# AN IMPLEMENTATION OF THE C PROGRAMMING LANGUAGE FOR THE HARRIS /6 MINICOMPUTER

by

SAMUEL JACOB LEFFLER

Submitted in partial fulfillment of the requirements
for the Degree of Master of Science

Thesis Advisor: Charles W. Rose

Department of Computer Engineering and Science
CASE WESTERN RESERVE UNIVERSITY

January 7, 1981

CASE WESTERN RESERVE UNIVERSITY

GRADUATE STUDIES

We hereby approve the thesis of

Samuel Leffler

candidate for the Master of Science

degree.

Signed: [illegible]
(Chairman)

[illegible]

[illegible]

Date 10 September 1980

# AN IMPLEMENTATION OF THE C PROGRAMMING LANGUAGE FOR THE HARRIS /6 MINICOMPUTER

Abstract

by

SAMUEL JACOB LEFFLER

As part of a project to port the UNIX operating system to a Harris /6 minicomputer, a programming environment for the C programming language has been developed. A C compiler based on the portable C compiler has been constructed, along with the necessary support utilities -- assembler, link-editor, etc. The architecture of the Harris /6 posed numerous problems to the porting effort, necessitating modifications to the machine independent portions of the portable compiler. This document describes the porting effort and modifications to the compiler. An evaluation of the code quality produced and the efficiency of the compiler are included. Finally, experiences gained from the porting project are employed in lending observations about the generality of the portable compiler, and the portability of the C language and programs written in the C language.

To my parents, Amos and Florence.

## ACKNOWLEDGEMENTS

I would like to thank William Shannon who was my partner on the UNIX/24V project. With his porting of the UNIX operating system this project became a significant contribution to the UNIX community. I must also acknowledge his efforts in helping test the compiler. Our endless discussions were important to the success of the project.

The efforts of Robert Gingell must be acknowledged. He was instrumental in initiating the UNIX/24V project and played an important role in it by moving the Release 7 shell.

I would like to thank my advisor, Professor Charles Rose, for his guidance in preparing this document, and Professors George Ernst and Raymond Hookway for serving on my committee.

C and UNIX exist because of the efforts of Dennis Ritchie and Ken Thompson. Also, Steve Johnson and Allan Synder were the originators of the portable C compiler. This project was possible because of the previous work of these people.

Finally, the A. R. Jennings Computing Center must be acknowledged for their support in this venture.

TABLE OF CONTENTS

## LIST OF TABLES AND FIGURES

# CHAPTER I

## INTRODUCTION

In 1976 Case Western Reserve University entered into a project to construct a campus-wide resource sharing network which would tie together existing minicomputers and allow for additional hardware to be supported at minimal connection cost. The machines chosen for this venture were Harris /6 minicomputers. By the summer of 1979 the proposed network plans had been reevaluated due to unforseen circumstances involving the development of networking software by the Harris corporation.

During the initial phase of the network development, the native /6 operating system, VULCAN, was found to have a number of weaknesses that made software development difficult. Consequently, one part of the reevaluation process involved the selection of an alternate operating system on which to base future work. The operating system chosen to replace VULCAN was the UNIX operating system. The choice of UNIX was due mainly to its proven ability as a base for software development and text processing, and for its known portability. The latter point was a major factor in establishing credibility of the selection, for at that time UNIX was not available for a Harris /6.

The UNIX system and its central software were originally written in assembly language, before the C language was invented, and have since been rewritten in C. Previous portability projects have involved moving the UNIX kernel to the Interdata 7/32 [19], the Interdata 8/32 [12], and the VAX-11/780 [18]. These efforts have had a major impact on the UNIX kernel, both in pointing out weaknesses in the C language, and in isolating nonportable sections of the kernel. The PDP-11 version of the kernel, on which the porting effort was based, consisted of approximately 10,000 lines of C code and about 1,000 lines of assembly code. Thus, a prerequisite to the porting of the UNIX kernel was a C compiler for the /6.

While it was clear that porting the UNIX operating system required a C compiler for the target machine, it was not clear exactly what route to take to create this compiler. At the initiation of the project there were three choices for creating a C compiler: write one from scratch, port the Ritchie C compiler [24] which runs on all PDP-11's, or use the portable C compiler written by S. C. Johnson [11]. Recent C compiler efforts had followed all of these routes: the

Interdata 7/32 C compiler was based on the Ritchie compiler, the VAX-11/780 compiler used Johnson's portable C compiler, and the University of Wisconsin's Harris /6 C compiler was written from scratch (in assembly language).

For the purposes of this project, the Wisconsin C compiler was unsuitable. The compiler supported an early version of C (version 6) and was known to have a significant number of bugs. The Ritchie compiler, a production compiler tuned to the PDP-11, was too firmly entrenched in the PDP-11 architecture to hope for reliable conversion to such a radically different architecture. Since the /6 C compiler would be needed before UNIX could be ported, and writing a compiler from scratch would take considerable time, this left only the portable compiler to consider. Previous efforts involving the compiler had shown excellent results, both in ease of movement and reliability of operation. The VAX-11/780 compiler, based on the portable compiler, required about one month of effort to create a first cut compiler, and more than two years after it was used to bootstrap a UNIX system onto the machine, it remains as the production compiler. Thus, it was decided that the C compiler for the /6 would be based on the portable compiler.

This research involved the movement of the portable C compiler to the /6 and creation/movement of related software necessary to support a programming environment compatible with that found on other machines supporting UNIX. The /6 architecture contains many features that made the porting effort very difficult and impacted the C language environment in a negative way. The solution of these problems will be discussed in somewhat general terms, in the hopes that they might be applied to machines with similar architectures. Necessary background for this document includes some familiarity with the C programming language.

The organization of the remainder of this document follows the development of the compiler, then steps back to consider the resultant programming environment created. Chapters II and III provide background material on the portable C compiler and the Harris /6 minicomputer, respectively. Later chapters presume a knowledge of these two topics commensurate with that presented in chapters II and III. Chapter IV gives an overview of the porting process, while chapter V covers the compiler modifications that were necessitated by the /6 architecture. Chapter VI discusses the quality of code presently generated by the compiler, and offers observations about possible future optimizations. Chapter VII considers the interaction between the compiler and its support tools, the assembler and link-editor. Chapters VIII evaluates some of the compiler´s good and bad points, while chapter IX looks at the impact of the /6´s word addressable architecture on the C programming environment. Finally, chapter X summarizes the results of this research.

# CHAPTER II

# A DESCRIPTION OF THE PORTABLE C COMPILER

## 1. Overview

This chapter discusses the structure and organization of the portable compiler. Rather than reiterate all that has been presented in previous descriptions of the portable compiler, [11], this chapter will introduce only those notions necessary for an understanding of the design issues involving the /6 version of the compiler. Some of the theoretical work on which the compiler is based, and its application to the compiler, is discussed elsewhere, [9], while a detailed analysis of the /6 adaptation of the portable compiler may be found in [15].

The compiler consists of two passes that together turn C source code into assembler code for the target machine. The two passes are preceded by a preprocessor which is highly portable in its own right.

Although the compiler is divided into two passes, this represents historical accident more than deep necessity. In fact, the compiler can optionally be loaded so that both passes operate in the same program.

This "one pass" operation eliminates the overhead of reading and writing an intermediate file, so the compiler operates about 30% faster in this mode. In this form the compiler also occupies about 30% more space than the larger of the two component passes. Because the compiler is fundamentally structured as two passes, even when loaded as one, this chapter primarily describes the two pass version.

## 2. Expression Trees

While there are a large number of important data structures involved in the operation of the compiler, the focus of interest in this document will be on the parse trees formed in the first pass, and used in the second pass by the code generation scheme. These parse trees are used to represent C expressions; almost all flow control constructs have code generated for them immediately in the first pass. The use of trees for an intermediate representation has simplified many of the complicated operations performed during code generation.

The definition of the nodes that comprise an expression tree differs from the first pass of the compiler to the second. The first pass must be concerned with a variety of symbol table related issues, while the second pass doesn´t use the symbol table, but must maintain information used in the register allocation and expression compilation schemes. The expression trees, along with other information, are communicated between the two passes by an ascii intermediate file. When the two passes are merged to form a single pass compiler this file is eliminated, and the trees are simply "handed off" to the appropriate second pass routine. Since the node definitions differ in each pass, the combination of passes requires node definitions to be made large enough to hold the union of the information needed in each pass.

Each node contains two members used in both passes: _op_, used to specify the "node number", and _type_. A node number identifies either a C language operator, or an operator internal to the compiler. The collection of node numbers defines the intermediate language used to communicate with the code generator. The internal operators defined are primarily used to process declarations in the first pass. The bottom-up construction of the parser and top down nature of C declarations requires declarations be used to build a "declaration tree", which is then processed in a top down fashion. Most C operators have a corresponding node number. For example, + is represented by PLUS, % by MOD, etc. A token such as MINUS may be seen in the lexical analyzer before it is known whether it is a unary or binary operator; clearly, it is necessary to know this by the time the parse tree is constructed. Thus, an operator (really a macro) called UNARY is provided, so that MINUS and UNARY MINUS are both distinct node numbers. Similarly, many binary operators exist in an assignment form -- for example -= -- and the operator ASG may be applied to such node names to generate new ones, e.g. ASG MINUS. Table II-1 shows some of the most common _op_ values which will be used in further discussions.

C has a rich typing structure, with a potentially infinite number of types. To begin with there are the basic types: CHAR, SHORT, INT, LONG, the unsigned versions known as UCHAR, USHORT, UNSIGNED, ULONG, and FLOAT, DOUBLE, and finally STRTY (a structure), UNIONTY, and ENUMTY. Three operators may be applied to types to construct others: if _t_ is a type, one may potentially have types _pointer to t_, _function returning t_, and _array of t's_ generated from _t_. Thus, an arbitrary type in C consists of a basic type, and zero or more of these operators. The _type_ member of each node contains a C type, as outlined

| Name | Description |
|---|---|
| NAME | named memory location |
| ICON | integer constant, possibly a symbolic address |
| PLUS | + operator |
| MINUS | -, U- has UNARY prefix |
| MUL | *, U* <=> UNARY MUL |
| AND | &, U& <=> UNARY AND |
| OR | \|, inclusive-or |
| ER | , exclusive-or |
| ANDAND | &&, logical connective |
| OROR | \|\|, logical connective |
| COMOP | `,´ operator |
| DIV | / |
| MOD | %, remainder |
| LS | <<, left shift |
| RS | >>, right shift |
| CALL | function call, UNARY CALL has no param´s |
| COMPL | ~, one´s complementation |
| INCR | ++, postfix and prefix |
| DECR | -- |
| EQ | ==, logical connective |
| NE, | != |
| LE | <=, ULE is unsigned version |
| LT | <, ULT is unsigned version |
| GE | >=, UGE is unsigned version |
| GT | >, UGT is unsigned version |
| REG | register |
| OREG | offset from register (U* REG + ICON) |
| STASG | structure assignment |
| STARG | structure argument to a CALL |
| STCALL | CALL returning a structure, UNARY STCALL |
| FLD | bit field |
| SCONV | storage conversion (e.g. pointer => int) |
| PCONV | pointer conversion (e.g. int => pointer) |
| PMCONV | pointer multiplication conversion, e.g. pointer + offset => pointer + offset*width |
| PVCONV | pointer division conversion, e.g. pointer - pointer => (pointer-pointer)/width |
| PACONV | pointer addition conversion, see Figure VI-1 |
| PSCONV | pointer subtraction conversion, see Figure VI-1 |
| FORCE | must have left tree in specific register |
| CBRANCH | conditional branch to label on right, according to comparison on left |
| INIT | initialize memory location with value on left |

Figure II-1. Portable Compiler Node Types

here, describing the type of the expression rooted at that node.

Remaining members of the node structure contain dimension table pointers, size table pointers, constant values, symbol table pointers, label numbers, register allocation information, Sethi-Ullman numbers (see section 4 of this chapter), etc. The exact application of each element of a node will be described where needed. For a complete description of all the data structures used in the portable compiler consult [15].

The formalism chosen in this document to represent the expression trees has been tailored to simplify their inclusion in the text, so it will be introduced here. Expression trees will be displayed "on their side", with each level of the tree marked by an extra level of indentation. To avoid ambiguity, the left son at each level will be displayed first. All unary operators therefore will have their single descendant on the left. Consider the expression *a* += *b* - *c*, the corresponding tree would be:

```
+=, <type>, ...
    NAME, _a, <type>, ...
    -, <type>, ...
        NAME, _b, <type>, ...
        NAME, _c, <type>, ...
```

Each node in the tree contains a leading node number (displayed symbolically), the C type of the node, and any further information that might be appropriate to the example. For the purposes of discussion, many of the details which would be present in the actual node representation (e.g. dimension table indices, sizes, etc.) may be omitted. Note that the NAME nodes, used to indicate that the variables, a, b, and c are statically allocated, have the symbol's name present, prepended with an underscore ("_"). The convention chosen for the /6 is that all C symbols will be constructed in this fashion to avoid name conflicts with assembly language defined symbols.

## 3. The First Pass

The first pass performs lexical analysis, parsing, and symbol table maintenance. It also constructs parse trees for expressions and keeps track of the types of the nodes in these trees. Additional code is devoted to initialization of static data structures. Machine dependent portions of the first pass serve to generate subroutine prologs and epilogs, code for switch statements, code for branches, label definitions, alignment operations, changes of location counter, etc.

The porter of the portable compiler has relatively few jobs to perform in the first pass. The work of lexical analysis, parsing, symbol table maintenance, semantic checking, and initialization of static data structures are, for the most part, completely handled by the machine independent portions of the compiler; the porter is left with only minor tasks.

For each machine, the expression trees built by the first pass will need specific massaging. There are two major areas where this is important -- NAME nodes and conversion operations. In the case of NAME nodes, the machine dependent portion of the compiler must rewrite the node to reflect the physical location of the name in the machine. In effect, the NAME node must be examined, the symbol table entry found (through a field in the node), and based on the storage class of the node, the tree must be transformed. Automatic variables and parameters are usually rewritten by treating the reference to the variable as a structure reference off the register which holds the stack or argument pointer. In the case of LABEL and internal static nodes, the node will be transformed to place the negative of the internal label number in the node. Finally, a name of class REGISTER must be converted into a REG node, and the encoded register number must be placed in the appropriate field of the node for use by the second pass. For machines with addressability problems (for instance the IBM 370) the work here may become fairly involved.

The conversion operator treatment is rather tricky. It is necessary to handle application of conversion operators to constants in machine dependent routines in order that all constant expressions can have their values known at compile time. In extreme cases, this may mean that some simulation of the arithmetic of the target machine might have to be done in a cross-compiler. In the most common case, conversions from pointer to pointer do nothing. For some machines, however, conversions from byte pointer to short or long pointer might require a shift or rotate operation which would have to be generated here.

The other machine specific issue involves the subroutine prolog and epilog generation. The hard part here is the design of the stack frame and calling sequence. While code for these jobs may be emitted in part in the first pass, the final stack size and the number of register variables is not known until the second pass, so these values must be referred to by assembler constants.

C has a finite, but fairly extensive, number of storage classes available. One of the compiler design decisions was to process the storage class information totally in the first pass -- the second pass has no access to the symbol table. This means that all of the storage allocation must take place in the first pass, so that references to automatics and paremeters can be turned into references to cells lying a certain number of bytes offset from certain machine registers. The first pass of the compiler deals with all address information internally in bits. It is the compiler writer's responsibility to convert these values to bytes or words, as appropriate.

## 4. The Second Pass

It is difficult to organize a code generator to be flexible enough to generate code for a large number of machines and still be efficient for any one of them. Flexibility is also important when it comes time to tune the code generator to improve the output code quality. On the other hand, too much flexibility can lead to semantically incorrect code, and potentially a combinatorial explosion in the number of cases to be considered in the compiler.

One goal of the code generator is to have a high degree of correctness. It is very desirable to have the compiler detect its own inability to generate correct code. This goal is achieved by having a simple model of the job to be done (e.g. an expression tree) and a simple model of the machine state (e.g. which registers are free). The act of generating an instruction performs a transformation on the tree and the machine state. If each of these instruction/transformation pairs is correct, and if the machine state model really represents the actual machine, and if the transformations reduce the input tree to the desired single node, then the output code will be correct.

For most real machines, there is no definitive theory of code generation that encompasses all of the C operators. Thus, the selection of which instruction/transformations to generate, and in what order, is necessarily heuristic in flavor. If, for some expression tree, no transformation applies, or more seriously, if the heuristics select a sequence of instruction/transformations that do not, in fact, reduce the tree, the compiler will report its inability to generate code and abort.

A major part of the code generator is concerned with the model and the transformations, most of which is machine dependent or depends on simple code tables. The flexibility comes from the heuristics that guide the transformations of the tree, the selection of subgoals and the ordering of the computation.

The remainder of this section involves a description of the scheme used by the code generator. It is based heavily on the machine model imposed by the portable compiler, so before the details are discussed, this model must be introduced.

The machine is assumed to have a number of registers, of at most two different types: A and B. Within each register class, there may be scratch (temporary) registers and dedicated registers (e.g. register variables, the stack pointer, etc.). Requests to allocate and free registers involve only the temporary registers.

Each of the registers in the machine is given a name and a number; the numbers are used as indices into various tables that describe the registers, so they should be kept small. One such table describes the status of each register. The status of each register is an expression formed from manifest constants describing the type of the register: SAREG for dedicated AREG's, SAREG|STAREG for scratch AREG's, and, similarly SBREG and SBREG|STBREG for BREG´s.

The actual code generation is done by a hierarchy of routines. Each tree to be processed is first scanned for any delayable operations, such as postfix ++ and -- operations. Also, an attempt is made to handle comma operators by computing the left side expression first, and then rewriting the tree to eliminate the operator. This is not always possible; for example, parameter lists involve the comma operator, but their evaluation order may not be altered. The code generation process takes as arguments a pointer to an expression tree, and a second argument that, for socio-historical reasons, is called a cookie. The cookie describes a set of goals that would be acceptable for the code generation. These are assigned to individual bits, so they may be logically or´ed together to form a number of possible goals. Among the possible goals are

| | |
|---|---|
| FOREFF | Compute for side effects only; don´t worry about the value. |
| INTEMP | Compute and store the value into a temporary location in memory. |
| INAREG (INBREG) | Compute the value into an A (B) register. |
| INTAREG (INTBREG) | Compute the value into a scratch A (B) register. |
| FORCC | Compute the expression for the condition |

codes it produces.

| | |
|---|---|
| FORARG | Compute the expression as a function argument; e.g. stack it if appropriate. |

The first step in the code generation process is a canonicalization of the expression tree. Canonicalization involves searching the tree for certain transformations that might be applicable. One, which is very common and very powerful, is to fold together an indirection operator (UNARY MUL) and a register (REG); in most machines, this combination is addressable directly, and so is similar to a NAME in its behavior. The UNARY MUL and REG are folded together to make another node type called OREG. In fact, in many machines, it is possible to directly address not just the cell pointed to by a register, but also cells differing by a constant offset from the cell pointed to by the register; such cases are also sought. Another transformation is to replace bit field operations by shifts and masks if the operation involves extracting the field. Finally, a machine dependent routine is called to calculate Sethi-Ullman numbers for the tree. A Sethi-Ullman number, [25], is an estimate of the number of registers required to evaluate an expression. These numbers are calculated in a bottom-up fashion. Each node in the tree has a number which is intended to reflect the number of registers required to evaluate the expression rooted at that node.

After the tree has been canonicalized, it is perused for subtrees that may be computed and (usually) stored before beginning the computation of the full tree. The selection of these subtrees is usually a result of the full tree requiring more registers (according to the Sethi-Ullman numbers calculated) than the machine has available. The trees handled in this manner must be computable without need for temporary storage locations. In effect, the only store operations generated while processing the subtree must be in response to explicit assignment operators in the tree. This division of the job marks one of the more significant, and successful, departures from most other compilers. It means that the code generator may operate under the assumption that there are enough registers to do its job, without worrying about temporary storage.

One consequence of this organization is that code is not generated by a treewalk. There are theoretical results that support this decision [1], [2], [25]. It may be desirable to compute several subtrees and store them before tackling the whole tree; if a subtree is to be stored, this is known before the code generation for the subtree is begun, and the subtree is computed when all scratch registers are available.

