1986

An Assessment of the Suitability of Nebula as a Target for C

John DiNovo

Rochester Institute of Technology
Graduate School of Computer Science

An Assessment of the Suitability of Nebula as a Target for C

by
John DiNovo

A thesis submitted to the
Faculty of the School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by:

_____
Professor Peter Anderson

_____
Professor Peter Lutz

_____
Professor Andrew Kitchen

Rochester Institute of Technology
Graduate School of Computer Science

An Assessment of the Nebula Instruction Set As a Target for the C Language

by
John DiNovo

TABLE OF CONTENTS

1 INTRODUCTION AND BACKGROUND 1
1.1 Problem Statement 1
1.2 Nebula Design Characteristics & Overview 2
1.3 Previous Work 6
1.4 Scope and Limitations 11

2 NEBULA AND C COMPATIBILITY 13
2.1 Introduction 13
2.2 Data Object Compatibility 14
2.2.1 Integers 14
2.2.2 Float 15
2.2.3 Logical 16
2.2.4 Register 16
2.2.5 Pointers 17
2.3 Data Object Conversion 17
2.4 Primitive Operators Supported by the ISA 20
2.5 Support for Data Structures 21
2.6 Support for Subprograms 27
2.6.1 The Nebula Context 27
2.6.2 Parameter Passage 28
2.6.3 Functions vs. Procedures in C 32

3 NEBULA AND VAX COMPARISON 34

4 PORTABLE C COMPILER IMPLEMENTATION 47
4.1 PCC Compiled Code 48
4.1.1 Null Function 48
4.1.2 Automatic Variables 52
4.1.3 Simple Expressions, Parameters, Return Values 55
4.1.4 External Variables 62
4.1.5 Array Assignments 65
4.1.6 Loops 67
4.1.7 Array Element as a Parameter 69
4.1.8 Array as a Parameter 72
4.2 Code Templates 76

5 CONCLUSIONS 79
5.1 Data Object Representation 80
5.2 Data Object Conversions 80
5.3 Operations on Data Types 80
5.4 Implementation of Data Structures 81
5.5 Implementation of Control Statements 82
5.6 Procedure Interface 82
5.7 Concluding Remarks 83

BIBLIOGRAPHY 86

1. Introduction and Background

1.1. Problem Statement

The primary question this thesis addresses is whether the Nebula Instruction Set, as defined in MIL-STD-1862,[DodM83] provides a suitable target, given its design intent, for the C language.

This investigation takes a number of things for granted. It is assumed that the notion of "suitable target" has validity, even though formalization of this notion will not be demonstrated. Rather it will be accepted as conventional wisdom that architectures and instructions[1] which provide more than the bare necessities with respect to data object definition, object manipulation, addressing modes, and primitive operations will be easier for a compiler writer to work with than architectures which do not.

Solutions of problems by computer can be viewed as a series of translations by which a problem in the physical world is made amenable to solution on a computer. The last few of these translations convert the abstractions of algorithms and data structures into specific manipulations of precisely defined data objects. Both the ease of

[1] For the programmer an architecture is represented not only by the hardware but also by the instructions available for use. In this study the terms architecture, instruction set, and instruction set architecture (ISA) will be used interchangeably.

translation and the efficiency of the solution will be better served when there is a reasonable degree of correspondence between the high order language (HOL) and the instruction set architecture (ISA).

The traditional hardware and instruction primitives useful to implement the data objects and data structures frequently used in high order languages have been known since the 1950's. Myers[Meye78] gives the following as approximate dates: index registers 1949, floating point data representations 1954, general purpose registers 1956, and indirect addressing 1959. Architectures which are designed to be the targets of general purpose high order languages, which were developed about the same approximate time, and which remain within the traditional model of machine design are likely to prove equally suitable as targets for the same high level language.

1.2. Nebula Design Characteristics & Overview

Nebula is the name given to the instruction set architecture described in MIL-STD 1862 as part of the Army's Military Computer Family (MCF). The MCF represents an attempt to standardize computer resources throughout Army battlefield applications.[Kogg81]

The Nebula instruction set represents an attempt to standardize machine operations within an architecture suitable for a family of computers ranging from microprocessors

to super mini-computers. It provides a rich assortment of powerful features designed to permit the writing of concise, reliable code. Among these features are parameter objects directly accessible as instruction operands,[2] architectural support for high level language procedure interfaces, an orthogonal instruction set, an independent register set for each procedure, flexible addressing modes, and automatic size conversions for data objects of similar type.

Two of the objectives which guided the design of MIL-STD-1862 were efficient implementation of high order languages and performance optimization.[Kogg81] This optimization was not meant to pertain only to hardware; the intention was to design a high performance instruction set architecture which would be a good compiler target, would provide a good match for algorithmic constructs found in high level languages, and would be implementable by various hardware designs. Although Nebula was designed with the implementation of Ada as a focus, it was also intended that it support other modern HOL's.[Kogg81]

---

[2] MIL-STD 1862 leaves many of the details of implementation to the hardware designer. For example, a particular implementation of the standard is free to provide each procedure with an independent set of registers by hardware, software, or a combination of the two. Nebula's parameter operands will often be referred to in this study as "hardware parameters" to distinguish them as separate objects even though they might be implemented in software.

Nebula is a 32-bit, byte addressed machine with sixteen registers; fifteen are available for general use and one is used as the program counter. It has several advanced architectural features:[Kogg81]

1. Instruction formats in which opcodes and addressing modes are mutually independent. (orthogonality)
2. Automatic truncation/extension of different sized operands.
3. Sophisticated procedure call mechanisms and matching stack structure.
4. Consistent handling of interrupts, exceptions, traps, SVC's, and illegal opcodes.
5. A variable segment virtual memory system for relocation and protection.
6. Separate I/O controllers.

The first three of these are of interest in this study. The addressing modes available in Nebula reduce the need for temporary stores. Also, each procedure has its own set of registers; no special care need be taken by the programmer to save registers.[3]

All opcodes are one byte long and, where necessary, specify the number of operands to follow. A notable feature of Nebula's architecture is that there are no size conver-

---

[3] It will be seen later that in some cases it will be necessary to simulate sharing of registers in software. In particular, a mechanism must be devised to implement C's return of a function value.

sion instructions nor are there different forms of an instruction for different sized operands. For example, the opcode ADD suffices for adding 8, 16, or 32 bit quantities, and these quantities may be mixed in the same expression. Additionally, this single opcode is used whether the instruction governs 2 or 3 operands.[4] Size information is associated with the operands either implicitly as is the case with defined memory locations and hardware parameters, or explicitly by the programmer for memory locations referenced indirectly.

Nebula has the capability to address parameters directly as instruction operands. This is accomplished by creating a parameter descriptor for each parameter passed in the call instruction in the procedure's context stack. The parameter descriptor contains address, size and location information for each parameter. This allows the called procedure to operate on parameter operands in the same manner as it would any other type of operand. For example, if the actual parameter is a memory location, the subroutine can take the address of the formal parameter with Nebula's MOVA instruction. It will be seen later, however, that most parameters will still need to be passed manually because Nebula's underlying procedure call is by reference as

---

[4] VAX requires six separate opcodes to account for the same cases, viz. ADDB2, ADDB3, ADDW2, ADDW3, ADDL2, and ADDL3.

opposed to C's call by value.

Nebula's design goals reflect a growing criticism of the traditional Von Neumann architecture. Myers[Meye78] argues that the continued refinement of Von Neumann's design, rather than a complete departure from it into an architecture more suitable as a target for high order languages, has resulted in a semantic gap. This gap is defined as the difference between the concepts in a HOL and the concepts in a computer architecture.[Meye78] Nebula is still a Von Neumann based machine, i.e. a single sequential memory in which no distinctions are made between instructions and data. It does, however, provide architectural concepts which are directly related to HOL concepts such as procedure communication links via parameters, and array addressing. Thus it is an attempt to close the semantic gap as much as possible within the confines of a Von Neumann architecture.

1.3. Previous Work

Portable compilers are interesting when they significantly reduce the amount of work required of the compiler writer to implement a language on a new machine in a way which takes advantage of the features of the new machine.

The notion of a portable compiler has intuitive appeal because much of the work of the compilation process is machine independent. This includes the lexical scan, parse,

table management, error handling, and intermediate code generation as well as some aspect of code optimization. Pascal P-Code provides a good example of compiler design which uses a standard front end to produce intermediate code that becomes the source for different code generators. In order to produce code for a new target, the compiler writer need only construct a new code generator or backend for that target.

In practice the situation is not so simple. Compilers are among the most intensively used programs, so efficiency is an important consideration. The efficiency of each phase is affected by the input to that phase (the output of the prior phase.) This project provides a good example. Nebula's parameter addressing mode allows the programmer to refer to parameters in a subroutine directly by its position in the parameter list, i.e. first, second, third, etc. In order to make use of this feature, the code generator must have access to the fact that an operand is a parameter and also the position of that parameter in the parameter list. This differs from the usual method of passing parameters as temporary locations on the run time stack. The different requirements of the code generator filter back to the prior phases of the compiler so that the symbol table, for example, should rather reflect that an identifier is a parameter and its location in the parameter list and rely on Nebula's parameter descriptors in the context stack to keep informa-

tion as to size and location of the data object the parameter represents. What generally is stored is that the object is an integer, and its offset into the run time stack. Parameters are often treated no differently from automatic variables in the compiler.

Code optimization and resource allocation are areas where the details of a specific architecture need to be given special consideration. It is often the case that arithmetic operations such as multiplication and division require the use of specific register pairs. On some machines only specific registers may be used as index registers though in all other respects they serve as general registers. Thus it is often not enough to know the number of available registers. It is often necessary to know that a specific register is available and to make it available if it is not. For these reasons good portable compilers are extremely difficult to produce. On the one hand it is desirable to separate the compiler into distinct modules with well defined, orthogonal interfaces. On the other hand, the code generator needs to have its input tailored to allow it to approach the ideal of a one to one translation of input to target code as closely as possible.

