5-20-1981

Language processors for the Motorola M68000 microprocessor

Joseph Ziarko

Master of Science in Computer Science

Thesis Approval Form

This is to certify that Joseph Ziarko has submitted a thesis entitled:

Language Processors for the Motorola M68000 Microprocessor

to the faculty of the School of Computer Science and Technology in partial fulfillment of the requirements for the degree of Master of Science.

Approval: Michael J. Lutz Michael J. Lutz 5/20/81
(thesis advisor) (date)

Peter H. Lutz Peter H. Lutz 5/20/81
committee member) (date)

Guy Johnson Guy Johnson 5/20/81
committee member) (date)

LANGUAGE PROCESSORS FOR THE MOTOROLA M68000 MICROPROCESSOR

Joseph P. Ziarko

Rochester Institute of Technology
School of Computer Science and Technology

ABSTRACT

A set of software tools for the Motorola M68000 microprocessor was developed to run under the UNIX* operating system. A 'C' language cross compiler was created by modifying the UNIX 'C' compiler for the PDP-11. A macro cross assembler was designed and implemented to produce relocatable object modules for the M68000 in the a.out format of PDP-11 UNIX object modules. The UNIX loader for the PDP-11 was changed to allow relocation of 32-bit quantities as required by the M68000. A small set of utility routines was also written to assist in the implementation effort.

The language processors and utilities provide the means by which high level 'C' programs can produce executable images for the M68000. All of the programs are currently running on a PDP-11/70 UNIX system.

May 10,1981

*UNIX is a Trademark of Bell Laboratories.

Dedication

I dedicate this project to my wife, Virginia, for her endless help and encouragement during the past year. I thank her for her great personal sacrifice of time -- that which she gave me and that which I could not give her. I am grateful for her attentive ear during my long-winded explanations of various technical crisises. I marvel at her ability to propose solutions to problems about which she has no background; many of her suggestions were successfully implemented. Thank you, Ginny. I hope that I can return the favor in the near future.

TABLE OF CONTENTS

FIGURES

APPENDIXES

1. Introduction

The 'C' programming language was originally developed for implementation of the UNIX operating system on the PDP-11. Over the years, 'C' has proven to be a versatile language and has been particularly useful in the development of system software. Production compilers for the language have been written for a variety of machines.

The main goal of this project was to develop a set of language processors for the Motorola M68000 that would run on the UNIX system. The M68000 is a 16-bit single chip microprocessor that is similar to the PDP-11 in both instruction set and addressing modes. (The architectural features of the M68000 processor are presented in Chapter 2.) The UNIX system was chosen because of its widespread use in the university and industrial environment. The lack of commercially available software development tools for the M68000 was the justification for the project.

The UNIX 'C' compiler (referred to as the "base compiler") was modified to produce M68000 assembly language source code as output. The process of converting the compiler was the most significant phase of the entire project and is discussed in detail in Chapter 4.

A cross assembler, written entirely in 'C', converts the assembly language output of the compiler to M68000 object code. The UNIX object module format of the PDP-11 is used so that

existing UNIX utility programs can be used, potentially without modification. The significant portions of the assembler are discussed in Chapter 5.

The UNIX loader required changes to satisfy special requirements imposed by the M68000; these are presented in Chapter 6. The modified loader is upward compatible with the UNIX loader and can be used to link either PDP-11 and M68000 object modules.

The remaining chapters discuss some additional software tools that were developed, a summary of results, and suggestions for further work. Appendix C gives the assembly code generated for a sample 'C' source program.

2. The Motorola M68000 Microprocessor

The M68000 is a 16 bit microprocessor developed by Motorola in the late 1970's. Its CPU design was influenced by recent advances in software technology, especially in the area of high level language support. The **link, unlk, chk** and **movem** instructions are oriented towards high level languages and their compilers. The M68000 possesses seventeen 32-bit registers plus a 32-bit program counter and a 16-bit status register. Its 24-bit address bus provides a direct memory addressing range of more than 16 megabytes.

2.1. Data Types

There are 5 basic data types that can be manipulated by the Motorola M68000: bits, BCD digits (4-bits), bytes (8-bits), words (16-bits) and long words (32-bits).

2.2. Registers

The bits in the seventeen registers are numbered right to left from 0; thus, bit 31 is the most significant bit (MSB) and bit 0 is the least significant bit (LSB). The registers are divided into two classes of eight data registers and nine address registers. The data registers (**d0 - d7**) support operands of 1, 8, 16 and 32 bits. In performing operations of less than 32 bits, only the low order portion of the register is used; the high order portion is neither used nor modified.

The address registers support operands of 32 bits and only allow word or long operations to be performed on their contents. In instructions involving a word operand and a destination address register, the word operand is sign-extended to 32 bits before any operation is performed. Two of the nine address registers are stack pointers (supervisory and user). Thus, only eight of the registers (**a0 - a7**) can be used as general purpose address registers.

The status register comprises two distinct bytes, the user byte and the system byte. The user byte is accessible to any program and contains the condition code bits (carry, overflow, zero, negative and extend). The system byte contains the interrupt mask, processor state bits, etc. and is only accessible in supervisor mode.

2.3. Addressing Modes

The M68000 provides 14 basic addressing modes, most of which can be used on byte, word, or long operands. These include register direct, address register indirect, absolute short, absolute long, PC relative, status register, implied, and immediate. The address register indirect mode is also available with postincrement, predecrement, or displacement. In addition, a variant form of both the PC relative mode and the address register indirect mode (with displacement) uses the contents of an index register in the calculation of the effective address of an operand. Both data and address registers can serve as index

registers.

The absolute short mode represents an address as a 16-bit quantity and sign-extends it to 32 bits before use. Thus it can reference only the low and high 32K bytes in the machine's direct addressing range.

The PC relative mode is represented by a 16 bit signed displacement from the address of the extension word of the instruction. This limits the range of relative addressing to plus or minus 32K bytes.

2.4. Operand Representation in Memory

Memory is byte addressable with all instructions and multi-byte data aligned on even byte addresses. Each word of memory is numbered right to left from 0; bit 15 is the MSB and bit 0 is the LSB. The even byte of memory is represented by bits 15 to 8 and the odd byte by bits 7 to 0. It is interesting to note that although the bits are numbered as in the PDP-11, the byte positions are reversed.

Long words are represented by two 16-bit words. The MSB of the long word is the leftmost bit in the even byte of the upper word and the LSB is the rightmost bit in the odd byte of the second word.

2.5. Instruction Set

At first glance, the Motorola M68000 instruction set appears

orthogonal with a rich set of useful instructions that can be generally used on byte, word or long operands. The obvious instructions such as add, subtract, multiply, divide, and, or, exclusive or, one's complement and negate are present. The conditional branch can be used to jump plus or minus 32K bytes relative to the instruction. A wide variety of shift and rotate instructions also exist that do logical or arithmetic shifts (left or right) with or without an extend bit (set like a carry bit).

A few instructions are also present that can be of significant use to the compiler writer. The **link** and **unlk** instructions can be used to set up and release the stack frame of a function call. A move multiple registers instruction allows transfer of any or all data and/or address registers to or from memory with a single instruction. Load and push effective address instructions are also available that can be very useful in the implementation of an "address of" operator.