When a tree is ready to be evaluated (i.e. it has been stripped of all subtrees that need to be stored), it is passed to a routine which handles the evaluation of expression trees that do not require temporary locations. This routine may make recursive calls on itself, and, in some cases, on routines higher up in the hierarchy. For example, when processing the operators &&, ||, and comma, that have a left to right evaluation, it is incorrect to examine the right operand for subtrees to be stored. In these cases, a recursive call to a higher level routine must be made when it is permissible to work on the right operand. A similar situation arises with the ?: operator.

The evaluation of expression trees works by matching the current tree with a set of code templates. If a template is discoverd that will match the current tree and cookie, the associated assembly code is generated. The tree is then rewritten, as specified by the template, to represent the effect of the output instruction(s). If no template match is found, first an attempt is made to match with a different cookie; for example, to compute an expression with cookie INTEMP, it is usually necessary to compute the expression into a scratch register first. If all attempts to match the tree fail, the heuristic part of the algorithm becomes dominant. Control is typically given to one of a number of machine-dependent routines that may in turn recursively call on the evaluation process to achieve a subgoal of the computation (for example, one of the arguments may be computed into a temporary register). After this subgoal has been achieved, the process begins again with the modified tree. If the machine-dependent heuristics are unable to reduce the tree further, a number of default rewriting rules may be considered appropriate.

To close this introduction, we will consider the steps in compiling the code for the expression

```
a += b
```

where a and b are static variables.

The canonicalization and Sethi-Ullman number computation are machine dependent, so assume they do not alter the tree noticeably. Then, to begin with, the whole expression tree is examined with the cookie FOREFF, and no match is found. Search with other cookies is equally fruitless, so an attempt at rewriting is made. Suppose we are dealing with the

Interdata 8/32 for the moment (it bears some resemblance to the /6 in many of its rewriting rules). It is recognized that the left and right hand sides of the += operator are addressable, and in particular the left hand side has no side effects, so it is permissible to rewrite the tree as

```
a = a + b
```

and this is done. No match is found on this tree either, so a machine dependent rewrite is done; it recognizes tha the left hand side of the assignment is addressable, but the right hand side is not in a register, so a request is made to place the right hand side of the assignment operator into a register. This invocation of the code generation scheme searches the tree for a match, and fails. The machine dependent rule for + notices that the right hand operand is addressable; it decides to put the left operand into a scratch register. Another recursive call is made to the code generator, with the tree consisting solely of the leaf <u>a</u>, and the cookie asking that the value be placed into a scratch register. This now matches a template, and a load instruction is emitted, and this third call to the code generator returns. The second call now finds that it has the tree

```
        reg + b
```

to consider. Once again, there is no match, but the default rewriting rule rewrites the + as a += operator, since the left operand is a scratch register. When this is done, there is a match: in fact,

```
        reg += b
```

simply describes the effect of the add instruction on a typical machine. After the add is emitted, the tree is rewritten to consist merely of the register node, since the result of the add is now in the register. This agrees with the cookie passed to the second invocation of the code generator, so this invocation terminates, returning to the first level. The original tree has now become

```
        a = reg
```

which matches the template for the store instruction. The store is output, and the tree is rewritten to become a single register node. At this point, since the top level call to the code generator was interested only in side effects, the call returns, and code generation for the expression tree is completed; we have generated a load, add, and store as might have been expected.

The effect of machine architecture on this scheme is considerable. For example, on the Honeywell 6000, the machine dependent heuristics recognize that there is an "add to storage" instruction, so the strategy is quite different: b is loaded into a register, and then an add to storage instruction is generated to add this register to a. The transformations, involving as they do the semantics of C, are largely machine dependent. The decisions as to when to use them, however, are almost totally machine independent.

Chapter V will consider much of the code generation scheme for the Harris /6. The design of the machine dependent tree rewriting rules and Sethi-Ullman number computations is extremely difficult; an analysis of a specific example should prove useful to future porters of the portable C compiler.

# CHAPTER III

# THE HARRIS /6 MINICOMPUTER

Since many of the design issues involved in the writing of a compiler are driven by the target machine´s architecture, this chapter introduces the basics of the /6 machine, and describes their use within the implementation of the C language. The material in this chapter describing the architecture of the Harris /6 is mostly from the /6 reference manual, [6].

## 1. The Register Set

The Harris /6 has no general purpose registers in the sense of the PDP-11 or VAX-11 architectures. There are five 24-bit registers available to a user in the main cpu and additional registers may be added with extra features (such as the Scientific Arithmetic Unit - SAU, bit processor, external timer, etc.). The register set is composed of three index registers and two arithmetic accumulators, as illustrated in Figure III-1. The two arithmetic accumulators may also be referenced as a pair, the D register, for double precision integer arithmetic; a byte register, the B register, is the lower byte of the A register.

In the scheme chosen for C, certain registers are either dedicated, or heavily used for a specific purpose. The K register is used as the frame pointer in the stack management scheme (i.e. it is equivalent to R5 on the PDP-11, or FP on the VAX-11). The A register is used when items must be forced into a specific register, as in returning a value from a function call. Within the compiler's machine model, the B register is never explicitly referenced, rather a need for the B register is effected by a request for the A register.

Two add-on registers are used by the compiler, the X register from the SAU, and the V register from the bit processor. The X register is the only register in which floating point arithmetic may be performed, while the V register is used for a temporary storage place for the K register while performing structure assignments.

The /6 instruction set forces many restrictions on the use of registers. Virtually all complex arithmetic operations and all character manipulations must be performed in a fixed register. This problem has similarities to the even-odd register pairing

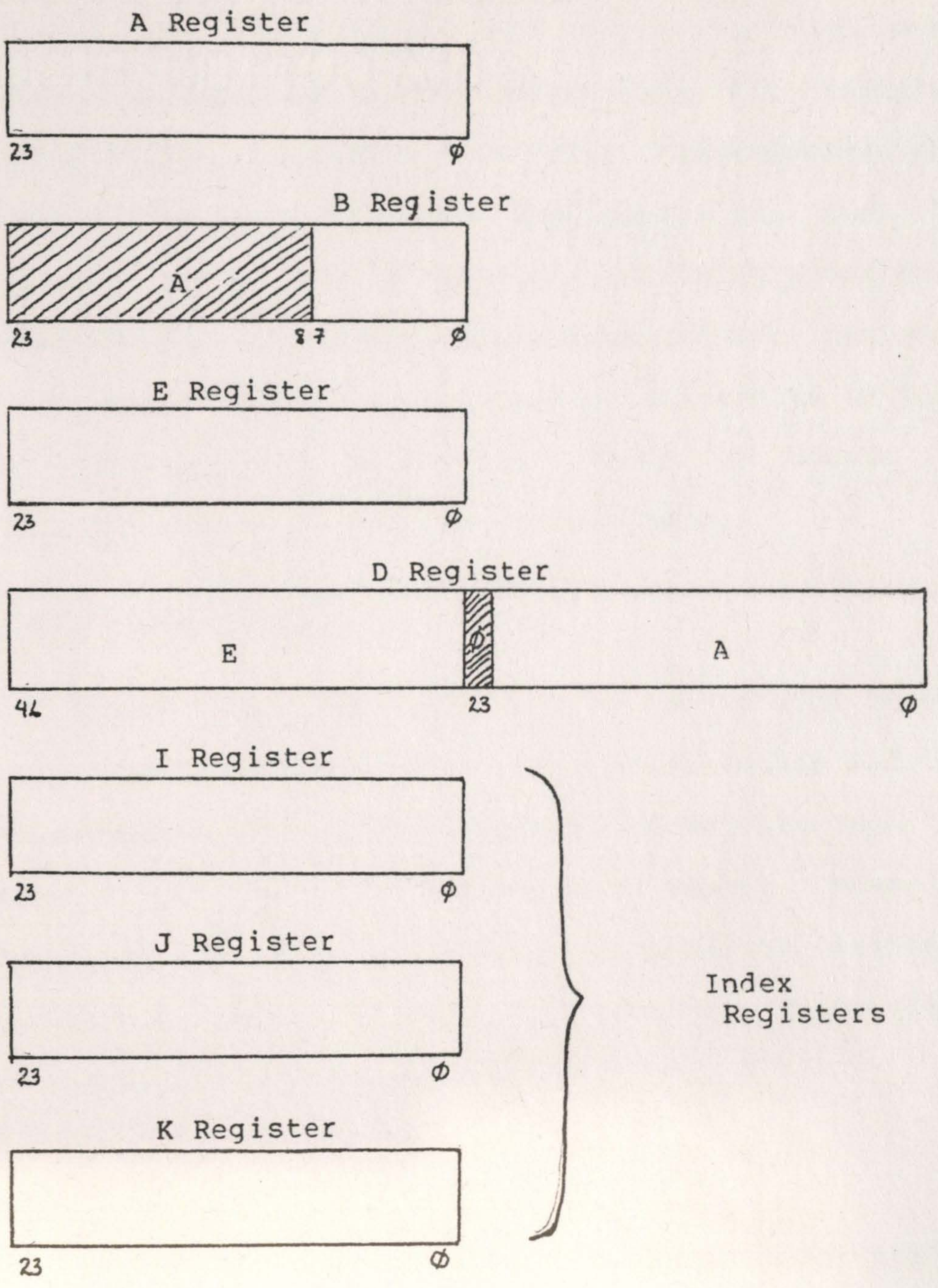


Figure III-1. The /6 Register Set
(used by the compiler)

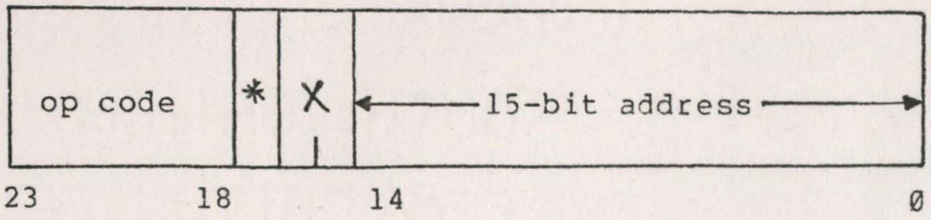
problem of the PDP-11, but forces a much more restrictive approach to code generation. For example, multiplication, division, shifting, remaindering, and extraction of arbitrary bytes from memory all must be performed using the A register. While this may seem attractive for overlapping operations without register to register transfers, in practice it results in just the opposite; the A register tends to become a bottleneck during expression calculations.

## 2. Addressing Modes

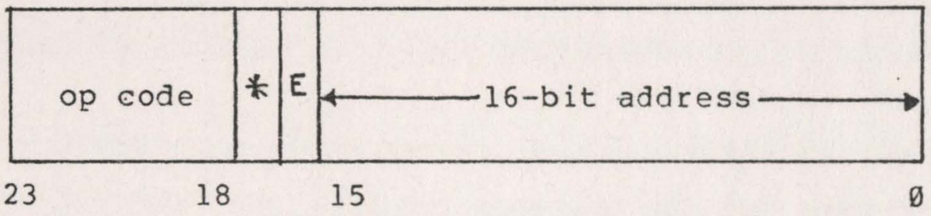
Total memory available to a /6 cpu is 256K 24-bit words. Because of the cpu's basic architecture and the corresponding addressing technique, executable code is confined to the lower 64K words of memory. However, memory above 64K may be addressed by means of indirect references. Figure III-2 illustrates the memory referencing formats available.

### 2.1. Direct Addressing

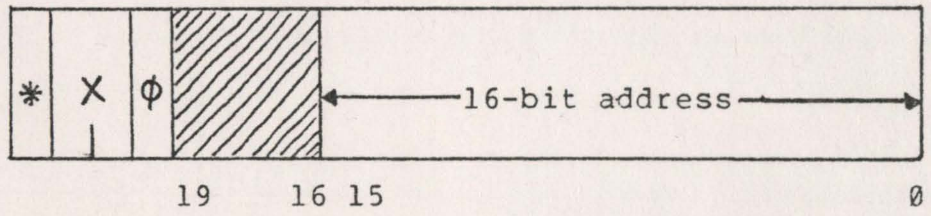
The standard memory reference instruction format allows the direct addressing of 32K words. The value of 32K words is a constraint imposed by the 15-bit address field of the instruction word. The addressing logic divides the lower 64K words of memory into two areas: Ø-32K; and 32K-64K. Under this method, the most



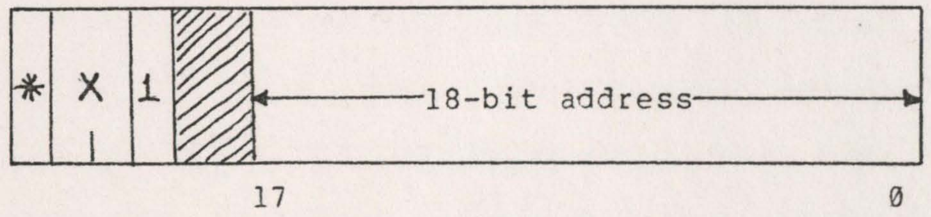
Direct Addressing



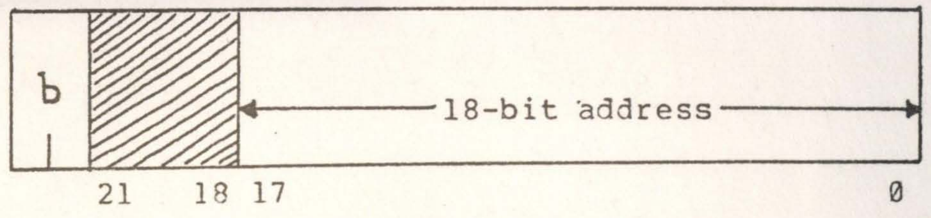
Long Branch Instructions



Standard Indirect Format (DAC)



Long Address Format (LAC)



Byte Address Constant (BAC)

Figure III-2. /6 Memory Reference Formats significant bit of the program counter is used to bias all direct address references. By performing a logical-or function between the immediate address reference and bit 15 of the program counter, instructions may directly address up to 32K words within their respective sections of memory.

Modification of a 15-bit direct address by means of indirection (*) and/or indexing (X) can permit an instruction to address any memory location up to 256K words. These provisions are discussed in sections 2.2 and 2.3.

A special group of _long branch_ instructions permits direct addressing up to 64K words. The instruction word format for this group is shown in Figure III-2. Note that these instructions may be modified by indirect references (*), but have no provision for indexing. Long branch instructions are not biased by bit 15 of the program counter.

## _2.2. Indirect Addressing_

Indirect address references permit the cpu to access up to 256K words of memory. When a memory reference instruction is decoded, bit 17 (*) of the instruction word is examined. If bit 17 is set, an indirect address reference is indicated. The word retrieved from memory when the effective address is calculated is treated as an indirect address word. Consequently, indirect addressing references may be chained together. The /6 imposes no restriction on the depth of chaining; however, the stall alarm feature may be enabled, limiting the total effective memory address calculation to 128 machine cycles.

The standard indirect format, with its 16-bit address field, permits access up to 64K words. Up to 256K words can be accessed by the 18-bit field in the long address word. Neither type of indirect address is affected by the program counter´s address bias.

Bit 23 (*) of either format may be set to specify another level of indirect addressing. Each level of indirect reference may be individually indexed to provide further address modification.

## 2.3. Indexing

A direct or indirect address reference may be modified by indexing. This operation adds the contents of a specific index register (I, J, or K) to the address in the current instruction or indirect reference to determine an effective address. A two bit field (X) in the instruction or indirect reference specifies which register will be employed in each indexing operation.

In the lower 32K memory section, direct address references may be indexed to access up to 64K words. However, instructions in the 32K-64K section of memory may not reference the lower section by indexing, since all immediate references will be biased by bit 15 of the pc.

## 2.4. Byte Addressing

The byte processing group of instructions permits program manipulation of all three bytes within a memory location. These instructions are divided into two classes: those that operate on a standard address format and always reference byte 3 (the right most byte), and those that work with a special byte address format (see Figure III-2) to access arbitrary bytes in memory.

The set of instructions that work with standard address formats includes operations to add a byte to the B register, subtract a byte from the B register, compare a byte to the B register, and transfer a byte in memory to/from the B register. The instruction formats permit both indirect address references and indexing.

The collection of instructions that operate on the byte address format is very restricted. They consist entirely of instructions to transfer bytes in memory to and from the B register, and instructions to increment a byte pointer residing in the I or J register. All arithmetic operations performed on bytes inaccessible by a standard address format must be carried out in registers. The instruction to fetch a byte from memory (_emb_) uses a byte address constant placed in the J register to locate the desired memory location, while the replacement of a byte in memory (the _rbm_ instruction) employs an address found in the I register.

## 3. Data Types

The /6 architecture supports data types for single word integer arithmetic and floating point arithmetic. In addition, a double word, double precision integer data type is supported by a small set of instructions. Within C, this double word data type is used for longs, while the standard data type is used for integer types (i.e. unsigned, int, and short). Floating point arithmetic is supported by the hardware only in double precision. The double word data type, shown in Figure III-3, has a particularly ugly property to it: the sign bit in the low word must be zero, or unpredictable results may occur. As a result, many conversions and arithmetic operations involving longs require extensive cleaning up.

## 4. Stack Management

The /6 instruction set provides no direct support for a stack. The architecture is very much FORTRAN oriented, unlike most of the previous machines to which C has been moved.

Because of the architecture of the /6, it was decided that no registers offered to a user process were suitable for dedication to a stack pointer. Instead, a memory location was allocated for it. Since the stack pointer must reside in data space, to allow creation of a pure text segment, the memory location can not be fixed, and must be referenced symbolically. Also, as a result of the limitations of the direct addressing mode, all references to this memory location must be indirect to allow its placement anywhere in the 256K word address space. The cost of adding an extra level of indirection for each reference to the stack pointer was considered at length. However, to allow for the eventual creation of programs of maximum size, one must simply pay the price.

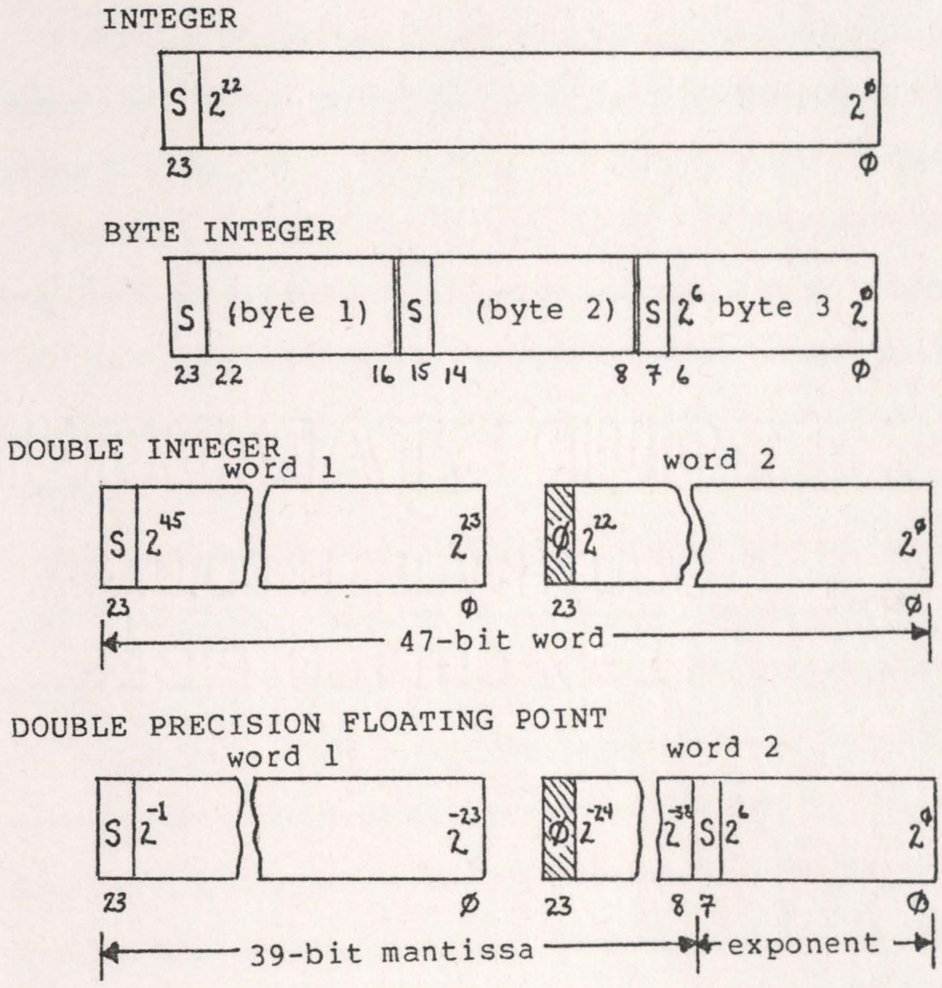


Figure III-3. Data Type Representations

The C language, as developed on the PDP-11 under UNIX, has the stack managed by the hardware and the operating system. The stack is initialized to the top of the user´s logical address space and allowed to grow downward as needed. A fixed size segment is allocated for an initial stack segment, with memory faults interpreted by UNIX as an indication of the stack needing expansion. This uniform treatment of memory faults implies that inadvertant traps, caused by faulty user programs, expand the stack needlessly. Under UNIX/24V (the official name for the UNIX implementation on a 6Ø24/6 cpu) a similar approach was selected. A users´ stack is treated as on the PDP-11, but the maximum stack size is set by the loader and is integral to each executable program. The heuristic used by the loader to set the stack size is based on the program size. The user may override the heuristic and specify the stack size. The reason for having the stack size vary is related to the /6 virtual memory hardware; it suffices to say that uniformly starting the stack at the top of the 256K word address space is too costly in terms of operating system resources. For a complete discussion of the UNIX/24V handling of the stack the reader is referred to [26].