The portable C compiler (PCC) supplied with the UNIX operating system has been described elsewhere.[John??] At the time of that writing, the compiler had served as the basis for compilers on roughly a dozen machines including

the Honeywell 6000, IBM 370, and Interdata 8/32. The compiler was originally designed to make two passes utilizing a file of intermediate code. In practice, the compiler can be converted to one pass operation and to emit target code during the syntactic analysis of the source code.

The goals which guided the design of the portable C compiler are:[John78]

1. Be easily movable to new hardware
2. Provide reasonable code quality initially yet allow tuning
3. Be self diagnosing
4. Use state of the art tools
5. Be compatible with existing support and system software

The PCC is a cross compiler written in C and intended to be used in a UNIX environment. The parser is generated by YACC but the lexical analyzer was constructed by hand rather than by use of LEX. The paper by Johnson[John78] points out a number of theoretical and practical considerations that went into the design of the compiler. The limited address space of the PDP-11, for example, constrained the method of resolving hash collision in the symbol table.[John78] Because research at the time had shown that the problem of identifying and eliminating common subexpressions in an expression is NP-complete, (even if register allocation is not considered), the decision was made to rely

on C's rich set of operators which tend to reduce the number of common subexpressions rather than spend a lot of compiler resources on this type of optimization.[John78]

The most sophisticated part of the compiler is the code generator. It will be discussed in more detail later. At this time it is simply noted that code is generated as each expression is recognized and that it is not generated by a simple walk of the expression tree built by the parser. For example, if it is found that a sub expression is to be computed and stored, this will be completed before attempting to compute the main expression. The idea is to have as many registers as possible available to compute the main expression.[John78]

The heart of the code generator is a table of templates which are matched against the current expression sub-tree and resource requirements to generate the given code. When a match is found, code is emitted. This process is not as simple as it might seem since there are input rewriting rules which can be applied as needed and the process is inherently recursive. Also, there is not a one to one correspondence between expression types and templates since the number of possible expressions in C, given it's ability to define types as needed, is potentially limitless.

1.4. Scope and Limitations

Only those features of Nebula which bear a direct relation to C language features are considered in this study. Thus Nebula's facilities for task management, virtual addressing, string processing, and exception handling will not be considered. Also those features which relate more to operating systems and process control such as memory management, service calls, and I/O controllers will be similarly ignored.

With respect to the modification of the portable C compiler, only a subset of C language features will be implemented. The use of the portable compiler is intended to provide a rough index of the degree of similarity between Nebula and currently available architectures. It was found that those features of Nebula which are furthest away from current architectures are the ones most difficult to represent in the abstract machine modeled by the PCC. It is recognized that the ease with which a portable compiler can be modified to target Nebula is ultimately of little relevance with respect to assessing Nebula's suitability as the target of a language. If it is determined that an architecture is a suitable target for a given language, we know that an adequate compiler can be built.

In the next chapter some criteria for assessing the suitability of an architecture as a target for a language

are developed and then they are applied to Nebula and C.

2. Nebula and C Compatibility

2.1. Introduction

The suitability of an architecture and its instruction set as a target for a high level language is directly related to the following:

1. Data types defined by the language are directly represented in hardware.
2. Conversions, which the language allows or enforces, of one data object into another type or size are supported by the architecture.
3. Primitive operations on data objects defined by the language have counterparts in the instruction set.
4. The data structures defined in the language can be easily implemented by the addressing modes available to the instruction set.
5. The language statements, especially with respect to control structures can be implemented by the instruction set in a straightforward manner.
6. The procedure interface defined by the language can be efficiently implemented by the instruction set.

Good alignment on the above criteria is sufficient to guarantee that a given architecture is a suitable target for a high level language.[1] Each of the six will be considered in turn.

---

[1] The six criteria given assume that we are constrained to a traditional Von Neumann architecture. It has been argued that no Von Neumann architecture can serve as an optimal target for modern high level languages. Myers suggests many extensions to the Von Neumann model to enhance its efficiency with respect to the implementation of modern high level languages.

2.2. Data Object Compatibility

2.2.1. Integers

There is straightforward compatibility between C data objects and Nebula address boundaries. The Nebula is a byte addressable machine with a 32-bit word size. The instruction set allows for legal addressing on byte, halfword, word, and double word boundaries. Thus any data object can be located on any boundary; however, on a given implementation of Nebula it may be more efficient to align particular objects on specific boundaries.

C distinguishes several fundamental data types. Characters (char) are objects which must be of sufficient size to store the integer code of the implementation's character set. Plain integers (int) are signed quantities in the natural size of the host machine. Integers can be characterized as short, long, or unsigned all of which may be the same size as plain integers; however, a long integer is guaranteed to be as large as a short integer.[Kern78] Characters and the various integer types are referred to as integral types. The size of the standard C types on various machines is given in Figure 2.1.[Kern78]

Nebula integers are represented in 2's complement notation. The architecture provides full support for 8, 16, and 32 bit integers including instructions for adding and subtracting with carry. Also, there are extended multiply and

	DEC PDP-11 ASCII	Honeywell 6000 ASCII	IBM 370 EBCDIC	Interdata 8/32 ASCII
char	8 bits	9 bits	8 bits	8 bits
int	16	36	32	32
short	16	36	32	32
long	32	36	32	32
float	32	36	32	32
double	64	72	64	64

Figure 2.1 -- Sizes of C Data Types on Some Machines

divide instructions which allow 64 bit results. Unsigned integer types are supported by instructions for unsigned addition, subtraction, multiplication and division. Modulus operations are supported by the REM instruction, which is analogous to the C '%' operator, and the MOD instruction which is similar except that the result always carries the sign of the modulo integer.

2.2.2. Float

The floating types in C are defined in two precisions. Single precision is called float, and double precision is called double. Nebula supports floating point data objects in word and double word lengths. By convention in C all floating point constants are represented as doubles.[Kern78]

Nebula provides extensive support for floating point numbers and operations. It distinguishes among five classes

of floating point numbers including plus and minus zero and plus and minus infinity. It has several maskable exception conditions including divide by zero, floating point overflow and floating point underflow.

2.2.3. Logical

C provides no distinct type for logical variables. It does provide operators for shifting integers a specified number of bits to the right or left. If the operand is defined as an unsigned integer, the right shift is guaranteed to be logical (zero-fill). The fill for right shift of integers is implementation dependent. Nebula treats logical operands in the same manner as unsigned operands described later.

2.2.4. Register

Nebula provides each procedure with a set of 16 general purpose registers. Of these however, register Ø is the program counter and register 1 is used as the stack pointer because of certain characteristics which will be described later. As will be seen later, register 2 will be reserved as a local stack pointer to reference a procedure's automatic variables and temporary storage. This leaves registers 3 to 15 available for general use. C provides a register type which allows the programmer to suggest to the compiler that a particular variable will be heavily used and should be kept in a register if possible.

2.2.5. Pointers

C pointers to any object are the size of a machine address. In Nebula this will be a word, the same size as an integer. Nebula provides an instruction to retrieve the address of a memory operand.

2.3. Data Object Conversion

C allows characters or short integers to be used wherever an integer may be used.[Kern78] Conversion from a shorter representation to a longer one always involves sign extension. Nebula specifies that all operands in signed integer arithmetic instructions are to be sign extended internally. The size conversion from an 8 bit through a 64 bit quantity is performed automatically.

When an integer is converted from a larger to a smaller representation both C and Nebula specify truncation of the high order bits.

All floating point arithmetic in C is performed in double precision. Single precision operands are converted to double precision by zero padding the fraction part. Nebula supports 32 and 64 bit floating point objects and, as with integers, size conversion is implicit.

C enforces a sequence of several object conversions referred to as the usual arithmetic conversions.[Kern78] First all character and short objects are converted to

integers, and all floats are converted to double. This conversion involves only lengths and so would require no intervention on the part of the compiler because of Nebula's automatic size conversion. Next, if either operand is double, C enforces the conversion of the other operand to double, and the result is double. Nebula has an instruction, FLOAT, which converts an 8, 16, or 32 bit signed integer to either a 32 or 64 bit floating point representation.

Next, if either operand is long, the other is converted to long and the result is long. Once again, only size conversions are involved and no intervention is required by the Nebula compiler. Next, if either operand is unsigned, the other is converted and the result is unsigned. Unsigned objects in C behave like unsigned objects in Nebula; both obey the laws of arithmetic modulo 2 to the n-th power where n is the number of bits in the representation. Unsigned integers are 8, 16, 32, or 64 bits in length.

C allows the coercion of one type into another by use of the cast operator. Nebula provides a specific instruction, FIX, which converts a 32 or 64 bit floating point quantity into an 8, 16, or 32 bit signed integer by means of truncation. There is an alternative instruction, RNDI, which rounds a 32 or 64 bit floating point representation to an integer value which is stored as a floating point number. C allows floating to integral conversions to be machine dependent.

C converts actual float arguments to double before a function call, and converts characters or short actual arguments to integers before the call.

Other conversions would not affect the internal representation of the object. A conversion of a pointer to one type of object to a pointer to another type of object would not require any modification in the executable code since both operands would be machine addresses.

Nebula supports the C type conversions very well. All length conversions are done implicitly in hardware and are transparent to the programmer or Nebula compiler. Specific instructions are provided to convert from C's integral to floating types and vice versa. There are no boundary restrictions for object addressing; this contributes to the simplicity of Nebula object manipulations.[2] As will be seen later, automatic conversion in conjunction with flexible addressing modes results in sparse assembler code.

