselves when such errors are encountered, a programmer will often include code to capture the trap. However, the code required in each code block may be different. On entry to a code block the programmer will set up appropriate code to handle traps. On return the handler code in force on entry must be re-instated. Hence such handler code is related to code block structuring.

Many processors provide special hardware instructions to help in the implementation of this block structuring. These relate to such things as :-

a. Allocating space for local data objects when the code block is called.

b. Providing facilities for accessing data objects of lexically containing code blocks. This is often referred to as a static link mechanism.

c. Providing facilities for accessing arguments and result areas.

d. Providing facilities for deallocating space used for local data objects and arguments, and for reinstating the environment of the caller code block and returning to the instruction after the call. This is often referred to as a dynamic link mechanism.

e. Provide facilities to save registers on entry to a subroutine and then reinstate saved values on exit.

f. Provide facilities to save the program status. In the GAS Machine this comprises the Less_Than, Greater_Than, Equal, Carry_Set, and Overflowed status "flags". On exit from the subroutine, the saved values should be reinstated.

g. Provide facilities for managing exception handlers on entry/exit. On entry to a subroutine, the current exception handler should be saved, and on exit the saved handler should be re-instated.

The use of any facilities provided in the target machine will usually help to implement block structure constructs efficiently. However, the type of facilities offered by many processors are varied, and often only cover certain of these issues. For example, the DEC VAX processor "CALLS" instruction provides facilities for easy management of arguments, register saving/reinstatement, dynamic linking, status saving and reinstatement, and exception handlers. However, it does not provide automatic allocation/deallocation of space for local data objects or for automatic maintenance of access paths to objects in lexically enclosing code blocks.

Code Block Environment

The GAS Machine must be able able to cater for a range of target machine facilities for implementing block structuring. It must therefore provide a fully conceptualised implementation of block structuring, not a partial implementation.

The basis of the GAS machine support for block structuring is the concept of the "environment" of a code block. An environment consists of :-

a. Its local data objects.

b. Its arguments list; the arguments passed to it when called.

c. Its' result area; where to place any result value that is to be returned on completion of the block.

d. The environments of lexically enclosing code blocks. This is required to allow access to objects in these blocks in accordance with the scope rules.

e. The environment of the code block which called it. This is required for returning on completion of the current code block.

f. The currently active exception handler routine for the code block.

g. The current program counter value for the code block.

The register contents are also part of a machine environment, but this is very target machine related and is not important at the GAS machine level.

The environment of a code block is maintained in a display vector. This consists of a set of pointers to objects on the code stack. The display vector interacts with the PUSHMARK, POPMARK and NEWDISPLAY GAS Code instructions. The way in which the Display and these GAS Code instructions interact in the operation of the GAS Machine is detailed in the "GAS Code Reference Manual". The access to data objects in outer lexically enclosing blocks is assumed to be provided automatically, given a display pointer identity, lexical level, and offset from the pointer contents. The GAS Machine therefore does not bias in favour of any particular method of implementing such an access (this is left up to the target machine code generating stage where techniques will be used that make use of any special features provided by some particular target machine), but provides sufficient information to allow any method of implementation to be used.

The importance of providing these runtime structures is that GAS Code can represent the steps involved in implementing subroutine calls in the family, the structure of the subroutine call is still clearly described (the Call and Declare_Routine Rcodes may not be retained by the GAS Code generator), and a standard calling convention and stack structure is provided which the target code

generator can attempt to provide as closely as possible regardless of target machine, providing in so doing a structure that will assist with portability of any code that manipulates the stack.

Datatypes Supported

The data type support provided reflects the support typically provided by the machines in the family, except that for a data form for which there is no "typical" family support, the structure of objects of this form should be clearly retained so that efficient target code can be selected. An example of this in family No 1 is bit-strings. Target machines in the family provide a wide variety of facilities for bit-string operations. Therefore in the GAS Machine bitstrings can consist of either a statically or dynamically defined number of bits. No restrictions are placed on possible size of the bitstrings other than by some implementation restriction which typically uses a 16-bit object to indicate the number of bits. If the target machine provides limited operations on statically sized bitstrings that are multiples of bytes, or a wide range of operations on dynamically sized bitstrings with or without a size limitation, this is a problem for the target code generator. For machine family No 1 the types supported are -

a. **Basictype Objects**. These reflect the fact that the machines in the family typically provide operations on static sized objects that are multiples of bytes. The Rcode Basictype coding scheme therefore provides the ideal mechanism for describing these objects.

b. **Blocks of Basictype Objects**. Provision of these reflects the fact that many of the machines in the family provide operation on blocks of memory, considered either as a block of bytes or a block of Basictype values. The operations typically provide block moves (LOAD, PUSH, POP, IN, OUT), compare and scan operations. Again the facilities provided on the machines in the family vary so that it is necessary to retain the full structure of the block in the GAS Code operand. Some target machines in the family may provide limited operations on blocks of bytes only, whereas some will provide operations on blocks of Basictype objects.

c. **Bitstrings**. Strings of arbitrary numbers of bits are frequently manipulable by target hardware in a restricted way.

Temporary Objects

A temporary storage reference involves an operand which is a reference to an object which has been allocated by the GAS Machine to temporary storage. These records are created when a temporary result or value requires storage. Such

temporaries are results generated whenever an Rcode subtree is required to return a result. When converted to GAS Code, the code for the subtree will be generated with the value to be returned left in a temporary. The value in the temporary is used either in generating target code or working out addressing modes for the node with the subtree. The subtree could involve the generation of the length of an operand, or the initial value of an index of a "FOR" loop. Temporary storage declarations contain the "id" of the element and the machine data type to be stored.

When a temporary storage object is required, the USEVAR GAS Code will be generated and inserted in the Rcode tree. The unique-id allocated to this object is used to reference this object. The DELVAR is used to allow deallocation of the entity database record for this object, and also for the generation of target machine code as temporary objects are obviously very important in register allocation, during final conversion from GAS Code to target machine code.

Temporary objects may be allocated space on the stack or to a register, or to global storage. The choice depends heavily on the target machine architecture and the lifetime required for the temporary object. The "temporary data store" concept therefore clearly identifies objects that are temporary and the USEVAR/DELVAR pseudo GAS Codes provide the information needed for their efficient allocation/deallocation.

The use of temporaries can be considered to be the equivalent of modelling a machine with an unlimited number of registers. Whenever a temporary storage location is needed, a register can always be found, that is a temporary in GAS Machine terminology. Extending the analogy with machine registers, it is assumed that "currently active" temporary data objects are saved when a CALL GAS Code is encountered.

Note that with block structured languages, a temporary data object is normally considered to be local to the code block in which it is defined. If the block can be called recursively, then at one time there can be several instances of the temporary. When a temporary is referenced, the most recent instance of the temporary is required. The GAS machine automatically achieves this.

In the target machine, a temporary can only be allocated to a global storage location if it is not possible for there to be more than one instance of it at one time. This is definitely true if there is no CALL GAS Code between the USEVAR and DELVAR for the temporary. If there is, the temporary must be allocated space on the stack or to a register. This ensures that the most recent activation of a temporary is accessed when a temporary is referenced, because if the object is allocated on stack then the temporary in the current frame will be accessed, and if that temporary has been allocated to a register then the register will only hold data

for the currently active code block.

Registers for active but suspended code blocks are saved on the stack. A temporary can be moved between stack and registers. The entity database record must be kept up to date by the code generator to show where it is located at all times.

When the DELVAR GAS Code is encountered, the storage for the temporary can be deallocated, but this may not happen immediately. If temporaries are created and deleted on a nested basis, storage can usually deallocated immediately by popping them off the stack, since the temporary to be deleted will always be on the top of the stack. The details of handling this is a problem for a GAS Code interpreter or target machine code generator.

A temporary jump label record is created to implement such high-level language constructs as WHILE, IF-THEN-ELSE, CASE, for which a jump to a label is required. The USELABEL pseudo-GAS Code is added to the Rcode tree when a temporary label is required, and a DELLABEL pseudo-GAS Code is added when the label is no longer needed, so that any GAS entity database record can be deallocated. The information required for these jumps is different from Rcode labels which are source language related entities and have additional information such as lexical level. Jump labels are merely the entities required for implementing this kind of control flow structure and the information required is simpler. Hence the use of separate records of jump labels.

Immediate Objects

Immediate objects occur when a value has been statically evaluated by the back-end. These objects can be 1,2,4 or 8 byte length objects, and the datatype must be one of the Rcode basic datatypes.

Standard Conditions

Standard elements are used to implement the characteristic of many machines to allow extended size arithmetic to be implemented via several steps, with the use of carry and overflow flags to allow carry over from one stage to the next, and also to control the flow of control depending on the result of previous instructions

a. CARRY - if previous exact arithmetic produced a carry, then the standard element CARRY will contain 1, else it will contain zero. For example in the sequence

```
ADD     X, Y, Z
ADD     A, CARRY, M
ADD     M, B, RESULT
RET
```


In this example X and Y are added to produce Z, the least significant part of a double precision add. The next two add instructions, add the most significant components and any CARRY from the least significant add. The result is placed in the RESULT standard storage area since it is the result of a function call.

b. OVERFLOW - This last operation resulted in an overflow.

c. POSITIVE - The last operation resulted in a "Positive" value, or in a test operation the first operand is larger than the second.

d. EQUAL - The last operation resulted was test of two equal values or the result was zero.

e. NEGATIVE - The last operation resulted in a negative value, or in a test operation the first operand in less than the second.

Machines in family No 1 provide an overflow and carry bits that provide the basic mechanism for multiple-precision arithmetic operations. They also provide the equivalent of Positive, Equal and Zero flags and these are used to provide the main basis for control of program flow.

In many machines, some flags reflect factors intrinsic to the machines own architecture. From a conceptual, non machine specific view, the only significant flags are interrupt enable/disable, carry/overflow flags, and Positive, Negative and Equal flags.

The GAS standard conditions can be used as booleans in tests, or to provide a "1" (TRUE) or 0 (FALSE) value. This is done to reflect the fact that there are times when "carry" can be used for example, to provide one of the source values in an addition operation and at other times can be used as the condition in a conditional branch. Many processors have a "zero" status bit but this is the same as "equal" where the previous operation would have set the "equal" condition to true if it had resulted in a zero result.

4 Run-time Entities

The term "entities" refers to objects whose declarations are encountered in the Rcode, or objects that are created during GAS Code generation, such as temporary labels. Additionally, objects will be identified that are important to the GAS Code generator and optimiser, and target code generator, but which are a sub-field of an Rcode or GAS Code entity. An example is a Basictype object located in an Rcode storage area or variable storage area. Such an object will be important to the GAS optimiser and the target code generator as it will no doubt be a candidate for register allocation. During GAS Code and target machine code generation, an entity data base will be maintained that records information about these objects.