A closer look at each instruction reveals a abundance of special cases and idiosyncrasies. The condition codes are not set on operations (add and subtract) that have an address register as a destination operand. The **clr** and **tst** instructions can only be used on data registers; an immediate zero must be moved into an address register to clear it and the same must be compared against an address register to set the condition codes. All addressing modes are not legal with all instructions. Even though there are four major groupings of allowable addressing

modes, special cases are not a rarity on a per instruction basis. The shift and rotate instructions have different rules for different types of operands. Memory operands must be words and can only be shifted by 1 bit. Data register values can be shifted by a constant amount when the constant is from 1 to 8. Variable length shifts and constant shifts of more than 8 bits require a second data register to hold the shift amount. A compare instruction can only compare memory to memory if the addressing mode for both operands is postincrement. These examples show that the machine is not as symmetric as its designers claim. Such peculiarities in the architecture can cause some unexpected problems in the development of a compiler.

2.6. Contrast with the PDP-11

The M68000 is in many ways more advanced than the PDP-11, yet it also lacks some useful properties that are present in the latter. The biggest advantage of the M68000 is that it can operate on long operands directly. The registers are not only greater in number but also allow manipulation of the 32-bit quantities. In addition, the stack linking and move multiple registers instructions can greatly reduce the cost of subroutine entry and exit.

On the other hand, the biggest disadvantage is that indirect accesses can only be done via an address register. This means extra overhead is incurred where pointers are not kept in address registers. A related problem is that the registers are not gen-

eral purpose. Data registers cannot be used as address registers and vice versa. A final aesthetic inconvenience is that the instruction formats are highly encoded at the bit level, making human comprehension of object code virtually impossible.

3. Justification for Approach

The technical approach chosen for the implementation of the compiler was to use the existing UNIX 'C' compiler for the PDP-11 as the base of a cross compiler for the Motorola M68000. This compiler, written by D. M. Ritchie, is well known for its speed of execution and quality of code generated for the PDP series computers. The internal structure of the compiler is well documented in an article by Ritchie entitled "A Tour through the UNIX C Compiler".[1] Despite these appealing characteristics, the choice of a base compiler was not apparent because of the existence of a Portable 'C' Compiler that is also available in Version 7 UNIX.

The Portable Compiler is the obvious candidate to base a cross compiler on since it was designed with machine independence in mind. It can be mutated to generate code for other machines and has been shown to produce good code for PDP-11's. It is also well documented in the UNIX manuals.[2]

Since using either compiler involved changing an existing PDP-11 product, the choice was based on the similarities and differences in the architectures of the two machines and on the advantages of using one compiler base over the other. Despite the peculiarities in some of the instructions of the M68000, it is very close to the PDP-11 in its addressing modes and instruction set. The one type of addressing mode that the M68000 lacks is the ability to use indirection with displacement, postincrement, or predecrement although this is more of a performance deficiency rather than an a functional one. Comparison of the

two instruction sets reveals a superiority in the M68000 because of its ability to directly perform 32-bit operations. The largest difference between the two machines is that the M68000 has two sets of special purpose registers while the PDP-11 has a smaller set of general purpose registers. After briefly reviewing both compiler implementations, it was apparent that neither would provide a significant edge in solving the problem of dissimilar register types. Thus, from a machine architecture point of view, there was little to recommend one compiler over the other.

In comparing the two compilers, the Ritchie code executes faster and is much smaller. The Portable 'C' Compiler possesses many machine independent features that are lacking in the UNIX compiler. The bottom line is that the implementation advantages of the UNIX 'C' Compiler outweigh the machine independent features of the Portable Compiler due to the similarities between the PDP-11 and the Motorola M68000. Therefore, the UNIX 'C' compiler was chosen as the base compiler.

The cross assembler, on the other hand, had to be designed and implemented from scratch. It was decided at the outset to use the a.out format of the PDP-11 as the object format of the assembler. The main advantage of this is that all the UNIX tools that are currently available (such as the loader) could potentially be used unchanged on the output of the assembler. The disadvantage is that programs are restricted in size due to addressing limitations of the PDP-11. For the purposes of this

project, the advantages far outweighed the disadvantages.

Much emphasis was placed on the human interface issues in the development of the cross assembler. It was designed so that it could efficiently handle a macro capability and produce a listing that was extremely beneficial when working in a "cross environment". A further goal was to be as compatible as possible in syntax and function with the cross assembler that was available from Motorola.[3] The main problem with the Motorola cross assembler is that it is an absolute assembler, making it unacceptable as a candidate to be used with a cross compiler. It also lacks some important features that were required by the compiler such as a **comm** directive. (The **comm** directive treats a variable as an undefined external at assembly time; the link-editor will define storage for the variable if it is not explicitly defined elsewhere. This allows space to be allocated only once for multiple declarations of a variable.) A final design constraint for the assembler was that it be able to run on the small PDP-11 processors so that it would be usable in a small UNIX shop.

3.1. Implementation Approach

A bottom-up approach was taken in the implementation of the cross compiler and assembler. The assembler and a new UNIX tool to dissect a.out format, were designed and written first. This provided a means by which a gradual familiarity with the M68000 could be achieved. It also defined a concrete syntax for the

assembly language that the compiler was to generate. The macro capability of the assembler was accounted for in the design and the implementation of the I/O routines in the assembler although the macro feature itself was not added.

The UNIX loader was then modified so that it could link 32-bit quantities. This was a requirement since all pointers in the M68000 are naturally 32 bits long. The compiler was the last job undertaken and proved to be the most difficult. Once the compiler began generating code, though, it could be immediately run through the assembler to check for illegal assembly language constructions. At the same time, it provided a cross check on the assembler so that the whole path from 'C' source to object code could be tested at once.

4. Compiler Design Decisions

The first step in writing a 'C' compiler is to determine how the various data types will be represented for the target machine. The most natural representation for integer data on the M68000 is 16 bits, so this was chosen to be the size of types **int**, **short** and **unsigned**. The 16-bit instructions are not only faster than the 32-bit instructions, but the multiply and divide instructions only support 16-bit operations (32-bit multiply and divide must be supported at run-time). Data of type **char** are stored as 8-bit signed quantities, and data of type **long** are 32-bit signed values. These representations are identical to those used on the PDP-11.

Pointers, on the other hand, are best represented by 32-bit quantities on the M68000. In fact, some instructions that operate on pointers require a 32-bit representation. This is different from the PDP-11 'C' compiler (and those of most other machines for that matter) in which pointers are interchangeable with integers. This incompatibility may pose a portability problem for some existing 'C' source. If code is written properly, though, with the appropriate "casting" operations, then it should compile and execute correctly with no modifications.

The other decisions that were made involved the setup of the stack frame on function calls and the allocation of registers. Since **a7** is the hardware stack pointer, **a6** was selected for the frame pointer. The compiler uses **a0**, **a1**, **d0** and **d1** as scratch registers. The remaining registers can be allocated by the user:

a2 to **a5** are for pointer variables, and **d2** to **d7** are for variables of other types.

Since subroutine entry and exit overhead must be kept at a minimum, the **link** and **unlk** instructions are used to allocate the stack frame of a function. The frame includes space for all automatic variables and for all registers (other than the compiler scratch registers) that are referenced within the function. The **movem** instruction saves only this select group of registers on the stack frame at the function entry point. (See Figure 5.1 Stack Frame Layout).