---

[2] Addressability is not the only consideration in object alignment. The particular way in which an implementation fetches memory operands has a significant impact on the most efficient way to store objects. A good example is provided by the INTEL 8086 microprocessor which has 16 bit registers and a 16 bit internal bus. Although any byte is directly addressable, memory I/O fetches only words beginning at even boundaries. The unneeded byte is simply discarded. If an array of words is stored beginning at an odd address two fetch cycles, one for the high order byte and one for the low order byte, will be required to reference each element of the array. If the same array is stored beginning at an even address, only one memory fetch would be required for each element.

2.4. Primitive Operators Supported by the ISA

C's set of operators provide it with its conciseness and power. An ISA which can implement a given language operator in a single instruction is a more desirable target for that language than an ISA which must implement the operation as a series of instructions.[3] Nebula has the full complement of compare and test instructions including a version for unsigned quantities which would facilitate the comparison of C pointers. Nebula generally has an efficient sequence of instructions to implement the any of C's operators. It has a full set of branch and test instructions with which the relational operators can be implemented. The instructions are sufficiently varied so that operands of any type may be compared. Nebula has a number of unusual instructions which might be useful in limited contexts. For example there is an instruction to return the number of one bits in an operand and an instruction to scale an operand by a power of 2. There is an instruction to set (make all 1's) or clear (make all Ø's) an operand depending on the current state of the condition codes. The latter would be useful in

---

[3] For example, an architecture limited to an instruction to ADD and an instruction to form the bitwise one's complement of an object can support all the arithmetic operations, but given a choice who would choose it over an architecture with the full range of add, subtract, multiply, and divide instructions other things such as cost being equal? Direct implementation of primitives can also have the desirable side effect of allowing the hardware to check for run time errors. The generally available DIVIDE by ZERO check is an example.

boolean operations. C's sizeof operator has an analogous Nebula instruction which returns the size in bytes of its operand though it is limited to Nebula objects.

2.5. Support for Data Structures

C gets much of its power from its ability to define higher level data objects such as arrays and structures. A structure in C is a collection of objects which may either be one of the fundamental types listed above or one of the defined higher level types. An array is a sequential series of one of the fundamental or defined types. The suitability of an ISA to implement these defined data objects depends on the scope and flexibility of its addressing modes.

Nebula has ten addressing modes, including three for directly addressing parameters.[4] These addressing modes and their effects are illustrated in Figure 2.2.

The non-parameter modes are divided into memory and non-memory modes. The non-memory modes consist of the short literal mode which can access an unsigned constant in the instruction stream up to 5 bits, a literal mode, which can access signed quantities of up to 8 bytes in the instruction stream, and a register mode.

---

[4] In the discussion of addressing modes, the symbol '#' indicates a literal value, '%' a register, and '?' a parameter.

MODE	EXAMPLE	VALUE IN %4 AFTER OPERATION
Short Literal	mov #5,%4	5
Literal	mov #-5,%4	-5
Register	mov %3,%4	Copy of %3
Register Indirect	mov @%3^B,%4	Sign extended contents of the byte at address in %3
Register Indexed	mov 8(%3)^W,%4	Sign extended contents of the word at address 8 + %3
Absolute	mov @(6)^H,%4	Sign extended contents of the halfword at address 6
Parameter	mov ?3,%4	Value of parameter 3
General Parameter	mov ?(%3),%4	Value of the parameter number indicated by %3
Unscaled Index	mov @%3(%2)^W,%4	Sign extended contents of the word at the address formed by adding the contents of %2 to the contents of %3
Scaled Index	mov @%3[%2]^W,%4	Sign extended contents of the word at the address formed by scaling the contents of %2 by the size of a word (4 bytes) and adding the result to the contents of %3

Figure 2.2

The modes which access memory contain size specification bits in the operand specifier which indicate the size of the quantity being addressed. All memory references may be to 1, 2, 4, or 8 byte quantities also referred to as byte, halfword, word, and doubleword quantities. The size is

obtained either from the assembler for direct storage references, or it can be given in the operand specifier as a postscript for indirect storage references.

The first of the memory modes is the register indirect mode which specifies a register which contains the effective address of the object. There are two versions of the register indexed mode in which the effective address of the operand is found by adding either a byte or word displacement to the contents of a specified register. An absolute mode is provided in which the address of the operand is specified explicitly in the operand specifier.

There is a short version of the parameter mode and an extended version. The short mode is used to specify a parameter from 1 to 7. The extended mode can specify a parameter in the range Ø to 255. The implementation of the parameter mode of addressing will be discussed in detail later.

The last three modes are the compound modes. Each of the compound modes includes one or more of the non-compound modes described above as part of the operand specifier. The general parameter mode uses any of the non-compound modes which can be evaluated to an unsigned integer, and then uses that integer as the parameter to access. For example, suppose register three contains the address of a byte in memory and that this byte in memory contains the number of the

parameter to be used in the operation. Then what is required is the general parameter mode using register indirect addressing. Suppose the byte in memory contains the value 4, then the following instruction would move the value associated with parameter number 4 into register 5.

```
mov     ?(@%3)^B,%5
```

The second compound mode is the unscaled index mode. The first operand specifier is followed by any of the non-compound modes which evaluate to a signed integer quantity. This quantity serves as the index. The second operand specifier is any of the non-compound modes which evaluates to an address. This address serves as the base. The effective address of the operand is then the base plus the index.

The last of the compound modes is scaled index mode which is similar to unscaled index mode except that the index is scaled by the object size. The size is taken either from the definition of the base specifier or is provided explicitly in the instruction. Figure 2.3 lists Nebula's addressing modes with the number of bytes required for each.

As mentioned earlier the different addressing modes are used to implement common data structures. Much of the power of Nebula derives from the flexibility of its addressing modes in combination with implicit size conversion for

MODE	SIZE
Short Literal	1 Byte
Literal	1 + Data Bytes
Register	1 Byte
Short Parameter	1 Byte
Extended Parameter	2 Bytes
Register Indirect	1 Byte
Register Byte Indexed	2 Bytes
Register Word Indexed	5 Bytes
Absolute	5 Bytes
General Parameter	1 + Non-Compound Bytes
Unscaled Index	1 + Base + Index Bytes
Scaled Index	1 + Base + Index Bytes

Figure 2.3 -- Nebula Addressing Modes and Sizes

object of the same type. In addition, the orthogonality of its instruction set greatly simplifies the code needed for typical operations.

For example, Figure 2.4.c is a function which adds together two short integers and stores the result as an

```
function sum(a,b,c)
short *a, *b;
int *c;
{
        int i,n;

        for ( i = 0; i < n; i++ )
                c[i] = a[i] + b[i];
}
```

Figure 2.4.c

integer.

One version in Nebula is shown in Figure 2.4.neb.[5] Assume short integers are stored as halfwords and integers are stored as words.

Nebula's scaled index mode in conjunction with implicit size conversions and complete orthogonality of its instruc-

```
                clr     %4
forloop:        add     a[%4]^H,b[%4]^H,c[%4]^W
                iblss   %4,n,forloop
```

Figure 2.4.neb

[5] Actually Nebula would not require the size specifications in this case; they are shown only for clarity.

tion set allows the loop operation to be expressed in assembler in nearly as concise a form as in the high level language. By way of comparison, the same loop in VAX might look as in Figure 2.4.vax.

2.6. Support for Subprograms

2.6.1. The Nebula Context

As a design, Nebula provides a clean and efficient mechanism for the implementation of subprograms.[6] The fundamental unit of execution in Nebula is the procedure. Associated with each procedure is a context that defines the current state of the machine within which a given procedure is executing. A context includes information about the processor status word (PSW) that contains control information related to the current context, the register set for the procedure, the descriptors for each parameter associated

```
                clrl    R4
forloop:        cvtwl   a[R4],RØ
                cvtwl   b[R4],Rl
                addl3   RØ,Rl,c[R4]
                aoblss  n,R4,forloop
```

Figure 2.4.vax

[6] It will be seen later that this design has some limitations for a C compiler.

with the procedure, and the location of the exception handler associated with the procedure.

Independent registers for each procedure mean that there is no need for the compiler to save and restore registers; the context for each procedure provides this service automatically. A procedure declares the number of registers it will use. This number can be from 1 to 15. Upon entry, the values in the registers are undefined with the exception of the value in register 1 which is inherited from the caller which allows it to be used as a stack pointer.

The context stack in Nebula is not equivalent to the run time stack associated with executing procedures. The compiler must maintain a procedure's local stack. In general the management of the context stack is transparent not only to the applications programmer, but also to non system programs such as compilers. References to stack shall be taken to mean the run time stack which is managed by the compiler for storage of temporaries and automatic variables.

2.6.2. Parameter Passage

Nebula provides explicitly for parameter variables. The actual parameters in the call statement are available to the called procedure in the same order in which the parameters appear in the call statement. As many as 255 parameters can be passed, though this requires a special form of the call instruction. Parameters are available by referencing the

parameter number. The Nebula assembler refers to '?n' where "n" is the parameter number. Parameter operands are valid operand specifiers in Nebula instructions and can often be used to simplify the code.

The parameter number is an index into to the list of parameter descriptors which is maintained by the architecture in the procedure's context stack. This descriptor contains size, location, and address information for each actual parameter. The underlying parameter passing mechanism for Nebula is by reference. It is legal for a parameter operand to be passed as an actual parameter. In this case the original callers variable space is accessed by reference to the parameter. For example if P1 calls S1 with register 4 as the only parameter, and S1 calls S2 with its ?1 as the second parameter in the operand list, then any assignment S2 makes to its ?2 will be reflected in P1's register 4.

In general three types of objects comprise the actual parameters in a Nebula procedure call. The caller might pass a literal value, a register, or a memory location. Both register and memory parameters are read/write; a change to the formal parameter operand in the called procedure will change the actual parameter in caller's variable space. In addition an attempt to change the value of a parameter operand whose actual parameter is a literal value will cause a run time exception in Nebula because a literal parameter is read only.