The implementation of C on the PDP-11 employs two general purpose registers to manage the stack. R5 is used as the frame pointer, and the normal hardware stack pointer R6 (SP) points to the location on the "top" of the stack. The stack grows down from high memory, with local variables being referenced via negative offsets from the frame pointer, and function arguments, placed on the stack prior to the register save area, referenced via positive offsets from the frame pointer (see Figure III-4). The management of this stack arrangement is handled by a pair of linkage routines, *csv* (for saving live registers on procedure entrance) and *cret* (for cleaning things up on procedure exit). For the /6, the management scheme is virtually identical to that used for the PDP-11. The register save area differs in size (there are no *register* variables on the /6, so only the frame pointer and stack pointer need be saved), and the stack pointer points to the next open location on the top of the stack (to minimize the number of pushes required when passing parameters for procedure calls). At one point there was some thought to expanding the routines *csv* and *cret* in-line, however the eventual coding of these routines showed this to be too costly. For a more detailed discussion of the C calling sequence consult [17].

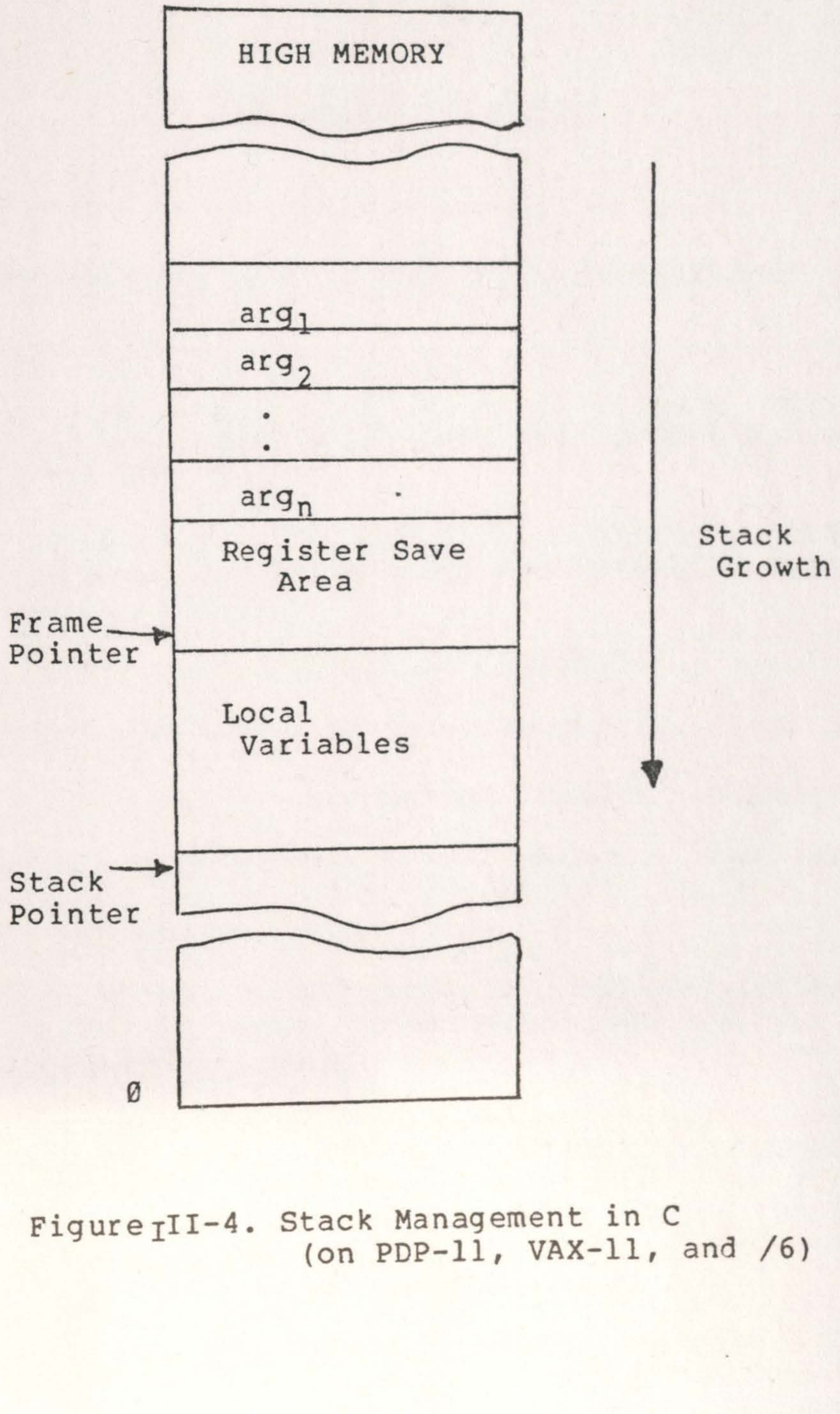


Figure III-4. Stack Management in C (on PDP-11, VAX-11, and /6)

# CHAPTER IV

# THE PORTING PROCESS

The porting process basically consisted of three steps:

1) Create a version of the compiler on a PDP-11/45 running UNIX, which generated symbolic assembly code for the /6.

2) Cross-compile the compiler to create a compiler running on the /6 under the native /6 operating system, VULCAN.

3) Re-fit the compiler for UNIX/24V.

While there may appear to be no reason for step 2), a number of major stumbling blocks required the intermediate step of going to VULCAN. Some of the problems encountered were:

1) The 11/45 had no compatible physical medium with which to communicate with the /6. All files transfered from the 11/45 to the /6, and vice versa, had to be sent over terminal lines at 300 baud (/6 => 11/45) and 1200 baud (11/45 => /6). With such an inefficient method for communication it became imperative to minimize the amount of work carried out on the 11/45. Consequently, little UNIX/24V development was carried out on the 11/45.

2) There were only two people involved in the majority of the project, and moving directly to a UNIX system would have required a /6 assembler and link-editor running on the 11/45, plus binary conversion programs to be able to boot a system directly off of tape or disk. By moving to VULCAN first we were able to take advantage of the existing assembler, link-editor, and debugger.

3) The existing VULCAN environment, augmented by the C compiler and standard I/O library, allowed parallel conversion of user programs such as the shell and editor.

Expanding the first list then, a more detailed list of the steps taken in the porting effort would be:

1) A C compiler was developed on the 11/45.

2a) A standard I/O library was tailored to the VULCAN operating system.

2b) UNIX/24V development work was carried out on the /6.

2c) The C compiler was cross-compiled and moved (numerous iterations) via the terminal link.

2d) The C compiler was modified to eliminate PDP-11 dependencies. This mostly involved alterations to data structures that required a long on the PDP-11, but only an int on the /6.

3a) A swapping version of UNIX/24V was brought up.

3b) YACC was moved to VULCAN to make the compiler completely self-sufficient under VULCAN.

3c) A version 7 shell was brought over to UNIX/24V, while a UNIX/24V assembler and link-editor were being written.

3d) The C compiler on VULCAN was modified to run with the UNIX/24V assembler, then compiled into .o files and moved to UNIX/24V where it was linked.

4) All other necessary user programs, not already on UNIX/24V, were moved.

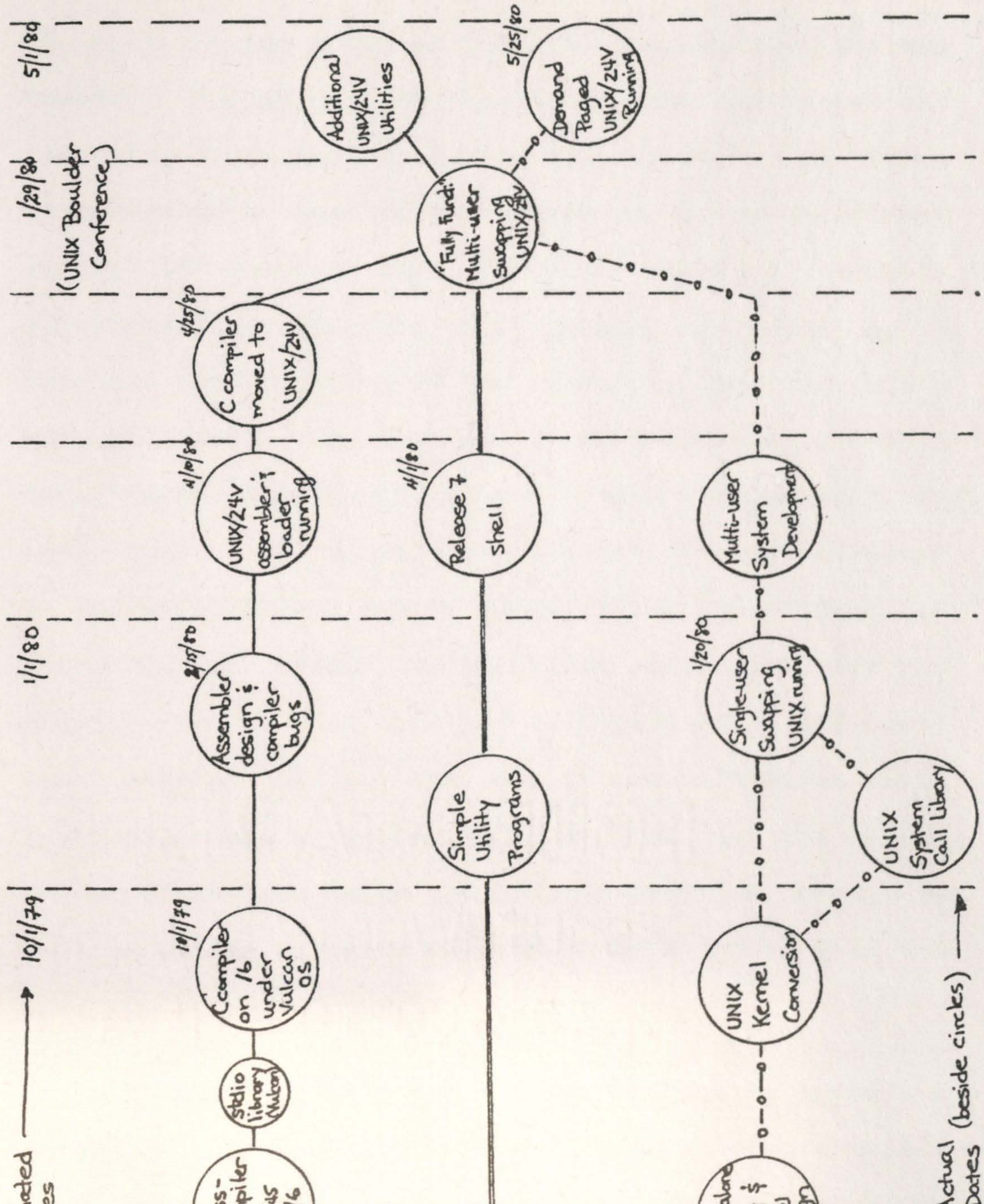
5) The UNIX/24V kernel development was moved to UNIX/24V.

After the last step, UNIX/24V was completely self sufficient. The current programming environment includes all essential programs, such as the shell, editor, assembler, link-editor, yacc, and compiler, plus numerous other tools such as make, sed, etc.

Because the compiler had to run under the VULCAN operating system for a long period of time (during kernel development), some compromises in the cross-compiler had to be selected. The VULCAN assembler and loader were not to be the eventual assembler and loader under UNIX, so little effort was put forth to make the compiler compatible with these programs. Rather a program external to the compiler was constructed to massage the assembly language output generated by the compiler into a format suitable for input to the VULCAN utilities. This post processing program, called cpop, was developed by the A. R. Jennings Computing Center in accordance with a previous agreement. The decision to handle the stay on VULCAN in this manner was caused by certain deficiencies in the assembler and loader (see chapter VII for a more detailed discussion).

The actual process of moving the portable compiler from the 11/45 to UNX/24V was fairly painful. The step between the 11/45 and VULCAN, over a 300 baud terminal line, took a number of weeks. This process was complicated by the involvement of cpop. Early on, bugs were traced as often to cpop as to the compiler, though very quickly the compiler became the standard culprit. Once the compiler initially reached the /6, very few trips back to the 11/45 were made. About one week after the compiler was deemed "usable" on the /6, it began to be fully used for kernel development. The trip from VULCAN to UNIX/24V was easier, but no less painful. While files could be moved by magnetic tape, the variable here was the assembler. The assembler was initially tested by use of a "mini-loader" (a simple-minded loader which relocated only a single file into the executable a.out format), but many instruction table typos and a few misunderstandings of the architecure resulted in a number of weeks worth of debugging. Once the assembler was stable, it was straightforward to move the compiler, and the UNIX/24V kernel soon followed. Since it is always interesting to look back on the work spent in such a project, Figure IV-1 has been included to allow comparison of the actual time spent on each phase of the porting effort to that which was originally expected. This figure should be taken lightly, as division of certain steps is nearly impossible.

The initial testing of the compiler involved the creation of a "validation suite". This collection of test cases was used extensively during the early debugging phases that took place on the 11/45. Once



8/1/79
Estimated Dates
10/1/79
1/1/80
1/29/80 (UNIX Boulder Conference)
5/1/80
Receive Release 7 Distribution
S.J. Leffler
C cross-compiler PDP-11/45 ⇒ /16
stdio library (Vulcan)
C compiler on /16 under Vulcan OS
10/1/79
Assembler design & compiler bugs
2/15/80
UNIX/24V assembler & loader running
4/10/80
C compiler moved to UNIX/24V
4/25/80
ARJCC
Simple Utility Programs
Release 7 shell
4/1/80
"Fully Func." Multi-user Swapping UNIX/24V
Additional UNIX/24V Utilities
Demand Paged UNIX/24V Running
5/25/80
W.A. Shannon
Standalone Testing & Kernel Design
UNIX Kernel Conversion
UNIX System Call Library
Single-user Swapping UNIX running
1/20/80
Multi-user System Development
8/1/79
Actual Dates (beside circles)

Figure IV-1. Development Schdeule for UNIX/24V

the compiler was moved to the VULCAN operating system, however, regression testing using the test cases was virtually abandoned, since the size of the test suite (approximately 200 files) required an extraordinary amount of compile time. Later testing usually consisted of specific test cases, followed by a complete recompilation of the compiler and operating system. While abandonment of the validation suite is not a recommended practice, most bugs encountered at this stage were usually as a result of "edge effects" in the Sethi-Ullman number calculations, and fixes had virtually no affect on unrelated constructs. If the compiler had been an order of magnitude more efficient (see chapter VIII), the use of the validation suite would have been possible. However, due to the large number of changes being made to the compiler during its stay on VULCAN, lengthy testing of this sort was not feasible.

In addition to the basic compiler being moved to VULCAN, a prerequisite was the creation of a standard I/O library to interface to the VULCAN operating system. The C language provides no I/O support within the language. Thus, in order to move the compiler from one machine to another, a friendly I/O environment must be provided. The standard I/O library was brought up in two stages. First, a version of the library was implemented under the Wisconsin C compiler. This allowed for testing of many of the algorithms involved, as well as the development of a small number of useful utility programs. When the cross-compiler was ready for use, the library was then recompiled on the 11/45 and moved along with the compiler. The stack design of the Wisconsin compiler precluded any possible mixing of compiled code.

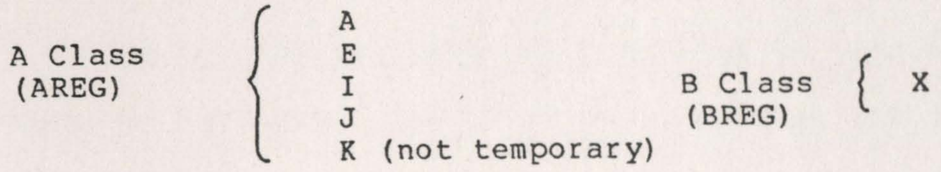
# CHAPTER V

# MOVING THE PORTABLE C COMPILER TO THE /6

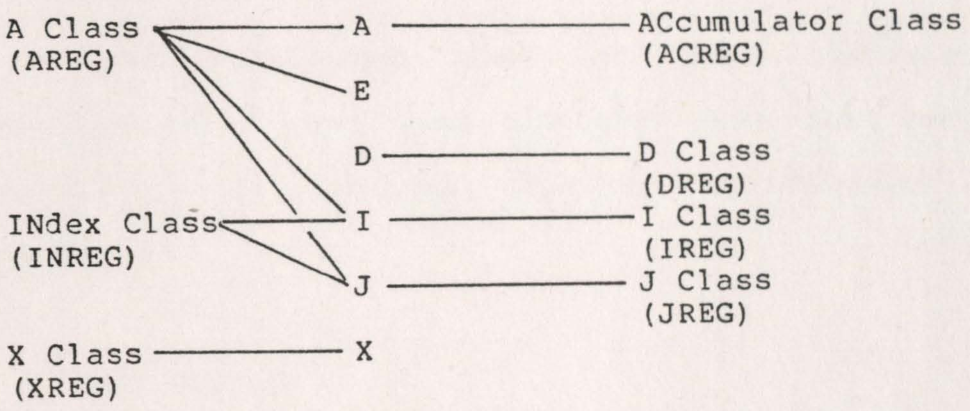
## 1. The Machine Model

As described in chapter II, the portable compiler has an abstract machine model to which a target machine must be mapped. Unfortunately, the /6 failed to fit into this model. The first problem involved the index registers. With only two register classes available in the original model, the /6 register set was divided such that index registers and arithmetic accumulators were in the same class, while the floating point register was kept in a class by itself (see Figure V-1). This caused considerable problems, the most difficult being the allocation of an index register. Since the machine model assumed all registers placed in the same class were identical, it was not possible to guarantee allocation of an index register when needed. For this reason, and because of problems encountered in handling the allocation of the D register, it was decided that more than two register classes were necessary. Since the abstract machine model was to be altered, it was decided to tailor it specifically to the needs of the /6 (to minimize the work that had to be done in other machine dependent portions of the compiler). The register classes shown in Figure V-1 were the result of these modifications to the machine model. Note that many of the register classes overlap, and that the class SDREG contains only the D register. The choice of placing the D register in a separate class implied that the register allocation scheme no longer needed to be concerned with register pairing. However, this decision also implied that the allocation routines were now required to handle the problem of physically overlapping registers. Certain of the register classes (SIREG and SJREG, in particular) were added soley to allow recognition of specific shapes in the code tables; these classes are never requested for allocation.

The modifications that were necessary to implement the new machine model were simple. The register allocation routines were altered to be aware of the new register classes. The low level allocation routines no longer needed to be concerned with register pairing, but instead had to handle the physical overlapping of the D register. Finally, the shape matching routine used in the template matching algorithm were made aware of the new shapes associated with the new register classes.



Original /6 Register Classes



Final /6 Register Classes

Figure V-1. Original and Final Register Classes in the /6 Machine Model

## 2. Address Arithmetic

By far the most difficult problem in moving the compiler involved the handling of address arithmetic. The difficulty stemmed from the /6's basic word addressable architecture, and the format of the pointers used to retrieve bytes from memory.