4.1. Conversion Process

The compiler was converted to generate code for the M68000 in a series of well defined phases. After studying the source code for the compiler, milestones were created in an attempt to define a reasonable time frame in which the project would be completed. In the final analysis, the logical divisions of work were accurately defined although the time required to complete any single phase was generally underestimated.

It is necessary to look at the original structure of the base compiler to completely understand the conversion process. The compiler command itself is really just a controlling program which parses the command options and executes each pass of the compiler as needed. The first of these passes is the 'C' preprocessor. It processes "include files" and expands macros defined by the "define" statement. The preprocessor takes as input the

original source code and creates a temporary file on output that contains only 'C' source code. This file is then processed by the first pass of the compiler, known as pass 0. This pass performs all the lexical analysis and parsing of the 'C' code and produces a pair of temporary files that contain an intermediate code, most of which is machine independent. These files are then input to the final pass of the compiler, which is known as pass 1. This pass is primarily responsible for generating machine dependent assembly code from the intermediate code of pass 0. This assembly code may be optionally processed by an optimizer to produce a "better" assembly language program. Depending on the compiler options chosen by the user, the controlling program may invoke the assembler to produce an object module and the loader to create a final executable image. This basic structure was maintained in generating the M68000 cross compiler.

The entire process of converting the PDP-11 compiler required a complete study of all its components. The sequence of conversion steps is discussed chronologically rather than top-down.

4.1.1. Preprocessor Conversion

The 'C' preprocessor did not require any changes for the M68000 because of its machine independence.

4.1.2. Pass 0 Conversion

The conversion of pass 0 was a simple task compared to the

effort required in pass 1. Minor changes were made to the algorithm which assigns registers to local variables of functions. These changes were necessary because there were more registers available for allocation, and because the registers were of different types. This pass of the compiler was also modified to reflect the change to 32-bit pointers.

The first major conversion consisted of changes to the intermediate code that was generated for function epilogues and prologues. Even though most of the modifications were localized to one function, they involved a complete rewrite in this area. Some new intermediate pseudo-codes were introduced that optimized the subroutine entry and exit sequences for the M68000.

The area of implicit casting of unlike operands in expressions was also effected, especially in light of the fact that pointers increased in size to 32 bits. On the M68000, the implicit cast between pointers and longs need not exist while that between pointers and integers had to be maintained.

Via a new compiler option, the intermediate code that was generated in pass 0 could be converted to ascii and copied into a temporary file. This made examination of the intermediate code effortless and allowed it to be studied even before pass 1 existed.

4.1.3. Controlling Program Conversion

The controlling program was modified to reflect the M68000

repertoire of compiler passes, its assembler and its loader. Also, the compiler option was added to put the intermediate code of pass 0 into a temporary file as described above. Pass 0 of the compiler could now be executed and the resulting intermediate code of various 'C' programs examined. In this manner, the pass 0 modifications were virtually debugged without the code generation pass of the compiler.

4.1.4. Pass 1 Conversion

Pass 1 logically consists of three major components: an input analyzer, an expression optimizer, and a code generator. The input analyzer is responsible for the reading of the intermediate files of pass 0 and for the building of expression trees. An expression tree (representing a single expression) is passed as a whole to a group of expression optimizer routines that rearrange the tree so that more efficient code can be generated for the target machine. This "optimized" tree is given to the code generator for conversion to assembly language. The code generation process is driven by a series of actions located in the code generation tables. These tables contain all the information that is required to produce assembly code for a particular operator.

The first task that was undertaken in pass 1 was to make the 'C' source "lint free". (**Lint** is a UNIX utility that attempts to detect code sequences that are non-portable or of dubious meaning.) It is not surprising that the compiler is written in poor "UNIX Version 7 C" since it was bootstrapped from "UNIX Version 6

C" which lacks many of the current features of the 'C' language.

The initial modifications to this pass of the compiler were made to functions that generated code for specific compiler subtasks without the use of code generation tables. Examples of this type of code generation are initialization code, "switch" code, structure assignments, and function calls. The primary reason for this approach was that these coding changes were somewhat localized. A second benefit was that incremental familiarity with the internal structure of the compiler was possible. The changes that were required due to the two different types of registers on the M68000 also became apparent during the modification of these areas.

The next phase of the conversion involved changes to the expression optimizer. Given a tree representing an expression formed by the intermediate code, certain machine dependent optimizations can be performed by the optimizer that will help in the generation of efficient code. Since the M68000 can manipulate 32-bit quantities while the PDP-11 cannot, more code was removed from the expression optimizer than was added. Also, some special case code was present that was difficult to understand in the context of the optimizer alone. Much of it was the result of interaction with the code generator, so it was removed to prevent unexpected problems from surfacing at a later time.

The most tedious phase of the project was the production of new code generation tables and the definition of many new code macros. The macros are special ascii characters in the code

generation tables that are intercepted and processed by the code generator. They are primarily used to begin code generation activity of an expression tree and to handle special cases in the code generation process. Due to its complexity, this phase will be discussed in detail in the next section.

The last phase of the project (excluding debugging, which is continuing), involved tuning the compiler. This consisted of changes to the expression optimizer so that code could be more efficiently generated by the code generator. These changes replaced the PDP-11 tuning code that was originally present in the expression optimizer.

The total effort required to generate the cross compiler in its present form from the base compiler was about 600 man-hours. This does not include the work that was done on the cross assembler or loader.