If GAS Code is generated and the Rcode tree is written to disc, then any interpreter or target machine code generator that reads this Rcode file will have to reconstruct the entity database. All Rcode declarations for static storage areas and constants must be left on the Rcode tree to define storage areas, the task of identifying all objects important to the GAS Code optimiser and target code generator will be much easier if all declaration Rcodes are left on the Rcode tree. This of course will depend on the storage available. The subtrees of these Rcode nodes will contain GAS Code.

The "run-time" data base consists of a set of records, one record for each object encountered. Objects are of one of the following types :-

a. Rcode Variable storage areas - space allocated on stack.

b. Rcode Static areas - global space allocated.

c. Rcode Constants - also allocated global storage.

d. Labels - Rcode labels or GAS temporary labels.

e. Symbols.

f. Routines.

g. Temporary storage locations required by the GAS Machine!

h. Optimiser and target code generator "variables".

Entity records for Rcode objects should be constructed when declarations are encountered in the Rcode. Entity records for temporary data objects and labels should be created during GAS Code generation as they are required. They may be deleted when no longer needed. The GAS optimiser and target code generator should create and delete records for variables they identify as important. The entity records are used to relate the declaration of objects and their later use.

Note that the entity database records for Rcode objects have to be accessible via their Rcode Identification when GAS Code is being generated. For example when an Rcode REFER_ VARIABLE requires conversion to an object offset. They also have to be accessible via their object offset and lexical level when target machine code is being generated so that a GAS Code with reference to an object by object offset and lexical level must be convertible into a byte offset to the object or it's descriptor if dynamic in length.

5 GAS Instructions

Instruction Format

GAS-code instructions will have the format

instruction_ code operand_ 1, operand_ 2, ...

Each instruction has a fixed number of operands, except for the SVC instruction. For an instruction such as ADD, the number of operands in a register-based machine will be three

Operand_ 1	source operand
Operand_ 2	source operand
Operand_ 3	destination operand

This is the most general form for a BINARY operation. The add instruction code is therefore *generic*.

Rcode *vs*. GAS Code

The following paragraphs attempt to explain the relationship between the two intermediate codes - Rcode and GAS Code.

Rcode is a tree structured intermediate language. Operands of an Rcode operation can be defined by a routine which is a subtree of Rcode that returns the required operand value. An example is an Add Rcode where the first operand is obtained from a subtree that involves much arithmetic. Execution of Rcode would therefore essentially be recursive. When an Rcode instruction is to be executed, a halt in execution would be needed when a subtree is encountered, and execution of this subtree would be performed. When complete, the value returned (if any), would be used to continue the execution of the "suspended" Rcode instruction. At any time there could be a stack of suspended Rcodes. GAS Code, on the other hand, is essentially a *linear* code. It is generated by removing much of the tree structure of Rcode. GAS Code is generated for any subtrees of an Rcode instruction and any values returned by these subtrees are left in temporary storage

locations. When GAS Code has been generated for all subtrees, then the GAS Code will be generated for the Rcode itself.

Most "topological" Rcodes will probably be left on the Rcode tree by a typical GAS Code generator after GAS Code has been generated. The term "topology" is used here to mean Rcodes that help describe the semantic structure of the code. Examples of this are Loop, Case, For, and Call_ . This information will still be very useful during target machine code generation. However the subtrees of these Rcodes will now point mainly to linear sequences of GAS Code. The literal fields of these Rcodes are also retained.

Declarative Rcodes are also retained. This is because the entity database is not saved if the GAS Code is written to disc. As stated earlier, the database contains information on declarations encountered and is used for relating references to objects, to the declarations of objects, and also for recording offsets allocated to objects on the stack. Declarations include the size and alignment of objects. When a GAS Code containing tree is read from disc, this database must be regenerated. Therefore the reader will need to use the declaration Rcodes encountered to rebuild the database. Any sub-trees of these declaration Rcodes will contain GAS Code.

GAS Code nodes are implemented via the Reserved Rcode. Each of these Rcodes *is* a GAS Code instruction. The format of the Rcode will be :-

byte_ one :	The Reserved Rcode indicator
byte_ two, byte_ three :	Number of bytes of literal data from byte_ four to the end of this Rcode
byte_ four :	The indicator of a particular GAS Code
rest :	GAS Code operand bytes

If several GAS Codes are linked in a sequence, this will be done using the standard LINK Rcode.

Rcode Groups and GAS Code

Rcodes consists of several groups. This division into groups is relevant to the actions which should be taken by a GAS Code generator.

Group 0 This group is of no major significance to the back-end of a compiler and has no effect on a GAS Code generator. They are prelude operations used to reconstruct the Rcode tree when reading from backing store.

Group 1 This group contains declarations such as program units, variables,

constants.

They are used by the back-end to produce the GAS entity data base, and for the allocation of storage.

Group 2 These are used to obtain values as operands for Rcode instructions without actually using the contents of storage locations. These provide operands that are evaluable addresses and constants. They are such things as Refer-Constant, Refer-Variable, Refer-Result, Refer-Area, Literal. Refer-Variable(X) means use the address of the variable "X" as the operand. Refer-Area(F) means return the address of the area "F". If the variables are global, the operand for Refer_ Variable(X) will be statically evaluable, if not an instruction such as

LOADADDR @SP+4, R1

could eventually need to be generated in the target machine. In GAS Code the operand is a pointer to the GAS Code entity record for X, and the mode of addressing will be "address of".

Group 3 This group of Rcodes provides the facilities for generating operand values that are computed from values in storage locations. It also includes operations on storage locations themselves. An example of what is essentially an operand reference is "Block-Load" which returns a block of bytes at a specified address.

An example of an Rcode that involves operand manipulation is the "Block_ Store" code which will store a given value in a specified location.

The group also includes the Call Rcode. This will produce GAS Code for setting up the argument list, and Call of the procedure code.

This group of codes therefore involves operations on storage entities, that involve -

a. Returning all or part of the contents of a storage location - effectively operand specification.

b. Setting all or part of the contents of a storage location.

c. Invoking a procedure - a procedure is a piece of storage that is treated as code.

This Group of Rcodes will therefore produce in GAS Code

a. Operands of GAS Code instructions.

b. GAS Code memory reference instructions.

c. GAS Code instruction sequences to set up argument lists for and then to CALL procedures.

It is important in GAS Code to retain via addressing modes, as much of an address computations structure as possible.

Group 4 This group provides control logic information - Loop, For, Case, Goto, Link, Label. Most of these Rcodes are left untranslated on the Rcode tree, so as to provide vital information to the phase that produces target machine code from GAS Code. However, most of these codes will also involve the generation of BRC GAS Code instruction of some kind and the creation of destination label records (see notes on these above). Also included are Rcodes which control sequences of statements, statements that are to be executed in strict sequential order, or a set of statements that can be executed in any order or in parallel if the target machine supports concurrent execution. Information on whether statements must be executed in sequence or can be executed concurrently is very important to the phase that converts GAS Code to target machine code - which is why it is important that group 4 Rcodes are left on the Rcode tree.

Group 5 These are the codes which provide arithmetic features. They include codes to load/store values from/to storage locations, and control operations on shared storage locations. Some arithmetic operations will convert directly to single GAS Code instructions, whereas others will require several GAS Code instruction - for example, extended precision operations where data type sizes exceed the target machine capabilities. Also included in this group are facilities for runtime checks and error message generation.

Group 6 The Rcodes in groups 0 to 5 will be generated independently of the target machine or operating system. The back-end must support all of the Rcodes in these groups. Group 6 involves Rcodes which have semantics which are frequently dependent on the target machine or, possibly, its operating system. Some of these, therefore, will not be supported by all compiler back-ends. The Rcodes in group 6(a), however, are all required of every target machine and are therefore processed into GAS Code. Other Rcodes in this group are passed untouched to the phase that converts the tree with GAS Code into target machine code. It is only in this phase that details of the specific target machine are known.

Group 6(a) Rcodes are referred to as "Compiler Standard Extensions". These are generated by the compiler front-end, to implement such source language features as Modula-2's get and set register facilities, input and output, and service calls. These facilities will not necessarily be supportable on all target machines and the back-end will give error messages if they are not supported.

Other codes in group 6 will only appear if included in-line in the user's source program. This implies that the source language supports in-line Rcode. The front end will pass these codes on unprocessed, as will the back-end phase that generates GAS Code. These machine specific Rcode extensions provide the ability to take advantage of specific features of a machine. Only the user who writes the in-line Rcode, and the back-end phase that produces target machine code are involved in the implementation of these extensions. Therefore it provides a streamlined mechanism for taking advantage of machine specific features in a way that has minimal effect on the compiler code.

NOTE It should always be noted that any user program containing in-line machine specific Rcode extensions will not be portable to other machines, if at all.

Group 7 This group provides type information that is used for :-

a. Implementation of separate compilation by providing a data structure to record symbol table information.

b. Generating information to be passed to a source-language debugger. The back-end must therefore process these codes to produce whatever format is required for an interpreter or debugger to be used. These Rcodes will not be processed during the phase of producing GAS Code, but should be handled in the following phase, where the details of the target debugger or interpreter will be known.

Group 8 This group allows the possibility of extending Rcode to include either general purpose, or implementation specific extensions. Each extension code is to be followed a second code indicating the extension within the extension code group. GAS Code *is* the "Reserved" Rcode.

Special Instructions

Most of the GAS instructions are fairly self explanatory, because they are simple instructions. However, the special instructions group requires some further elaboration.

Input/Output. The IN/OUT instructions are used to code input and output Rcode instructions specified in group 6(a) "Compiler Standard Extensions". These codes provide a portable conceptualised representation of IO. They involve transfers between a storage area and an I/O port. It is up to the user programmer to ensure the port address is valid for the target machine, and that the port specified handles the data type specified.

If the source language supports in-line Rcode, then IO can be done with machine specific extensions, rather than with the IN/OUT instructions. However

this will make the program less portable to other machines. Most languages, such as MODULA-2, do not support in-line R code, and therefore group 6(a) Rcodes will be generated for IO.

Note that the port-id can be dynamically evaluable where the target machine supports this. This means that no checking can be made for a valid port-id by the back-end, and that the back-end should generate an error message when the target machine cannot support this facility.

State Set/Read. When a user program refers to a register (eg in MODULA-2 using the get and set register facility) the front-end will examine the register codings available in the portable compiler MACHINE module and determine if the register is a special register such as a status register or segment register. If it is, the front-end will generate an Rcode instruction for manipulating special registers, otherwise, it will generate the set/get general register Rcodes.

The data type is relevant in that the front-end checks data type length is not greater than the register length. If it is shorter, the back-end will generate appropriate code to replace the lower order bits by the source, leaving the remaining bits unchanged!

Note that special registers are treated differently from general registers. General registers are allocated by the automatic register allocation mechanism of the back-end. Special registers are not.