This is contrary to C which passes all parameters, except arrays, by value. An array is passed by reference to the address of its first element. If the application requires a reference parameter, the C programmer must explicitly pass the address of the variable. This is accomplished either by declaring a variable to be a pointer to the type of data object in question, or by using the unary operator '&' which takes the address of the variable. The procedure accepting the parameter must declare the formal parameter to be a pointer to the type of data object being passed. The difficulties which might be encountered are illustrated by the typical C function in Figure 2.5.[Kern78]

The use of the formal parameter "n" as the control variable in the "for" loop will cause the compiler some difficulty. In the code generated for the above function, the variable "n" would be referred to as "?2" or parameter

```
power(x, n)                /* raise x to n-th power; n > 0 */
int x, n;
{
        int p;

        for ( p = 1; n > 0; n-- )
                p = p * x;
        return(p);
}
```

Figure 2.5 -- Segment of C Source Code

number two. In a straight forward expansion of the source code into Nebula code, one might find the instruction to decrement ?2. If the function were called with a literal value as the actual parameter, an illegal write exception would be raised at run time because ?2 would reference a read only object. If the call were made with a register or memory location as the actual parameter, the function would have the unexpected and, in C, illegal action of changing the caller's register or memory value.

One solution to the problem is to ignore Nebula's hardware parameters and to pass arguments in a conventional manner making use of the stack. The cost of this approach is increased size of the code generated and the loss of Nebula's parameter operands. Using Nebula's hardware parameters can result in fewer instructions in the called program and in the savings of register resources also. In some cases using the parameter addressing mode will save the called function from having to move an address into a register and then using the register as an index. In other cases using a parameter operand will result in an instruction of fewer bytes than an instruction using an alternative addressing mode.

If the hardware parameters are to be used, either the caller or the called function must make a local copy of the actual parameters which are literals or value parameters. The method selected for this project has been to have the

caller make the local copy of any parameter which is not a reference parameter on its stack and then to pass the copy as the actual parameter. This insures that all parameters will be writable and that the caller's variable space will not be corrupted by assignments in the code of the called procedure. If a reference type such as a pointer or an array is being passed as an argument, then the variable itself will be used in the calling sequence and a local copy will not be made. Upon return from the called procedure, any local copies will be discarded to prevent uncontrolled growth of the stack.

The code in the called routine will be free to make whatever use of the hardware parameters and addressing modes are most suitable to the task at hand.

2.6.3. Functions vs. Procedures in C

No distinction is made in C between function and procedure subprograms. All subprograms return a value; if not explicitly declared, this value is assumed to be of type integer. C provides two ways to exit from a procedure. Either a value is returned by an explicit "return" statement, or no explicit value is returned.

A conventional manner of passing a return value to the caller is to utilize one of the shared registers. In Nebula this is not possible since the caller's registers, with the exception of any which were passed as parameters, contain

the values they had prior to the call. A simple solution is to pass one of the caller's registers as the first parameter to all functions. Upon return, this register will contain any value assigned to it by the called procedure. Register 3 will be used for this purpose.

3. Nebula and VAX Comparison

Nebula was intended to be a good target for high level languages. This chapter will compare the relative efficiency of Nebula and VAX implementations of segments of C code with respect to the size of object code in bytes.

The focus will be on the capacity of the respective ISA's to implement common, representative operations. Thus the difficulties of parameter passing which have been dealt with in the previous chapter will not be addressed again here. A related issue which will not be addressed is the amount of complexity one is willing to introduce into the compiler in order to take advantage of a specific feature of a given architecture. The analysis of this chapter will focus only on the instruction set available; the assumption is made that when faced with an actual compiler decision, the designer will make those trade offs between complexity of design and implementations of language features based on the current environment.

In an unpublished study comparing several architectures' ability to implement Ada, Anderson,[Ande85] used a similar method of analysis to compare Nebula with other available architectures. The architectures compared were MIL-STD1750A, AN-UYK/43, IBM/370, PDP-11, and VAX, though VAX was not compared in all cases. With the exception of VAX, Nebula code was significantly smaller for all the pro-

grams examined. For those programs in which VAX was compared, the results were about the same as with Nebula. This result is not surprising given that both Nebula and VAX were the newest of the architectures reviewed by that study.

Register structure and usage restrictions must be considered at almost every stage in the compilation process. Efficient register management is one of the more difficult and time consuming aspects of code generation. Both machines have sixteen 32-bit general purpose registers. On the VAX registers 12, 13, 14, and 15 have been reserved for use as the stack pointer, frame pointer, argument pointer, and program counter respectively. All registers are globally available; thus it is common to pass data in either direction between procedures.

This differs considerably from the situation in Nebula. Only register Ø is specifically reserved as the program counter. All other registers are available for general use; however, each procedure has its own set of registers. In Nebula, registers have scope in a manner analogous to automatic variables. In particular, with a single exception, a called procedure does not inherit the values contained in the caller's registers. The exception is register 1 which does inherit the value of the caller's register 1.[1] However, this is a one-way link; the called procedure

[1] In practice, the situation is slightly more complicated. The value in register 1 is inherited only if that

cannot return a value in register 1. Register 1 is also incremented and decremented by Nebula's PUSH and POP instructions.

Another difference which has significant implications for the compiler is Nebula's parameter addressing mode. Parameters are listed explicitly as arguments in the syntax of the procedure call; they are then available to the called procedure as directly addressable objects. Because Nebula passes its parameters by reference while C defines most parameters as value parameters it is not possible to rely entirely on Nebula's facilities for parameter definition. An additional method of communication must be enforced by the compiler. The proposed mechanics of the implementation will be discussed later.

A third area of significant difference is Nebula's automatic conversion of objects to the appropriate size. There are no Nebula equivalents of the VAX instructions which convert byte objects to words or words to longs. To add an integer stored as a word to an integer stored as a halfword no conversions need be done nor do special forms of the ADD instruction need be used. Sign extension for signed

---

register exists in both the caller and called procedures, and if the call is not the result of a task initiation, interrupt, or trap invocation. Register 1 may not be available in either the caller or called procedure because at entry each procedure declares the number of registers it will require. In these cases the called procedure's register 1 is undefined.

arithmetic is provided automatically by the architecture. In VAX, the programmer must explicitly insure sign extension for signed arithmetic by converting the halfword to a word and then using that result as the operand to the subsequent ADD instruction. This strict orthogonality allows Nebula to work with a reduced set of opcodes and simplifies the compiler.

A final area of difference which has a significant impact on the number of bytes of code generated is the flexibility of the addressing modes available in Nebula. Aside from the parameter mode mentioned earlier, Nebula allows more direct addressing than does the VAX. A typical example is indexed addressing.

Both machines have an index mode of the general form "base[index]." VAX, however, requires that the index be a register. This generally results in the need for an additional instruction to move the value of the index into an available register. Nebula does not have this constraint. For example, to access the i-th element of a globally defined array, say A, where the value of i is stored in a local variable, the Nebula operand would have the form

```
mov     A[24(%2)],%4 [2]
```

where register 2 contains the base address of a block of

[2] Actually Nebula provides an even more explicit mode of reference to array objects. In the above example we might

storage and the variable "i" is offset from this base by 24 bytes. This is in contrast to a reference to the same data object on the VAX which might have the form

```
mov     24(R2),R3
mov     A[R3],R4
```

Beside saving the code for the additional move instruction, the Nebula version saves the register resource and the time to allocate it.

Some examples will illustrate the differences. Figure 3.1.c is a C function to compute the vector inner product of two vectors. Figure 3.1.neb is the same function coded in Nebula, and Figure 3.1.vax is the VAX code. The code in

```
float function VIP(a,b,n,result)
float a[],b[],*result;
int n;
{
        float tempf;
        int i;

        *result = 0.0;
        for ( i = 0; i < n; i++ )
                *result = *result + a[i] * b[i];
        return;
}
```

Figure 3.1.c -- VIP in C

have been accessing the 5th. word in an array. An equivalent instruction in Nebula would be mov A[#5],%4. The index is automatically scaled by the size of the object referenced by A.

```
2                clrf            result
2                clr             i
3                cmp             i,n
2                bgeq            done
8        top:    mulf            a[i],b[i],tempf
3                addf            tempf,result
4                iblss           i,n,top
         done:   <continue>
```

Figure 3.1.neb -- VIP in Nebula

```
3                clrf            result
2                clrl            i
4                cmpl            i,n
2                bgeq            done
8        top:    mulf3           a[i],b[i],tempf
4                addf2           tempf,result
5                aoblss          n,i,top
         done:   <continue>
```

Figure 3.1.vax -- VIP in VAX

Figures 1.neb and 1.vax do not reflect the actual appearance of the code in an implementation of the VIP function. Symbolic names were used rather than the assembler mnemonics for parameters and registers in order to make the code easier to read. Thus a, b, result, and n are all parameters and can be accessed by the one byte parameter operand mode. Tempf and i are registers.

For sake of reference, Figure 3.2.neb and Figure 3.2.vax give the code for VIP in assembler mnemonics. In future cases this will not be done. We know that we can

```
        clrf            ?4
        clr             %1
        cmp             %1,?3
        bgeq            done
top:    mulf            ?1[%1],?2[%1],%2
        addf            %2,?4
        iblss           %1,?3,top
done:   <continue>
```

Figure 3.2.neb -- VIP in Nebula Mnemonics

```
        clrf            @16(ap)
        clrl            R1
        cmpl            R1,12(ap)
        bgeq            done
top:    mulf3           @8(ap)[R1],@4(ap)[R1],R2
        addf2           R2,@16(ap)
        abolss          12(ap),R1,top
done:   <continue>
```

Figure 3.2.vax -- VIP in VAX mnemonics

always convert code with symbolic names into code utilizing assembler mnemonics.