4.1. Stack Frame Layout

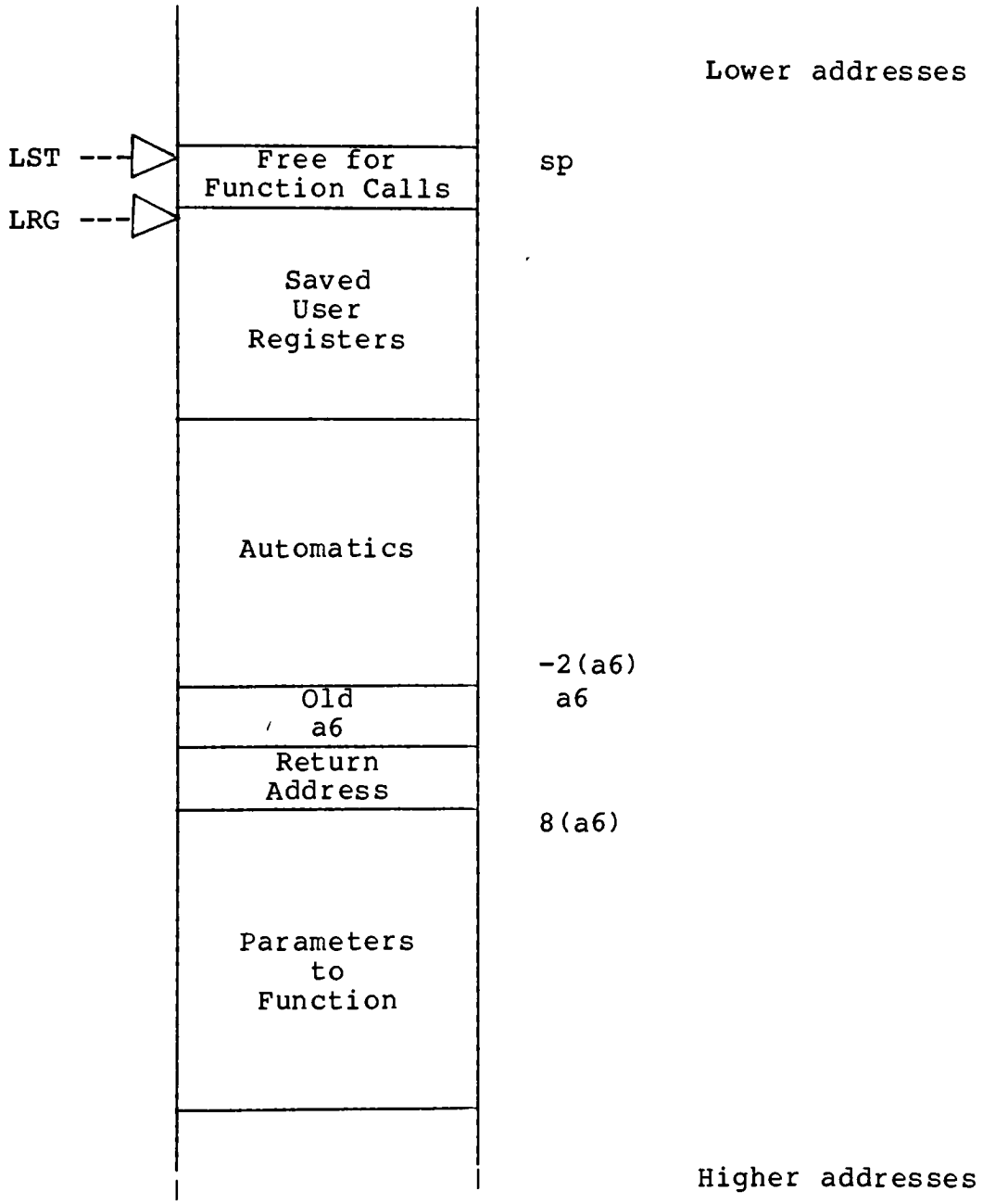


Figure 4.1 Stack Frame Layout

4.2. Compiler Intermediate Code

As mentioned in the last section, communication between the two passes of the compiler is achieved through two intermediate language files. The files are asymmetric clusters of binary numbers whose structure is defined solely by the first number in the cluster called the operator. The operator consists of an 8-bit operator number and an 8-bit sync byte. The operator number defines the operation to be performed and the type (ascii, binary) and number of operands that follow. The sync byte is used as a sequence check to catch phase errors between the two passes. There are two basic types of operators, expression operators and control operators; they are described in Appendixes A and B respectively.

The most significant modifications to the intermediate code can be seen in the **BGNFUNC** and **ENDFUNC** control operators. The function number in both of these operators (represented by # in the following discussion) generates special constant labels that are only used in the context of that particular function. The first of these labels (**LST#**) represents the number of bytes that will be added to the frame pointer by the **link** instruction to define the function stack frame. A second label (**LRM#**) is a constant that represents the mask of registers that will be pushed onto and popped from the stack by the **movem** instruction. The last label (**LRG#**) represents the displacement from the new frame pointer where the saved registers will be located in the function stack frame. (See Figure 4.1 Stack Frame Layout).

It is necessary to look at a typical function entry and exit to fully understand the purpose of these labels. In generating code at the function entry point, the only information available is the number of registers that were allocated for local variables and the amount of stack space required to hold automatic variables and the previous values of the allocated registers. Unfortunately, it is not known how much additional stack space will be required for registers and temporaries used by the code generator. Thus, all three labels are undefined at the entry point (and must be resolved by the assembler as forward references). The code generated at the entry point to function number one, for example, is shown in Figure 4.2.

```
        link    a6,#LST1
        movem.l #LRM1,LRG1(a6)
```

Figure 4.2 Entry point code for function #1.

When all code has been generated for the function body, the registers used by the code generator are known as well as the additional stack space that is required. Figure 4.3 gives an example of an exit sequence for the same function.

```
        movem.l LRG1(a6),d5-d7/a5
        unlk    a6
        rts
LRM1    equ     $20e0
LST1    equ     -34
LRG1    equ     -32
```

Figure 4.3 Exit code for function #1.

In this example, it can be noted that the pop register sequence of the **movem** instruction uses an ascii representation for the registers. This is solely part of the human interface; the LRM1 mask provides an equivalent machine representation to satisfy the initial **movem** instruction at the function entry point. It can also be seen that the register save area in the stack begins 32 bytes below the frame pointer (LRG1), while the stack frame actually includes 2 more bytes (LST1). The discrepancy is accounted for by the generation of function calls within the function body. Under favorable circumstances, the compiler will pass the first actual parameter (rightmost) using the stack pointer (**(sp)**) rather than the predecremented stack pointer (**-(sp)**) for performance reasons. If there are no nested calls in a function body, then the **LST** label would be identical to the **LRG** label. In the above example, enough space is allocated for a 16-bit parameter to be passed in this manner, although the compiler will allocate up to four bytes of stack space for this purpose.

The other class of intermediate language operators, as mentioned above, comprises those associated with expressions. A single expression is parsed in pass 0, forming a binary tree. This tree is transmitted to pass 1 in reverse Polish notation and reconstructed using an input stack. At the conclusion of the reconstruction, only the root of the tree should remain on the stack. No control operators may be part of the tree and thus cannot be part of the sequence of operators forming the tree. An **EXPR** or **CBRANCH** control operator will always be generated after

transmission of the tree to initiate generation of code for that particular tree.

4.3. Expression Optimization

The first analysis that is performed on an expression tree in pass 1 is to optimize the tree so that efficient code is generated for the target machine. Many of the original base compiler optimizations are applicable to the M68000 and remain intact in the cross compiler. Numerous M68000 machine dependent optimizations were also added to the compiler to aid in the code generation process. Several of these optimizations will be described to show the considerations which arise in generating good code.

The compare instruction of the M68000 generally requires that the destination operand be in a register unless the source operand uses immediate mode addressing. In light of this, the expression optimizer attempts to rearrange relational operations to prevent redundant register moves. If the destination operand is a register node, no optimization is performed. If the source operand is a register node, or if the source operand is more difficult to compute than the destination, the operands are exchanged and the relation is inverted. This is also done if both operands are equally difficult to compute, but the destination operand is a name node while the source is not a name node. In reversing the operands, the non-name node can be evaluated in a register and directly compared to the name node. Without this

rearrangement, a second register would be required into which the name node value would be moved before the comparison could take place.

The assignment operator also takes advantage of cases where the destination operand is a register node. In mixed mode assignments where the destination is a data register of larger bit size than the source operand, any type conversion operations such as ITOL need not be performed in a separate register before assignment; the destination register can be used directly. In cases where the destination is an address register and the source operand is a signed 16-bit quantity, the operand can be directly assigned to the register since automatic sign extension occurs when words are assigned to address registers. The assignment plus (+=) and minus (-=) operators can also take advantage of this particular optimization since the internal code sequence for incrementing and decrementing a pointer is to use a 16-bit signed quantity on the right hand side of the expression.