To understand some of the difficulties encountered, one must consider how the compiler normally forms addresses. Consider the following code sample,

```
int foo[2Ø];
...
.. = foo[...];
```

On the PDP-11 and VAX-11, a portion of the expression tree to be passed to the second pass of the compiler appears as

```
U*, int, ...
    +, PTR int, ...
        ICON, _foo, PTR int, ...
        <expression>, int, ...
```

where the expression contains the index in bytes. However, on the /6, the appropriate index should be in words, not bytes. The index is formed, in a machine dependent manner, by converting the internal representation of the offset, maintained in bits, to the numeric value it should have for transmission to the code generator. By interpreting the type of the node to which an offset is to be added, in most cases the conversion from bits to bytes/words is obvious. However, not all cases are so obvious. For example, consider another sequence of code,

```
struct foo {
     int   a;
     char  b;
} bar[10];
...
..= bar[..].a;
..= bar[..].b;
```

This example hits at the core of the problem. The structure contains two members, one a character, the other an integer. The natural addressing structure on the /6 for each of these items is very different; one may be accessed by indexing, while the other requires a byte pointer. Consequently, all offsets constructed in addressing the integer member must be in words, while offsets for the character should be in bytes (in order to optimize the address calculations). If one considers the address formation as the trees are built, both have the common base

```
+, PTR strty, ...
     ICON, _bar, PTR strty, ...
     PMCONV, int, ...
        <index expression>, int, ...
        ICON, <width of structure>, int, ...
```

The PMCONV node refects the semantics of addition to a pointer in C. The addition results in the index expression being multiplied by the width of the structure, so that one may point to the appropriate element of the aggregate. At this point in the address formation, the tree building routines do not know which element of the structure will be accessed (if, indeed, an element will be addressed at all). Thus, a decision as to which type of calculation (byte or word) should be performed for the offset is impossible.

The problem of deciding between byte and word offsets permeates much of the machine dependent portion of the first pass of the compiler. As shown above, there are cases where the compiler can not, in a straightforward manner, decide whether to create an address offset in bytes or words. This raises the possibility of leaving all offsets in bytes. Unfortunately, this is far too expensive to be a viable solution. Assume all address offsets are maintained in bytes. If one now considers the first example of this section, the original tree used to form the array index would appear as

```
U*, int, ...
    +, PTR int, ...
        ICON, _foo, PTR int, ...
        PMCONV, int, ...
            <expression>, int, ...
            ICON, 3, int, ...
```

The 3 in the last line indicates that each element of the array is 3 bytes wide (since the /6 has a 24-bit word). Because the offset is maintained in bytes, the PMCONV node can not be collapsed to the previous form for the tree. Without intervention before the second pass of the compiler to convert the byte count to a word count, this expression tree will have incorrect code generated for it. The problem of intervening in instances such as this is very difficult, thus one may conclude that maintaining all address offsets in bytes is impractical.

Since it has been shown that both byte and word offsets are required, a logical question to ask is how they might be distinguished? Within the original scheme of the compiler it was not possible. In most cases, offsets are merely integer constants -- ICON nodes. However, the only distinguishing characteristic of an ICON node is its type. To handle the identification problem, a new characteristic was added to all ICON nodes describing their "offset-type". That is, a new node element, sym x, was added to all ICON nodes. Sym x has one of the values BYTES, WORDS, or NOTYPE to indicate that the ICON is an offset in bytes, an offset in words, or a constant introduced outside of the compiler (e.g. a program constant).

By using this added information, the problem of forming the appropriate offset type (bytes or words) may be delayed, in the most difficult cases, until information is available to make a decision. When a decision is formed, a pass is made through the expression tree to convert appropriate expressions. For instance, if we reconsider the structure reference, we find that the partial tree from before takes the form,

```
+, PTR strty, ...
    ICON, _bar, PTR strty, ...
    PMCONV, int, ...
        <index expression>, int, ...
        ICON, 2 (WORDS), int, ...
```

The size of the structure is maintained in words, as all structures are aligned to word boundaries according to the semantics of C. Now, if the expression refers to the integer member of the structure, the resultant expression tree is,

```
U*, int, ...
     +, PTR int, ...
          ICON, _bar, PTR int, ...
          *, int, ...
            <index expression>, int, ...
            ICON, 2 (WORDS), int, ...
```

While the "intermediate" tree for the character member is,

```
U*, char, ...
     +, PTR char, ...
          PCONV, PTR char, ...
               +, PTR strty, ...
                    ICON, _bar, PTR strty, ...
                    PMCONV, int, ...
                      <index expression>, int, ...
                      ICON, 2 (WORDS), int, ...
          ICON, 1 (BYTES), int, ...
```

This tree is intermediate because the word offset has yet to be converted to a byte offset. To create the final tree we must consider first what happens to the PMCONV node. For the /6, this is always transformed to straight multiplication:

```
U*, char, ...
     +, PTR char, ...
          PCONV, PTR char, ...
               +, PTR strty, ...
                    ICON, _bar, PTR strty, ...
                    *, int, ...
                         <index expression>, int, ...
                         ICON, 2 (WORDS), int, ...
          ICON, 1 (BYTES), int, ...
```

However, a machine independent local optimization will normally convert this to a left shift (see section 3 of this chapter), since one of the operands is a constant and a power of two. Thus, the true form of the intermediate tree is,

```
U*, char, ...
     +, PTR char, ...
          PCONV, PTR char, ...
               +, PTR strty, ...
                    ICON, _bar, PTR strty, ...
                    <<, int, ...
                         <index expression>, int, ...
                         ICON, 1 (WORDS), int, ...
          ICON, 1 (BYTES), int, ...
```

Now, to convert this to a suitable byte address, a pass is made through the tree, converting all word offsets to byte offsets. The construct involving the << node is recognized as the result of an optimization, and converted to the appropriate byte calculation. The result of the transformations is,

```
U*, char, ...
     +, PTR char, ...
          PCONV, PTR char, ...
               +, PTR strty, ...
                    ICON, _bar, PTR strty, ...
                    *, int, ...
                         <index expression>, int, ...
                         ICON, 3 (BYTES), int, ...
          ICON, 1 (BYTES), int, ...
```

The following list summarizes the reasoning behind the design of the eventual scheme for handling address calculations:

1) The first pass of the compiler must handle the machine dependent translation of all offsets from bits to bytes/words; the second pass of the compiler expects all this work to be previously performed.

2) Since the /6 has two distinct techniques for addressing memory, one for word-oriented items, the other for bytes, offsets must necessarily be maintained in both bytes and words.

3) The simple cases are handled with little trouble by interrogating the type of the expression to which an offset is to be added; the only difficulty exists in the formation of addresses for structures.

4) The intermediate address calculations involving structures are always performed in words since the constants involved will all be offsets to a word-aligned memory location, and because the resultant address calculation may not be used for the addressing of an element of basic type (e.g. bar[3], for the above example, is a perfectly legal construction).

5) All references to items of character type (_char_ and _unsigned char_) that involve address offsets have _these offsets_ maintained in bytes (with the exception of NAME nodes, which will be described shortly).

The result is that the second pass of the compiler may assume any address construction involving a word-oriented item will involve only word offsets, while offsets for byte items are almost always in bytes.

The reader may be asking why all offsets involved in byte address calculations _must_ be maintained in bytes? In the previous example dealing with the structure, it appears that a more viable approach to the calculation of the byte pointer is to form the word address, convert to a byte pointer, then add the byte offset left over. Clearly this approach would result in the same address as the approach taken. There are two problems with such an attack. First, the cost in manipulating byte pointers, both in conversion and calculation, is enormous; this approach tends to increase the number of such calculations. Second, the compiler attempts to minimize the number of byte pointer calculations by combining constants wherever possible; by maintaining offsets in both words and bytes in the same expression subtree, such optimizations are greatly complicated. In general, the resulting code for byte pointer calculations has been found to be very good. This is due mainly to the approach taken in handling address offsets.

In the previous discussion, it was assumed that the character item the compiler was attempting to address was in an arbitrary memory location. In many instances this is not the case, and the compiler may use an alternate form of addressing which is far cheaper than creating a byte pointer. For those items located in the third byte of a word, the character may be retrieved by supplying only the word address of the word it is contained in. In addition, while arbitrary bytes may only be brought to or from memory when byte pointers are used, bytes addressable in this special case may be used in limited arithmetic calculations (addition and subtraction). As a result, the compiler trys, whenever possible, to establish the feasability of this word addressable format. Cases where bytes may be retrieved in this manner include those bytes at constant offsets within aggregate structures (both on the stack and in "main" memory). When such an addressable byte is recognized, if possible, the address offset is folded to form a NAME node. The second pass understands that all NAME nodes of character type are examples of this special addressing form.

Further candidates for the above addressing format are single characters allocated static storage. However, some problems arise in performing address calculations in this manner. The portable compiler has a simple notion of the way bytes are laid out in a machine word. One may specify only that bytes are always placed left to right, or right to left. Since the /6 is word addressable, the attachment of a label to a memory location (in the assembler) allocates a _word_ of storage, not a byte as the portable compiler likes to believe. Consequently, the compiler assumes that a statically allocated character is allocated a memory cell (on the /6 a word) and placed in the first byte of a word (on the /6 the left most one). Thus, if one is to arbitrarily assume that statically allocated bytes are situated in the third byte (to aid in addressing), inconsistencies may appear. For example, if the user attempts to initialize a statically allocated character or take the address of such a variable (using the & operator), the result will reflect the notion that the value is located in the first, rather than third, byte of the word. Consequently, to optimize addressing of static character items, some modifications (special cases) were necessary to bypass the compiler´s idea of byte layout in statically allocated characters.

## 3. Sethi-Ullman Number Computations

The movement of the portable C compiler to a new machine requires the creation of a number of machine dependent modules which are hooked into the machine independent portions of the compiler. Examples of these modules include those used to handle tree transformations in processing type conversions, the generation of code for subroutine prologs, epilogs, switch statements, etc., and the compilation of expressions. In the process of compiling expressions one of the most important routines calculates Sethi-Ullman numbers for parse trees. As described in chapter II, Sethi-Ullman numbers are intended to reflect the minimal number of registers needed to evaluate an expression. The numbers are calculated when a tree is initially handed to the second pass of the compiler (be it from an intermediate file or directly), and after every transformation is performed on the tree (e.g. after a code sequence has been emitted). Each node of an expression tree has a Sethi-Ullman number stored in it to reflect the relative difficulty of evaluating the tree rooted at that node. This section will describe the algorithms employed in the /6 compiler for calculating Sethi-Ullman numbers. It should be noted that the problem of calculating "perfect" Sethi-Ullman numbers (i.e. estimating the minimal number of registers required to evaluate an expression tree) for any but the simplest machines is NP-complete [3]. Thus, the algorithms found in almost all versions of the portable compiler are swamped in heuristics, and the creation of such algorithms should be considered at best an art.

The Sethi-Ullman computations normally create a number for a node dependent on the numbers of each descendent, and the operator of the node under consideration.

```
        op, <type>, su, ...
            left, <typel>, sul, ...
            right, <typer>, sur, ...
```

That is, the calculation of su is dependent on op, sul, and sur. In addition, in most cases, the type of each node involved may come into the calculation. For a leaf node, the values for sul and sur are assumed to be zero. For machines with an orthogonal architecture, computations may normally be grouped according to operator classes or by the operand types. For instance, for the PDP-11 and VAX-11 virtually all simple arithmetic operations have identical addressing modes and allowable source and destination locations, allowing computations to be identical. However, the /6 instruction set has numerous special case restrictions and supports a highly iregular collection of operand addressing modes. Thus, the Sethi-Ullman computations for the /6 tend to be extermely complex, with almost every operator having separate calculation rules. Further, the many restrictions on register usage often requires computation for an operator be overestimated, to guarantee the availability of a necessary register. For instance, all operations involving longs must have the D register available for use. If the Sethi-Ullman number computed for that node were to specify it needed only two registers (the size of the D register), the code generation scheme could conceivably be left with the I and J register which would satisfy its requirements, but prevent it from generating code. Therefore, in many instances, operations involving a long must specify a requirement for all of the registers to insure that the D register will be available for allocation. This difficulty in reserving registers required the handling of floating point operations, and operations involving longs to treat the /6 almost as if it were a single accumulator machine.

All directly addressable items -- constants, static memory locations, indirect references, etc. -- are assigned a value of zero. Indirection operators (U*) normally require one extra register to hold the address. Certain indirect references, such as those requiring character extractions, will always require a register. Assignment operations are manipulated to insure that the right hand side is placed in a register when the expression is not addressable. Care must be taken in handling assignment statements to make sure that the right hand side isn´t placed in a register before the right hand side is addressable. For example, the construct

```
*f(a) = b
```

if handled as previously specified, would result in the right hand side being placed in a register before the function call was made, but function calls require all the registers to insure all scratch registers will be free.

Logical operations (i.e. comparisons) have the most compact calculation rules of any class of C operators. Depending on the size of the object involved in the comparison, the following calculation is used:

```
su = min (max (sul, sur+size),
          max (sur, sul+size))
```

The size is the number of registers required to hold an item to be compared: 1 for an integer or character, 2 for a long or float. The calculation rule reflects an attempt to evaluate the more difficult side first.

Storage conversion operations are special-cased to recognize certain of the more difficult possibilities. The conversion of a pointer to an integer type is a no-operation, with the exception of a byte pointer. This conversion is very expensive, requiring all but one of the scratch registers. Since the resource requirements are so great, and certain specific registers are needed to perform the operation (the A and E registers), an SCONV node involving a byte pointer is reserved all the registers. The handling of conversions to a long format requires the D register. However, as described at the beginning of this section, the allocation of the D register proved to be a difficult problem, so storage conversions to a long also require all the registers.

Assignment operators (i.e. +=, -=, etc.) are very difficult to evaluate, because the /6 has virtually no instruction support for them. Consequently, the machine dependent rewriting routines almost always transform these operations to their equivalent form,

```
e1 op= e2  => e1 = e1 op e2
```

after any side effects in the left hand side have been "weeded out". Thus, the calculations for assignment operators are used mainly in guiding Sethi-Ullman calculations higher up in the tree. In general, the handling of these operators is broken into two classes: those involving single word items, and all others. The latter category encompasses character items and double word items (longs and floats). If the assignment operator involves word-addressable items, the calculation is identical to a simple assignment operation (=). This allows the /6 instructions to add a register to memory, add a constant to memory, etc. to be used without penalty. Those operations that will require rewriting, as discussed previously, will be transformed and recalculated to reflect the true nature of the work which must be performed. All other cases fall into the difficult category that must be handled very carefully, and thus are specified to require all registers. One should note that the condition implied by the latter decision, while appearing to be costly, exists for only a "short period of time", since the operator is almost immediately rewritten. Thus, the difficulty in performing the operation is merely for the benefit of calculation at a higher level in the expression tree.

The handling of normal arithmetic operations (+, -, /, %, *, etc.) involves the most complexity since the quality of code generated for these operators has a heavy impact on the quality of code generated for many other operators (due to the way assignment operators are handled). The operations *, /, %, <<, and >> all require the A and E register, D register, or just the A register. After a number of months of trying various heuristics to optimize usage for these operations, it was decided to reserve all of the registers when handling these operators. This causes complex expressions involving one or more of the operators to invariably require a store into memory, but guarantees that compilation of expressions involving the operators will be successful (i.e. won´t result in an inability to allocate the needed register). The difficulty in approximating register needs for these operators indicates that expressions involving the operators are good targets for future optimization (see chapter VI). The remaining operators, + and -, require close scrutiny of the types involved. This is because, for instance, the addition of two integers requires far less work than the addition of an integer to a byte pointer. A myriad of heuristics are involved in computing estimates for operations involving byte pointers; the actual code is the definitive source for an exact description of what goes on. The overall approach to handling these operators, when word-addressable items are involved, closely resembles that used for logical operations. The rule applied to expressions where the left hand operand is directly addressable is

```
su = max (size+size,
          min (max (sul, size+sur),
               max (sur, size+sul)))
```

This calculation, as for logical operators, reflects an intent to place the more difficult expression in a register first, then the easier. The possibility of both operands having to be placed in registers is reflected in <u>size+size</u>. For expressions whose right hand operand is addressable, the simple estimate

su = max (size, sul)

suffices. This usually causes the left hand (unaddressable) side to be placed in a register before the operation takes place.

The remaining calculations of interest handle PACONV and PSCONV nodes. The work here involves checking to see whether the operands will require a full calculation, or whether an optimization may be performed (see chapter VI). If the offset operand is a constant, the code emitted will not require all the registers to perform the calculation, so fewer registers are needed, and fewer stores into memory will be generated.

The rules described comprise the majority of the Sethi-Ullman calculation procedure. However, certain side effects may also take place as a result of calculations. An attempt is made to place the tree in a canonical form, to insure consistent tree shapes may be expected in the rewriting routines. These canonical shapes normally are meaningful only when dealing with commutative operators, such as +. The rules employed in canonicalizing the trees are:

- if a float and double are involved, place the double on the right
- if a character and a word item are involved, place the character on the left
- if a long and something other than a long are involved, place the long on the right
- if two longs are involved, place the more complex one on the right
- if the right hand side has a larger Sethi-Ullman number than the right, swap the operands

The code should be consulted for further elaboration on this simplified description. The result of these transformations is to place the more difficult operand on the left of an operator. Rewriting routines may then assume this without checking the Sethi-Ullman numbers. (One shold note that once a quantity has been placed in a register it is never moved about in the tree by a canonicalizing transformation; this avoids destructive interference between the Sethi-Ullman computations and the machine dependent rewriting rules.)

## 4. Register Allocation Strategies

The register allocation strategy in the portable compiler is divided into two sections -- a machine independent portion and a machine dependent portion. The machine independent portion of the strategy involves keeping track of those registers which are busy, allocating registers within a certain class, and other bookkeeping-type chores. The machine dependent portion allows a designer to specify, in a more exact manner, register needs or preferences. Each node in an expression tree has a member, rall, which is used by the machine dependent allocation strategy to indicate if a specific register is needed or preferred in the calculation of the expression rooted at that node. Thus, if an expression must be calculated into, say, the A register, the designer may communicate this need to the machine independent allocation routines by placing the code for the A register in rall and or-ing in a flag that specifies the result of the calculation must be placed there (a MUSTDO flag). When code is generated for the tree rooted at the node, the machine independent routine which allocates registers will take this information into account. If the result of compiling the expression is not the required register, a transfer will automatically be generated to satisfy the MUSTDO condition. Should this transfer fail due to the register being busy, the compiler will abort. The specification of a register preference allows the designer to "steer" expression calculations away from needed registers, or allow possible optimizations to take place (e.g. keeping things in the A register on the /6 for an &, |, or operation). This section will describe the strategies involved in handling the machine dependent portion of the register allocation scheme. As in the Sethi-Ullman computations, algorithms provided here are totally heuristic, driven by an attempt to minimize the cost of compiling each expression.

The A and E registers, alternatively the D register, have already been mentioned as major bottlenecks in code generation. Consequently, a large portion of the register allocation strategy is involved in steering operands away from these registers. For instance, calculation of the expression *a*+*b*, where *a* is a character pointer and *b* is a byte offset, may best be done by placing *a* in the A register and *b* in the I or J register before performing the actual calculation. Should *b* end up in the A or E register, it would have to be moved before *a* could be placed in the A register and manipulated (shifted and divided). Hence, the register allocation routine tries to steer _b_ away from the A register, though it is not always possible (suppose _b_ were actually _f()_, then the value returned from the function call would necessarily reside in the A register). Further work involves the requirements of the _emb_ and _rbm_ instructions. These instructions, for extracting and replacing bytes in memory, respectively, require one of their operands, the byte pointer, be placed in the J and I registers (J for an _emb_, I for an _rbm_).

Requirements of floating point operations are fairly simple, because of the simple structure of the floating point unit. Division requires that the left side be placed in the X register, while the right resides in the D register. Negation (U-) operations act on values only in the X register. All other requirements are handled in the code tables by explicit cases that perform register interchanges when the operands are incorrectly situated.

Function calls, where the function is not directly addressable (i.e. a pointer to a function is involved), must have the address placed in an index register. Arbitrarily, the I register was selected.

Exclusive-or, bit-or, bit-and, and multiplication are most efficiently performed when one of the operands is placed in the A register. Therefore, both operands of these operators show a preference to be placed there.

Division and remaindering must have the left operand in the A register and the right not in either the A or E registers. Thus, the left operand is MUSTDO´d into the A register and the right into the I register. Most of the time the right operand will not have to be placed in a register, but in case it requires evaluation prior to the right hand side (a function call, for instance), this requirement will keep the A and E registers free for allocation.

Variable left and right shifts cause some problems on the /6 because the instruction set supports only shifts of a constant number of bits. To handle this weakness and insure reentrancy, tables of shift instructions are available in the C run-time library. A variable shift, then, performs an "execute memory", exm instruction, after placing the value to be shifted in the A or D register. To execute the correct shift instruction out of the tables, the amount to be shifted must be placed in an index register and then used to index into the shift tables. Thus, register allocation requires the left operand be placed in the A or D register (MUSTDO´d in) and the right operand placed in an index register.