Set-Interrupt. This instruction is designed to implement Rcode instructions that must be uninterruptible (as opposed to critical code in which only mutual exclusion need be enforced), and to allow return from interrupt handlers to be specified. For example

```
Add_ In_ Place_ Locked address, value_ to_ add
```

could be coded in GAS Code as

```
ADD X, Y, X
```

however, this gives no indication that this must be uninterruptible. Therefore it *should* be coded as

```
SETINT   DISABLE
ADD      X,Y,X
SETINT           ENABLE
```

SVC instruction. This implements the Rcode group 6(a) "Service_ Call" instruction. For languages which support concurrency, for example, the compiler front-end will generate system calls for LOADCONTEXT and SAVECONTEXT. Other

calls will be generated directly only by languages that support in-line Rcode. In the latter case the back-end will have to be aware of the meaning of the system call code and generate appropriate code. This requires co-operation between the programmer and the compiler back-end, which is inevitable when writing machine-related system software.

The codes 254 and 255 are allocated the meanings SAVECONTEXT and LOADCONTEXT across all languages and target machines. The concept of SVC is important. System calls such as GETDATE, GETTIME, GETRECORD are effectively only subroutine calls, and are not considered SVC's. SVC's are where the operating system kernel is effectively asynchronously interrupted and forced to perform some action such as setting a timer, performing physical IO via OS routines, setting up and saving process data. Using the kernel for facilities such as GETTIME will be provided by a runtime library procedure.

Appendix C

Rcode Reference Manual

University of Waikato
Department of Computer Science

PORTABLE LANGUAGE IMPLEMENTATION PROJECT:
RCODE REFERENCE MANUAL

Introduction

This reference manual contains the formal specifications of the meaning and purpose of each opcode within the intermediate code known as Rcode. Any implementation is at liberty to choose values and sizes for parameters and opcodes themselves, except as may be herein defined. The values and forms used within the project of which this specification is a part are defined in the appropriate Modula-2 Definition modules used within the compiler structure.

The following opcode descriptions and formats are, unless otherwise stated, common to all target machines. Opcodes in Groups 6b, 6c, etc are specific to particular processor hardware, designed to allow a compiler front-end to use special hardware facilities where this is (rarely) necessary.

Object Alignment

Alignment of objects referred to in Rcode, wherever required, is specified as a single octet (byte of eight bits). This indicates that the address of the allocated storage must be divisible by the corresponding power of 2.

For example, 0 indicates arbitrary byte-alignment, whereas a value of 1 means that the address must be word-aligned (divisible by 2). These are the two most useful values on most microprocessors, since some classes of object are required to be word-aligned because of the machine bus architecture. Alignment values up to 3 (indicating quadword alignment) are those most likely to be used in practice; a value of 9 (indicating page alignment) is occasionally required for the beginning of storage areas.

Data Types

The primitive data types recognised in Rcode are *boolean*, *tristate*, *signed exact*, *unsigned exact*, *real*, *decimal* and *unspecified*. Except for boolean and tristate, these may come in different sizes. Particular opcodes are defined for operating on multiple-byte blocks of data, and on strings of bits.

It is the responsibility of a compiler front end to map operations on source language data types into appropriate Rcode opcodes. For the common simple types (eg boolean, integer, real), the correspondence is obvious; operations on more complex data structures will need to be built up out of the available Rcode primitives.

Boolean A boolean datum has two possible values: true or false. Considered as integers, these have values 1 and 0 respectively. An attempt to use any other integer value as a boolean gives undefined results. A boolean may be stored as a component of a packed structure in a single bit; unpacked, however, it takes up a whole byte.

Tristate A tristate datum, as its name suggests, has three possible values, represented by the integers -1, 0 and +1. Use of other integer values where a tristate is wanted gives undefined results. Tristate values are returned by the comparison operators, where they are used to denote 'less-than', 'equal' and 'greater-than' respectively. A tristate value allows concise expression of the state of CPU condition codes. As part of a packed structure, it requires two bits for storage; unpacked, a single byte.

Signed Exact Signed exact numbers (integers) may be represented in any appropriate form for the target machine. This is most often twos-complement form. At least two sizes (byte and word) of arithmetic are supported within Rcode. This may, of course, result in the generation of multiple length machine code arithmetic on those machines which do not support this in hardware.

Unsigned Exact Unsigned exact numbers (cardinal numbers) are available in the same range of sizes as signed ones, for all machines. However, it should be noted for machine code generation purposes that, on most machines, hardware arithmetic for unsigned numbers is restricted to addition and subtraction, as hardware multiply and divide instructions only operate on signed quantities.

Real Real arithmetic hardware is unavailable on many microcomputers. Where machine code generation is required, therefore, it will often be necessary to emulate hardware floating point facilities. The lengths provided for real numbers include at least 32, 64 and 80-bit (temporary) as defined in the proposed IEEE standard. Where hardware facilities for longer real arithmetic is available on any

particular machine then a size coding greater than 2 should be used.

Decimal Decimal arithmetic operates on blocks of packed decimal digits, each digit occupying four bits.

Unspecified Unspecified is the data type for which arithmetic is really exact arithmetic in another guise. However, whereas signed and unsigned exact arithmetic is meant to cause a run-time error should the result be outside the representable range, unspecified arithmetic merely returns the truncated result. For example, unspecified 8-bit, 16-bit and 32-bit addition and subtraction is the same as unsigned addition and subtraction modulo 256, 65_536 and 4_294_967_296 respectively.

Type Coding

Wherever required, the data type is represented in a single octet (byte of eight bits), encoded as follows :-

a. *Bits 0 .. 2.* These contain the size, interpreted for exact and unspecified values as octets times a power of 2, for reals in a standard way up to 2 (0 - 32-bit, 1 - 64-bit, 2 - 80-bit) - values of 3 and 4 are target machine dependent in interpretation. The size code must be 0 for boolean, tristate and decimal data. This is because unpacked boolean and tristate values always occupy 1 byte. For decimal data, the size is dynamically determined from that of the operands.

b. *Bits 3 .. 5.* These specify the data type, as follows:

0	- boolean
1	- tristate
2	- signed exact
3	- unsigned exact
4	- unspecified
5	- real
6	- decimal
7	- reserved

c. *Bit 6.* If the data type is a signed or unsigned exact, bit 6 equal to 1 causes the result of the operation to have, prepended to it, a byte contain a boolean indication of whether overflow occurred or not. Also, the signalling of a run-time error on overflow is disabled for that operation. Bit 6 equal to zero causes the operation to return its normal result, and to signal an error on overflow. For other data types, the use of this bit is reserved for conveying implementation-specific information. For the VAX, for example, if the datum

size is 1 and the type is 5 (indicating double-precision real arithmetic) this bit distinguishes between the two kinds of double precision available: it is 0 for D_ floating, and 1 for G_ floating.

d. *Bit 7.* For real operations (data type 5), this bit is available to indicate whether the result of the operation is to be truncated (bit 7 = 0) or rounded (bit 7 = 1). For other data types, this bit is reserved for conveying implementation-specific information.

Notation

The opcodes in this Reference Manual are divided into functional groups and subgroups, indicating the purpose for which they were originally conceived. No implementer is restricted to using any opcode for any specific purpose suggested in this manual, but the use of Rcode at all implies that the user intends to conform to the form and function specified for each individual opcode.

In addition to being divided into groups, all opcodes are divided into a small number of classes :-

a. **Static (S).** This opcode requires no run-time action.

b. **Declarative (D).** This opcode declares some run-time entity. It may, or may not, therefore require run-time action dependent upon whether the entity is statically determinable or not.

c. **Functional (F).** This opcode class returns some value at run-time.

d. **Procedural (P).** This opcode performs some run-time action without returning any value.

e. **Symbol_table (T).** An opcode in this class provides information which is not meaningful for code generation purposes, but may be of use to provide symbolic debugging information.

The bracketed letter code used above appears against each Rcode entry in the reference list. Where some opcode may or may not return some value dependent upon its operands, then (F/P) will be seen in the list.

Each opcode has zero or more operands, listed below the entry heading. Those items marked "-" describe operands to the instruction which occur as subtrees. Items prefixed by "*" describe information which occurs literally (ie is not dynamically computable), immediately following the opcode. Whether any of these subtrees is expected to return a value will naturally depend on the opcode.

Group 0: Pseudo-operations

Opcodes in this group cause special actions to be taken during the building or rebuilding of the Rcode tree.

Group 0a: Miscellaneous

Noop

This opcode is ignored on input. The opcode value *must be* 0, permitting null bytes to be inserted anywhere in an Rcode file, for example as filler at the end of a disk block.

Null (S)

This opcode represents a null subtree. This opcode is used whenever an operand to the parent node is being omitted as, for example, when some compiler optimisation detects unreachable code and wishes to replace it - with a Null.

Refer_ Generic (T)

* generic nesting level of argument group
* argument number within group

This opcode is used wherever a reference occurs to the corresponding generic argument. The back end of a compiler should never see this, except as a part of the symbol table information.

Identifier (T)

- the character string representation

This only occurs as an operand which is giving a name to something. The subtree is expected to be an appropriate literal - usually a Short_ Block_ Lit.

Group 0b: Subtree Stack Manipulation

These operations are provided to allow a compiler front-end, operating under tight memory constraints, to output the subtrees of a node in any convenient order, possibly before it has determined the correct order, and even before it has discovered whether it will need those subtrees or not. Instead, it can direct the subsequent compiler phases to perform the re-ordering, and deletion of unwanted operands, as it is rebuilding the tree.

NOTE 1. These operations do not themselves occur as part of the tree structure of the Rcode.

2. The semantics are described as equivalent operations on a stack which is being used to rebuild a tree. Any equivalent actions are, of course, permissible.

Tree_ Rotate (S)

* Depth into subtree stack to rotate
* How many steps to rotate (signed)

A single-step rotation in the positive direction moves the subtree at the top of the stack to the bottom of the affected part, moving the rest of the affected part one place up. A single-step rotation in the negative direction has the opposite effect. Multi-step rotations effectively consist of the appropriate number of single-step rotations.

Tree_ Duplicate (S)

* Depth of entry in stack to copy (1 = top)
* how many copies to make

Makes the required number of copies of the specified subtree, and pushes them onto the stack being used in rebuilding the tree.

Tree_ Delete (S)

* Number of subtrees to delete

Deletes the specified number of subtrees from the top of the tree-building stack.

Group 0c: Source Text Symbol Information

These opcodes allow the inclusion of source line and column number information in the Rcode, from where it can be included in some target-system debugger symbol-table. Like the operations in the previous sub-group, they occur 'free-floating', not as part of the Rcode tree structure.

Line number information, if present, takes the form of two explicit operations. Column number information, however, may optionally (see Standard Pragmas) be associated with every Rcode tree node, where it occurs as an extra byte following the opcode, but preceding the additional opcode-specific literal information. This byte specifies the column position in the current source line corresponding to the Rcode operation, as a non-negative increment over the position of the operation which preceded it in the file. The column number is initialised to one at the start of each source line.

Set_ Sourceline (T)

* new source line counter value

Sets the current source line counter to the specified number, and resets the column counter to one.