Since the 'C' language requires that all byte operands be converted to integer in the evaluation of expressions, the code generator will automatically sign-extend characters moved into registers -- no explicit cast operation is required either internally or externally. The code generation tables have the responsibility for the correct handling of mixed mode operations where one operand is of type **char**. In fact, the cast that is generated when a byte interacts with a long word is integer-to-long (ITOL) rather than the expected character-to-long (CTOL). This type of

treatment in an assignment operator would generate naive code without the help of the expression optimizer. For example, if one were to assign an addressable word quantity to an addressable byte quantity (excluding register operands), the unoptimized code might be that shown in Figure 4.4.

```
move.w      wval,d0
move.b      d0,bval
```

Figure 4.4 Unoptimized assignment of word to byte.

The extra move instruction is required because the least significant byte of the 16-bit word operand is located at the address of the word plus one (odd byte). This is detected by the expression optimizer in the processing of the assignment statement and is solved by generation of an explicit integer to character cast (ITOC). Figure 4.5 shows the improvement in the code of Figure 4.4 using this optimization.

```
move.b      wval+1,bval
```

Figure 4.5 Optimized assignment of word to byte.

This optimization is also performed with the LTOI cast where the integer part is really a byte operand.

Besides tree optimization, the single most important task performed by the expression optimizer is labeling each interior node of the expression tree with a register count. This count represents the minimum number of registers required to generate code for that node without temporary stores. The algorithm that

is used in the original compiler is based on one proposed by Ravi Sethi and J. D. Ullman in a paper entitled "The Generation of Optimal Code for Arithmetic Expressions".[4] A problem immediately encountered is that the algorithm assumes the target machine has general purpose registers (like the PDP-11). It was decided to allow the algorithm to use only the data register set in its calculations since address registers cannot be used for general arithmetic operations. For this to work, two modifications had to be made. First, the code generator needed to have two scratch address registers at its disposal for temporary pointer indexing (a0 and a1). Second, it was necessary to modify the node marking algorithm so that it did not count indirect nodes (those with STAR as the operator) in the register requirement calculation. These nodes could be resolved without the aid of any data registers by using a free scratch address registers. An inherent limitation of this implementation is that temporary address registers can only be used for indexing and not for general arithmetic operations. The only exceptions to this rule are cases in which it is known that the operation being performed is simple enough so as not to require the use of any other registers as temporaries (this is often the case in pointer arithmetic). It is the responsibility of the code generator to insure that no other liberties are taken with the address register set.

4.4. Code Generation

After the expression tree has been processed by the expression optimizer, it is given to the code generator to produce

assembly code. The main generator routine has three arguments: a pointer to the expression tree, a pointer to the code generation table that is to be used, and an encoded integer representing the current data and address registers available for use. There are five possible choices for code generation tables; the choice made determines whether the result should end up in a data register or an address register.

The first code table, dregtab, is the data register table. It is used to compile code for the current expression tree into a data register. Aregtab is the corresponding table for address registers. It is used for indirection and for calculating special case arithmetic operations into address registers.

Efftab is used when an expression is compiled only for its side-effects rather than its value. Such is the case in assignment statements where the value of the result need not be left in a register for further analysis.

Cctab is used when only the condition codes of the expression are desired rather than the value of the expression. The if statement is the primary user of this table since the truth value of a conditional (0 or 1) is not generally required.

The last table, sptab, is used to compile code for the current expression tree onto the stack. It exists solely to prevent redundant register moves from being generated in this situation.

The tables are structured in such a way that not all

operators need to be in all tables, nor do all cases of possible operands have to be present in the table entry for an operator. The main code generation routine can attempt to generate code using the register table dregtab if it determines that code cannot be generated for the current operator with the desired code table. If this succeeds and the desired table was sptab, a move instruction is generated to move the operand onto the stack. If the desired table was cctab, the appropriate test instruction is generated so that the condition codes are set. If code cannot be generated using dregtab, then a "No code table for op" diagnostic results. Thus, the data register table is the largest of the code generation tables. It is used in a last ditch effort to generate code for any operators that are missing or incomplete in other tables. In fact, the other tables are relatively small and generate code in only a limited number of situations. The data register table is really the only table required for the compiler to generate correct (but inefficient) code for the M68000.

The code generation tables are written in a pseudo-code for ease of generation and modification as well as to achieve a certain degree of portability. An undocumented UNIX program takes as input the pseudo-code file and produces as output an assembly language file that is assembled and linked with the cross compiler. In order to transport the cross compiler to another machine, only this program has to be rewritten so as to generate assembly code for the machine that the compiler is to execute on. This is a much less complex and less error-prone job than the effort that would be required to rewrite the code generation

tables themselves.

Each of the five main code tables consists of two parts. The first is a dispatch table containing (operator, subtable pointer) pairs. This table is searched linearly using the operator as the key. If the operator is not in the table, a failure indication is returned. Otherwise, the pointer to the subtable is used to locate the place where the code table actually begins for that particular operator.

A subtable consists of a series of operand specifications. Associated with each specification is a list of actions, many in the form of macros, that are to be performed if this set of operand specifications is matched. As above, this table is searched linearly until either an appropriate set of operand specifications is found or until the bottom of the subtable is reached. In the latter case, a failure indication is returned (in the same manner as if the operator were never found in the original table). When a match is determined, the actions associated with the particular subtable entry are performed, resulting in the generation of assembly code.

Before studying the actions of generating code, it is important to look at the matching criteria that are used in the selection of a subtable entry. A subtable entry is introduced in the pseudo-code by a % character and is followed by two comma separated lists of characters each representing one operand. For unary operators, the second operand specification must be present although it is ignored in the matching process. The format of

the specification line is shown in Figure 4.6.

```
%dt,dt
        (Action list)
```

Figure 4.6 Operand specification format.

The 'd' after the percent sign is the degree of difficulty of the destination operand (first operand); the other 'd' represents that of the source operand (second operand). The 't' in either case represents the type of the operand.

The concept of degree of difficulty can be best explained using a bottom up approach. A degree of difficulty represented by the letter **n** can be defined as the most difficult specification in the repertoire, i.e., any operand will match this specification. Thus, a subtable entry as shown in Figure 4.7 is guaranteed to match any given operand.

```
%n,n
        (actions for this specification)
```

Figure 4.7 Most difficult operand specification.

Since the **n** degree conveys no information about either operand, the only recourse is to generate code to compile the second operand on the stack and the first operand in a scratch register, and then to perform the operation. It is easily seen that if a simpler degree of difficulty existed that was able to pass more information about the operand, considerably better code could be generated. To enhance the previous example, a degree of diffi-

culty represented by the letter **a** is introduced that will match any operand that is directly addressable (i.e., any operand that can be represented as the source or destination of a Motorola M68000 instruction). Thus, the code subtable of Figure 4.7 is enhanced as is shown in Figure 4.8.

```
%n,a
        (actions for this specification)
%n,n
        (actions for this specification)
```

Figure 4.8 Enhanced code subtable.

With this subtable, it is possible to avoid compiling the second operand on the stack if the operand is indeed addressable. It can also be pointed out that if the %n,a specification was positionally beneath the other, it would never be matched. This is because the degree of difficulty is a relative quantity rather than an absolute one.