Assignments involving characters require that the character to be manipulated be placed in the B register and that the pointer to the memory location reside in the I register. Since the B register is never allocated by the allocation routines, the A register is requested instead. To steer intermediate calculations away from the I register, assignment operators force the left hand side´s address to always be placed in the I register and the right hand side´s address in the J register. The specification of I and J registers for addresses involved in character pointer manipulations often results in extraneous register transfers. This is because the decision to place an address in the I or J register is made at the assignment operator level in an expression tree, and forcing this decision on lower level calculations in the same tree often results in previous allocation decisions being reversed.

Addition, subtraction, PACONV, and PSCONV operations are primarily concerned with steering expressions into the most cost efficient configuration. The handling of character pointers is best done by placing the byte pointer (word address for PACONV and

PSCONV nodes) in an index register and the byte offset in the A register. If one or more of the operands fails to be situated correctly special cases in the code tables will have to generate register transfers or interchanges to set the operands up correctly before a calculation is performed. Usually, something of this sort occurs only when an operand is tied to a register by a previous register allocation (for instance a function call).

Storage and pointer conversions require the left operand be placed in the A or D register when a byte pointer or long is involved. These are MUSTDO situations.

The overall scheme of register allocations can best be seen in the handling of the assignment operators. The compiler attempts to maintain address type operands lying to the left of an assignment operator in the I register and those lying on the right in the J register. Much of the reasoning behind this scheme is due to the handling of characters, but it also is applicable to word oriented items (in forming OREG nodes by indexing off the I or J registers). Register transfers or stores into a compiler-generated temporary may arise when expressions cross the "borders" imposed by the assignment operators. This may be seen in expressions like

```
*(p = q + n) -= a * b
```

Where nested assignment statements are involved, incorrect placement of the operands is often observed. The resultant register transfers are very difficult to optimize out, since they reflect a belief, on the part of the compiler, that an expression should go in a specific register, and it doesn't recognize its error until deep into the calculation. Thus, to eliminate the register transfer, it is usually required to rework many previous calculations to insure the final result will reside in the correct register, thus eliminating the register transfer.

## 5. Machine Dependent Rewriting Rules

Whenever a search of the code tables for a match of the current expression tree fails, the tree is handed to a machine dependent rewriting routine. The routine is expected to manipulate the tree in such a way that another search of the code tables may result in a match. Manipulations usually take the form of evaluating a portion of the tree into a register or compiler temporary, though it is not required that one of these actions take place. Further, the transformation on the tree needn´t be of a "global" nature. Rather, only a small portion of the tree may be manipulated. Should further rewriting be required, the same routine, or a similar one may be called again. The process of rewriting trees involves close cooperation between all the rewriting rules as well as cooperation with the Sethi-Ullman calculation routine and the register allocation strategy. Since the rewriting of trees may not be successful, the compiler maintains a count of the number of recursive calls it makes to the rewriting rules to avoid infinite recursion.

The rewriting routines are divided according to the class of operators on which they act. For instance, routines must be supplied to rewrite binary operators, assignment operators (both =, and op= types), increment and decrement operators (++ and --), structure assignment operations, etc. This section will consider the work performed by the rewriting routines used in the /6 compiler. The focus will be on the general "attack" used in handling an expression tree; each rewriting routine will be considered separately.

The routine offstar is called whenever an indirection (U*) operator is to be rewritten. Offstar tries to form an OREG node whenever possible. This may be done if a tree of the form

```
U*, <type>, ...
     +/-, <type>, ...
          <expression1>, ...
          ICON, ....
```

is present. The subtree labeled expression1 must be placed in an index register to allow indexing to be used with the constant offset. The types involved are important, as a character pointer may not be used in an indexed addressing format. However, should a PACONV or PSCONV node lie under the U* operator, indexing is possible if a constant offset is present and the offset is "well formed" (the constant must be a number, say n, with n+1 mod 3 = 0). If it is not possible to form an OREG node (i.e. no +, -, PACONV, or PSCONV is a descendent), the subtree is forced into a register to allow a straight indirection to be performed.

The routine setasop is used to handle op= constructs. In most instances, the /6 instruction set is not equipped to efficiently handle these operators, so rewriting transforms the tree to an equivalent form:

```
e1 op= e2 => e1 = e1 op e2
```

after any side effects have been removed from the left hand side. As one might expect, recognizing side effects is a nontrivial chore. The scheme used to recognize them relies heavily on the Sethi-Ullman calculations. The routine assumes that a Sethi-Ullman number greater than zero indicates some calculation must be performed on the left hand side. Consequently, it tries to "evaluate out" any side effects before transforming the tree. Unfortunately this is not always possible. Consider the expression

```
*(p+3) op= f()
```

Should the left hand side be partially evaluated into a register before rewriting, the function call on the right may be executed with one or more scratch registers occupied with temporary calculations. The choice here is to hold off evaluating the left hand side, causing it to be evaluated twice, or to try to place it in a temporary memory location. The code generator won't place the expression in memory on its own, since it can't possibly have a Sethi-Ullman number greater than the number of free registers (the requirement to form a store operation), so setasop would have to make a decision for itself. Since most instances do not require a store, arbitrarily storing everything would be very costly. However, the work involved in optimizing store selection is also very costly. Thus, the routine chooses the first approach described: when a complex calculation is being performed on the right hand side that might possibly be a function call, the left hand side is not evaluated until after the tree has been rewritten. This approach leads to instances where poor code will be generated, but avoids adding a great deal of complexity to the rewriting process.

The remainder of setasop deals with optimizing those operations which may be performed with the add register to memory and add operand to memory instructions. To allow use of these instructions the right hand side is scrutinized for constants, registers, etc. If use of one of these instructions is possible, setasop delays forcing full evaluation of the right hand side, attempting instead to make the left hand side addressable, in the hopes that one of the instructions described may be applied.

Straight assignment operations are handled by the routine <u>setasg</u>. This routine, and the routine to handle binary operators, plays an important role in the overall rewriting scheme since most assignment operators are rewritten to form a binary operator and a simple assignment. Therefore, this routine is fairly complex. The /6 has no memory to memory transfer instructions, so the right hand side of an assignment operator must always be placed in a register (with the exception of an assignment of zero or -1). While placing the right hand side in a register is the main priority of the routine, it must be careful not to tie up a register needed for an address calculation on the left hand side. For instance, the expression

```
foo[a*b].bar = c
```

will probably require the A, and possibly the E, register to perform an address calculation for the structure reference. Therefore, setasg can not immediately place the right hand side in a register. Fortunately the Sethi-Ullman numbers may be used to decide when the left hand side is complicated enough to require an approach different than simply placing the right hand side in a register. The result is that the left and right hand sides are scanned for indirection operations and the like, with calculation priority going to the one with highest Sethi-Ullman number. This approach is clearly not infallible and has led to a few cases where a partial evaluation of one side has resulted in the other being untenable (the compiler reacts to such a situation by aborting).

The other difficulty in handling assigment operations stems from the possibility of cascaded assignment operators, i.e.

a = b = c = d = ... = expression

In this case one must be careful not to move too quickly to place the left hand side in a register, because too many registers may be tied up in calculating addresses for the memory locations associated with _a_, _b_, etc. Since the right hand side will occupy at most 2 registers room must always be left for its placement in a register. To avoid filling up registers a check is always made on the right hand side, to insure it requires no extra registers to be made available before acquiring new registers for the left hand side.

The final routine that will be considered is _setbin_. Setbin handles rewriting of binary operators. As mention previously, setbin plays a major role in producing quality code because other rewriting rules often introduce new binary operators.

Setbin must handle two classes of operations: logical and arithmetic. The handling of logical operators attempts to generate a "compare memory to addressable "a little at a time". That is, the routine tries to bounce back and forth between the operands, evaluating each side piece by piece, until eventually both are addressable. When this occurs, one of the two is selected to be placed in a register, and a comparison is performed with the other in memory. This scheme can be very dangerous, since the partial evaluation of the component expressions, if done in the wrong order, may result in both sides holding registers which are needed for the completion of the calculation of the other side (a deadlock of sorts). Consequently, checks are present to recognize cases where it is clear which side should be placed in a register first (e.g. a function call must be performed before a multiplication). In addition to the "rocking" scheme described, setbin may also reverse the sense of a comparison when one side is placed in a register. This is because the code tables were only made aware of comparisons where the left hand side has been placed in a register. Thus, when the right hand side is the first to make it into a register, setbin must reverse the sense of the comparison, and flip the tree to allow a match in the code tables.

The handling of arithmetic binary operators is guided mainly by the Sethi-Ullman numbers. As described in section 3 of this chapter, a by product of the Sethi-Ullman number calculations is that the left hand operand almost always has a higher Sethi-Ullman number than the right (i.e. it is "harder" to evaluate than the right). Setbin uses this to good effect, by usually working on the left hand side, going to the right only if the left hand side is addressable. This work is complicated by longs, since the routine assumes it can, in the worst case, place both operands in registers. Longs may be placed only in the D register, so simultaneous placement of long operands in registers is not feasible; setbin must handle this case specially.

## 6. Machine Independent Modules

The compiler, as distributed with release 7 of UNIX, had two "bugs" in it. One, more a machine dependency than a bug, involved the lexical scanner. Within the routine that handled recognition and conversion of numeric constants, a PDP-11 dependency had crept in. The C language specifies that those constants too large to fit in a single word are automatically typed long. In the original routine, the test performed to check overflow was based on a constant fitting in a 16-bit number. A trivial fix resulted in the machine dependency being removed from this supposed machine independent module.

A second bug was more serious in nature. The machine independent routine which handles the building of expression trees performs constant collapsing whenever both operands of the expression are suitably formed. However, in collapsing the tree the routine failed to correctly coerce types. This resulted in expressions such as 1L+1 being turned into (integer)2. The fix for this problem required a fair amount of code to be added. The problem appears to be basic to all versions of the compiler, since it was present in both PDP-11 and VAX-11/78Ø versions of the portable compiler.

## 7. The 64K Word Boundary

As discussed in chapter III, the only way for a program to address memory above 64K words is via an indirect reference through a memory location which has bit 2Ø set (a lac in the /6 terminology). Since this has a major impact on the code generation scheme, the current version of the compiler supports only programs less than or equal to 64K words in total size. Because this appears to be a detraction from the compiler, a discussion of the reasoning behind the restriction seems in order.

Since a lac is required to access memory, this implies that code can not be generated which performs any indirect references through user defined pointers stored in memory. That is, if one considers the construct *p in C, in order to insure the memory cell pointed to by p is addressable, the contents of p must first be placed in an index register, then an indirect reference must be made through a lac. For programs restricted to at most 64K words this is not required. Indirect references through a memory location of any type allow addressing memory in the lower 64K words of the address space, so the retrieval of *p is possible by an indirect reference through the memory location where p is stored. In terms of /6 assembly language the two code sequences that would be required to retrieve *p, assuming p is a pointer to an integer, are

```
64K words             256K words
---------             ----------
tma   *@_p            tmj   !_p
                      tma   !0,j
```

It is not feasible to maintain all pointers in a lac format, since this would imply that address arithmetic, comparisons, etc. would require special calculations. To handle programs of maximum size, then, it is necessary that the compiler always place user-defined pointers in an index register before performing an indirection. Having to place all pointers in an index register has a significant impact on the code generation scheme. Whereas before, certain expressions might have been addressable without any registers, to handle the expression under the restriction of a full address space requires at least one register. Thus, to handle a full 256K word address space the Sethi-Ullman number computations must be reworked. This portion of the compiler is by far the most difficult to construct and tune; any major modifications such as this, requires extensive work. In addition to adding complexity to the code generation scheme, code size and efficiency suffer when the address space is expanded. For these reasons the decision to limit program size to 64K words appears to be sound.

# CHAPTER VI

# AN EVALUATION OF CODE EFFICIENCY

This chapter furnishes qualitative and quantitative observations concerning the code generated by the /6 C compiler. One must remember that the current version of the compiler has had very little tuning performed on it. In addition, almost all C compilers running on other machines have a later pass which performs peephole optimizations. Later sections of this chapter contain observations concerning the impact a peephole optimizer will have on code quality. The feasability of a machine independent global optimizer is also considered, something along the lines of the optimizer built into the BLISS-11 compiler [32].

When considering code quality produced by the portable compiler, one must take into consideration the delicate balance between generation of efficient code and the reliability of the compiler. As mentioned in chapter II, the code generation scheme used by the compiler is very simple and takes great care to insure the compiler is consistent. When squeezing efficient code out of such a scheme, one must be careful not to cause the compiler to lose the ability to generate code for expressions it was previously able to handle. The tuning of the Sethi-Ullman computation routine tends to have a major impact both on code quality and compiler reliability. Thoughtless alterations to this routine to generate better code for a class of expressions may cause the resource calculations for related expressions to be underestimated.

Finally, since the compiler views individual expressions as disjoint objects, the generation of code is inherently limited to optimization within a single expression. The notion of cross-statement optimizations, or even cross expression optimizations, are out of the realm of the basic code generation scheme. Consequently, when viewing the quality of code generated by the portable compiler, one must localize it to individual expressions. At this level, the most notable item is the number of unnecessary stores that are generated.

## 1. Overview

At the highest level, the code generated by the compiler leaves a lot to be desired. This is due almost entirely to the problems encountered with redundant loads and stores that appear when individual code sequences are juxtaposed. However, with respect to individual expression trees, the code appears to be very good. This opinion is based on nearly 6 months of viewing assembly code generated by the compiler for the UNIX/24V operating system and its utilities.

The manipulation of multiple characters appears to be the most difficult area for the compiler to handle. This is true because nearly all character arithmetic must be performed in registers, and there is normally a great deal of pressure placed on the compiler to simultaneously stuff multiple characters in registers. Unfortunately, the complexity of this problem tends to overwhelm the compiler's simple-minded notion of the /6 architecture. The result is that stock code sequences must be used to insure multiple characters are placed in registers. When this is combined with the bottleneck imposed by the I and J registers for operations dealing with byte pointers, the code generated tends to be of a "worst case" variety. For example, the construct

```
char *p, *q;
...
while (*p++ != *q++)
        ;
...
```

is very common in C programs. The code the compiler will normally generate, assuming the character pointers are statically allocated, is

```
        Lnnn:
                tmj     !_p
                ----------
                bbj     .+1
                imj     !_p
                ----------
                emb     !Ø
                esb
                ----------
                tmj     !_q
                ----------
                bbj     .+1
                imj     !_q
                ----------
                tae
                emb     !Ø
                esb
                iae
                ----------
                cae
                bnz     Lnnn
```

The dotted lines serve to separate individual code sequences generated. The code created for the expression is fairly good. The difficult task of getting both characters into registers is handled nicely by the register interchange. With the exception of the interchange near the bottom, this code is nearly optimal when only local information is taken into account. Clearly, the looping nature of the construct would warrant the pointer p being maintained in a register througout the loop. However, enhanced register allocation strategies such as this are not possible within the scheme used by the portable compiler.

A more difficult situation for the compiler is an assignment operator dealing with characters. Consider

```
char *p, c;
...
*p++ |= c;
```

the resultant code would be

```
tmi    !_p
----------
bbi    .+1
imi    !_p
----------
tij
emb    !0
esb
----------
tae
tmb    !_c
esb
iae
----------
oea
----------
rbm    !0
```

Once again, this code is fairly good, aside from the register shuffling that must inevitably occur when two characters are brought from memory. The most notable step in the evaluation of the expression is that the pointer p was brought from memory only once and the ++ operation was performed early on, thus minimizing the number of memory-register transfers that had to take place. With more context available to the code generation scheme it is possible to expect the code might be improved to the following.

```
        tmi     !_p
        bbi     .+1
        imi     !_p
        tij
        emb     !0
        tae
        tmb     !_c
        oea
        rbm     !0
```

To generate code of this quality, the code generator would have to know that the result of the bitwise-or was to be used for assignment to a character data type. This would allow deletion of the conversions from 8-bit quantities to 24-bit quantities (the esb instructions). Rearranging the values in the A and E registers would be not be necessary if the code generator knew their values would be discarded after the bitwise-or was performed.

## 2. Optimization of Address Calculations

The original scheme for generation of code optimized word address calculations fairly well. However, the handling of addresses for character items left quite a bit to be desired. To improve the code that was generated for byte manipulations, modifications were made to the intermediate language; two new node types were added, specifically for character pointer manipulations. These new nodes, PACONV and PSCONV, are formed by squashing trees as shown in Figure VI-1. This collapsing process must perform the appropriate coersions to word quantities where needed. The form of the node corresponds to converting the expression

$$(\text{char } *)\underline{\text{word address}}+\underline{\text{byte offset}}$$

to a single operation.

The result of adding these new nodes is a compaction in the height of a tree which is passed to the code generator. With more information stored in a single node, code sequences may be tailored to handle operations in a more efficient manner. The introduction of these new nodes had a major impact on the complexity of the code generator; a number of machine independent modules had to be modified to be aware of their existence. However, the benefits, in terms of code quality, far outweighed the difficulties encountered in augmenting the intermediate language.

## 3. Machine Independent Local Optimizations

The compiler performs a small number of machine independent local optimizations on the expression trees in the first pass. These optimizations primarily

```
PCONV, PTR char
     +, ~PTR char                      PACONV, PTR char
          <base-address>    =>              <base-address>
          <word offset>                     <byte offset>


PCONV, PTR char
     -, ~PTR char                      PSCONV, PTR char
          <base-address>    =>              <base-address>
          <word offset>                     <byte offset>


+, PTR char
     PCONV, PTR char                   PACONV, PTR char
          <subtree>         =>              <subtree>
     <byte offset>                          <byte offset>


-, PTR char
     PCONV, PTR char                   PSCONV, PTR char
          <subtree>         =>              <subtree>
     <byte offset>                          <byte offset>
```

Figure VI-1. Definition of PACONV and PSCONV Nodes for the Intermediate Language

involve constant folding and variations on this theme. Some of the optimizations which were assumed to be machine independent turned out not to be applicable to the /6, while certain optimizations, inspired by the addition of the PACONV and PSCONV nodes, were added especially for the /6. Rather than give elaborate detail of the optimizations performed, a tree transformation will be shown, followed by an explanation of any fine points.

```
U*, <type>, ...              =>   NAME, <type>, ...
     ICON, PTR <type>, ...
```

This sort of collapsing reflects the addressability of arbitrary memory locations on a machine. For the /6, this optimization may be performed for any word addressable item, but only in special instances for character items.

```
U&, ...                      =>   ICON, PTR <type>, ...
     NAME, <type>,
```

This optimization is the inverse to the previous transformation.

```
*, <type>, ...                          <<, <type>, ...
   <expression>, ...         =>             <expression>, ...
   ICON, n, ...                             ICON, log2 n, ...
```

This standard transformation requires the constant n be a power of two, as is easily verified by the condition,

```
n >= Ø and n&(n-1) = Ø =>'s a power of 2
```

This transformation also checks for multiplication by one.

```
+/-/*/|, ...
    <expression>, ...
    ICON, Ø, ...
```

Operations with zero are transformed or eliminated. Since these transformations are applied only to integer expressions, one needn't worry about eliminating operations with Ø that might be performed for their side effects, e.g. normalization of a floating point number by adding Ø.Ø.

```
/, <type>, ...                          <expression>, ...
    <expression>, ...          =>
    ICON, 1, ...
```

Division by the constant one is removed.