Inc_ Sourceline (T)

Increments the current source line counter by 1, and resets the column counter to one.

Advance_ Columnpos (T)

* one-byte amount to be added to the current column position

Adds the specified amount to the column counter.

This is useful where a gap of more than 255 characters occurs between successive Rcode operations coming from the same source line. It is also usable when column information is not present on every opcode.

Group 0d: Code Generation Control

The operations indicated by the opcodes in this subgroup also occur free-floating - independent of the tree structure.

Pragma (S)

- pragma operands
- text information as a literal
* pragma code id

A pragma is meant to convey implementation-specific information to control aspects of the code generation process. Standard pragmas are defined at the end of this Reference Manual. The list of pragmas which may be defined for a particular language should be contained in a Language System User Guide or equivalent document.

Group 1: Declarations

Group 1a: Globally-occurring Declarations

These global declarations may only occur at the outermost (top) level of a compilation unit.

Begin_ Unit (S)

- unit name
- unit body itself
- the end_ unit code (see below)
* year (four characters)
* month
* day of month
* hour (twenty-four hour clock)
* minute
* second
* hundredths of a second (all two characters)
* Target machine type (code values specified for project)
* Target machine version
* Target machine variant
* Target OS type (code values specified for project)
* Target OS version
* Target OS variant
* Rcode version
* Rcode variant
* Bits per byte
* Bytes per short integer
* Bytes per default integer
* Bytes per address

This opcode marks the beginning of a compilation unit, and provides some information which may be used for consistency checking.

The literal parameters form a *time-stamp* which corresponds in format to an ANSI proposed standard over all but the last six 'fields'. All code values specified for the Portable Language Implementation Project are contained in appropriate compiler/linker Definition modules. They need not, of course, be used by any other system making use of Rcode. The time-stamp facilitates version checking. It is meant to be passed on to a linker program, so that, when linking together the complete user program, the linker can check that different units importing the same unit have not imported different versions of that unit.

End_ Unit (S)

* module kind (face, body or entire)
* number of imported modules

* number of generic argument groups (0 if non-generic)
* number of internal declarations

This opcode marks the end of a compilation unit.

'Module face', 'module body' and 'entire module' are Peano language terms. There are corresponding terms in other modular languages. A module face consists of definitions which are read by the compiler front end in order to perform interface consistency checking when compiling an importing unit, and also when compiling the coresponding module body - the compiler back end should never see them. A module body is that compilation unit which corresponds to a known module face, making direct use of the definitions therein. An entire module is complete and self-contained, having no separate module face containing definitions. The difference between a module body and an entire module is not important to the back end of a compiler.

Import_ Unit (S)

- name of imported unit
* id number assigned to unit
* date/time/version information

Specifies a compilation unit which is imported by this one. Import information is passed on to the linker, where it is used to sort out what units are needed in the final program, and in what order they are to be initialised and finalised. The date/time/version information is identical in form to the literal fields of the Begin_ Unit opcode.

Declare_ Area (S)

- optional explanatory text, to be displayed on a linker map
* id number assigned to area
* alignment required
* area protection :- read-only or read/write

This opcode declares a static storage area. The size of the area is initially zero; space is allocated in it using extend-area and append-area instructions.

The explanatory text, if present, should take the form of a short block literal instruction. This literal information, interpreted as a text string, is passed on to the linker, which may display it on a storage allocation map.

Group 1b: Other Declarations

Declarations in this group may occur at any point within a compilation unit, dependent upon the source language from which they were generated.

Extend_ Area (S)

* number of area to extend
* how many bytes to extend area by

Increase the size of the storage area by the specified number of *uninitialised* bytes. It is up to the front end of a compiler to keep track of (and adjust) the current size of the area, in order to satisfy any alignment requirements for the new extension.

Append_ Area (S)

- literal information
* number of area to append to
* how many bytes of literal information to store

This increases the size of the storage area by the specified number of bytes, and initialises the newly-allocated storage with the given information. It is up to the front end of a compiler to keep track of (and adjust) the current size of the area, in order to satisfy any alignment requirements for the new extension.

Declare_ Constant (D)

- expression giving value
* assigned id number
* lexical level
* alignment required

Declares a constant entity and specifies its value.

Declare_ Variable (D)

- expression giving size in bytes
* assigned id number
* lexical level
* alignment required for the allocated storage

Declares a variable entity and specifies how much storage it should be allocated - but NOT where this storage should be!

The lexical level is present on this and the Declare_ Constant operations in order to allow optimisation of access to statically-sized

entities. A compiler back end may simply allocate the appropriate amounts of space in the stack frame, and remember the (static) starting offsets, with which it can replace all references to those entities.

At any point where access to an outer block is required, the complete tree for the outer block need not have been built, since the code generator may be operating on a routine at a time, rather than for an entire compilation unit. However, if a compiler front end could be written to run as one pass (as it could be for Pascal say), the declaration of a constant or variable would certainly have been seen before any references to it. The explicit presence of a lexical level on the declaration therefore completes the information necessary for a compiler back end to recognise that this has indeed happened.

To further support the generation of code for a routine at a time, it is also required of a compiler front end that entity declarations local to different blocks at the same lexical level are *not* intermingled. Particularly, for two successive blocks at the same lexical level, the declarations local to the second block should all occur after the declaration of the first block. In this way, successive entity declarations with the same lexical level (with no intervening ones at outer lexical levels) may be assumed by the back end to belong to the same block.

Declare_ Type (T)

- size of objects of the type in bytes, or null if unbounded
- size of objects of the type in bits, or null if unbounded
- code for performing run-time consistency check
- type definition itself
* assigned id number
* lexical level
* nesting level within compilation unit
* alignment required for objects of the type
* packed or not
* machine type or not
* basic machine type coding
* length in bits
* length in bytes
* is an unsafe type
* is a forward definition

This opcode forms an Rcode type descriptor - in terms of the high-level language definition. The run-time consistency checking code could be something like ensuring that the lower bound of a subrange is not greater than the upper bound plus one. The Rcode expression comprising the type definition consists of one of the operators in group 7c (below).

Note that all type definitions (whether they are directly given a name via a type definition or not) in the source language get translated into corresponding type entity declarations. Types which were just part of other types, get referred to in the appropriate places, in the latter, using one of the operations in group 7b (below).

Declare_ Routine (D)

- code for the routine itself
* assigned id number
* lexical level
* is it forward
* is it immediate
* is it a macro routine
* is proper routine entry/exit code to be generated
* size of function result, or 0 if none or variable-length

An immediate procedure or function is one that may be used in constant expressions. This information is really only meaningful to a compiler front end.

Macro routines are those for which calls are expanded in-line. Hence, no code need be generated for them; the back end of a compiler system should never see them.

The static size of the function result is included for deciding whether the result should be returned in a register or not. To ensure consistency with importing units, the decision should be made based on this size alone.

Declare_ Module (T)

- zero or more generic arg group definitions
- zero or more contained declarations
* assigned id number
* lexical level
* whether face, body or entire

* number of generic argument groups, or 0 if non-generic
* number of contained declarations

This opcode declares a module entity. If this is a non-generic module then the code generator should inspect its contents to generate machine code and/or interpret the Rcode as required.

Declare_ Label (D)

* assigned id number
* lexical level

This defines a label. Note that Goto's and labels are provided only for implementing Pascal-like languages. They are not required for any other purpose.

Define_ Symbol (S)

- name of symbol
- expression giving value
* data type

Defines a global symbol in the target-system-dependent relocatable object language. This permits Rcode to express the necessary interfacing to operating-system library routines and other software.

Group 1c: Object Characteristics

This opcodes in this sub-group identify general characteristics for all kinds of objects defined in symbol table form, characteristics which are of use when a compiler is importing definitions to form symbol table entries, but which could/would be ignored in the back-end of a compiler, by a debugger, etc.

Whole_ Object (T)

- formal argument list
* is built-in to the source language
* forms part of a standard library
* argument group number
* is exported
* kind of parameter (or not)

This defines characteristics which are common to complete objects of whatever kind they may be. The kind of parameter relates to the kind of calling convention appropriate to this particular object (eg by value as variable, by value as constant, as fixed type, as variable type, etc).

Part_ Object (T)

- field offset
- field width
* is a bit field or not
* is exported
* kind of parameter (or not)

Defines characteristics which are common to components of objects of whatever kind they may be.

Group 2: Values of Things

The operations in this group return values without operating on any operands. Multiple versions of the 'return-literal' opcode are provided, to save space where only small amounts of literal information are wanted.

Byte_ Lit (F)

* one byte of literal information

Returns the literal information as value.

Short_ Word_ Lit (F)

* two bytes of literal information

Returns the literal information as value.

Long_ Word_ Lit (F)

* four bytes of literal information

Returns the literal information as value.

Short_ Block_ Lit (F)

* one-byte count of how many bytes of literal information
* one or more bytes of literal information (255 bytes max)

Returns the literal information as value.

Literal (F)

* count of how many bytes of literal information
* one or more bytes of literal information (up to 65536)

Returns the literal information as value.

Refer_ Constant (F)

* id number
* lexical level

Returns the value of the specified constant entity.

Generic_ Refer_ Constant (T)

* id number
* lexical level
* generic nesting level

This opcode only occurs within a generic entity, when referring to some constant entity declared within it. It facilitates re-assignment of lexical levels and id numbers, when instantiating the generic entity.

Refer_ Variable (F)

* id number
* lexical level

Returns the address of the specified variable entity.

Generic_ Refer_ Variable (T)

* id number
* lexical level
* generic nesting level

This opcode only occurs within a generic entity, when referring to some variable entity declared within it. It facilitates re-assignment of lexical levels and id numbers, when instantiating the generic entity.

Refer_ Area (F)

* id number

This returns the address of the start of the specified storage area in the current compilation unit.

Refer_ Imported_ Area (F)

* area number within module
* module number

Returns the address of the start of the storage area from the specified imported unit.

Refer_ Symbol (F)

- name of symbol
* data type

Returns the value of an externally-defined global symbol in the target-system-dependent relocatable object language. This allows Rcode to interface to operating-system library routines and other software.

Refer_ Arglist (F)

* lexical level

This opcode returns the address of the start of the argument list for the specified block. This address is intended for use in address arithmetic (to perform offsetting) prior to doing a load, to obtain the value of some argument. It should not be used to store into the argument list.

WARNING If the argument list is zero-length, the address returned from the operation specified by this opcode is undefined.

Refer_ Result (F)

* lexical level

Returns the address of the start of the function result area for the specified block. If the result area is zero-length, the address returned is undefined.

Refer_ Routine (F)

* id number
* lexical level

This returns a descriptor for the specified routine. The lexical level should agree with that in the routine declaration. A descriptor consists of the address of the routine, followed by an environment pointer value (eg the static link, or alternatively a pointer to a display array).

Generic_ Refer_ Routine (T)

* id number

* lexical level
* generic nesting level

This opcode only occurs within a generic entity, when referring to some routine entity declared within it. It returns a descriptor for the specified routine.