The total number of bytes for the Nebula code is 24; the number of bytes for each instruction is given in the left hand column. The total number of bytes for the VAX code is 28. Inspection shows that each of the 4 additional bytes is related to a parameter access. Nebula functions with 7 or fewer parameters can address parameters with one byte. Functions with more than 7 parameters must use the extended parameter mode of operand access which requires two

bytes. Extended parameter mode can access up to 255 parameters. Since few functions have more than seven parameters, Nebula will generally take fewer bytes than VAX for functions with parameter access.

Figures 3.2.neb and 3.2.vax again illustrate the orthogonality of the Nebula ISA. Whether an arithmetic operation has two or three operands, the opcode format is the same in Nebula. In VAX the instruction opcode has a 2 and 3 operand format. In addition, the VAX opcode specifies the size of the operands while the Nebula code needs no length specification.[3]

A slight change in the specifications illustrates one of the most significant advantages that Nebula has over VAX. If the two arrays are stored as 16 bit integers (halfwords in Nebula; words in VAX) and the result is to be a 32 bit quantity (word in Nebula; longword in VAX) the only changes needed in the Nebula version would be to change the clrf, mulf, and addf instructions to their integer equivalents of clr, mul, and add respectively. No additional code would be required because size conversion is implicit. This is in

[3] The VAX convention of specifying operand size as part of the opcode is useful in those situations in which the operands are an indirect reference to memory. In this case no further size information need be provided. In Nebula, one is often required to state the size of a memory operand explicitly if this information is not already known to the assembler. In the case of parameters no additional specification is needed since the architecture maintains this information in the parameter descriptor.

marked contrast to the VAX version which would need to change not only the opcode, but also add the instructions to do the conversions from word to longword quantities before the arithmetic operations were performed. In addition to the extra code, VAX would require the temporary storage locations to hold the converted results.

Figure 3.3.c[Kern78] is a C function to perform a binary search on a sorted array and to return the index of the target if it is found and a -1 if it is not found. It has been selected because it is typical of functions which use control loops and parameters.

```
binary(x,v,n)
int x, v[], n;
{
        int low, high, mid;

        low = 0;
        high = n - 1;
        while ( low <= high ) {
                mid = ( low + high ) / 2;
                if ( v[mid] < x )
                        low = mid + 1;
                else if ( v[mid] > x )
                        high = mid - 1;
                else
                        return ( mid );
                }
        return ( -1 );
}
```

Figure 3.3.c -- Binary Search in C

```
2                clr             low
4                sub             #1,n,high
3        again:  cmp             low,high
2                bgtr            nomat
4                add             low,high,result
3                div             #2,result
5                cmp             v[result],x
2                beql            done
2                blss            lower
4                sub             #1,result,high
2                br              again
4        lower:  add             #1,result,low
2                br              again
3        nomat:  mov             #-1,result
         done:   <continue>
```

Figure 3.3.neb -- Binary Search in Nebula

Figure 3.3.neb shows the Nebula code for Binary Search. We can assume without loss of generality that n, v, x, and result are parameters, and that high and low are registers. The total number of bytes for the Nebula code is 42.

Figure 3.3.vax is the code for Binary Search in VAX. The total number of bytes is 44. As in the previous case x, v, and n are taken to be parameters while high and low are registers. In VAX, result can also be treated as a register. Figures 3.3.neb and 3.3.vax demonstrate another of the differences between Nebula and VAX. Since the caller has no access to the registers of the called program, all function results must be passed back via a parameter. This is in contrast to the situation in VAX where function results are commonly passed back in a designated register.

```
2                clrl            low
5                subl3           #1,12(ap),high
3        again:  cmpl            low,high
2                bgtr            nomat
4                addl3           low,high,result
3                divl2           #2,result
6                cmpl            v[result],x
2                beql            done
2                blss            lower
4                subl3           #1,result,high
2                brb             again
4        lower:  addl3           #1,result,low
2                brb             again
3        nomat:  movl            #-1,result
         done:   <continue>
```

Figure 3.3.vax -- Binary Search in VAX

```
struct tnode {
        int t_item;
        struct tnode *t_left, *t_right;
        }

        btsearch(p,x)
        struct tnode *p;
        int x;
        {
                if ( p != NULL )
                        if ( x < p->t_item )
                                p = btsearch(p->t_left, x);
                        else if ( x > p->t_item )
                                p = btsearch(p->t_right, x);
                return (p);
        }
```

Figure 3.4.c -- Recursive Binary Tree Search

Figure 3.4.c is a C function to recursively search a binary tree. It returns NULL if the item is not found, and a pointer to the node if the item is found. Recursion is

directly supported in Nebula. But it is also very well supported in VAX.

Figures 3.4.neb and 3.4.vax illustrate a version of btsearch in Nebula and VAX code respectively. The code for both the Nebula and VAX versions assumes that a node

```
3                   mov             ?1,?
3                   cmp             ?1,%1
3                   cmp             ?1,NULL
2                   beql            done
4                   cmp             ?2,Ø(%1)^W
2                   beql            done
2                   bgtr            gtr
9                   call            btsearch, 4(%1)^W,?2,?3
2                   br              done
9           gtr:    call            btsearch, 8(%1)^W,?2,?3
1           done:   ret
```

Figure 3.4.neb -- Recursive Binary Tree Search in Nebula

```
4                   movl            4(ap),RØ
3                   cmpl            RØ,NULL
2                   beql            done
4                   cmpl            8(ap),(RØ)
2                   bgtr            gtr
2                   beql            done
3                   pushl           8(ap)
3                   pushl           4(RØ)
6                   calls           #2,btsearch
2                   brb             done
3           gtr:    pushl           8(ap)
3                   pushl           8(RØ)
6                   calls           #2,btsearch
1           done:   ret
```

Figure 3.4.vax -- Recursive Binary Tree Search in VAX

structure has been defined such that the word at offset 0 contains the data value, the word at offset 4 contains a pointer to the left descendant, and the word at offset 8 contains a pointer to the right descendant.

In this case, the Nebula code requires 40 bytes while the VAX code needed 44 bytes. Nebula's calling sequence needed only 9 bytes compared with the 12 needed by VAX. As shown in Figure 3.5 Nebula's code is marginally smaller in some cases than the code for VAX, but when one takes into account the creation of temporaries needed to compensate for Nebula's call by reference mechanism discussed earlier, this advantage is soon lost. Some examples in the next chapter which show actual code generated by the VAX and Nebula implementations of the PCC will give some indication of what might be expected in practice.

Program	Nebula	VAX
VIP	24	28
Binary Search	42	44
BT Search	40	44

Figure 3.5

4. Portable C Compiler Implementation

The modifications made to the portable C compiler in order to implement a subset of C will be outlined in this chapter. Examples of code generated by the modified compiler will be compared with VAX code generated by the PCC distributed with the college's version of UNIX.

The portable C compiler was chosen for this project so that some actual experience generating Nebula code could be acquired as quickly as possible. It was not the intention of this study to make an assessment of the PCC or to make observations about the suitability of Nebula as a target for C based on the experience with the PCC. Rather it was intended that the experience of converting the compiler for a subset of C would give a flavor of the difficulties which might be encountered when attempting a complete implementation for Nebula.

The mechanics of the compiler and the philosophy behind the design are described in two papers by S.C. Johnson. [John78] [John??] Briefly the compiler attempts to specify a model of an abstract machine. Retargeting the compiler should amount to a respecification of the model. The more closely an actual machine resembles the abstract machine, the easier the retargeting process. For the most part, Nebula was very similar to the abstract machine. Nebula has one class of general registers, a conventional word size, no

restrictions on register usage, and does not utilize register pairs.[1] As might have been expected, generating a procedure call and accessing a parameter in a subroutine caused the most difficulty.

4.1. PCC Compiled Code

4.1.1. Null Function

Figures 4.1 show a C program which calls a null function along with the Nebula code generated by the modified compiler and, for comparison, the VAX code generated by the PCC. In the Nebula code the assignment of symbolic names to registers 1 and 2 would ordinarily not be part of the compiler output. Normally these assignments would be made as

```
main()
{
        dummy();
}
dummy()
{
}
```

Figure 4.1.c

[1] Nebula does provide for 64 bit operands; however, registers are defined to be word operands. Therefore, if a register is specified as the target of an operation which generates a result which is too large to fit into a word, the high order bits are lost and the T (truncate) condition code is set.

```
; PCC generated nebula code--Sun Sep  7 21:Ø2:53 1986

        .sect   bss,data
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=Ø,reg=15

        br      L14
L15:
        call    _c_dummy,%3
        ret
L14:
        br      L15
        .sect   bss
        .sect   prog

        .global _c_dummy

_c_dummy:               .entry  np=1,reg=15

        br      L19
L2Ø:
        mov     %3,?1
        ret
L19:
        br      L2Ø
        .sect   bss
```

Figure 4.1.neb

```
        .data
        .text
        .align  1
        .globl  _main
_main:
        .word   L12
        jbr     L14
L15:
        calls   $0,_dummy
        ret
        .set    L12,0x0
L14:
        jbr     L15
        .data
        .text
        .align  1
        .globl  _dummy
_dummy:
        .word   L17
        jbr     L19
L20:
        ret
        .set    L17,0x0
L19:
        jbr     L20
        .data
```

Figure 4.1.vax

part of the run time environment established by the system. They are shown here for clarity.