The previous optimizations are very simple. The remaining optimizations involve tree transformations to combine constants.

```
+, ...
   -, ....                                    +, ...
      <expression>, ...      =>                  <expression>, ...
      ICON, nl, ...                              ICON, n2-nl, ...
   ICON, n2, ...
```

This corresponds to

```
        (el-e2) + e3 => el + (e3-e2)
```

when e2 and e3 are constants.

```
+, PTR char, ...                  PSCONV, PTR char, ...
   PSCONV, PTR char, ...              <word address>, ...
      <word address>, ...  =>     ICON, n2-nl (BYTES), ...
      ICON, nl (BYTES), ...
   ICON, n2 (BYTES), ...
```

This optimization is similar to the previous one, except it deals with collapsing address calculations for character pointers. The optimization corresponds to

```
   ((char *)(el-e2) + e3) => (char *)(el - (e2-e3))
```

For the /6, this optimization and others like it are very powerful. The result of this transformation may be a halving of the code generated for the tree. Note that there is no need to check that the constants involved are byte quantities because the PSCONV node always has a byte offset as its right son, and, since the PSCONV node appears below the addition, one knows the type must be "PTR char." A similar transformation is performed for trees with PACONV nodes.

```
PACONV, PTR char, ...               PACONV, PTR char, ...
    +/-, <type>, ...                     <expression>, ...
         <expression>, ...   =>          ICON, ? (BYTES), ...
         ICON, n1, ...
    ICON, n2, ...
```

This is the first optimization in which the offset-type of a constant must be checked. Since a PACONV node is formed by collapsing a PCONV node into a PLUS node, the operations performed underneath it may be performed with word offsets (e.g. in structures). Hence, when calculating the new ICON node one must coerce byte and word quantities. Without the offset-type attribute added, this would not be possible. A similar transformation is performed for PSCONV nodes.

```
                                     <op>, ...
<op>, ...                                 <expression>, ...
    <op>, ...                             ICON, n1 <op> n2, ...
        <expression>, ...  =>
        ICON, n1, ...
    ICON, n2, ...
```

This transformation may be carried out only for commutative operators.

## 4. Machine Dependent Local Optimizations

While the previous section described optimizations that would normally be carried out for any machine at the intermediate language level (by transformations to expression trees), this section concerns itself with various optimizations performed just before generating code. As such, these optimizations should properly be considered machine dependent.

### 4.1. Switch Statements

The code generated for switch statements attempts to optimize the operation based on the range of case values. Two different types of switch statements are generated: a direct switch through a table of addresses, and a test and branch sequence. It would be simple to add further variations such as hashed switches, looped table lookup, etc.

### 4.2. Parameter Passing

A by-product of the stack design is that a free location is always maintained at the top of the stack. This implies that function calls with a single parameter (of one word) may be performed without a "push" of the stack pointer. This was taken from the Ritchie compiler for the PDP-11 [24].

## 4.3. Structure Assignments and Structure Parameters

Depending on the size of the structure to be moved, either a series of moves is emitted, or a loop is built. Currently, all structures of six words or less are moved without a loop. In certain cases the building of a loop requires three index registers (to use a bwk, bwj, or bwi instruction). When this is necessary, the K register is saved in the bit processor´s V register, and reused for the loop.

## 4.4. Byte Pointer Additions and Subtractions

Checks are performed to determine whether the byte offset involved in the calculation is a constant. In this case, two possibilities arise. If the constant is a multiple of three, the operation may be performed without placing the pointer in a register. Otherwise, the operation may be performed by an addition and a series of byte pointer increments (using bbi or bbj instructions). Should the offset be unknown at compile-time, a "full" calculation must be performed.

## 4.5. PACONV and PSCONV Calculations

These optimizations are similar to those used for byte pointer arithmetic. In the case of PACONV and PSCONV, nodes the word address must always be placed in a register. However, if the offset is a constant, the operation may be performed using an addition and sequence of byte pointer increments.

## 4.6. Special Instructions

The code tables special case certain expressions to allow use of many low cost special purpose /6 instructions. For example, the assignment of the constants -1 and 0 may performed by a single instruction. The placement of constants in a register may often be performed by instructions using an immediate addressing mode, e.g. toi, tna, etc. instructions.

## 5. Statistics

Up to this point, the claims made concerning the compiler's effectiveness have been of a qualitative nature. To substantiate them, a number of statistics have been collected on the performance of the compiler. While it is somewhat difficult to pinpoint weaknesses with simple things such as program size and execution time (the latter being somewhat dependent on the operating system, and the former being inconclusive when comparing widely different architectures), they still merit some thought and explanation.

Figure VI-2 shows relative size, in words, of certain programs found on the /6, PDP-11, and VAX-11/780. The numbers for the VAX come from London and Reiser [18] and represent the state of the VAX compiler early in its development. The figures are fairly misleading. While the compilers are based on the same program, internally they are vastly different. The fairest comparison is the C preprocessor, since it is virtually identical at the source code level across all machines; however, figures for the C preprocessor were not available. Words were chosen for comparison (instead of bytes) since it tends to even out the differences in instruction sets and word sizes. One should note that output for the PDP-11 and VAX-11/780 were produced using a peephole optimizer, while the /6 compiler is sans optimizer. The editors considered are identical for the PDP-11, VAX-11, and Interdata 8/32. The /6 editor, <u>em</u>, is a superset of the <u>ed</u> editor, and as such would be expected to be somewhat larger. The drastic difference in size of the second pass of the /6 C compiler can be attributed to the extra work that must be carried out in handling the irregular instruction set. That is, a large portion of the extra size is due to the code tables. Since the /6 requires many special cases in the code tables, their size grows significantly. It has been estimated that a peephole optimizer, equivalent to that found on the PDP-11 or VAX-11, may result in a 10-20% savings in code size. However, even if this estimate is applied to the figures collected, the text sizes presented for the /6 will still be larger that those for other machines. This is due mainly to the other machines being byte addressable. C is a heavily byte oriented language, word addressable architectures such as the /6 make implementation of C difficult and costly.

Figure VI-3 shows execution times for the compilers. The numbers for the /6 were collected on a swapping system with three other large compilations in progress and a compute-bound artificial intelligence program running. The file _pftn.c_ contains the symbol table management routines for the first pass of the portable compiler. The second table shows execution times for other portions of the portable compiler. Once again the black box approach is poor, since external factors play an important role in the numbers collected. Figure VI-4 shows a more detailed timing of

| Program | System | Size (words) | | | |
|---|---|---|---|---|---|
| | | Text | Data | Bss | Total |
| C, pass1 | PDP-11 | 18368 | 9913 | 8828 | 37109 |
| | VAX-11 | 9380 | 7373 | 5878 | 22631 |
| | Interdata 8/32 | 15152 | 8048 | 6230 | 29430 |
| | Harris /6 | 22096 | 6102 | 10463 | 38661 |
| C, pass2 | PDP-11 | 10624 | 3127 | 2623 | 16374 |
| | VAX-11 | 5852 | 2273 | 1888 | 10013 |
| | Interdata 8/32 | 8913 | 2258 | 1890 | 13061 |
| | Harris /6 | 15620 | 6390 | 4946 | 26962 |
| ed/em | PDP-11 | 5376 | 151 | 2195 | 7722 |
| | VAX-11 | 2888 | 53 | 1139 | 4080 |
| | Interdata 8/32 | 5471 | 1144 | 6735 | |
| | Harris /6 | 8549 | 341 | 2945 | 11835 |
| grep | PDP-11 | 2368 | 204 | 953 | 2525 |
| | VAX-11 | 1216 | 119 | 484 | 1819 |
| | Interdata 8/32 | 2987 | 290 | 484 | 3761 |
| | Harris /6 | 3761 | 267 | 1402 | 5523 |
| ls | PDP-11 | 3552 | 384 | 1928 | 5864 |
| | VAX-11 | 1721 | 285 | 1441 | 3447 |
| | Interdata 8/32 | 3915 | 480 | 1442 | 5837 |
| | Harris /6 | 4713 | 389 | 2588 | 7690 |

Figure VI-2. Representative Program Sizes

selected compilations on the /6. The values were calculated without the benefit of floating point arithmetic, so roundoff errors are significant. The figures for the assembler seem to indicate that an inordinately large part of the compilation process is spent here. The C preprocessor appears to be very efficient, while the first and second passes of the compiler are heavily compute-bound (a result of extensive table searching and a heavy use of recursion). Since the C preprocessor deals almost entirely with characters, it appears the added effort devoted to optimizing code for character manipulations has paid off.

## 6. Further Optimization for the Portable Compiler

It has become very clear that the use of the portable compiler as a production compiler is possible only with some sort of optimizer. The major reason for this statement is that the code generation scheme does not consider possible cross-statement optimizations or local common subexpression eliminations. As a result, while isolated expressions are normally of a high quality, the juxtaposition of code sequences often results in poor code.

| Machine | Execution Time (seconds) | | |
|---|---|---|---|
| | Real | User | Sys |
| PDP-11/70 (Ritchie Compiler) | 86.0 | 43.5 | 11.8 |
| VAX-11/780 (portable compiler) | 82.0 | 64.0 | 10.5 |
| PDP-11/70 (portable compiler for Interdata 8/32) | 153.0 | 114.6 | 16.6 |
| Harris /6 (portable compiler without optimization) | 975.0 | 318.3 | 51.9 |

Figure VI-3a. Execution Times For the Compilation of the File pftn.c (cc -c -O pftn.c)

| Command | Execution Time (seconds) | | |
|---|---|---|---|
| | Real | User | Sys |
| cc -c local.c | 238.0 | 38.1 | 21.9 |
| cc -c optim.c | 497.0 | 99.8 | 26.4 |
| cc -c scan.c | 324.0 | 158.4 | 33.2 |
| cc -c trees.c | 740.0 | 291.6 | 48.3 |
| cc -c xdefs.c | 72.0 | 14.4 | 16.7 |
| cc -c address.c | 176.0 | 41.4 | 19.3 |
| cc -c allo.c | 314.0 | 107.8 | 27.5 |
| cc -c comm2.c | 94.0 | 27.1 | 20.0 |

Figure VI-3b. Execution Times For Selected Compilations By the Harris /6 C Compiler

| Command | Phase | Execution Time | | |
|---|---|---|---|---|
| | | Real | User | Sys |
| | cpp | 169.0 | 15.9 | 35.8 |
| | c0 | 471.0 | 104.4 | 14.5 |
| cc -c cgram.c | c1 | 678.0 | 167.1 | 14.8 |
| | as | 510.0 | 109.8 | 17.6 |
| | (total) | 1828.0 | 397.2 | 82.7 |
| | cpp | 61.0 | 8.5 | 12.9 |
| | c0 | 105.0 | 27.3 | 5.5 |
| cc -c clocal.c | c1 | 186.0 | 47.5 | 5.7 |
| | as | 132.0 | 30.9 | 7.2 |
| | (total) | 484.0 | 114.3 | 31.4 |
| | cpp | 81.0 | 8.0 | 12.5 |
| | c0 | 95.0 | 20.8 | 4.8 |
| cc -c code.c | c1 | 106.0 | 29.5 | 3.6 |
| | as | 159.0 | 33.0 | 6.8 |
| | (total) | 581.0 | 91.5 | 27.8 |
| | cpp | 64.0 | 7.4 | 10.3 |
| | c0 | 80.0 | 15.7 | 4.4 |
| cc -c comm1.c | c1 | 86.0 | 17.7 | 2.6 |
| | as | 104.0 | 19.6 | 5.7 |
| | (total) | 334.0 | 60.6 | 23.0 |
| | cpp | 63.0 | 6.3 | 9.8 |
| | c0 | 52.0 | 8.5 | 4.6 |
| cc -c local.c | c1 | 60.0 | 12.0 | 2.4 |
| | as | 63.0 | 11.2 | 4.9 |
| | (total) | 238.0 | 38.1 | 21.9 |
| | cpp | 74.0 | 8.1 | 11.3 |
| | c0 | 121.0 | 25.4 | 4.6 |
| cc -c optim.c | c1 | 201.0 | 44.1 | 4.8 |
| | as | 101.0 | 22.0 | 5.5 |
| | (total) | 497.0 | 99.8 | 26.4 |
| | cpp | 199.0 | 15.3 | 24.0 |
| | c0 | 206.0 | 77.6 | 8.4 |
| cc -c pftn.c | c1 | 287.0 | 120.1 | 7.1 |
| | as | 283.0 | 165.3 | 12.2 |
| | (total) | 975.0 | 318.3 | 51.9 |
| | cpp | 29.0 | 12.8 | 14.9 |
| | c0 | 47.0 | 38.6 | 5.2 |
| cc -c scan.c | c1 | 124.0 | 56.5 | 5.9 |
| | as | 124.0 | 50.4 | 7.0 |
| | (total) | 324.0 | 158.4 | 33.2 |

Figure VI-4. Detailed Execution Profile For the /6 C Compiler

To enhance the quality of code generated, there are two alternatives. The most appealing is the development of a machine independent global optimizer that could be incorporated somewhere in the first pass, or between the two present passes. For such an optimizer to be implemented, the first pass of the compiler would have to undergo major modifications. The modifications would be necessary to maintain global context for the expression trees constructed during the first pass of the compiler. In the current code generation scheme, each expression is treated as a separate entity. For a global flow analysis to be performed on a program, expression trees would have to be maintained within context, implying a significantly different treatment of the expression tree as a data structure. A second problem with the scheme used in the portable compiler, assuming a global optimizer is to be added, is that the first pass generates certain portions of code. For a flow analysis to take place, an entire "block" of context would have to be formed and analyzed before any code could be emitted. This sort of treatment implies that constructs presently processed in the first pass would have to be handled in a significantly different manner.

The second alternative, and the one presently found in many C compilers, is a separate peephole optimizer that processes the assembly code produced by the compiler. For most applications, this approach appears to be the most viable. As mentioned previously, the portable compiler can produce high quality code for individual expressions. The instances where it breaks down are generally due to a lack of context with which to make decisions. This lack of context normally results in redundant loads and stores being generated. By performing a backwards pass through the assembly code to calculate register usage information, problems such as redundant operations may be easily recognized (the VAX-11/780 peephole optimizer presently works this way). Even without a backwards register pass through the code, an optimizer of this sort may employ a fairly small "window" into the assembly code to locate redundancies of the sort mentioned.

To choose between the alternatives presented, one must consider the applications for which the compiler is to be used, as well as the target machine on which the compiler is to be run. The cost of performing a global flow analysis on a program can be very high, both in compiling speed and in memory overhead. The fact that context must be maintained to perform a global flow analysis normally implies those expression trees involved must be maintained "tied together" in core until the end of a block is reached, at which time the full analysis may be performed. For machines with a small address space this is probably not feasible. Further, the results of performing global analysis, as opposed to a limited local analysis, quite often are not significant. Since C was developed for minicomputers, and presently is found mostly on minicomputers, the problem of a limited address space appears to have been a determining factor in the selection of peephole optimizers. A similar analysis tends to indicate that further optimization for the /6 minicomputer might best be done in a peephole fashion. Previous chapters have alluded to possible candidates for optimization on the /6 (see sections 3, 4, and 5 of chapter V).

# CHAPTER VII

# THE INTERACTION BETWEEN COMPILER, ASSEMBLER, AND LOADER

The C programming environment provided under UNIX/24V is dependent on the facilities provided by the utilities supporting the compiler -- in particular, the assembler and loader. The major impact of the assembler and loader is in the treatment of statically allocated global data structures. This chapter will discuss the interaction between the /6 C compiler and its assembler and loader. The problems encountered in developing a single compiler for multiple assembler/loader combinations will also be discussed.

## 1. An Overview of the Assembler and Loader

The UNIX/24V assembler, *as*, bears some resemblance to the PDP-11 UNIX assembler of the same name. However, the two programs are totally different internally. The assembler for the /6 is totally written in C and uses the parser generator *yacc*, [8], to handle syntax analysis, while the PDP-11 assembler is written in assembly language. The UNIX/24V assembler is two passes, and generates only relocatable output. That is, unlike the PDP-11 assembler, the output produced by as must be processed by the loader, ld, to create an executable program. This departure from the normal convention of producing executable output when possible was due to the addressing structure of the /6. Simply, the necessity to support literal constructions in the assembler would have required a third pass to create executable output. Since the third pass would have simulated the loader´s actions, there was little reason to include this ability in the assembler.

The UNIX/24V assembler syntax is very compatible with the VULCAN assembler. The instruction mnemonics remain the same, and all of the addressing modes may be expressed identically. The major additions to the UNIX/24V assembler, which the compiler uses, are:

1) A uniform handling of numeric constants as operands. The VULCAN assembler had a limited notion of what values could be used as an operand. In particular, negative numbers were not allowed where the operand was intended to be an unsigned (positive) value. This precluded the use of negative numbers as offsets in forming an indexed addressing mode. All values that overflow an operand field are truncated to fit by as; an option on the assembler allows the user to be notified of this action.

2) A nicer handling of externally defined symbols. In the VULCAN assembler all references to external symbols requires the symbol be prefaced by a "$" (to distinguish the symbol from a common symbol). This is impractical for the compiler, as it is impossible to have this sort of information at the time the compiler outputs symbolic names.

3) Temporary labels as introduced by Knuth [14]. Temporary labels were found most useful in simplifying the logic needed to construct loops for structure assignments and passing structures as parameters.

4) A more complete set of storage allocation directives. In particular, the ability to initialize memory locations by bytes.

5) An additional literal construct to handle byte address constants as operands.

As discussed in chapter IV, before the UNIX/24V assembler was written, some of the deficiencies in the VULCAN assembler were so difficult to work with that a post-processing program was required to massage the assembly language output. If one counts the time spent in this program as time spent assembling, the assembly process was more than halved by moving from the VULCAN assembler to as. Further information concerning the assembler may be found in [16].

The UNIX/24V loader was created by partially rewriting the PDP-11 UNIX loader, ld. As such, the semantics of the loading process are nearly identical to that found on the PDP-11. For the most part, the effort involved in moving the loader was in converting byte-oriented portions to be word-oriented. For example, I/O buffering on the PDP-11 was done in a byte-oriented manner which was most inefficient on the /6, so it was converted to be performed in word quantities. The porting of the loader turned out to be very easy; the loader virtually worked on the second try, and required only about two or three days of effort to get entirely working under UNIX/24V. It was originally expected tha the loader would require more time to get working than the assembler (even though the assembler was being written from scratch), but little more than a week of real time was spent on it.

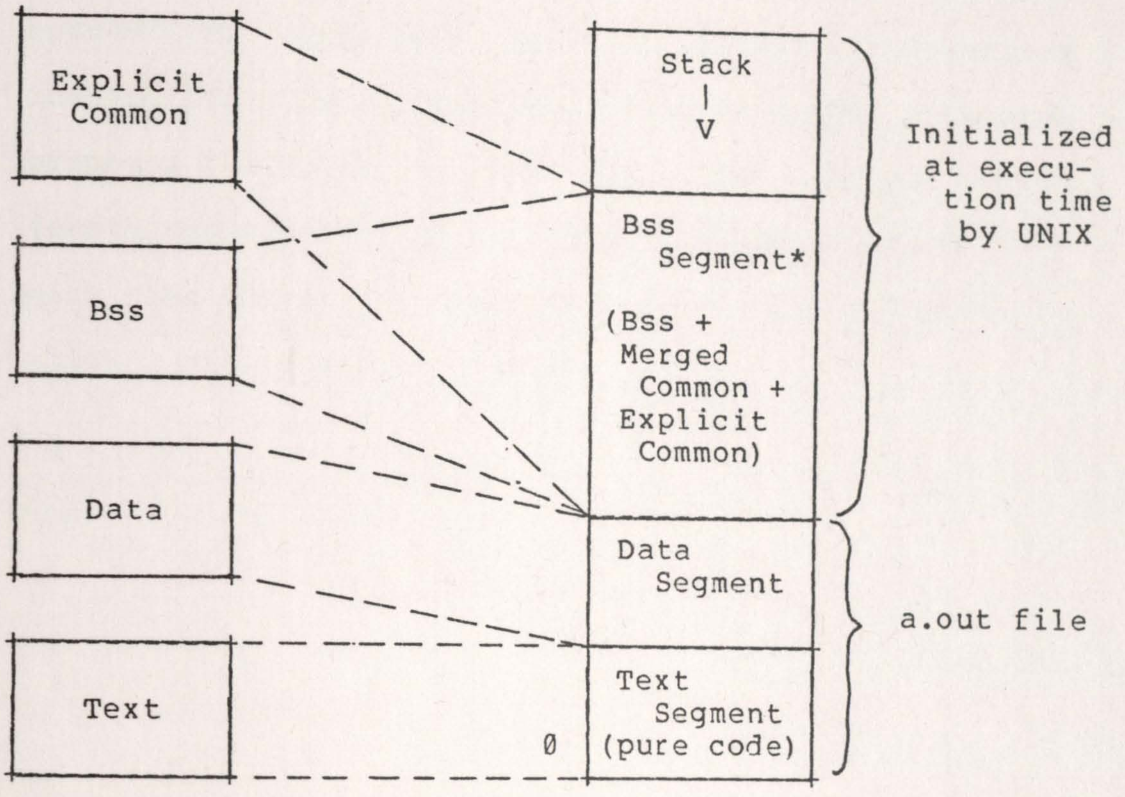
## 2. Incompatabilities With VULCAN

As designed and implemented, there are a few noticeable incompatabilities between the programming environment found on VULCAN and that found under UNIX/24V. The significance of these differences stems from the porting path taken. Since many of the programs moved to UNIX/24V came from the PDP-11, via VULCAN, considerable effort was expended to minimize the amount of work necessary to move programs along this path. The differences noted were due to the VULCAN assembler and loader, and as such, resolving them was not within the scope of this project.