The degrees of difficulty that are defined for the M68000 cross compiler are much more refined than those found in the PDP-11 base compiler. As is shown above by the example, the more refined the degree of difficulty, the greater amount information is passed to the code generator. This is more of a requirement in the M68000 since not all addressing modes are allowed with all instructions and certain optimizations can only be performed in some special cases. The following is an ordered list of the degrees of difficulty used in the cross compiler (from easiest to most difficult):

z is satisfied when the operand is a constant that has a value of zero. This specification applies to both integer and long constants and catches many special cases such as assignment of zero to an operand (allowing a clear instruction to be generated instead of a move).

1 is satisfied when the operand is a constant that has a value of one. This applies to both integer and long constants and optimizes such cases as shift by 1.

q is satisfied when the operand is a constant that has a value between one and eight. It corresponds to the 'quick' mode available with many M68000 instructions.

c is satisfied when the operand is any integer or long constant (positive or negative). It is an extremely valuable degree specification since it allows the code generator to take advantage of those instructions that have special "immediate" addressing mode formats.

r is satisfied when the operand is an address register operand. This degree specification must appear before the data register specification (below) since address registers require special treatment in certain contexts (in comparison with zero, for example).

d is satisfied when the operand is a data register operand. This specification takes advantage of the fact that quantities can only be sign extended in registers.

t is satisfied when the addressing mode of the operand is postincrement. This catches some special cases where the addressing mode must be postincrement.

p is satisfied when the addressing mode of the operand is predecrement. This catches some special cases where the addressing mode must be predecrement.

i is satisfied when the operand is represented by the "address of" operator. It is only used to optimize code in extremely rare cases since any addressable operand would match it.

a is satisfied when the operand can be represented as the source or destination of an M68000 instruction.

e is satisfied when the operand can be resolved in 'n' or fewer data registers, where 'n' represents the number of data registers that are currently available to the code generator for use. This degree specification also requires that there be at least one extra address register available for use by the code generator.

f is satisfied in the same manner as the **e** specification except that no address register restriction is imposed.

n is satisfied by any operand.

b means that the bottom of the subtable has been reached. The base compiler implementation used implicit end of table markings that were placed between subtables by the assembly language conversion program. This degree specification

guarantees at the table level that one subtable will never run into the next.

The type field associated with each operand specification is an optional set of characters that limit the scope of the operand. The following types are supported:

(a missing type) will match anything of type character, integer, long or pointer.

w will match anything of type integer (signed or unsigned).

b will match anything of type character.

f will match anything of type float.

d will match anything of type double.

s will match anything of type structure.

m is used with binary operators and requires that the bit size of both operands be identical.

l will match anything of type long.

u will match anything of type unsigned integer.

ip is an infrequently used type that matches anything of type pointer.

A * can be optionally appended to the end of each half of an operand specification to require that the highest operator in the current tree be an indirect operator (STAR).

In view of this information, the match algorithm can be easily understood. Assuming a binary operator is at the top of the tree, the match routine will first call a function that returns a number representing a degree of difficulty for each of its two immediate nodes (operands). After the subtable for the operator has been located, the first operand specifications are compared against the the actual operands to check for both degree of difficulty and type compatibility. An indirect operand check is performed if the "*" specification is present. If any of the match criteria fail, the algorithm proceeds to the next subtable operand specification until either a match is found or until the bottom of the subtable is reached. The match function returns a pointer to the matched subtable entry on success, and a NULL pointer otherwise.

4.5. Code Generation Macros

The actions that are associated with an operand specification are comprised of assembly language code fragments and macros. By strict convention, all macros begin with upper case letters. (In fact, the program that converts the code generation tables to assembly code is so rigid that it cannot even tolerate any upper case letters in comments!) The main code generation loop processes one character at a time from the action string. If the character represents a macro, the macro action is performed. All characters that are not special to the code generator are simply passed to the output assembly language file. A list of actions is terminated by a blank line.

There must always be at least one available data and address in the compilation of any node in the expression tree. These registers are called the current registers. If an additional data (or address) register is available to the code generator, this register is called the next register. (The next data or address register is always one greater than its corresponding current register.) An initial set of macros can be defined from these concepts:

R causes the name of the current data register to be printed.

Rl causes the name of the next data register to be printed.

Q causes the name of the current address register to be printed.

Ql causes the name of the next address register to be printed.

For example the operand specification

```
%n,z
        clr.w     R
```

would expand in the assembly language file to

```
clr.w     d0
```

if the current data register were d0.

Another set of macros is associated with addressable quantities (those represented by the **a** degree of difficulty):

Al causes the generation of code representing the destination operand (the first operand in the specification) to be printed to the assembly language file. This macro is only

valid if the operand can be represented as the source or destination of an M68000 instruction.

A2 is the same as A1 except that code is generated for the second operand (source).

The following macros are used to generate the correct type specifier that is appended to the opcode of most M68000 instructions.

B1 causes a **.b**, **.w** or **.l** to be appended to the instruction opcode if the destination operand can be represented as an 8-bit, 16-bit or 32-bit quantity respectively.

B2 is the same as B1 except that the source operand is used.

D1 is the same as B1 except that 8-bit quantities cause a **.w** to print rather than a **.b**. This is used in arithmetic operations where byte operands have already been sign-extended to 16-bits.

D2 is the same as D1 except that the source operand is used.

O is used with indirect operands and causes a **.b**, **.w** or **.l** to be appended to the instruction opcode if the type of object pointed to by the operand is an 8-bit, 16-bit, or 32-bit quantity respectively.

```
%rl,au
        clrB1     R
        moveB2    A2,R
        moveB1    R,A1
```

Figure 4.9 Assignment using efftab.

Figure 4.9 shows that if an unsigned quantity is moved into an address register, it first must be moved into an data register so that the high order word can be cleared. The code that would be generated from the following specification (assuming d0 is the current data register, -2(a6) is the source operand, and a5 is the destination operand) is shown in Figure 4.10.

```
clr.l     d0
move.w    -2(a6),d0
move.l    d0,a5
```

Figure 4.10 Example of assignment using efftab.

This example also suggests that the size of the code tables can get quite large in representing all operand combinations for all operators. For this reason, a facility exists that allows up to two separate code strings to be defined in a side table per operator. The macros that allow access to the side table are the following:

I causes the primary code string associated with the operator to be printed.

I' causes the secondary code string associated with the operator to be printed.

J cause the secondary code string associated with the operator to be printed if the type of the root of the current tree is unsigned, else the primary code string is printed. This macro is useful in the handling of signed and unsigned

multiplication and division.

J' is the same as **J** except that the type of the destination operand is tested.

J'' is the same as **J** except that the type of the source operand is tested.

For the assign operator, move is the primary code string and clr is a secondary one. Thus, the operand specification in Figure 4.9 can be rewritten as is shown in Figure 4.11.

```
%rl,au
        I'Bl      R
        IB2     A2,R
        IBl     R,Al
```

Figure 4.11 Use of space saving code macros.

These macros can also assist in reducing the number of subtables. The same subtable can be shared between similar operators (such as add and subtract) if these macros are used in place of specific opcodes in the action list. The net reduction in the size of the data may be critical when running the compiler on small PDP-11's.

A series of macros exists that are primarily responsible for the recursive generation of code for subtrees. They are the following:

F causes the left operand of the current tree to be recursively compiled into the current data register.