In the Nebula code the entry to each function is clearly identified and the number of parameters and registers available to a function are declared. By convention all registers were made available to each function. The number of parameters depends on the source code. Even though there are no parameters to the function, dummy, in the source code, the compiler was made to generate a dummy

parameter, register 3, in case the return value from the function was required. Even the null function generates at least one data transfer instruction to move a return value to parameter 1 prior to a function return.[2]

Most of the modifications were straightforward. The external function names were changed from "_" to "_c_" in the file local.c. The entry statement was generated in code.c. The major difficulty was keeping track of the number of parameters. This was accomplished in pftn.c where a parameter table is managed. The index into the table is also the number of parameters. It is interesting to note that the number of parameters is available directly to the code which generates the calling sequence in the VAX version. Unfortunately, the data is needed by the called function in Nebula and it has already been discarded.

---

[2] This is an expedient. The PCC is designed to allow the designation of one register to contain the return value of a function. Parameters as addressable objects are unknown to the VAX version of the compiler. Rather than get too involved in the details of the PCC at this early stage, by insuring that the data move instruction would be generated only when necessary and transferred into parameter 1, it was easier to change the specification of the return register and always generate a the move instruction just prior to a function return. These extra instructions would be early candidates for deletion when tuning the compiler. Also, the caller could take advantage of Nebula's parameter mechanism by including the left hand side of an assignment to a function value in the parameter list in the call instruction. This would save the data move instruction at the function return.

4.1.2. Automatic Variables

Figures 4.2 show a C program which allocates temporary storage for automatic variables in both the main program and in a function. One of the differences between Nebula and VAX is the use of registers for special purposes. VAX creates a frame pointer automatically; it is actually register 13. The file local2.c has a table, rnames, which associates a symbolic name with a register. It also has a table, rstatus, which describes the uses to which a register may be put. In VAX, for example, register 12, the stack pointer, can not be used as a scratch register. These tables were changed to reserve registers 1 and 2 in Nebula for the stack pointer and local stack pointer respectively,

```
main()
{
        int a,b,c;
        a = 3;
        b = 2;
        dummy();
}

dummy()
{
        int a,b,c;
        a = 3;
        b = 2;
}
```

Figure 4.2.c

```
; PCC generated nebula code--Sun Sep  7 21:03:01 1986

        .sect   bss,data
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                    .entry  np=0,reg=15

        br      L14
L15:
        mov     #3,0(lsp)^W
        mov     #2,4(lsp)^W
        call    _c_dummy,%3
        ret
L14:
        sub     #12,sp
        mov     sp,lsp
        br      L15
        .sect   bss
        .sect   prog

        .global _c_dummy

_c_dummy:                   .entry  np=1,reg=15

        br      L19
L20:
        mov     #3,0(lsp)^W
        mov     #2,4(lsp)^W
        mov     %3,?1
        ret
L19:
        sub     #12,sp
        mov     sp,lsp
        br      L20
        .sect   bss
```

Figure 4.2.neb

```
        .data
        .text
        .align  1
        .globl  _main
_main:
        .word   L12
        jbr     L14
L15:
        movl    $3,-4(fp)
        movl    $2,-8(fp)
        calls   $0,_dummy
        ret
        .set    L12,0x0
L14:
        subl2   $12,sp
        jbr     L15
        .data
        .text
        .align  1
        .globl  _dummy
_dummy:
        .word   L17
        jbr     L19
L20:
        movl    $3,-4(fp)
        movl    $2,-8(fp)
        ret
        .set    L17,0x0
L19:
        subl2   $12,sp
        jbr     L20
        .data
```

Figure 4.2.vax

and to make registers 3 through 15 available as general purpose registers. There are also global constants, for example ARGREG and STACKREG, which identify special purpose registers. These were changed to conform to Nebula requirements.

The code for temporary storage allocation is also generated in local2.c. The only changes which were required were to change the instruction mnemonics and to add the code to generate the instructions to create the local stack pointer. The local stack pointer is analogous to the frame pointer in VAX, and is used by a procedure to access its automatic variables. Since the register sets for each procedure are independent, there is no need to have either the caller or the called function restore the lsp to its prior value upon subroutine exit.

The final features of note are the stack offsets for the automatic variables. It was decided to use positive offsets in Nebula in contrast to the negative offsets in VAX. This change was accomplished by setting a single switch in the compiler and is a testament to the well thought out design of the original PCC.

4.1.3. Simple Expressions, Parameters, Return Values

Figures 4.3 show the code for a function which returns the product of its two arguments. It also contains an ADD instruction in the calling program for comparison. The Nebula code for the addition operation illustrates the simplicity of the opcodes. The VAX opcode must specify the size of the operands and whether it is the two or three operand format. Nebula need only add the size indicator in the operand specifier. It is noteworthy that if one operand

```
main()
{
        int a;
        int c;
        int d;
        c = a + d;
        d = dummy3(4,5);
        a = dummy3(c,d);
}

dummy3(a,b)
int a,b;
{
        return(a*b);
}
```

Figure 4.3.c

```
; PCC generated nebula code--Sun Sep  7 21:04:32 1986

        .sect   bss,data
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=0,reg=15

        br      L14
L15:
        add     8(lsp)^W,0(lsp)^W,%3
        mov     %3,4(lsp)^W
        push    #4
        push    #5
        call    _c_dummy3,%3,4(sp)^W,0(sp)^W
        add     #8,sp
        mov     %3,8(lsp)^W
        push    4(lsp)^W
        push    8(lsp)^W
        call    _c_dummy3,%3,4(sp)^W,0(sp)^W
        add     #8,sp
        mov     %3,0(lsp)^W
        ret
L14:
        sub     #12,sp
        mov     sp,lsp
        br      L15
        .sect   bss
        .sect   prog

        .global _c_dummy3

_c_dummy3:              .entry  np=3,reg=15

        br      L19
IL20:
        mul     ?3,?2,%3
        mov     %3,?1
        ret
        mov     %3,?1
        ret
IL19:
        br      L20
        .sect   bss
```

Figure 4.3.neb

```
        .data
        .text
        .align  1
        .globl  _main
_main:
        .word   L12
        jbr     L14
L15:
        addl3   -12(fp),-4(fp),r0
        movl    r0,-8(fp)
        pushl   $5
        pushl   $4
        calls   $2,_dummy3
        movl    r0,-12(fp)
        pushl   -12(fp)
        pushl   -8(fp)
        calls   $2,_dummy3
        movl    r0,-4(fp)
        ret
        .set    L12,0x0
L14:
        subl2   $12,sp
        jbr     L15
        .data
        .text
        .align  1
        .globl  _dummy3
_dummy3:
        .word   L17
        jbr     L19
L20:
        mull3   8(ap),4(ap),r0
        ret
        ret
        .set    L17,0x0
L19:
        jbr     L20
        .data
```

Figure 4.3.vax

to the addition was a halfword, the Nebula code, except for the size specifiers, would not be changed at all; the VAX code would first need to make a size conversion from half-word to full word size and then use the converted value.

VAX pushes its two parameters onto the stack in reverse order just prior to the function call and indicates the number of parameters as the first argument to the call. This allows the automatic framing mechanism to know the number of stack locations to release upon return. Nebula lists the parameters as operands in the call statement. In this case, because the arguments were literal values, the Nebula implementation of the PCC first had to create stack temporaries for the parameters and pass the temporaries as the arguments.[3] The VAX calling mechanism automatically releases the stack space for its parameters upon return. The Nebula implementation must adjust the stack by adding the requisite number of stack locations when the function returns.

The modifications to the calling sequence required more involved changes to the source code. The function gencall in file local2.c controls the creation of the calling sequence. It was modified to traverse the expression subtree for the subroutine call in reverse order.

Figure 4.4 shows two new functions written for the Nebula implementation. Genargtemps generates the Nebula code to create the stack temporaries and atcnt returns the

[3] If the literals had been passed there would have been no difficulties in this particular case since the function did not update the value of the arguments. In general, the compiler must take precautions to make all arguments writeable.

```
#ifdef NEBULA

atcnt( p )
register NODE *p;
{
        int count = 0;

        while ( p->in.op == CM ){
                if ( !ISPTR(p->in.right->in.type) )
                        count++;
                p = p->in.left;
        }
        if ( !ISPTR(p->in.type) )
                count++;
        return(count);

}

genargtemps( p, ptemp ) register NODE *p, *ptemp; {
        register NODE *pasg;
        register NODE *ql, *qr;
        register align;
        register size;
        register TWORD type;

        /* generate code to make all arguments writeable and to
           create local temps for c value passed arguments */


        /* do arguments from left to right */

        while( p->in.op == CM ){
                genargtemps( p->in.left, ptemp );
                p = p->in.right;
        }


        /* ordinary case */

        /* register, literal, and single valued memory operands
           are given temporary locations on the stack.  The temps
           are passed as arguments so that original values remain
           uncorrupted by subroutine actions and so that all
           parameters are read/write. */

        switch(p->in.op){

                default:
                        break;
```

```
                /* create temp for arguments, change type so that
                   procedure call will generate temp location as
                   parameter.  If argument is array or pointer, no
                   temp needed.  */

                /* take care of automatic arrays */

                case PLUS:
                        if ( ISPTR(p->in.type) ) {
                                ql = p->in.left;
                                qr = p->in.right;
                                if ( ql->in.op == REG && qr->in.op == IC
                                        p->in.op = OREG;
                                        p->tn.rval = LSP;
                                        p->tn.lval = qr->tn.lval;
                                        p->in.name = "";
                                        ql->in.op = qr->in.op = FREE;
                                        break;
                                }
                        }
                        printf("check PLUS in genargtemps0);

                case REG:
                        if ( ISPTR(p->in.type) ){
                                p->in.op = OREG;
                        }
                        break;

                case NAME:
                case OREG:
                        expand( p, FORARG, "    push    AR0);
                        p->in.op = OREG;
                        p->tn.rval = SP;
                        p->tn.lval = --atmps * 4;
                        p->in.name = "";
                        argtemps += 4;
                        break;

                case ICON:
                        if ( ISPTR(p->in.type) ) {
                                break;
                                }
                        expand( p, FORARG, "    push    AR0);
                        p->in.op = OREG;
                        p->tn.rval = SP;
                        p->tn.lval = --atmps * 4;
                        p->in.name = "";
                        argtemps += 4;
                        break;
        }

}
#endif
```

Figure 4.4

number of stack locations to free after a subroutine return. The input to both functions is an expression tree generated by the parse of a function call. Genargtemps distinguishes among literals, registers, and memory operands. In the first two cases a temporary is always created. If a memory location represents the base address of an array, then no temporary is created since a reference parameter is appropriate in this case. The actual code is emitted by the function expand which is given a code string containing internally defined macros.