The first problem was that the VULCAN assembler and loader allowed global symbols to be at most six characters long. Under the scheme chosen for C, all symbols defined in a C program have an underscore ("_") prepended to them to avoid name collisions with assembly language routines. As a result, the six character limit really imposed a five character limit on C variables. This applied to all C variables, not just globally defined ones, since the compiler cannot easily distinguish between local and global symbols. Hence, all programs that were originally written for the PDP-11 had to be checked for name conflicts within the first five characters (the PDP-11 assembler and loader handle eight character symbols). In most cases, the fix for conflicts involved using the C preprocessor define statement to map the conflicting names into distinct symbols. It was originally hoped that the program lint, [10], would be helpful in locating the offending symbols, because it has an option to perform checking of this sort. However, lint checks for conflicts only within the first six characters because its application was targeted for the Honeywell 6000 machine.

The second problem dealt with the notion of common storage. VULCAN treats common as a separate segment during assembly and linkage. Hence, all symbols typed common must be identified as such. Further, externally referenced symbols are in a different segment than common, implying that a reference to a symbol must specify if the symbol is common or external. Under UNIX, common is formed at linkage time by merging all external references to symbols of the same size (though it may also be explicitly declared). This allows a reference to an external symbol that may be merged common to be identical to a reference to a symbol which is explicitly common. The latter scheme greatly simplifies the compiler's task of creating symbol references. Figure VII-1 shows program segment layout under UNIX/24V and how common is handled to remain compatible with this scheme.

The result of this second problem is that global declarations in C programs must be carefully matched across files. If a global variable is initialized at compile time, the variable is placed in porg space under VULCAN. To compatibly merge other files with this declaration, all references to the variable in other files must be as an external variable. A globally defined variable which is uninitialized is placed in common, requiring references and declarations in other files to be common also. Almost all programs written for the PDP-11 fail to maintain this convention, allowing the loader to merge common and resolve the incompatibilities in declarations.



* Paged file formats may result in a portion of bss being placed in the data segment. This occurs when rounding the size of the data segment to a page boundary.

Figure VII-1. Program Segment Layout Under UNIX/24V

Since the UNIX/24V assembler and loader maintain conventions compatible with the PDP-11 assembler and loader, the problems mentioned fail to arise in moving programs directly to UNIX/24V. The maximum symbol length supported by _as_ was chosen to remain compatible with the PDP-11 (it could easily have been extended to allow a nine character limit).

# CHAPTER VIII

# DEFICIENCIES AND FEATURES OF THE COMPILER

Since the compiler was moved to UNIX/24V, there has been minimal opportunity to evaluate the programming environment. Little, if any, software development has been undertaken except in the area of the operating system. Nevertheless, some fairly solid opinions have been formed concerning the facilities available, as compared to the other UNIX systems with which the author has had experience.

## 1. Compiling Efficiency

The most notable item in the software development cycle on the /6, is that the compiling process is very slow. This observation is colored somewhat by the inefficiency of the system as a whole. The time spent in compiling C programs is more than doubled under UNIX/24V. Table VI-3 shows statistics collected under the swapping version of the system. Disecting the compilation process has shown that an abnormal percentage of the time is spent assembling programs. The two passes of the compiler and the preprocessor appear to be fairly efficient, though some consideration has been given to improving the compiler´s performance by merging the two passes into a single pass.

Possible reasons for the poor performance of the assembler are directly related to its design. The focus behind the assembler devlopment was to get it working as quickly as possible. This is not to imply that efficiency was totally neglected, merely that it was considered secondary in importance. The assembler´s two passes are driven uniformly by the parser. The first pass involves the normal scanning and parsing of the input file, while the second pass eliminates rescanning the input file by reading tokens from a binary intermediate file (created in the first pass). It appears that the decision to reparse the file in the second pass was costly. The decision was based on the handling of expressions within the assembler. A common technique in assemblers is to handle, as much as possible, expressions in the first pass of the assembler, and backpatching undefined expressions in a subsequent pass. This is possible if the expressions are limited enough to allow a symbol table entry to completely describe all partially defined expressions (alternatively storing this information in the intemediate/output file).

Unfortunately, with the myriad of types, operators, and most importantly, literals allowed in as, such a scheme is not feasible. The reason being that partial expression evaluation, on which this sort of scheme relies, requires that values for items such as literals be known in the first pass. However, since literals are placed at the end of the text segment, this precludes definition of literal and data values until the end of the first pass. Consequently, to define partial values for expressions in the first pass would require a tree representation of the expression, most likely in the symbol table. The cost of such a scheme was believed higher than making values defined only in the second pass via a second parsing.

In profiling the assembler, an inordinately large portion of its execution time appears to be involved in handling character input. The input routine is fairly complex because the assembler suports the inclusion of text files. This feature has not been heavily used, and its removal may result in a speed improvement.

## 2. User Feedback

One of the most pleasing attributes of the portable compiler is the comprehensive error checking performed. Unlike previous C compilers encountered, the portable compiler produces excellent diagnostics in the area of type incompatibilities. Two of the most nonportable constructs are type punning (an implicit type conversion carried out by an assignment) and illegal pointer uses (e.g. using pointers to point to members of a structure other than their own). The portable compiler gives warnings in each of these cases. It is interesting to note that the Ritchie compiler, which has been firmly entrenched on the PDP-11 for nearly as long as UNIX has been, does not produce dignostics for equivalent constructs.

The "noise" created by the portable compiler has proved instrumental in aiding in the porting of programs. In many cases, the diagnostics produced by the C compiler led to the detection of nonportable constructs. The most notable example of this was the Release 7 shell. The original shell compiled under the Ritchie compiler without any diagnostics, while the portable compiler produced hundreds of lines of messages. A large number of these messages were traced back to nonportable constructs in the implementation of the shell.

## 3. Expandability

The portable compiler has been found very easy to work with in integrating run-time and compile-time facilities. The compiler currently supports the standard UNIX run-time profiling, as well as an entirely new feature designed to aid in debugging programs on a system without a debugger.

Run-time procedure tracing has been added to allow a user to specify that procedure invokations should be communicated to the user. This facility allows for run-time selection of which functions to trace. The output produced by the tracing shows the symbolic name of the function, as well as the parameters passed to the function. The run-time selection of which functions to trace is communicated via a shell variable stored in the environment. Once a file has been compiled with a tracing option, a user may manipulate the tracing of functions contained in the file by setting the shell variable TRACE according to the syntax

```
TRACE=[on|off]|[all]|[all-]f1,f2,...
```

This allows a user to selectively trace a class of functions, all but a class of functions, etc. The facility has proven useful, particularly in the absence of a debugger. Approximately 15 lines of code were added to the compiler to support tracing; the majority of the work went into designing and building the run-time support.

The one facility the compiler/assembler system does not support is the simulation of floating point arithmetic. This is due to the architecture of the /6. Most machines without an optional floating point unit treat floating point instructions as illegal instructions. Under UNIX, this allows a program to trap the instructions and interpret them. Unfortunately, a /6 without an SAU treats floating point instructions as no-operations. Thus, if floating point simulation is to be performed some other approach must be taken. The usual alternatives are: have a non-floating point compiler which generates different code than produced for floating-point machines (presumably function call to library routines), or have the assembler map the floating point instructions into some set of illegal instructions which UNIX may catch and return control to a simulation package. The latter approach appears to be the most appealing, since it is expected that changing the compiler is a more difficult proposition than altering the assembler. Further, the implementation in the assembler of the latter scheme can be done trivially by inserting a special trap before each floating-point instruction in the assembler code tables. While the methods needed to handle the lack of an SAU have been explored, nothing has been implemented, since the effort required to write a floating-point simulation package for the /6 is quite large.

## 4. Current Status

The programming environment developed for UNIX/24V was created on a /6 cpu that lacked a floating point unit. As noted above, the simulation of floating point arithmetic is not currently supported. Consequently, little, if any, real testing has been performed on the floating point facilities provided by C. The compiler generates floating point code; however, a number of bugs are certain to be present. The initial validation of the compiler on the PDP-11/45 included numerous floating point test cases, but since the compiler has been moved to UNIX/24V no further testing has taken place.

The assembler was developed under similar circumstances and lacks only one routine to be able to support floating point constants (the .float directive, see [16]). All floating point instructions are assembled correctly, and the syntax of floating point constants is supported. The routine that is missing, _realize_(), has the following calling sequence:

```
long
realize(whole, frac, exp)
        long whole, frac;
        int exp;
```

The routine is passed the three portions of a floating point number, the mantissa (whole.frac) and the exponent (exp), and is expected to construct the binary representation for the number, returning it as its result (a _double_ type is the same size as a _long_ in C on the /6). This routine was left unimplemented because of the lack of an SAU on the test machine; its implementation is straightforward.

Other than the floating point deficiencies noted above, the compiler and its supporting utilities have undergone extensive testing. All features of the C programming language described in [13] are supported and tested. In addition, the compiler supports many features added to Release 7 of the C programming language. These features include structure assignments, passing structures as parameters and returning them from functions, enumerated types (they are packed as tightly as possible), extensive bit field arithmetic and typing (i.e. a bit field may be defined of "enumerated type", for instance), and the initialization of structures containing bit fields. The portable compiler, in general, supports a dialect of C that is more advanced than that provided by the Ritchie compiler. All syntactic constructs deemed legal by the portable compiler are supported on the /6, except for fields of character type.

The supporting utilities are likewise fairly advanced. The assembler, aside from the floating point caveat noted previously, has been employed to write the machine language assists for UNIX/24V, and the system call library and portions of the standard I/O library. Besides a flexible syntax, the assembler supports a number of "frills" to support the future addition of a macro preprocessor. The assembler supports constructs, similar to those of the C preprocessor, to allow the definition of the current source line number and input file. Should a macro preprocessor be needed, these features will allow consistent diagnostics to be generated by the assembler. The link editor, <u>ld</u>, supports the loading of normal assembler output files (type 407), as well as libraries (handled by the <u>ar</u> program, and random libraries (managed by <u>ranlib</u>). The loader can create files in a variety of formats (see Figure VIII-1). All other program development utilities moved to UNIX/24V (make, ar, ranlib, size, nm, syms, etc.) function exactly (or in a logically equivalent manner) as specified in the UNIX Programmers Manual, Release 7, [29]. Appendix A contains a list of all utilities available under UNIX/24V.

| Magic # | Description | ld Options |
|---|---|---|
| 405 | Paged overlay. Normal UNIX overlay, except text and data have been rounded to page (1K words) boundaries -- as in 413. Not supported by UNIX/24V. | -O -z |
| 407 | Relocatable loader input. Produced by assembler -- not executable, except by early versions of UNIX/24V. Header sizes are exact sizes of text, data, and bss. | |
| 410 | Paged reentrant. Text segment is write protected and shared. Text and data are rounded to a page boundary after bss is merged into data segment. Maximum program size is indicated in a.out header. | -n -z |
| 412 | No stack. Program size specified in a.out header does not include stack allocation. Unsupported by UNIX/24V. | -K0 |
| 413 | Paged executable. Text segment is writable. Size of text plus data segment is rounded to a page boundary after bss has been merged into data segment. Maximum program size is indicated in a.out header. | -z |
| 177545 | Archive or random library. Loader input. | |

Figure VIII-1. Loader File Formats for UNIX/24V

# CHAPTER IX

# C ON A WORD-ADDRESSABLE MACHINE

The /6 architecture has had a noticeable impact on the C programming environment. That is, the implications of a word addressable machine have had a tendency to filter into the programs written on it. The byte pointer representation used on the /6 implies that conversions between word and byte pointers may result in a loss of information. Hence, the common practice, at least on the PDP-11, of using a character pointer as the "common denominator of all types" is not possible on the /6. In addition, a lack of attention to parameter type compatabilities across function calls can cause problems. A more detailed discussion of these problems follows.

## 1. Handling Data Types

With multiple representations existing for pointers and with one of these representations, the byte pointer, being vastly different from normal numeric representations, a number of basic rules must be established to understand the consequences of conversion operations. In addition, in C, numerous storage conversion operations take place, and their semantics must be understood.

All pointer conversion rules designed for the /6 are based on two underlying assumptions:

1) The value stored in a pointer to a word oriented item is always assumed to contain the number of words the item is offset from zero.

2) The value stored in a byte pointer contains the number of bytes the item is offset from byte zero.

If one uses these two basic assumptions, conversion operations between pointers may be "consistently" defined. Table IX-1 summarizes the rules for conversion between pointers, as well as the storage conversion rules involving pointers. One should note that while conversions are defined in all instances, some conversions may result in a loss of information. The loss of information across certain conversions is unavoidable in the case of the /6. Despite this information problem, the design of conversion rules must be as consistent as possible. In a byte addressable architecture, where the representations for pointers and numeric items is identical, these rules are usually trivial (i.e. they do nothing). For the /6 the rules defined appear to minimize inconsistencies while remaining logical.

| From | To | Rule |
|---|---|---|
| char * | int<br>unsigned<br>short<br>unsigned short<br>long<br>unsigned long | Convert format to give byte number of memory location pointed to, e.g. 3 * (number of words) + number of bytes. |
| char * | int *<br>struct *<br>long *<br>char **<br>etc. | Return address of word pointed to, e.g. mask with 17777777 (octal). |
| int *<br>struct *<br>long *<br>char **<br>etc. | char * | Return address of first byte in word pointed to, e.g. or with 20000000 (octal). |
| int *<br>struct *<br>long *<br>char **<br>etc. | short<br>unsigned<br>short<br>etc. | Copy. |
| int *<br>struct *<br>long *<br>char **<br>etc. | long<br>unsigned long | Treat pointer as unsigned integer, e.g. zero fill from 24-bit number to 47-bit number. |

Figure IX-1. C Pointer Converion Rules for the /6

The storage conversion rules are fairly standard. The /6 numeric representations and instruction set allows these conversions to be one or two instructions normally, and fail to require the extra thought necessary in the case of pointer conversions. Figure IX-2 summarizes these conversion rules.

One notable exception to the scheme presented above is the handling of a NULL pointer. The C language definition states that "it is guaranteed that assignment of the constant Ø to a pointer will produce a null pointer distinguishable from a pointer to any object" [13]. The natural conversion operations, outlined above, imply that an assignment of this sort fails to follow this convention. In addition, rules found in most other C compilers, for the storage conversion of a character pointer to an integer, specify that a straight copy is carried out. This also conflicts with the rules for conversion set forth for the /6. Consequently, to maintain compatability with other C compilers, the /6 compiler handles the null pointer as a special case. The decision to make this exception to the general conversion rules resulted from experiences in porting various user programs (in particular the Bourne shell). A result of this special casing is that certain code sequences generated for

| From | To | Rule |
|---|---|---|
| char * | int<br>unsigned<br>short<br>unsigned short<br>long<br>unsigned long | Return the byte number of the memory location pointed to, e.g. 3 * (number of words) + (number of bytes). Result is treated as an unsigned value when converting to longs. |
| long<br>unsigned long | int *<br>struct *<br>char **<br>etc. | Truncate 47-bit value to 24-bit value, then copy. |
| long<br>unsigned long<br>int<br>short<br>unsigned<br>unsigned short | char | Truncate to 8-bit value, then copy. |
| int<br>unsigned<br>int *<br>struct *<br>etc. | long<br>unsigned long | Lengthen 24-bit value to a 47-bit value by sign extension or zero filling. |
| int<br>short | float<br>double | Machine defined - fax instruction. |
| long<br>unsigned long | float<br>double | Machine defined - fno instruction. |
| unsigned<br>unsigned short | float<br>double | Zero fill to a 47-bit number, then treat as a above. |
| float<br>double | int<br>unsigned<br>char<br>unsigned char | Machine defined by fxa instruction. Characters are truncated following conversion. |
| float<br>double | long<br>unsigned long | Convert to 24-bit number by fxa instruction, then sign extend/zero fill to 47-bit value. |

Figure IX-2. C Storage Conversion Rules for the /6

storage conversions must perform a check for the null pointer. In addition, the handling of the null pointer raises the complexity of the code used for handling constants within the compiler. It appears that the language definition is lacking here. An improvement might be the introduction of an explicit null pointer to the language definition, as found in Pascal, ADA, etc.

## 2. The Impact of the /6 on Programming in C

As mentioned previously, the C compiler for the /6 employs two different representations for pointers. The byte address format is employed for any pointer to a byte oriented object. All non-byte storage items (longs, fields, enumerations, etc.) are manipulated with a word address format. As one might expect, the impact on C goes beyond the compiler's difficulty to generate code. Because the conversion between these two formats is not a one-to-one mapping there are cases where statements in C may result in information "slipping between the cracks". In particular, consider the following statements:

```
char *p, foo[20];
int *q;
...
p = &foo;
p++;
q = (int *)p;
p = (char *)q;
```

While this code probably won't occur in practice, one should notice that the assignment to q results in a "rounding" of the address to a word boundary. If one considers the values that are involved here, the following will be seen (assume the array foo is placed at location octal 1000):

| Statement | Value for p | Value for q |
|---|---|---|
| p = &foo; | 020001000 | <undefined> |
| p++; | 040001000 | <undefined> |
| q = (int *)p; | 040001000 | 00001000 |
| p = (char *)q; | 020001000 | 00001000 |

The information transfer from p to q, and back to p again, results in the byte position being lost. This is to be expected, since it is not possible to maintain this information across the assignment statements. Occurences of this sort can not occur on the PDP-11, since byte and word pointers are treated as objects having an identical format (i.e. all conversion operations have no effect on the internal value of a pointer).

A more serious problem with the differences in data type representations is the handling of parameters. While word pointers and integer items are identically formatted, the shape of a byte pointer causes the interchange of a byte pointer and an integer to have drastic effects on operations. While conversions may be applied when a type cast takes place in-line, parameter passing may hide the need for a conversion. Since the C language definition specifically avoids checking actual parameters against formals (in fact the syntax of the language doesn´t always supply enough information to allow this to be done), the programmer must be held at fault when something of this sort occurs. Once again, incompatabilities of this type have no impact on a byte addressable machine, since the representation for pointers will be equivalent.

A final problem caused by the data type representations for the /6 involves longs. This problem is independent of a word addressable architecture; it is particular to the /6. There is a large temptation on many machines to treat longs as something other than an indivisible storage location. That is, a long may be used to allocate storage, but in some instances operations are performed on subcomponents of the long. Since the /6 representation of a long requires that the sign bit in the low word be zero, the manipulation of a long as two separate words may result in inconsistent results. Admittedly it is poor programming practice to use a trick such as this, but, when porting programs, one must be aware of the problem.

## 3. The C Programming Environment

From a users´ standpoint, programming in C on the /6 is little different from any other machine to which C has been moved. As noted above, a user must be aware of certain machine problems that preclude the "free" programming style exemplified on machines such as the PDP-11 and VAX-11. The deficiencies in the /6 in fact tend to improve the portability of C code written on the /6. Since a user must follow the typing conventions in C more closely than on most other machines, programs written on the /6 tend to move quite easily. A user must be careful to consider the consequences of pointer conversions, match parameter types in function calls, and treat data structures at "face value". When all the pitfalls of byte pointers are treated carefully, the result is a program that is very readable, and most importantly, very portable.

For efficiency, a user programming in C on the /6 will usually steer clear of unwarranted use of character variables. Since the cost of accessing arbitrarily aligned bytes via an indexing operation is quite high, the use of pointers is important. That is, when deciding whether to implement an algorithm with indices or pointers, the use of pointers is recommended, because there is a large cost involved in forming byte addresses by adding an index to a byte pointer. Thus, the `.´ operator for structures tends to be used less than the `->´ operator, and array indexing (`[]´) tends to be used less than straight pointer manipulations. As an example of the relative cost of indexing and pointer manipulations, consider a simple loop to step through a character array and initialize each entry to zero.

```
char foo[20], *p;
int i;
...
for (i = 0; i < 20; i++)
        foo[i] = 0;
for (p = foo; p < &foo[20]; p++)
        *p = 0;
```

The code for each loop is displayed below, side by side. Note the added cost involved in using array indexing versus pointer manipulations.

```
      Indexing
      --------
      tzm  !_i
L0:
      tma  !_i
      cma  $20
      bon  L1
      tmi  !_i
      tma  !_p
      tze
      lld  2
      rla  2
      aei
      myo  3
      aia
      dvo  3
      aoe  1
      lla  2
      rld  2
      tai
      toa  0
      rbm  !0
      aum  !_i
      buc  L0
L1:
```

```
      Pointers
      --------
      tma  #_foo,0
      tam  !_p
L0:
      tma  !_p
      tme  #_foo,20
      kea
      bon  L1
      tma  !_p
      toa  0
      rbm  !0
      tmj  !_p
      bbj  .+1
      buc  L0
L1:
```

Similar comparisons may be made for other pointer/index related operations with equivalent results.