S causes the right operand of the current tree to be recursively compiled into the current data register.

G causes the left operand of the current tree to be recursively compiled into the current address register.

W causes the right operand of the current tree to be recursively compiled into the current address register.

Optionally, there is a sequence of characters that may follow any of the above macros. Each character is a flag to the code generator to cause certain tuning to take place.

+ causes the next register of the "opposite" type to be used in the generation of code for the operand. For example, the macro **F+** causes the next address register to be used ("opposite" meaning not a data register as implied by **F**). This flag can only be specified if a next register of the target type actually exists (i.e., the **e** or **f** degree of difficulty was matched).

***** forces special generation of code for indirect operands of the form *A or *(A + c) where 'A' represents an address and 'c' is a constant. In either case, code is only generated for the address 'A'. Another specification exists which allows the constant 'c' to be picked up at a later time.

S causes **sptab** to be used in the generation of code for the specified subtree.

C causes **cctab** to be used in the generation of code for the specified subtree.

l causes the next register of the specified type (data or address) to be used instead of the current register. This flag can only be used if the next register actually exists (i.e., the **e** of **f** degree of difficulty was matched).

! guarantees that the result will end up in the requested register.

There are a few macros that are not represented by upper case letters. These macros use characters that are illegal in the M68000 assembly language so that they can be intercepted by the code generation loop.

{ causes a **#** to be printed to the assembly language file (introducing an immediate operand).

} is used when code for a push or pop from the stack is going to be generated. If the character after the **}** is a '-', a push is assumed. This macro keeps track of the number of 'items' on the stack.

#1 causes the collected constant of an indirect destination operand (*(A + c)) to be printed.

#2 is the same as #1 except that the source operand is used.

@ is the same as #1 except that the current subtree root is used.

```
%n,fw*
   F
   W*+
   J      #2(Q),R
   E
```

Figure 4.12 Divide, multiply and mod using dregtab.

According to the match criteria specified in Figure 4.12, the source will only match if the operand is STAR of type integer or unsigned and it can be compiled in any number of data registers less than or equal to the number that are currently available. The action list first shows that the destination operand is to be compiled into the current data register. Then, the source operand is to be compiled into the current address register. The '+' flag is specified since the current data register is holding the destination operand and cannot be used for temporary storage in compiling the source operand. The '*' specification makes sure that only the address of an indirection is compiled into the current address register. The next line of actions generates code for the instruction, and uses the #2 macro to pick up the constant (if any) that was part of the indirection. This is the implementation of the address register indirect with displacement mode of addressing. The **E** macro will generate a **swap** instruction of the current data register if the operator is **mod** and nothing otherwise. This allows divide and mod to share the same subtable (since the only difference between the two is the location of the result within the register).

A handful of special case macros also exist.

Y causes generation of an **ext.w** instruction of the current data register (or of the next data register if the next action byte is an **A**) if the type of the current subtree is **char**. This macro is extremely useful in allowing bytes and integers to be handled in the same subtable specification.

Y' is the same as **Y** except that the type of the destination operand is tested.

Y'' is the same as **Y** except that the type of the source operand is tested.

X causes some support code to be generated for shift instructions.

Z causes the printing of the immediate data associated with the mask used in bit field operations.

H is the same as the **F** and **S** macros except that code is generated for the current subtree using dregtab. It is used in the generation of code for cctab to prevent redundant **tst** instructions from being generated when the operation itself will set the condition codes correctly.

T causes the current subtree to be compiled using aregtab. It is used to implement the STAR operator in dregtab.

There are about 33 pages (1800 lines) of code generation tables that implement the full 'C' language less floating point. This converts into 2100 lines of totally unreadable assembly language code that is eventually assembled and linked with the

compiler.

5. Assembler Design Decisions

Mas is a two pass cross assembler for the M68000 that produces relocatable object code in the a.out format of the PDP-11. The assembler command is a controlling program the parses the command options and executes each pass of the assembler as needed. The first assembler pass, **pass 1**, builds the user and macro symbol tables and performs syntax checking on the assembler source. The output of pass 1 is a pair of files containing the two generated symbol tables and a temporary file that contains the assembler source in an internal format. This format allows status information about each assembler source line to be transmitted to the second assembler pass, **pass 2**. Based on this information, the pass 2 input analyzer can decide to ignore a source line if serious errors were detected in pass 1. Pass 2 produces an object file and a listing file as output. The listing includes a printout of the object code associated with each source line, a sorted symbol table listing, and a list of error diagnostics.

Aside from the external influences on the design of the assembler (see 2.0 Justification for Approach), there were some important internal issues that influenced the design. The **yacc**[5] processor of UNIX was used to generate the parser for the assembler. A series of grammer specifications (similar to BNF) are input to yacc and a 'C' function (parser) is produced as output. This approach was especially appealing because the grammer is self documenting and easy to maintain.

Symbol table management was designed around the fact that memory space would be scarce. A buffer management package was developed to support an "infinite" symbol table space for any number of variable size symbol tables. The implementation will be discussed in the next section.

Some performance items were also designed into the cross assembler. The lexical analyzer was written without the use of the **lex**[6] utility so that performance tuning would be easier to accomplish. All symbol table lookups are done via a hashing algorithm (see 6.0 Software Tools). The I/O routines of pass 2 use double buffering to prevent the input line from being copied to a separate buffer for output.

The format of the listing output file was optimized for space. Key output items are strategically placed on 'tab' boundaries so that the size of the listing can be kept at a minimum. The listing is also formated to fit on 80 column paper.

The approximately 12,000 lines of 'C' code that make up the cross assembler were written and tested in about 600 man-hours. About 25% of the total time was invested in implementation of the listing output.

5.1. Symbol Table Management

There are three symbol tables in the cross assembler. The permanent symbol table contains all the instruction opcodes and directive mnemonics. The user symbol table consists of all user

defined labels. The macro symbol table contains the names of all macros that are defined in the assembler module and of those invoked from a macro library.

The lexical analyzer employs a number of strategies in searching for a symbol. If a token begins in column one, it is assumed to be a label; only the user symbol table is searched. If an operator (instruction opcode, directive) is being searched for, the macro, permanent, and user symbol tables are searched in that order. This allows macros to redefine permanent symbols. When looking for an operand, the lexical analyzer first checks the user symbol table. If a match is not found, a side table of permanent symbols (register names) is searched. The permanent and macro symbol tables are not used in operand searches.

The memory cache management of symbol table buffers is accomplished in the buffer management routines. For each symbol table, a buffer header is defined that contains control and linkage information. Each header is linked to a variable number of fixed size, self-contained symbol table cache buffers. (The permanent symbol table buffer is unique in that it is always maintained in memory.) Only one cache buffer per symbol table can be in the process of being filled; all other buffers are full of previously defined symbols. As each buffer is filled, it is assigned a unique buffer number for that symbol table. A filled buffer will remain linked into the buffer cache as long as there is enough memory to support the current needs of the cache system. Each time a symbol is located in a particular buffer, the

buffer is aged. If a new cache buffer is required and the dynamic memory pool is exhausted, the entire cache system is sorted to find the "youngest" buffer in the cache (the buffer with the least number of cache hits). This buffer is then written (swapped) to a disk file associated with its symbol table (assuming that it contains information that has not already been written). The position of the buffer in the file corresponds to its buffer number. A buffer that is in the process of being filled is not eligible for swapping.