Refer_ Imported_ Routine (F)

* id number of routine within unit
* unit number

This returns a descriptor for the specified routine from the specified imported unit. Note that within that unit, the routine must be marked as being exported. This may be done by source-level declaration or internally by a compiler front end - as appropriate for the language concerned.

Group 3: Basic Operand Manipulation

This group of opcodes defines all forms of manipulation of operands excluding any form of transformation (ie not including logic or arithmetic).

Group 3a: Memory Operations

Note that loading returns an object found in store, while storing places an object into a given location. A conventional machine Load operation comprises both of these since both a source and a destination are involved.

Storage is meant to occur at some time after the "issue" of a Store opcode, aand then only if necessary. This allows for delayed storage operations to reduce memory accesses when generating machine code. The Update opcodes are therefore provided to indicate that any holdup must be cleared and the location immediately updated.

Block_ Load (F)

- source address for value to load
- length in bytes of value to load
* alignment that may be assumed for source address

Returns the specified number of bytes of data beginning at the specified address. The block may be zero-length.

The alignment value is provided to allow a machine code generator to produce code to move more than a byte at a time, if this

would be more efficient on the target machine.

Bit_ Block_ Load (F)

- base byte address of source value to load
- bit offset from source base address
- length in bits of value to load

This opcode returns the specified number of bits of data beginning at the specified bit offset from the given base address. The block may be zero-length.

Block_ Store (P)

- destination address
- value to store
* alignment that may be assumed for destination address

This opcode stores the value in memory as a whole number of bytes, beginning at the specified address.

The alignment value is provided to allow units larger than a byte to be moved at a time, if this would be more efficient on the target machine.

Bit_ Block_ Store (P)

- base destination address
- bit offset from base destination address
- value to store

Stores the value in memory, as some number of bits, beginning at the specified offset from the given base address.

Block_ Update (P)

- address of block
- size of block, in bytes
* alignment that can be assumed for address of block

This operation indicates that optimisations performed on accesses to the specified block of store (delayed loads or delayed stores) are not to be carried across an occurrence of the operation. It allows the programmer to specify checkpoints for atomic operations on shared

variables.

Bit_ Block_ Update (P)

- base byte address for block
- bit offset of start of block from base address
- size of block to be updated, in bits

The effects of this opcode are similar to Block_ Update, except that the block is a bitstring beginning on an arbitrary bit boundary.

Group 3b: Operations on Structures

Select (F)

- byte offset into value for part to select
- size in bytes of part to select
- value to select from

Returns the specified portion of the given value. This operation is used to implement selection of components of structures.

Bit_ Select (F)

- bit offset into value for part to select
- size in bits of part to select
- value to select from

Returns the specified portion of the given value. This operation is useful for implementing selection of components of packed structures.

Construct (F)

- zero or more components
* number of components

This opcode combines the given component values into a physically-contiguous block value. Each component occupies some whole number of bytes, with no gaps. This operation is used, for example, to implement constructors in the Peano language, or to assemble the list of parameters for s routine invocation.. The components must each be (at the top level) a Construct_ Component operation, which

is described below.

Packed_ Construct (F)

- zero or more components
* number of components

This opcode has similar meaning to a Construct operation, except that each component is packed into some number of bits, again with no gaps.

Construct_ Component (F)

- replicator count, defaults to 1 if null.
- size to give component within overall structure
- value of component

Specifies a component of either a Construct or a Packed_ Construct operation.

The size to give the component is interpreted as bytes for a Construct, and bits for a Packed_ Construct. If the actual component value is smaller than this, it gets padded with zero bits at the high-address end to the specified size; if larger, it gets truncated at the high-address end. It is the job of a compiler front end to determine size values such that, if the complete structured value is given its required alignment, each component ends up with an alignment suitable for its data type.

Group 3c: Other

Call_ (F/P)

- size of result area, or null if fixed
- composite value of argument list
- routine descriptor
* size of function result area, or 0 if variable or zero-length
* alignment required for argument list
* alignment required for result area

This opcode calls the specified routine. The argument list may be built, for example, by using the Construct operation. Used in a context where a value is required this opcode implies a function call, otherwise a procedure call.

Group 4: Control

Group (P)

- body of group

This operation implements the grouped statement such as a 'compound statement' source language style of construction. A grouped statement may be left in the middle with an Exit_ operation (see below). The body of the group is simply a statement or sequence of statements, none of which returns a value.

Loop_ (P)

- body of loop

This operation implements the loop statement in typical high-level languages. It may also be used for the other loop constructs present in several languages.

Local_ Block (F/P)

- size of result area, or null if fixed
- composite value of argument list
- body of block
* lexical level
* alignment required for result area
* alignment required for argument list
* size of result area, or 0 if none or variable-length

This declares a local block, which is equivalent to an inline routine call. It may have its own local declarations, and may also be left with an Exit_ operation (see below). If it is used in a context where a value is required then it implies a function local block, otherwise a procedure.

For_ (P)

- one or more for-index definitions
- body of for-loop
* lexical level
* number of for-indexes

Implements a for-statement, with the for-indexes declared as local constants, and with static increments of either plus or minus 1. This directly corresponds to the for-statement of languages such as Peano

or Ada. Other languages, with different semantics, could still implement their for-statements in terms of this primitive.

The body of the for-loop is performed for all possible combinations of values of the for-indexes, starting from their initial values up to their final values, in steps of plus or minus 1 (as specified in each index definition), with the last for-index varying the most rapidly. This operation is like a local block, in that the body may contain declarations local to the for-statement. Indeed, the for-indexes themselves are effectively declared as constant entities within the body of the for-loop, with id numbers assigned in sequence, with the first for-index being 1.

For_ Index (D)

- initial value
- final value
* data type of for-index
* backwards or not

Defines a for-index for a for-statement. The data type must not be real. If 'backwards' is true, the step is minus 1, so if the initial value is less than the final value, the for-loop is not executed at all. Otherwise, if 'backwards' is false, the step is plus 1. Then for the number of iterations of the for-loop to be zero, the initial value must be greater than the final value.

Exit_ (P)

* number of levels to exit

Exits the specified innermost number of levels of group, loop, for-statement or block. If an Exit_ operation causes an exit from a block, it must be the last (outermost) construct of the ones being left. An Exit_ may not cause an exit from more than one block.

Case_ (F/P)

- selecting expression
- zero or more alternatives
- else part
* data type of selecting expression (must not be real)
* number of alternatives

This opcode is used to implement both case- and if-statements in

typical high-level languages. The selecting expression is used to select an alternative that has a case-label (see below) with a matching value. If no such alternative is found, the else-part alternative is selected for execution.

An if-statement is represented as a case-statement with a boolean selecting expression, and appropriate labels on its alternatives.

All the alternatives (except the else-part) must, at the top level, be Case_ Alternative operations (defined below). This operation may also occur in expressions, where it returns a value. This will be the value returned by the alternative which was selected for execution. In this case, all the alternatives must be capable of returning a value of an appropriate type.

NOTE A case-operation is implementable on several computers using jump tables, on others using a case instruction of some kind. However, if the selecting expression is known to have only two or three values (for example, if the data type is boolean or tristate), an implementation using comparisons and conditional branches may be more efficient.

If there is more than one case-label matching a given value for the selecting expression, the result is undefined.

Case_ Alternative (F/P)

- one or more case label definitions
- alternative statement/expression itself
* number of labels

Defines an alternative in a case-statement or case-expression. The parent node must be a Case_ operation. An alternative is selected for execution if one of its associated labels matches the value of the selecting expression.

Byte_ Case_ Label (S)

* one-byte label value

Defines a single case label value of some byte-sized data type (the data type is specified in the Case_ operation).

Byte_ Range_ Case_ Label (S)

* one-byte low bound

* one-byte high bound

Defines a range of label values of some byte-sized data type (the data type comes from the Case_ operation). This label matches any value of the selecting expression from the low bound up to the high bound, inclusive. If the low bound is greater than the high bound, this represents the null range, which matches no values.

Shortword_ Case_ Label (S)

* two-byte label value

Defines a single case label value of some shortword-sized data type (the data type is specified in the Case_ operation).

Shortword_ Range_ Case_ Label (S)

* two-byte low bound
* two-byte high bound

Defines a range of label values of some shortword-sized data type (the data type comes from the Case_ operation). This label matches any value of the selecting expression from the low bound up to the high bound, inclusive. If the low bound is greater than the high bound, this represents the null range, which matches no values.

Longword_ Case_ Label (S)

* longword-sized label value

Defines a single case label value of some longword-sized data type (the data type is specified in the Case_ operation).

Longword_ Range_ Case_ Label (S)

* longword-sized low bound
* longword-sized high bound

Defines a range of label values of some longword-sized data type (the data type comes from the Case_ operation). This label matches any value of the selecting expression from the low bound up to the high bound, inclusive. If the low bound is greater than the high bound, this represents the null range, which matches no values.

Case_ Label (S)

- large label value

Defines a single case label value of some data type with the same size (the data type is specified in the Case_ operation).

NOTE This operation is reserved for expansion to integer data sizes larger than a longword.

Case_ Label_ Range (S)

- label value low bound
- label value high bound

Defines a range of label values of some data type with the same size (the actual data type comes from the Case_ operation). This label matches any value of the selecting expression from the low bound up to the high bound, inclusive. If the low bound is greater than the high bound, this represents the null range, which matches no values.

NOTE This operation is reserved for expansion to integer data sizes larger than a longword.

Sequence (F/P)

- zero or more statement subtrees
* number of subtrees in sequence

The statements of the sequence are to be executed in strict sequential order. This operation may occur in an expression, where it returns whatever value is returned from the last subtree in the sequence.

Set_ (F/P)

- zero or more statement subtrees
* number of statements in set

The statements of the set are to be executed in some unspecified order. The compiler back end may even take advantage of parallelism in the target hardware, and execute parts of the set concurrently. This operation may occur in an expression, where it returns whatever value is returned from the last subtree in the set, irrespective of which subtree completes 'execution' last.

No_ Transplant (F)

- subtree to be isolated

In general, the back end of a compiler may take advantage of the associativity of operators to rearrange expressions such as

(a + b) + c into a + (b + c), to permit further transformations in the interests of efficiency. However, this may have implications on the accuracy of the resulting computation. The No_ Transplant operation permits a compiler front end to set limits to this rearrangement; specifically, no portion of the subtree below the No_ Transplant may be moved into the part of the tree above, nor is movement in the opposite direction permitted. Apart from this, the No_ Transplant operation simply returns whatever value is returned from the subtree.

Link (P)

- head of list
- rest of list

This opcode enables the construction of linked lists of opcodes as, for example, in the Sequence and Set_ codes above. Except for the last element in the list the 'rest' will be another Link opcode.

Goto (P)

* id number of destination label
* lexical level of destination label

Implements the Goto statement found in some programming languages.