# CHAPTER X

# CONCLUSIONS

The previous chapters of this document have detailed work carried out in establishing a C programming environment under UNIX/24V. Most of these discussions have been concerned with the target machine, the Harris /6. However, the work carried out has served to illuminate several issues that are pertinent to a large class of machines and which need further study. These issues will be tied together with a summary of some of the important points brought out in earlier chapters.

## 1. The Portable C Compiler

The portable C compiler has proven to be an excellent vehicle whereby a C compiler may be effectively bootstrapped onto a new target machine. The compiler is clearly equipped for machines with a byte addressable architecture, while its adaptation to word addressable architectures is less obvious. The major reason for problems encountered in moving the compiler to a word addressable architecture is the necessity to maintain two different types of address offsets. The portable C compiler does not directly support such a notion, adding a great deal of complexity to the handling of address arithmetic. The underlying assumption in the design of the compiler is that all address arithmetic will be performed in bytes, with conversion to words being performed at the last possible instance, or that all addresses will be maintained in words. Handling all addresses in bytes is impractical if efficient code is to be generated, while the consequence of using only word addresses is that packing of aggregate structures may not be performed -- very costly on machines with a large word size. A parallel /6 C compiler project, [31], chose the latter route, packing one character per 24-bit word. They reported that packing one character per word simplified code generation, but proved costly when performing input/output (packing and unpacking of data structures was required). The Wisconsin C compiler packs character arrays, but does not pack structures. This compiler is structured completely different from the portable compiler.

The question of how to handle multiple offset types has been addressed in this /6 C compiler, within the framework of the normal code generation scheme of the portable compiler. The resultant implementation has proven satisfactory in that it generates correct code, and of a decent quality, but has been unsatisfactory with respect to compilation efficiency. The first pass algorithms used to build expression trees can be very slow when numerous passes must be made over the trees to investigate possible byte pointer problems. It appears that a closer integration of the machine dependent byte/word offset calculation algorithms and the machine independent tree building algorithms would result in a more efficient compiler. The reason this was not immediately done was to maintain the structure of the compiler during development stages. The alterations required to carry out such a plan are not straightforward, and the greatest need during the development was a working compiler. Should the compiler be considered for further use closer study of its internal structure is clearly warranted.

The other significant problem encountered in adapting the portable compiler to the /6 was in mapping the compiler's abstract machine model to the target machine. The notion of an abstract machine to which a target machine may be mapped, is quite old and has undergone many studies; as yet no definitive model for a universal abstract machine has been constructed. The earlier work on the Janus abstract machines [5], LSD programming language [4], BCPL programming language [22], IBM S/360 FORTRAN(G) compiler [7], Pascal P-code [20] and EM-1 [28] abstract machines, and the original portable C compiler [27] have been enlightening, but by no means conclusive. The abstract machine defined for S. C. Johnson's portable C compiler borrows a number of ideas from Snyder's portable C compiler, [27], and appears to be suitable for most all machines presently being designed. The machine model assumes collections of homogeneous general purpose registers, a stack of some sort (either in software or hardware), and a uniform addressing scheme of memory cells. It tolerates some deviation from a purely orthogonal architecture through the machine dependent register allocation scheme and flexibility in constructing the code generation tables. Unfortunately, it fails to handle severe deviations from this model, as found in the /6. There appears to be little reason for the compiler to support a wider range of architectures. If a designer can understand enough of the internal workings of the compiler, the modifications necessary to tailor the abstract model to a given architecture are not difficult. Greater generalization of the abstract machine model (for instance to support more than two register classes) would result in a degradation in the performance of the compiler at little benefit. Thus, it seems fair to say the machine model is satisfactory for most implementations. That this model has been so succesful implies that its properties should be closely studied before future attempts at new and improved abstract machine models are made.

## 2. The C Language As a Portable Implementation Language

One may consider the C language on its merits as an implementation tool for portable software systems. In particular one may ask how C stacks up against other languages normally considered in implementing large software systems. In short, this author believes C is one of the best, if not the best, language for implementing large software systems that are to be used on many different machines. While this opinion is heavily colored by a lengthy exposure to the language, the statement is not made without some justification.

There appear to be a number of qualities required of a language to be successful as an implementation tool. The first is that it should support the basic constructs to program in a structured manner. These constructs include those used for control flow and the abstraction of data types. Secondly, the programmer shouldn´t be overly burdened with "syntactic sugar". This fault, common to many of the programming languages currently in vogue, tends to stifle programmer creativity. Third, recursion should be available. The notion of recursion is one of the most powerful programming concepts available. Its use tends to enhance the correspondence between "natural" algorithms and their implementation. Finally, pointer data types must be supported. Similar to recursion, pointers are usually the natural tool to use in implementing algorithms; their presence tends to allow implementation to correspond closely to a paper algorithm. Needless to say, the C programming language supports all these notions, as do many other programming languages. However, C is virtually the only programming language to support address arithmetic in a comprehensive form that meshes quite naturally with the remainder of the language. C has its detractions, the most common complaint is its weak type checking. In truth, saying C performs weak type checking is fairly charitable. There are many good reasons for having strong type checking in a language, but how to enforce a rigid type checking environment, when combined with the sort of extensive address arithmetic possible in C, is not currently well understood. The lack of type checking performed in C allows faster compilation, and promotes a "robust" environment in which arbitrary memory cells may be used. Some of the bad points of weak type checking are consider later. Overall, it appears C is an excellent tool for implementing software. In a very short period of time C has distinguished itself through its use in a large number of software projects (operating systems, compilers, graphics packages, text processing systems, etc.)

The question of portability in a language can, for the most part, be separated from an analysis of its usefullness as an implementation tool. In considering why C has been so successful as an implementation language, one invariably recognizes the rich set of mechanisms offered a programmer. However, it is also this rich collection of mechanisms which can inhibit portability of a program. Certain high level languages promote portability through restrictions on the operations which may be performed on a data structure, while others attempt to supply an abstract machine for the user. The former technique (e.g. Pascal) tends to be unduly costly for the programmer. To perform a "natural" operation within the structure of the language one must resort to subterfuge (i.e. use an assembly language assist), or work around it (for instance, use indices instead of pointers). The latter approach to portability (APL for instance) requires a simulated environment be moved with each new implementation of the language, so penalizes the language´s implementer. C, on the other hand, is easy to implement and program in. It allows a user to directly employ the power of the target machine. This ability to "touch" the bare machine also allows the creation of totally nonportable constructs.

In considering C as a portable language, a number of specific reasons have been recognized. The ability to abstract data structures avoids the simulation of a natural aggregation. For instance, a need for a collection of heterogeneous items needn´t be simulated within an array. Because the constructs needed for the representation of a natural data structure are present in the language, moving a program from one machine to another may depend on the atomic operations expected of the language.

Secondly, while C supports extensive address calculation features, problems involved in word size differences, data type incompatabilities, etc. are, for the most part, eliminated by natural operator definitions and built-in primitives. In particular, the notions of pointer addition and subtraction have the possibility of being completely nonportable. However, by consistently defining their semantics, C avoids requiring a programmer to know the width of aggregate structures when working with pointers to such objects. Further, when constants of this sort are necessary, the sizeof operator is very useful in maintaining a portable program.

On the negative side, C's notoriously weak type checking allows highly nonportable constructs to be created without any noise from the compiler. London and Reiser [18] have listed four problems they encountered during their experiences in moving UNIX and C to the VAX-11/78Ø. Two of the four suggestions they make are directly related to the weak type checking performed in C.

1) The actual arguments in a procedure call should be type checked against the procedure declaration, and a "dummy" declaration which specifies types be permitted, even if the called procedure is not actually declared in the same compilation.

2) The `->' operator should be checked to insure that the structure element on the right is a member of a structure to which a pointer on the left may point.

3) A structure element should be declarable with any name, as long as the name is unique within the immediately surrounding structure. (The current requirement that a structure element must uniquely correspond to an offset from the beginning of the structure, across all structures in a compilation, creates naming problems, and frequently leads to errors of the type noted in item 2 above.)

4) The issue of alignment to an even-byte (or other)

boundary should be brought into the open, so that arbitrary data structures can be accurately described.

Of the suggestions noted above, the problem addressed in the first was found to be the most difficult to deal with in experiences with the /6. The second item is a by-product of the Ritchie C compiler, the portable compiler produces diagnostics for constructs of this sort. The third is definitely a worthwhile suggestion, and warrants further consideration. The last item suggests a view of aggregate data structures similar to that found in BLISS, [33], and in this author´s opinion diverges from one of the nicer facilities of C. The issue of alignment normally comes up only when trying to model machine dependent data structures (i.e. byte/word/bit layout of a particular structure). As such, the the question of representing a structure in a portable fashion is a moot point. The notion of an aggregate structure in C is to group together items in a single logical unit. If specific layout is required, bit fields, and the like, may be employed to construct a structure of arbitrary shape. Thus, the latter suggestion appears to introduce unnecessary complexity into C.

In summary, the C programming language seems to be a successful tool for implementing portable software systems because it has a rich set of primitives which allow a programmer to use the full power of an underlying architecture. It is also this ability to get at the basic machine which allows introduction of nonportable constructs. Consequently, unlike many other programming languages, portability in C is easily possible, but mostly up to the programmer. It is interesting to note that the consideration of portability is subject to consideration, most other languages force "portability" on a programmer by techniques of the sort noted previously in this section.

## 3. Portability Between Widely Different Architectures

The question of portability, in general, is currently not fully understood. Most people can recognize a program, language construct, or the like, as being nonportable, but few can pinpoint exactly what makes a program or programming language "portable". Further, when considering portability between machines specifics of the particular architectures invariably enter into any consensus formed. The basic variants of word size, data types, and addressability play a major part in forming conclusions, but are, by definition, of little consequence when considering portability between machines with similar architectures (for instance a

PDP-11 and VAX-11). Thus, if one is to formulate opinions about portability it seems appropriate to consider a worst case phenomenon. Since the /6 architecture is so much different from the PDP-11 architecture, it is worthwhile to consider the experiences garnered from this research (the porting project originated on a PDP-11, and had the /6 as its target machine).

Word size, in and of itself, posed few problems. Rather, the specific size of a word on the /6 introduced incompatibilities. A 24-bit word, with an 8-bit byte, invariably introduces the number 3 into many calculations; while programs developed for the PDP-11, where a 16-bit word and 8-bit byte are used, introduce the constant 2. Worse, since 2 is a power of 2, division by 2 was often removed in favor of a right shift. A portable construct of the form

```
(sizeof int/sizeof char)
```

quickly became standard in programs developed on the /6. If the /6′s word size had been a power of 2, further problems might have been eliminated, or more easily dealt with. A machine with a word size which is a power of 2 offers many optimizations of the sort mentioned. Constructing a portable expression involving a constant, as shown above, will be the rule of thumb only if the compiler will perform the optimization from a division to a shift. For a signed number, an optimization is not possible (consider -1 divided by 2 and -1 right shifted 1 bit). Thus, at least in C, it seems worthwhile to recommend expressions be created as above, and that compilers be aware of their existence as common practice and optimize accordingly (this requires the expression be unsigned in type).

Data type incompatibilities cause problems mostly when a programmer assumes no side effects will be created when mixing them. In a language such as Pascal mixing types is illegal, or well defined. C, on the other hand, has been developed on a machine where mixing types normally has no effect on the value of an object. Consequently, many programs misuse this property of the PDP-11 architecture. The recent introduction of "type casts" to C was of major importance in handling the multiple data type formats of the /6. It appears portability between machines with incompatible data type formats must be handled either implicitly by a strongly typed language, or with much foresight, and constructs like type casts, in a weakly typed language. The straight-line interpolation of type conversions, in a weakly typed language, is simple when compared to the nightmares introduced by parameter passing. Handling parameter incompatibilities is by far the most difficult problem to manage in porting programs between architectures with multiple data type formats; at the very least the language support must be present to handle conversions.

A final problem is the method by which a target machine addresses main memory. Most modern architectures support the notion of byte addressability, and some even allow bit strings to be directly addressable. When considering the problems introduced by movement from a machine which is byte addressable to a machine which is not, the first thing that comes to mind is how to handle the inevitable incompatability between pointers. Should the language being used not allow mixing of types of this sort, the problem is nonexistant. However, should it be possible, the architecture has a major impact on the portability of a program. As discussed in chapter IX, there is bound to be an information loss when converting between formats. This problem appears, as much as anything, to motivate a need for type checking, for at least some cases of pointer manipulations. Strong type checking shouldn't be required, but at the very least a programmer must be notified of an irregularity.

## REFERENCES

[1] Aho, A. V., and S. C. Johnson, "Optimal Code Generation for Expression Trees"; JACM, 23(3); pp. 488-5Ø1; 1975. Also in Proceedings ACM Symposium on Theory of Computing; pp. 2Ø7-217; 1975.

[2] Aho, A. V., S. C. Johnson, and J. D. Ullman, "Code Generation for Machines with Multiregister Operations"; Proceedings 4th ACM Symposium on Priciples of Programming Languages; pp. 21-28; Jan. 1977.

[3] Bruno, J., and R. Sethi, "Code Generation for a One-Register Machine"; JACM 23(3); pp. 5Ø2-51Ø; July, 1976.

[4] Calderbank, V. J., and M. Calderbank, "LSD Manual"; CLM-PDN 9/71; Culham Laboratory UKAEA; Abington, Berkshire; 1971.

[5] Coleman, S. S., P. C. Poole, W. M. Waite, "The Mobile Programming System: Janus"; National Technical Information Center PB22Ø322; U. S. Dept. of Commerce; Springfield, Va.; 1973.

[6] Harris Corporation, "Reference Manual - Slash 6 Digital Computer"; September, 1976.

[7] IBM FORTRAN IV (G) COMPILER, Program Logic Manual; 1967.

[8] Johnson, S. C., "YACC - Yet Another Compiler Compiler"; Bell Laboratories Computer Science Techinical Report No. 32; July, 1975.

[9] Johnson, S. C., "A Portable Compiler: Theory and Practice"; Proceedings 5th ACM Symposium on Principles of Programming Languages; pp. 97-1Ø4; January, 1978.

[1Ø] Johnson, S. C., "Lint, A C Program Checker"; Bell Laboratories Computer Science Technical Report No. 65; 1978.

[11] Johnson, S. C., "A Tour Through the Portable C Compiler"; UNIX Programmer´s Manual, Vol. 2 -

Supplementary Documents, Seventh Edition; January. 10, 1979.

[12] Johnson, S. C., and D. M. Ritchie, "Portability of C Programs and the UNIX System"; Bell System Technical Journal, Vol. 57, No. 6, Part 2; pg. 2021-2048; July-August, 1978.

[13] Kernighan, B. W., and D. M. Ritchie, The C Programming Language; Prentice-Hall, Englewood Cliffs, New Jersey; 1978.

[14] Knuth, D. E., The Art of Computer Programming - Volume 1; Addison-Wesley; Reading, Mass.; 1975.

[15] Leffler, S. J., "A Detailed Tour Through the /6 Portable C Compiler"; Case Western Reserve University, internal document; June 1980.

[16] Leffler, S. J., "UNIX/24V Assembler Reference Manual"; Case Western Reserve University, internal document; 1980.

[17] Lesk, M. E., S. C. Johnson, and D. M. Ritchie, "The C Language Calling Sequence"; Bell Laboratories internal memorandum; 1977.

[18] London, T. B., and J. F. Reiser, "A UNIX Operating System for the DEC VAX-11/780 Computer"; Bell Laboratories Technical Memorandum 78-1353-4; July 7, 1978.

[19] Miller, R., "UNIX - A Portable Operating System?"; Australian Universities Computing Science Seminar; February, 1978.

[20] Nori, K. V., U. Ammann, K. Jensen, and H. Nageli, The Pascal (P) Compiler Implementation Notes; Institut fur Informatik, Eidgenossische Technische; Hochschule, Zurich; 1975.

[21] Poole, P. C., "Hierarchical Abstract Machines"; Proceedings Culham Symposium on Software Engineering; April, 1971.

[22] Richards, M., "The Portability of the BCPL Compiler"; Software Practice and Experience; Vol. 1; pp. 135-146; 1971.

[23] Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C Programming Language"; BSTJ,

57(6), Part 2; pg. 1991-2020; July-August, 1978.

[24] Ritchie, D. M., "A Tour through the UNIX C Compiler"; UNIX Programmer´s Manual, Vol. 2 - Supplementary Documents, Seventh Edition; Jan. 10, 1979.

[25] Sethi, R., and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expression Trees"; JACM 17(4); pp. 715-728; October, 1970. Reprinted as pp 229-247 in Compiler Techniques; ed. B. W. Pollack; Auerbach; Princeton, New Jersey; 1972.

[26] Shannon, W. A., "A Demand Paged UNIX Operating System for the Harris /6 Computer"; Case Western Reserve University, forthcoming Master´s Thesis; Cleveland, Ohio.

[27] Snyder, A., "A Portable Compiler for the Language C"; Master´s Thesis; M.I.T, Cambridge, Mass.; 1974.

[28] Tannenbaum, A. S., "Implications of Structured Programming for Machine Architecture"; CACM, 21(3); pp. 237-246; March, 1978.

[29] UNIX Programmer´s Manual, Vol. 1, Seventh Edition; January, 1979.

[30] Weber, L. B., "A Machine Independent Pascal Compiler"; Master´s Thesis; University of Colorado, Boulder; 1973.

[31] Williams, M.; personal communications; Bell Laboratories, Murray Hill, N.J.

[32] Wulf, W., R. K. Johnson, C. B. Weinstock, S. O. Hobbs, and C. M. Geschke, The Design of An Optimizing Compiler; Elsevier Press; 1975.

[33] Wulf, W., D. B. Russel, and A. N. Habermann, "BLISS: A Language for Systems Programming"; CACM 1(12); pp. 780-790; December, 1971.

# APPENDIX A

## UNIX/24V UTILITIES

This appendix lists the user utility programs that have been ported to UNIX/24V. Most all programs were taken from the PDP-11 Release 7 distribution of UNIX. In some instances the utility runs under both the VULCAN operating system and UNIX/24V; these are marked with a *. A list of the major utilities not provided under UNIX/24V is also included. Items in the latter list were not moved because their implementation was highly nonportable, or because they were of little utility to the project.

## Current Utilities

| | | | |
|---|---|---|---|
| ar | arithmetic | as(*) | at |
| basename | cal | cat(*) | cb |
| chgrp | chmod | chown | clri |
| cmp | comm | cp | date(*) |
| dcheck | dd | df | diff |
| du | dump | dumpdir | echo |
| em(*) | fgrep | file | getty |
| grep(*) | icheck | init | kill |
| ld | learn | ln | login |
| lpd | lpr | ls(*) | mail(*) |
| make | mkdir | mkfs(*) | mknod |
| mount | mv(*) | ncheck | newgrp |
| nice | nm(*) | od(*) | passwd |
| pr | ps | pwd | ranlib |
| restor | rm(*) | sed | sh |
| size(*) | split | strip | stty |
| su | sum | syms | sync |
| tail | tar(*) | tee | test |
| time | tr | tsort | tty |
| umount | uniq | update | wall |
| wc | who | write(*) | wump |
| yacc(*) | | | |

## Major Missing Utilities

| | |
|---|---|
| adb | requires ptrace system call |
| awk | requires lex. |
| bc & dc | bc requires dc which is complicated, and as yet hasn't been looked at for portability |
| lex | highly nonportable |
| lint | requires sort |
| sort | tried to port it, but still buggy |
| tp | no need for it, always use tar |

## Text Processing Programs

| | |
|---|---|
| eqn, ptx, pubindex, roff, tbl, nroff, troff, etc. | must have nroff/troff to be useful, and nroff/troff is highly nonportable |

## Fortran Processors

| | |
|---|---|
| f77, m4, ratfor, struct | f77 uses the portable compiler, so careful study must be made regarding the changes to the intermediate language |

## Uucp Utilities

| | |
|---|---|
| uucp, uux, uucico, etc. | low priority, and require conversion of a packet driver at kernel or user level |