4.1.4. External Variables

Figures 4.5 illustrate the compilation of external variables and somewhat more complicated arithmetic expressions. The VAX and Nebula code for this example begins to illustrate some of the severity of the mismatch between C's value parameters and Nebula's reference parameters. The VAX version is 45 bytes long. The Nebula version is 57 bytes long, nearly a 25% increase. Ten of the extra bytes are needed to create the stack temporaries and release them upon return.

```
int boy;
main()
{
        int c;
        int d;
        c = 7;
        boy = c;
        d = sumup(7,boy,c);
}

sumup(a,b,c)
{
        return(a+b+c);
}
```

Figure 4.5.c

```
; PCC generated nebula code--Sun Sep  7 21:03:13 1986


        .sect   bss,data
_c_boy:         .blkw   1
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=0,reg=15

        br      L15
L16:
        mov     #7,0(lsp)^W
        mov     0(lsp)^W,_c_boy
        push    #7
        push    _c_boy
        push    0(lsp)^W
        call    _c_sumup,%3,8(sp)^W,4(sp)^W,0(sp)^W
        add     #12,sp
        mov     %3,4(lsp)^W
        ret
L15:
        sub     #8,sp
        mov     sp,lsp
        br      L16
        .sect   bss
        .sect   prog

        .global _c_sumup

_c_sumup:               .entry  np=4,reg=15

        br      L20
L21:
        add     ?3,?2,%3
        add     ?4,%3
        mov     %3,?1
        ret
        mov     %3,?1
        ret
L20:
        br      L21
        .sect   bss
```

Figure 4.5.neb

```
        .data
        .comm   _boy,4
        .text
        .align  1
        .globl  _main
_main:
        .word   L13
        jbr     L15
L16:
        movl    $7,-4(fp)
        movl    -4(fp),_boy
        pushl   -4(fp)
        pushl   _boy
        pushl   $7
        calls   $3,_sumup
        movl    r0,-8(fp)
        ret
        .set    L13,0x0
L15:
        subl2   $8,sp
        jbr     L16
        .data
        .text
        .align  1
        .globl  _sumup
_sumup:
        .word   L18
        jbr     L20
L21:
        addl3   8(ap),4(ap),r0
        addl2   12(ap),r0
        ret
        ret
        .set    L18,0x0
L20:
        jbr     L21
        .data
```

Figure 4.5.vax

4.1.5. Array Assignments

Figures 4.6 show the compiled code for an assignment to an array. No additional modifications were needed for this case.

```
main()
{
        int a[10][20];
        a[5][0] = 7;
}
```

Figure 4.6.c

```
; PCC generated nebula code--Sun Sep  7 21:03:22 1986

        .sect   bss,data
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=0,reg=15

        br      L14
L15:
        mov     #7,400(lsp)^W
        ret
L14:
        sub     #800,sp
        mov     sp,lsp
        br      L15
        .sect   bss
```

Figure 4.6.neb

```
        .data
        .text
        .align  1
        .globl  _main
_main:
        .word   L12
        jbr     L14
L15:
        movl    $7,-400(fp)
        ret
        .set    L12,0x0
L14:
        movab   -800(sp),sp
        jbr     L15
        .data
```

Figure 4.6.vax

4.1.6. Loops

Figures 4.7 illustrate code with loops. The only additional modifications required to compile this code segment were changes to the table of instruction mnemonics, ccbranches, in the file local2.c

```
main()
{
        int a,b,i;
        for ( i = 0; i < 10; i++ ) ;
        while ( i > 0 )
                i--;
}
```

Figure 4.7.c

```
; PCC generated nebula code--Sun Sep  7 21:03:26 1986


        .sect   bss,data
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=0,reg=15

        br      L14
L15:
        clr     8(lsp)^W
L18:
        cmp     8(lsp)^W,#10
        bgeq    L17
L16:
        inc     8(lsp)^W
        br      L18
L17:
L19:
        test    8(lsp)^W
        bleq    L20
        dec     8(lsp)^W
        br      L19
L20:
        ret
L14:
        sub     #12,sp
        mov     sp,lsp
        br      L15
        .sect   bss
```

Figure 4.7.neb

```
        .data
        .text
        .align  1
        .globl  _main
_main:
        .word   L12
        jbr     L14
L15:
        clrl    -12(fp)
L18:
        cmpl    -12(fp),$10
        jgeq    L17
L16:
        incl    -12(fp)
        jbr     L18
L17:
L19:
        tstl    -12(fp)
        jleq    L20
        decl    -12(fp)
        jbr     L19
L20:
        ret
        .set    L12,0x0
L14:
        subl2   $12,sp
        jbr     L15
        .data
```

Figure 4.7.vax

4.1.7. Array Element as a Parameter

Figures 4.8 are examples of passing an array element as a parameter. The modifications to compile this example were minimal. An array element is still considered a value parameter in C so the creation of a stack temporary was still required.

```
main()
{
        int a[10];
        dummy(a[1]);
}

dummy(t)
int t;
{
        t = 7;
        return;
}
```

Figure 4.8.c

```
; PCC generated nebula code--Sun Sep  7 21:03:30 1986

        .sect   bss,data
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=0,reg=15

        br      L14
L15:
        push    4(lsp)^W
        call    _c_dummy,%3,0(sp)^W
        add     #4,sp
        ret
L14:
        sub     #40,sp
        mov     sp,lsp
        br      L15
        .sect   bss
        .sect   prog

        .global _c_dummy

_c_dummy:               .entry  np=2,reg=15

        br      L19
L20:
        mov     #7,?2
        mov     %3,?1
        ret
        mov     %3,?1
        ret
L19:
        br      L20
        .sect   bss
```

Figure 4.8.neb

```
        .data
        .text
        .align  1
        .globl  _main
_main:
        .word   L12
        jbr     L14
L15:
        pushl   -36(fp)
        calls   $1,_dummy
        ret
        .set    L12,0x0
L14:
        subl2   $40,sp
        jbr     L15
        .data
        .text
        .align  1
        .globl  _dummy
_dummy:
        .word   L17
        jbr     L19
L20:
        movl    $7,4(ap)
        ret
        ret
        .set    L17,0x0
L19:
        jbr     L20
        .data
```

Figure 4.8.vax

4.1.8. Array as a Parameter

Figures 4.9 are the final examples of compiled code. Since the array itself is the parameter, no stack temporary was created. Inspection of the code for both the Nebula and VAX versions provides a good example of an instance where the the Nebula code is much clearer than the code generated by VAX. This is due to the greater flexibility of the

```
int b[10];
main()
{
        dummy(z);
}
dummy(t)
int t[10];
{
        t[5] = 6;
}
```

Figure 4.9.c

```
; PCC generated nebula code--Sun Sep  7 21:02:41 1986


        .sect   bss,data
_c_b:           .blkw   10
        .sect   prog,code
sp=%1
lsp=%2

        .global _c_main

_c_main:                .entry  np=0,reg=15

        br      L15
L16:
        call    _c_dummy,%3,_c_b
        ret
L15:
        br      L16
        .sect   bss
        .sect   prog

        .global _c_dummy

_c_dummy:               .entry  np=2,reg=15

        br      L20
L21:
        mov     #6,?2[#5]^W
        mov     %3,?1
        ret
L20:
        br      L21
        .sect   bss
```

Figure 4.9.neb

```
        .data
        .comm   _b,40
        .text
        .align  1
        .globl  _main
_main:
        .word   L13
        jbr     L15
L16:
        pushl   $_b
        calls   $1,_dummy
        ret
        .set    L13,0x0
L15:
        jbr     L16
        .data
        .text
        .align  1
        .globl  _dummy
_dummy:
        .word   L18
        jbr     L20
L21:
        movl    4(ap),r0
        movl    $6,20(r0)
        ret
        .set    L18,0x0
L20:
        jbr     L21
        .data
```

Figure 4.9.vax

indexed addressing modes available to Nebula. Nebula can use the parameter mode to specify the index. VAX can only index on a register. This requires an extra data move instruction.

Passing arrays as parameters began to strain the capabilities of the PCC. Up to this point all of the modifications preserved the spirit of the PCC in that they were

changes made to general cases so that each change tended to implement an entire class of cases. In this instance the modifications were very case specific, i.e. the code was modified to handle an assignment to an element of an array when the array was passed as a parameter and the array element was identified by a literal value. Since the concept of a parameter as a directly addressable object is not built into the structure of the PCC, most of the obvious solutions simply add code to treat special cases. Not only is this approach error prone, but it obviates the reasons behind the development of a portable compiler in the first place.

Some time was spent trying to introduce the syntactic category of PARAM into the structure of the compiler. Although the results were encouraging, a complete job would require going back to the actions of the parser and would have required more time than was available for the project.

4.2. Code Templates

In addition to the modifications mentioned above, the table of code templates in the file table.c was modified. The operation of the code generator is described by Johnson[John??]. Briefly, when the code generator is passed an expression tree, it first passes the tree to the routine, store, which returns a tree, or sub-tree, which can be computed without temporary stores because store itself will generate code and simplify the expression tree. The exact

mechanism by which this is accomplished is complex and beyond the scope of this project to explain in detail. A key element is an estimation of the register resources required. The concluding chapter will point out the implications this might have for Nebula.