Label (S)

* id number of label

Defines the label to point to the current place in the block. For example, if this occurs in a sequence, transferring control to the label will cause execution to resume at that point in the sequence.

Group 5: Arithmetic

Group 5a: Memory Operations

This subgroup models subgroup 3a, except that all objects are of basic types only.

Load (F)

- address to load from
* data type of value to load

Returns the value at the specified address.

Store (P)

- address to store value at
- value to store
* data type of value to store

Stores the value at the specified address. The size specified by the data type should agree with that of the value being stored.

Update (P)

- address to update
* data type of value to update

This opcode indicates that optimisations performed on accesses to the specified piece of memory (delayed loads or delayed stores) are not to be carried across an occurrence of the operation. It allows the programmer or a compiler front end to specify checkpoints for atomic operations on shared variables.

Group 5b: Numeric Operations

Add (F)

- first operand
- second operand
* data type

If the data type is boolean, this performs an inclusive-or operation on its operands, taken as boolean values. If the data type is signed, unsigned or unspecified exact, or real, it performs the appropriate numeric add operation on its operands, taken as values of the corresponding type. The tristate data type is not allowed.

Subtract (F)

- first operand
- second operand
* data type

If the data type is signed, unsigned or unspecified exact, or real, this performs the appropriate numeric subtract operation on its operands, taken as values of the corresponding type. Boolean and tristate data

types are not allowed.

Negate (F)

- value to negate
* data type

If the data type is boolean, this performs a not-operation on its operand, returning false if its value is true, true if its value is false. If the data type is tristate, this operation returns -1 if its operand value is +1, +1 if the operand is -1, and 0 if the operand is 0. It could thus be used to invert the sense of a comparison. If the data type is signed or unspecified exact, or real, this performs the appropriate numeric negate operation on its operand, taken as a value of the corresponding type. The unsigned exact data type is not allowed.

Absolute_ Value (F)

- operand value
* data type

If the data type is signed exact or real, this performs the appropriate absolute-value operation on its operand, taken as a value of the corresponding type. The boolean, tristate, unsigned exact and unspecified data types are not allowed.

Multiply (F)

- first operand
- second operand
* data type

If the data type is boolean, this performs an and-operation on its operands, taken as boolean values. If the data type is signed, unsigned or unspecified exact, or real, it performs the appropriate numeric multiply operation on its operands, taken as values of the corresponding type. The tristate data type is not allowed.

Divide (F)

- dividend
- divisor
* data type

If the data type is signed, unsigned or unspecified exact, or real, this

performs the appropriate numeric divide operation on its operands, taken as values of the corresponding type. The boolean and tristate data types are not allowed.

Modulus (F)

- dividend
- divisor
* data type

If the data type is signed, unsigned or unspecified exact, this returns the remainder on dividing the dividend value by the divisor. The boolean, tristate and real data types are not allowed.

Group 5c: Extended-precision Numeric Operations

These special opcodes are provided to simplify the task of a back-end code generator which will frequently have to adopt a different strategy for multiple length arithmetic of all kinds.

Block_ Add (F)

- first operand
- second operand
* data type

Performs a multiple-precision add operation on its operands, which may be any number of bytes long, but should be of the same size. The size field in the data type is ignored; the only valid types are signed exact, unsigned exact and unspecified.

Block_ Subtract (F)

- first operand
- second operand
* data type

Performs a multiple-precision subtract operation on its operands, which may be any number of bytes long, but should be of the same size. The size field in the data type is ignored; the only valid types are signed exact, unsigned exact, and unspecified.

Extended_ Multiply (F)

- multiplicand
- multiplier
* data type

This multiplies its two operands, returning a double-length result. The data type must be signed exact.

Extended_ Divide (F)

- dividend
- divisor
* data type

Returns both the quotient and remainder of the division, in that order, as a composite value. The size of the divisor (and of the quotient and the remainder) is given directly by the data type; the size of the dividend is twice this. The data type must be signed exact.

Extended_ Modulus_ Floating (F)

- real multiplicand
- real multiplier
- multiplier mantissa extension, integer
* data type for integer operands and integer part of result
* data type for real operands and real part of result

Does the floating multiplication, and returns the whole number and fraction parts of the result separately, in that order, as a composite value. The whole number part is returned as a signed integer, the fraction part as a real.

Group 5d: Conversions

Convert (F)

- source value
* result data type
* source data type
- source value

Does conversions between operand precisions and between exact and real values. The valid conversions are as follows :-

a. From signed/unsigned exact to signed/unsigned exact - this is always valid provided that the value does not overflow the bounds of the result type. This could occur if the result type is smaller than the source type, but it also means that a negative value of a signed exact type cannot be converted to an unsigned exact.

b. From unspecified to unspecified - this is always valid, but

the result type must not be larger than the source type. The truncated bits are simply thrown away - that is, overflow is ignored.

c. From signed exact to real - this should always be valid - provided overflow does not occur.

d. From unsigned exact to real - this is always possible, although there may be no hardware support for direct machine code generation.

e. From real to signed exact - this is valid provided overflow does not occur.

f. From real to unsigned exact - this is always possible provided that integer overflow does not occur. Fractions are truncated towards zero. This may not be supported by hardware for machine code generation.

g. From real to real - this is valid provided overflow does not occur. For conversion to a lesser precision, the result is truncated or rounded depending on whether bit 7 of the result data type is 0 or 1 respectively.

All other combinations of source and result data type are illegal.

Convert_ Tristate_ Boolean (F)

- source value
* byte containing conversion mask

Maps the source value, interpreted as being of tristate type, onto a boolean value, according to the 3-bit conversion mask :-

- Bit 0 indicates what result to give if the source tristate value is −1.
- Bit 1 indicates what result to give if the source value is 0.
- Bit 2 indicates the result if the source value is +1.

This operation is useful for mapping of the result of a conversion onto a boolean value, for implementing the comparison operators in Pascal and Modula-2, for example.

Group 5e: Comparisons
Compare (F)

- left-hand operand
- right-hand operand
* data type

Performs the appropriate comparison operation on its operands, returning a tristate result as follows :-

−1 means less-than,
0 means equal,
+1 means greater-than.

Any data type, except unspecified, is allowed.

Block_ Compare (F)

- left-hand operand
- right-hand operand
* data type

Performs a string-comparison operation on its operands taken data type unit size at a time, making the comparison on a data type basis. Both operands should be of the same size. It returns a tristate result, with the same meanings as for Compare.

Group 5f: Bitstring Operations
Bit_ Op (F)

- left-hand operand
- right-hand operand
* operation mask

This is a generalised two-operand bitwise operation. The operand sizes, in bits, should be equal; the result has the same size.

Each bit of the result is determined is follows :-

a. The corresponding bit of the right-hand operand is extracted, left-shifted one place, and inclusive-ORed with the corresponding bit extracted from the left-hand operand, to give an integer number in the range 0 to 3.

b. The bit (of the low-order 4) in the operation mask which has this number immediately gives the value of the result bit.

Some example masks are :-

2_ #1110 - bitwise-or (set union)
2_ #1000 - bitwise-and (set intersection)
2_ #0010 - bitwise-clear (asymmetric set difference)
2_ #0110 - bitwise-xor (symmetric set difference)
2_ #1011 - bitwise >= (reverse implication)
2_ #1101 - bitwise <= (implication)
2_ #1010 - identity operation on left-hand operand
2_ #0101 - complement first operand (set difference with universal set)

Either operand may be null if, according to the operation mask, its value is irrelevant to that of the result.

Arithmetic_ Shift (F)

- value to be shifted
* how many bit places to shift

Returns a result of the same size as the value, shifted arithmetically left (if positive) or right (if negative) by the number of places specified.

Logical_ Shift (F)

- value to be shifted
* how many bit places to shift

Returns a result of the same size as the value, shifted logically left (if positive) or right (if negative) by the number of places specified.

Rotate (F)

- value to be shifted
* how many bit places to shift

Returns a result of the same size as the value, rotated left (if positive) or right (if negative) by the number of places specified.

Singleton (F)

- size in bits of result to produce
- offset to bit to be set to 1

Returns a 'singleton set', i.e. a string with all the bits, except one, equal to 0. The offset of the bit to be set must be less than the size

of the result.

Zero_ Extend (F)

- size in bits of result
- value to be extended

If the result size is greater than that of the second operand value, the result is that value, with zero bits added at the most significant end, to make up the required size. If the result size is less than that of the operand value, the latter is truncated at the high-address end to the required size. The bits lost should all be zero, otherwise a run-time error occurs.

Sign_ Extend (F)

- size in bits of result
- value to be extended

If the result size is greater than that of the second operand value, the result is that value, with its sign bit duplicated at the most significant end, to make up the required size. If the result size is less than that of the operand value, the latter is truncated at the most significant end to the required size. The bits lost should all be equal to the sign bit of the result, otherwise a run-time error occurs.

Group 5g: Run-time Checks

Signal (D)

- error message text

This operation signals a run-time error. The message text should be given by one of the literal operations.

Assert (D)

- boolean expression
- error message text

This operation signals a run-time error if the given condition is found to be false. The message text should be given by one of the literal operations.

Range_ Check (F)

- value to be checked
- low limit to be checked against

- high limit to be checked against
- error message text
* data type

The data type should be boolean, tristate, or signed or unsigned exact. The value to be checked is returned as the result of the range check operation. However, if this value is less than the low limit, or greater than the high limit, a run-time error is signalled. The message text should be given by one of the literal operations.

Either of the high or low limit expressions could be null, meaning that the corresponding limit check is not to be performed. This allows one-sided range checks, if the front end is able to determine, from the logic of the program, which checks are not necessary.

Group 6: Machine/System-dependent Operations

This section of Rcode contains a number of 'standard' system-dependent operations which are needed on any machine which is to implement programs involving concurrency, input/output, interrupts, etc. In addition there are a number of machines which include particular instructions which could be used to achieve special effects only applicable to that machine. These special effects are not usually produced by a compiler front end generating Rcode, but more likely from using the two-level language facility for inline Rcode which Peano offers. It is for this reason that machine-dependent Rcodes are provided. Extensions for machines not covered in this manual are permissible, provided that such extensions are ONLY going to affect programmer generated Rcode and NOT compiler generated Rcode - which would then no longer be portable.

Group 6a: Compiler Standard Extensions

The opcodes in this sub-section may be generated by compiler front ends and will be understood by all code generators or interpreters provided as part of the Portable Language Implementation Project.

Input (P)

- i/o port address
- buffer location
- number of units to transfer
* data type

Transfer the specified number of data type objects from the addressed port into the machine buffer specified. The port address is expected to be valid for some input device on the machine concerned. The data type must specify a size of object which corresponds to the port concerned, for example a byte for a byte port!

Input_ Move (P)

- i/o port address
- buffer location
- number of units to transfer
* data type

This opcode specifies the transfer of the given number of data type units from the addressed port into the buffer (if non-null) and returning the value. The port address is expected to be valid for some input device on the machine concerned. The data type must specify a size of object which corresponds to the port concerned, for example a byte for a byte port!

Output (P)

- i/o port address
- buffer location
- number of units to transfer
* data type

Transfer the number of units specified from the buffer to the device attached to the output port address. The port address is expected to be a valid address for an output device. The data type must correspond to the kind of output port attached to the address, for example a word if the port is a word port!

Fast_ Call (P)

- address of routine to call

This opcode implies the direct generation of a machine code which does not set up any environment - which may be done for normal routine calls. It may thus be used to implement non-standard calling conventions - in particular the calling convention used by the native operating system.

Fast_ Return (P)

This opcode implies the generation of a machine code which merely returns from some routine call without carrying out any frame

cleanup that may be done for normal routines by the back end. It may be used to implement non-standard calling conventions.

Jump (P)

- address to jump to

This opcode implies the action of a jump instruction to the specified destination. This may be used to implement non-standard routine calling conventions.

Interrupt_ Return (P)

This opcode implies the action of a machine code to return from an interrupt routine, restoring any machine-dependent environment which the hardware may have saved. This could be used in a routine declaration which doesn't have the standard routine entry and exit code generated, to set up special-purpose interrupt handlers.

Get_ Priority (P)

- result location expression

This opcode places the current process priority into the location specified. Note that the bit-pattern which represents this priority must be interpreted in a machine-dependent manner.

Set_ Priority (P)

- value expression

This makes the current process priority be the given value expression (which must be a valid machine-dependent value!).

Save_ Raise_ Priority (P)

- value expression

This saves the current process priority (in some processor dependent manner) and increments the current priority by the given value expression - which must result in a machine-dependent valid value.

Restore_ Lower_ Priority (P)

- location

Place the current value of the priority of the current process in the given location and restore the priority to that value most recently

saved.

Get_ Special_ Reg (F)

- address for result (usually null)
* data type
* register number

Returns the value of the specified privileged CPU register. An encoding of the possible register numbers are contained in the Reg_ Kind enumerated data type contained in the project runtime library MACHINE module. The machine code generated by a compiler back end depends upon the register involved and, of course, the machine.

Set_ Special_ Reg (P)

- value to be given
* data type
* register number

Sets the specified privileged CPU register to the given value. An encoding of the possible register numbers are contained in the Reg_ Kind enumerated data type contained in the project runtime library MACHINE module. The machine code(s) which may be generated by a compiler back end are often register and machine-dependent.

Get_ General_ Reg (F)

- address for result (usually null)
* data type
* register number

Returns the value of the specified general-purpose register. The encodings of the possible register numbers are contained in the Reg_ Kind enumerated data type contained in the project runtime library MACHINE module.

Set_ General_ Reg (P)

- value to be set
* data type
* register number

Sets the value of the specified general-purpose register to that given.

The encodings of the possible register numbers are contained in the Reg_ Kind enumerated data type contained in the project runtime library MACHINE module.

Allocate_ General_ Reg (D)

- address for saving value (if needed)
* data type
* register number

Marks the specified general-purpose register as in use for the rest of the current block. Anything the code generator may have put in it is saved. This operation may be used to implement non-standard routine calling conventions, with arguments passed in registers, for example.

Deny_ General_ Reg (D)

- null opcode
* data type
* register number

Marks the specified general-purpose register as in use for the duration of the current block. This operation should occur right at the start of the block. The code generator never touches the register, so its initial contents are preserved. This operation may be used to implement non-standard routine calling conventions, with arguments passed in registers, for example.

Free_ General_ Reg (D)

- address last saved in (usually null?)
* data type
* register number

This opcode is used to inform a compiler back end code generator that it may now use the specified register in its generated code. This operation may follow an Allocate_ General_ Reg or a Deny_ General_ Reg operation on that register, in the same block.

Service_ Call (P)

- optional parameter
* one byte service call code

This opcode implies the generation of a machine code to 'call' a

kernel service within the 'operating system' of the machine concerned with the appropriate parameters. This is also used where specific hardware services (eg context saving - 254 or context loading - 255) are required.

Test_ Set (P)

- address of location to test-and-set
- address for result

This opcode requires a machine code generator to generate an uninterruptible (if possible) code for testing and setting the location specified - returning a boolean value, true indicating that the destination was previously set, false otherwise.

Test_ Clear (P)

- address of location to test-and-clear
- address for result

This opcode requires a machine code generator to generate an uninterruptible (if possible) code for testing and clearing the location specified, returning the result as a boolean value, true indicating that the destination was previously clear, otherwise false.

Add_ In_ Place (P)

- address to add value to
- value to add

This opcode requires the indivisible action of incrementing the value at the given location by the amount specified. It could be used to indivisibly update a memory counter which is being shared between asynchronous concurrent activities.

Subtract_ In_ Place (P)

- address to subtract value from
- value to subtract

This opcode requires the action of indivisibly subtracting the given amount from the address specified. It could be used to indivisibly update a memory counter which is being shared between asynchro-

nous concurrent activities.

Add_ In_ Place_ Locked (P)

- address to add value to
- value to add

This opcode requires the uninterruptible action of incrementing the value at the given location by the amount specified. It may be used for counting semaphore implementation.

Subtract_ In_ Place_ Locked (P)

- address to subtract value from
- value to subtract

This opcode requires the uninterruptible action of subtracting the given amount from the value stored in the address specified. It may be used for implementing a counting semaphore.

Inline_ Code (P)

* one-byte length of literal code to insert
* literal code itself

The specified literal code is to be copied, unchanged, by a compiler code generator, to its code output. This operation may be used to generate the very machine specific special instructions, not otherwise normally considered by the machine code generator. Care should be taken with the use of this operation, as it cannot be guarranteed that the code generator checks that the literal information constitutes a valid sequence of machine instructions.

Group 6b: Z8000-specific Operations

The following list of Rcode extensions permits access to all the special features of the Z8000-series microprocessors except :-

a. The translate and translate-and-test instructions.

b. The compare-and-decrement group of instructions.

c. Extended Processing Unit (EPU) instructions.

Extensions to cope with the first three may be added as necessary. However, these codes may be generated by the target code generator where suitable actions are detected. Extensions for the last will have to await the availability of documentation on the kinds of EPU's available.

Multi_ Micro_ Test (F)

Generates an MBIT instruction, and returns a boolean value indicating the state of the MI pin; true if the pin is high (inactive), false if it is low.

Multi_ Micro_ Request (P)

- 16-bit delay count (must be greater than 2)

Generates an MREQ instruction, and returns a tristate value based on the resultant setting of the condition codes, as follows :-

+1 (greaterthan)	- request not signalled (resource not available)
0 (equal)	- request not granted (resource not available)
−1 (lessthan)	- request granted (resource available)

Multi_ Micro_ Reset (P)

Generates an MRES instruction.

Multi_ Micro_ Set (P)

Generates an MSET instruction.

Group 6c: VAX-specific operations

The following Rcode extensions are specific to the Digital Equipment VAX-11 target machine series of processors.

Exec_ Service_ Call (P)

- service call code

Generates a CHME instruction, with the specified argument value which should be a 16-bit code.

Super_ Service_ Call (P)

- service call code

Generates a CHMS instruction, with the specified argument value which should be a 16-bit code.

User_ Service_ Call (P)

- service call code

Generates a CHMU instruction, with the specified argument value

which should be a 16-bit code.

Group 7: Type Information

This final group of opcodes is included in Rcode for two purposes :-

a. To enable compilers to implement the separate compilation feature of languages such as Peano and Modula-2.

b. For generating information to be passed to a source-language debugger.

For the former purpose, when the front end of a compiler is compiling a compilation unit which is a module face or (for languages like Peano) an entire module, it produces an Rcode file called a *symbol file*. When compiling some later unit which imports this one, the compiler front end re-reads the symbol file, and rebuilds its symbol table from the information within, thus facilitating full type checking across compilation units.

The back end of a compiler never sees this symbol file. Instead, when compiling the corresponding module body or (for languages like Peano) simultaneously with the production of the symbol file, the front end produces another Rcode stream, containing the actual code which is to be passed to the back end for the generation of target machine code.

This latter *code file* may also contain symbol table information (some of which is just a copy of that in the symbol file), for debugging purposes. The portable language project implementation translates this into whatever format is appropriate for either the project editing environment used as a debugger or for a target-system-specific debugger.

Group 7a: Symbol Table Declarations

Expose (T)

* id number of module to expose
* exported or not

Implements the exposing-declaration in Peano or the Use facility in languages like Ada, for example.

Declare_ Constant_ Id (T)

- name
- general characteristics
- type of constant
- expression referencing value

Declares a constant identifier. The general characteristics should be one of the opcodes in Group 1c. The Rcode expression giving the type should be one of the operations in group 7b (below). The expression for referencing the value of the constant is expected to be inserted in-line in the Rcode generated by the front end, wherever a reference to the constant occurs.

The expression for referring to the value is expected to consist of a Declare_ Constant to give the appropriate id number and lexical level.

Declare_ Var_ Id (T)

- name
- object characteristics
- type of variable
- expression indicating the L-value of the variable

Declares a variable identifier. Object characteristics will usually be given by an opcode from group 1c. The Rcode expression giving the type should be one of the operations in group 7b (below). The expression which returns the address (L-value) of the variable would normally be a Declare_ Variable operation referencing the actual variable entity.

Declare_ Type_ Id (T)

- name
- object characteristics
- type itself

Declares a type identifier. Object characteristics is expected to be an Rcode from group 1c. The Rcode expression giving the type should be one of the operations in group 7b (below).

Declare_ Label_ Id (T)

- name
- object characteristics
- label value

This declares an explicit user-defined label. The label value is

expected to be a Declare_ Label opcode.

Declare_ Routine_ Ident (T)

- name
- object characteristics
- routine type
- list of arguments, or null if niladic
- name of result, or null if not function
- expression returning routine descriptor
* function precedence (or 0 if procedure)

Declares a procedure or function identifier. The Rcode expression giving the routine type should be one of the operations in group 7b (below). This should be a procedure or function type, though (for example in Peano) it may be a generic type whose subtypes are procedure or function types.

The expression which returns a descriptor for the routine is inserted in-line in the Rcode generated by the front end, wherever a reference to the routine occurs. This would normally be a Declare_ Routine operation referencing the appropriate routine entity. However, for routines declared externally, this would have to be an expression which built a routine descriptor out of a Define_ Symbol operation.

Declare_ Module_ Id (T)

- name
- object characteristics
* id number

Declares a module identifier. The id number corresponds to that in a Declare_ Module module entity declaration.

Group 7b: Type References

Refer_ Type (T)

* id number
* lexical level

This references the type entity declared with the specified id number

at the given lexical level.

Refer_ Module_ Ident (P)

- name expression

This opcode enables module identifier numbers to be translated back into the name of the imported module name.

Generic_ Refer_ Type (T)

* id number
* lexical level
* generic nesting level

This opcode only occurs within a generic entity, when referring to some type entity declared within it. Facilitates re-assigning of lexical levels and id numbers, when instantiating the generic object.