The structure returned from store is passed to the routine, order. Order searches the code templates until it is able to match the subtree for which the code is to be generated. The subtree must match with respect to goals, e.g. compute for effect or compute for condition codes only, shape of the right and left children, resources required, and reclamation rules. Clearly there cannot be a template for all possible subtrees. The algorithm for finding a match is recursive and has provisions for rewriting the input in numerous heuristic ways in order to find a match. Only when a match cannot be found after all the attempts at rewriting does the compiler fail. If this happens no code is emitted and a compiler error is generated.[4]

Because the compiler was so efficient at self-diagnosis specific templates needed to generate code for the segments of C source were identified either by inspection of the templates for VAX or running the VAX version and developing the

[4] Out of curiosity I put a counter into the search code. It was not unusual for the template table to be entered hundreds of times in order to find a match. Attempting to trace the operation by hand was futile.

modified templates as needed. The table for the PCC implemented for this study consisted of 14 templates compared with 120 for VAX. Many of the VAX templates were related to size conversion code and so would not be needed in a complete implementation for Nebula.

This completes the discussion of the PCC modifications. The next chapter will present conclusions and an assessment of Nebula.

5. Conclusions

The original goal of the study was to determine if Nebula is a suitable target for C. In many ways that question could have been answered before doing the study. Nebula, at base, is a traditional architecture utilizing features which have been well understood for years. C is a conventional language. There was no reason to expect that Nebula would not be a suitable target for C.

But there is also a practical side to the question which does make studies such as this valuable. In practical terms if a new architecture is to succeed, it must provide a comparative advantage over what is currently available. Whether the Nebula design accomplishes this is not so clear.

The Nebula design team was heavily influenced by the language description of Ada and by studies which had addressed the issues of architectural efficiency [Diet79] [Kogg81]. The Ada features of multi tasking and exception handling appear to be well supported by the architecture.

Given the criteria of suitability listed in Chapter 2 and addressing the question of whether Nebula has a comparative advantage over VAX, as a representative modern architecture, the answer would have to be a qualified "no". Each of the criteria will be considered in turn.

5.1. Data Object Representation

Nebula represents C's data objects very well, though no better than does VAX. C's data objects are standard types: integers, reals, logicals, and reference types. These types are implemented in a variety of sizes.

5.2. Data Object Conversions

Nebula excels in this category. One of the most convenient features of an HOL for a programmer is freedom from the concern of the proper sizing of operands. In this respect Nebula is very much like an HOL. This has an impact not only on the size of the code generated, but also on programmer productivity given the pernicious nature of some size conversion bugs. In some cases VAX allows the mix of different sized operands; however, no automatic sign extension is performed. For the most part size conversions and sign extensions are transparent to the Nebula programmer.

From the point of view of the compiler, the complicated and time consuming process of checking sizes and providing for the different combinations can be avoided.

5.3. Operations on Data Types

As in the case of the data objects themselves, C's operators are well supported in Nebula though no better than in VAX. This result is not surprising given the standard nature of the operations on data types.

5.4. Implementation of Data Structures

Nebula's addressing modes support C's data structures very well. In fact its scaled and unscaled addressing modes are more flexible than VAX's since the base and index values can be specified by any of the non-compound addressing modes. In many cases this flexibility will save both a data move instruction and a register which would be required by VAX.

This issue of saving a register resource is non-trivial with respect to efficient code generation. The mechanism for generating code from the expression tree described earlier depends heavily upon the estimation of the register requirements in order to determine which sub-trees can be computed and which must be stored. One of the obstacles to accurate estimation is the treatment given to the register needs due to indirection. An architecture with simpler register usage should allow a more accurate prediction of that usage, and hence allow the possibility of more efficient code generation. This conjecture is an open empirical/analytical question suitable for further research.

It is not immediately clear how a compiler might take advantage of some of the addressing modes on Nebula, such as the general parameter mode in which a parameter is specified indirectly by obtaining a value using one of the non-

compound modes.[1]

5.5. Implementation of Control Statements

Nebula is similar to VAX in its sequence control statements. In addition to the full complement of test and branch instructions, it has instructions for the direct implementation of counter increment loops.

5.6. Procedure Interface

The basic incompatibility between Nebula's passing parameters by reference and C's passing them by value is the most severe argument against Nebula as a target for C. The compiler writer is faced with two choices. Either give up the use of Nebula's hardware parameters, and their associated addressing modes, or have the caller create temporaries for all value parameters and pass the temporaries. Neither solution is entirely satisfactory.

If the objects themselves are passed, the called function will need to make local copies in order to avoid the potential for corrupting the caller's variable space. Also, the called function will lose the use of the parameter addressing modes; in many cases it will have to replace a

[1] An issue in the choice of an architecture for general use is assessing the trade-offs between its suitability as an HOL compiler target and its suitability as an architecture for direct coding of assembly language. The difficulties it presents to the compiler writer might be offset by the benefits it provides to the assembly language programmer.

one byte operand specifier with a multi-byte operand specifier.

If the caller makes the copies, each caller of a common function will need to generate the same code. In addition to the code to generate the temporaries, there is the code to release them. The exact cost of this approach will be situation dependent, but there are clearly many cases in which the size of the code will be increased by a significant amount.[2]

An additional difficulty is causes by Nebula's register independence among procedures. One of C's most heavily used features is its use of the value returned by all functions. Nebula must implement this in software rather than use the simple expedient of designating a common register to hold the return value. Though not nearly so severe as the parameter passing incompatibility, it is still annoying.

5.7. Concluding Remarks

With respect to the limitations of this study, the superiority of Nebula over other modern architectures has

[2] A possible strategy is to give up the hardware parameters entirely and pass parameters in the traditional manner of pushing them onto the stack. The solution offered in the implementation does this in any case. If we are willing to give up the parameter addressing modes we could save the bytes required in the call statement for the operand specifiers. This might offset the loss of the one byte parameter addressing operand which would need to be replaced by the usual register indexed operand.

not been convincingly demonstrated. There are, in fact, some serious implementation difficulties which require software work arounds which might prove too costly in terms of space efficiency. Parameter passing, in particular, was cumbersome. Nebula's reference mechanism is more suitable for a language like Ada which has different parameter modes, i.e. in, in-out, out. In that case it is illegal for a procedure to make an assignment to an in parameter, and such an assignment would be caught by the type checking at compile time. This means that the code generator need not concern itself with the potential corruption of the caller's variable space. If an assignment to a formal parameter is made, it is guaranteed that the parameter is an in-out, or an out parameter so the reference mechanism is appropriate. This design seems to depend to much on Ada. Reference parameters are simply not well suited to C's language definition.

It is appropriate to finish this study with some remarks about the personal benefits gained by doing it.

Before the actual analysis of Nebula, I spent some weeks going through the C source code for the VAX implementation of the compiler. Although the direct application of the benefits of this exercise are not readily apparent, it provided valuable training in developing a facility to mask out irrelevant details when trying to reconstruct the logic of an existing program.

In addition I was able to add several techniques to my technical skills. The lexical analyzer, for example, uses a "dope" vector in order to determine class membership of input characters. I have made use of this feature several times since discovering it in the code. There are utility functions to traverse a tree in post-order, pre-order, and in-order. One of the arguments to the function is itself a function to accomplish the appropriate visit to a node. This was the first time I saw good examples of passing functions as parameters.

This was the largest program I have had to deal with and effective data management, modular design, and selective use of global structures were critical to its robustness. The necessity of techniques to keep code well-structured is more apparent in large projects.

As a final note I have been convinced of the importance of good documentation. The PCC could be made a lot more portable with a little more commenting of some particularly obscure parts of the code.

BIBLIOGRAPHY

Aho, Alfred V. & Ullman, Jeffrey D., The Theory of Parsing, Translation, and Compiling, Prentice-Hall; New Jersey, 1972

Anderson, Peter G., "A Comparison of Nebula and Alternative Computer Architectures via Selected Ada Programs", Unpublished Manuscript RIT School of Computer Science & Technology, October 1985

Barron, D. W., Assemblers and Loaders, MacDonald/Elsevier; London, 1969

Calingaert, Peter, Assemblers, Compilers, and Program Translation, Computer Science Press Maryland, 1979

DoD Military Standard, MIL-STD-1862B, Nebula Instruction Set Architecture, 1983

Johnson, Stephen, C., "A Portable Compiler: Theory and Practice" in Fifth Annual ACM Symposium on Principles of Programming Languages, January 1978. Pages 97 - 104

Johnson, Stephen, C., "A Tour Through the Portable C Compiler"

Johnson, Stephen, C., "YACC: Yet Another Compiler-Compiler," Bell Laboratories; Murray Hill, New Jersey, 1978

Kogge, Peter & Olsen, Philip F., "The Army's MIL-STD-1862 and the Military Computer Family", IBM Federal Systems Division Technical Directives; Vol. 7 No. 2, Summer 1981

McNaughton, Robert, Elementary Computability, Formal Languages, and Automata, Prentice-Hall; New Jersey, 1982

Myers, Glenford, J., Advances in Computer Architecture. John Wiley & Sons, 1978.

Pagan, Frank G., Formal Specification of Programming Languages, Prentice-Hall; New Jersey, 1981

Poole, Peter C., "Portable and Adaptable Compilers" in Lecture Notes in Computer Science, Vol. 21 Compiler Construction an Advanced Course, Goos, G. & Hartmanis, J. eds.,

Sebesta, Robert W., VAX 11: Structured Assembly Language Programming, Benjamin Cummungs; Menlo Park, 1984

Waite, William M., "Relationship of Languages to Machines" in Lecture Notes in Computer Science, Vol. 21 Compiler Construction an Advanced Course, Goos, G. & Hartmanis, J. eds.,

Waite, William M. & Goos, Gerhard, Compiler Construction, Springer-Verlag; New York, 1984