Refer_ Imported_ Type (T)

* id number of type within unit
* unit number

This opcode references the type entity exported from the specified compilation unit.

Subtype (T)

- parent type
- zero or more actual arguments
* number of actual arguments

This instantiates the generic type which is given as the parent type. The root opcodes specifying the parent type, and any actual type arguments, may be any of the opcodes in this group, including further Subtypes.

Group 7c: Components of Type Definitions

The operations in this sub-group occur as the type definition operand, in a type entity declaration (see the Define_ Type operation in group 1b, above). They supply further information specific to each class of type. As always, references to other types are done with operations in group 7b.

Enumerated_ Type (T)

- zero or more identifiers making up the enumeration list

* number of identifiers in the enumeration list

Defines an enumerated type. The identifiers in the enumeration list are automatically declared as constant identifiers, with literal values starting from 0 up to one less than the number of identifiers in the list.

Char_ Type (T)

Specifies the "standard" type *char*.

Integer_ Type (T)

Specifies the universal basic type *integer*.

Cardinal_ Type (T)

Specifies the universal basic type *cardinal*.

Real_ Type (T)

* number of exponent bits

Specifies the universal basic type *real*. The number of exponent bits is merely a count and does not imply any particular underlying hardware exponent base or offset, etc.

Subrange_ Type (T)

- base type of subrange
- expression giving low bound
- expression giving high bound
- normalisation code
* signed or not
* minimum value
* maximum value

Defines a subrange type, and whether it is signed or not. Only subranges of the predefined type *integer* are signed. The normalisation code is used to define the correct storage values in minimum storage space. The minimum and maximum values are expected to be within the range of a machine word for representation purposes.

Array_ Type (T)

- subscript type
- array component type

Defines an array type. Note that in many languages, multi-dimensional arrays are treated as arrays of arrays! This is the technique for specifying multi-dimensional array structured types in Rcode.

Record_ Type (T)

- zero or more record field definitions
* number of fields

Defines a language record type. Each field definition is given by a Record_ Field operation, defined in group 7d below.

Note that records in Modula-2 and Pascal differ from the ones in Peano in that they allow variants. These are represented by extra, compiler-generated fields whose types are union types, since unions are the Peano equivalent to variants.

Union_ Type (T)

- zero or more selecting expressions
- zero or more union field definitions
* number of selecting expressions
* number of fields

Defines a union type. Each field definition is given by a Union_ Field operation, defined in group 7d below.

For languages which permit record variants these may be represented in the Rcode by extra record fields whose types are union types. In this case, there will be just one selecting expression, which is a dummy anyway, since the variants are not checked.

Set_ Type (T)

- set element type

This defines a language set type for relevant languages.

Channel_ Type (T)

- element type
* is FIFO or not
* direction (inward or not)

This opcode defines a type which is some kind of input/output

pathway from the machine (which could be virtual).

Routine_ Type (T)

- list of argument types, or null if niladic
- result type, or null if not function
- variable result size
* argument count
* fixed result size (or -1 if none or variable)

Defines a procedure or function type. This corresponds to the "signature" of a routine or function.

Pointer_ Type (T)

- pointer component type

This opcode defines a pointer type.

Capsule_ Type (T)

- name
- initialisation code reference
- finalisation code reference
* module id
* lexical level

This defines a capsule type of object used in object-oriented and similar languages.

Generic_ Type (T)

- size in bytes of arglist
- size in bits of arglist
- formal arg group
- type definition proper
* allow unconstrained or not

Defines a generic type. The formal argument group is given by an Arg_ Group operation, defined in group 7d, below. The type definition proper may be given by any of the operations in group 7c - for instance, it could be a further Generic_ Type.

Discrete_ Type (T)

This opcode specifies that a particular generic type is to be discrete. This opcode is normally handled only within the front end of any

compiler system. The back end will never see this since it should have been converted into the exact type specification before reaching that point.

Private_ Type (T)

- actual type

Defines a private or otherwise hidden type. The Rcode expression giving the actual type is one of those operations in group 7b, above.

Group 7d: Components of Type Definitions

Record_ Field (T)

- name
- component characteristics
- type of field
- default value, or null if none

Defines a field of a record structure. This operation only occurs as a subtree of a Record_ Type operation.

Union_ Field (T)

- list of associated case labels
- name
- component characteristics
- type of field
* number of associated selecting expression
* number of associated case labels

Defines a field of a union or variant record structure. This operation only occurs as a subtree of a Union_ Type operation.

The selecting expression number, if nonzero, is used to specify an element in the list of selecting expressions hanging off the parent Union_ Type node; these are numbered from 1 upwards. This field of the union is selected if and only if the selecting expression matches one of the case labels associated with this union field, and provided no preceding (lower-numbered) selecting expression selects any union field. If the union field comprises the else-part of the union - that is, it is to be selected only if none of the selecting expressions selects any union field, this is indicated by the number of the selecting expression, and the number of associated case labels.

both being zero.

Value_ Arg (T)

- name of argument
- type of argument
- expression giving field offset
* is value held in a bit field

Defines a formal value argument in a generic argument group. The field offset indicates the location of the value of the argument, within the composite value of the entire argument group. It is interpreted as a bit offset for a bit field (that is, the argument group structure is packed), and as a byte offset otherwise. This opcode only occurs as a subtree to an Arg_ Group operation (below).

Type_ Arg (T)

- name of argument
- formal type

Defines a formal type argument in a generic argument group, where the formal type has been explicitly specified. No storage is taken up in the composite argument group value, for holding run-time information specifically about this argument. This opcode only occurs as a subtree to an Arg_ Group operation (below).

Unspec_ Type_ Arg (T)

- name of argument
- expression giving field offset
* is size held in a bit field
* is size expressed in bits
* is private allowed for actual type

Defines a formal type argument in a generic argument group, where no formal type has been specified. On instantiation, the actual type may be any type, except that private types may be disallowed.

Within the generic type, nothing is known about the type except its size. The field offset expression indicates the offset, within the composite argument group value, to the field containing this size. The offset is to be interpreted in bits for a bit field (i.e. the argument group structure is packed), and in bytes otherwise.

The size itself may be expressed in bits or in bytes. The former

is needed if the formal type is used as part of a packed structure anywhere within the generic, otherwise the latter is adequate. This operation only occurs as a subtree to an Arg_ Group operation (below).

Arg_ Group (T)

- zero or more formal argument definitions
* number of formal argument definitions comprising group

Defines a group of formal arguments for a generic entity. The formal argument definitions are some mixture of Value_ Arg, Type_ Arg and Unspec_ Type_ Arg opcodes.

Group 8: Two-byte Extended Rcodes

The possibility of extending Rcode for either special implementation specific functions or general purpose extensions, is allowed for in the Rcode basic design by including three extension codes :-

a. **Implementation_ Specific**.

b. **Extension**.

c. **Reserved** - for compiler GAS Code (second-level intermediate language). See also GAS Code User's Guide and Reference Manuals.

Each of these is expected to have at least two bytes of following literal information. The first of these bytes is a count of the following literal bytes (up to 255 maximum) and the second is the extended opcode itself. The need for more literal information is, of course, dependent upon the semantics attributed to the extension opcode byte.

STANDARD PRAGMAS

The following are the currently-defined standard pragmas for communication between the front and back ends of a compilation system.

Side_ Effects. One byte Boolean literal parameter.

This pragma indicates whether the back end is free to optimise expressions as though they had no side effects (as in Peano), or not (e.g. Modula-2, Pascal). The flag is true to indicate that side effects may be present, false otherwise. The default assumption is that side effects are absent.

Column_ Info. One byte Boolean literal parameter.

This pragma indicates that each subsequent Rcode opcode includes a column number byte (if the parameter is true) or not (false). The default assumption is that column numbers are omitted.

Bibliography

[1] Portable Language Implementation Project-
Portable Linker Reference Manual

University of Waikato

[2] Portable Language Implementation Project-
Generic Editor Reference Manual

University of Waikato

[3] Portable Language Implementation Project
Runtime Library Users' Guide

University of Waikato

[4] Portable Language Implementation Project
Unix Runtime Library Guide

University of Waikato

[5] Diana Reference Manual, March 81.
G. Goos and Wm A. Wulf editors.

[6] Strong, J, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T.Steel [1958]. "The problem of programming communication with changing machines: a proposed solution", Comm. ACM 1:8 (August) 12-18. Part

2: 1:9 (September) 9-15. Report of the Ad-Hoc committee on Universal Languages.

[7] Anklam, P, D. Cutler, R. Heinen, M. MacLaren [1982]. "Engineering a compiler, VAX-11 code generation and optimization".

[8] VAX Architecture Handbook; Digital Equipment Corporation.

[9] Aho, A, R. Sethi, J. Ullman [1986]. "Compilers Principles, Techniques and Tools".

[10] Rustin, R. (editor) [1972]. "Design and Optimization of Compilers".

[11] Bornat, R. [1979]. "Understanding and Writing Compilers."

[12] Ganapathi M, J. Hennessy, C. Fischer [1982]. "Retargetable Compiler Code Generation", Computing Surveys, 14, Dec 573-592.

[13] Nori K, U.Ummann, K. Jensen, H. Nageli, Ch. Jacobi [1981]. "Pascal P implementation notes", Barren 125-170.

[14] Christain Rousning Ltd.

[15] Schmidt U, R. Voller [1986]. "The development of a machine independent multilanguage compiler system

applying the Vienna Development Method", System Decsription Methodologies 557-590.

[16] Ganapathi M, C. Fischer [1985]. "Programming Languages and Systems", Oct 560-599.

[17] Tanenbaum A, H. Van Staveren, E. Keizer, J. Stevenson [1983]. "Practical tool kit for making portable compilers", Communications ACM, 26, 654-660.

[18] Yankov B, S. Bonev, L. Nikolov [1985]. Microprocessing and Microprogramming, 16, Nov-Dec, 221-226.

[19] Graham S [1984]. "Table-driven code generation", Lorho 251-288.

[20] Powell M [1984]. "A portable optimizing compiler for Modula-2", ACM SIGPLAN notices 19:6, 310-318.

[21] Johnson S [1975]. "Yacc - yet another compiler compiler", Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.

[22] Johnson D [1979]. "A tour through the portable C compiler", AT&T Bell Laboratories, Murray Hill, N.J.

[23] Snyder C. [1975]. "A portable compiler for the language C", Project MAC, MIT, AD-A010218/6 May.

[24] Aho A, R. Sethi, J. Ullman [1986]. "Compilers principles, techniques and tools, 648-653.

[25] Heyliger G, L. McElhaney, T. Dwyer, P. Keziah [1980]. "Recommendations for a retargetable compiler", Martin Marietta Aerospace, AD-a084195/7 March.

[26] Akin T, [1981]. "A reusable code generator for the PRIME 50-series computers", Georgia Inst. of Technology, AD-A108820/2 AUG.