The symbol table search algorithm, in locating a particular symbol table entry, makes no guarantee that the corresponding buffer will remain in the cache when the search begins for the next symbol. For this reason, each time an entry is located, a special data structure is filled with the location (buffer number) and appropriate symbol table information for the entry. The yacc parser uses the information in this block in processing the symbol. If access to the buffer containing the symbol entry is required, it can be easily gained via the location information that was placed in the block. Also, if a symbol table cache buffer is updated, that buffer must be explicitly marked as having been written into. This is required in order for the buffer mechanism to write the buffer to the disk file in a swap. When the parser no longer requires access to a symbol, its block is released into a free pool.

6. Loader Modifications

A new loader, mld, was created from the existing PDP-11 UNIX loader ld to support relocation of 32-bit quantities. Mld is upward compatible with ld and can used to link object modules created by the PDP-11 UNIX assembler as or by the M68000 cross assembler mas.

Two major changes were made to support the requirements of the M68000 cross assembler. A new relocation bit was defined, signifying that a particular word of object code is the high order portion of a 32-bit relocatable quantity. The subsequent word of object code is the low order portion of the long; its corresponding relocation word contains the relocation information that will be applied to the whole 32-bit value.

The other major change was the addition of an option (-M) that directs the loader to perform addressing range checks on the PC relative and absolute short addressing modes of the M68000. This option is mandatory for M68000 object modules and is prohibited for PDP-11 object modules. Its presence permits the specified M68000 addressing modes to be used over source module boundaries.

The basic structure of the a.out format was not altered by the loader modifications. Thus, all UNIX tools that operate on this format should require no changes.

7. Software Tools

Some additional software tools were developed to assist in certain phases of the project. The expose program was critical to the debugging of both the assembler (mas) and the loader (mld). It contains options to selectively print the header, the text area, the data area, and/or the symbol table of an a.out image. It also provides the capability to begin examination of the text or data region from a particular location in the image. Each 16-bit word of object code is printed with its address and an ascii representation of the relocation information associated with it. The mld loader additions to the relocation fields in the a.out format are also recognized by this utility. Both octal and hexadecimal radixes are supported by expose.

Htab is a utility that generates the hash table of the permanent symbol table for the mas cross assembler. The input to htab is a file containing assembler opcodes and flags. The output is a 'C' source file that is compiled and linked to the M68000 cross assembler. An external hash function is compiled with both htab and the cross assembler; different hash functions can be chosen to test collision rate and to monitor performance (tuning). Options exist that allow manipulation of hash table size and the interactive insertion of symbols into the input stream.

8. Summary and Conclusions

This project resulted in a set of tools which convert 'C' programs into executable code for the Motorola M68000. The cross assembler has been successfully used in an industrial environment to produce M68000 diagnostic programs. Programs created by the 'C' Compiler have not been executed on an M68000 for a variety of reasons. First, enhancements to a downloader (not a part of this project) would be required to handle the text, data and bss regions as defined in the executable header. Second, the run-time support routines for long division and multiplication are not written.

The assembly language output of the compiler was "verified" by comparing it to the output of another M68000 cross compiler. This second compiler was written at MIT and is based on the Portable 'C' Compiler. The output of both compilers is similar; the differences were accounted for by tedious hand checking. In all cases, the compiler developed for this project generated code at least as efficient as that of the MIT compiler.

As a test of the compiler, the 'C' source code for the cross assembler was translated and linked with the modified loader. The link was successful after calls on UNIX library functions were replaced with "stubs".

The conversion of the UNIX 'C' Compiler was technically challenging and very interesting. In my opinion, this compiler would provide an excellent base for a cross compiler for any

"reasonable" machine. The amount of work to convert pass 0 is proportional to the similarity in data type sizes between the target machine and the PDP-11. The amount of code that can be retained in pass 1 is more closely related to the similarity in addressing modes (rather than instruction set) between the target and the PDP-11. In any case, the internal structure of pass 1 should provide a significant head start in the development of any cross compiler.

9. Further Work

The addition of floating point simulation to the cross compiler is the most significant, unfinished task. The required changes are localized to pass 1 in the expression optimizer routines and in the code generation tables. The suggested technical approach is to define some specific floating point operators that would be substituted for normal operators in the expression optimization process. (For example, the add operator PLUS might become FPLUS.) The benefit of independent code generation tables is twofold. First, it localizes the changes to the code generation tables; if a floating point capability is built into the M68000 instruction set in the future, these modifications can be easily removed. Secondly, a performance gain may be realized since the "normal" code tables (plus, minus, etc.) will not have floating point entries intertwined with non-floating point entries. The run-time support routines for the floating point simulation must also be designed and implemented. The full functionality of the 'C' language will only be achieved with the addition of a floating point package.

The optimization pass of the cross compiler does not currently exist. Many of the optimizations in the PDP-11 UNIX 'C' Compiler would be applicable to the M68000. Areas of optimization specific to the M68000 include: elimination of unnecessary **movem** instructions when no registers are saved in the function stack frame, removal of unneeded **link** and **unlk** instructions when no stack frame is required within a function, and insertion of

assembler directives (or rearrangement of code) so that forward references to local symbols in the same code region use the short modes of addressing rather than the absolute long mode.

A current deficiency is the cross assembler is the inability to generate optimum code for branch instructions. The conditional branch instructions of the M68000 support displacements of 8 and 16 bits; all forward references in the assembler use the latter. A significant code improvement could be achieved if short branches were used in all possible circumstances. A discussion of the solution to this problem can be found in "A Process for the Determination of Addresses in Variable Length Addressing" by Gideon Frieder and Harry J. Saal.[7]

The completion of the conditional compilation and macro capabilities in the cross assembler is also a candidate for future study. These features were designed into the assembler, but the implementation phase was never fully completed. This task also includes modifications to the macro archiver (mar) to accept the macro syntax of the M68000 cross assembler.

All project related software is currently being ported to the VAX-11/780. The purpose is to make the 'C' source code as machine independent as possible. This will minimize the portability problems that can occur if it becomes desirable to use this software in a non-PDP-11 environment.

10. References

1. D. M. Ritchie, "A Tour through the UNIX C Compiler," Bell Laboratories, Murray Hill, New Jersey.

2. S. C. Johnson, "A Tour Through the Portable C Compiler," Bell Laboratories, Murray Hill, New Jersey.

3. M68000 Cross Macro Assembler Reference Manual, Third Edition, Motorola Inc., (1979).

4. R. Sethi and J. D. Ullman, "The Generation of Optimal Code for Arithmetic Expressions," J. Assoc. Comp. Mach. **17**(4) pp. 715-728 (October 1970). Reprinted as pp 229-247 in Compiler Techniques, ed. B. W. Pollack, Auerback, Princeton NJ (1972).

5. S. C. Johnson, "Yacc - Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep, No. 32, Bell Laboratories, Murray Hill, New Jersey (July 1975).

6. M. E. Lesk and E. Schmidt, "Lex - A Lexical Analyzer Generator," Bell Laboratories, Murray Hill, New Jersey.

7. G. Frieder and H. J. Saal, "A Process for the Determination of Addresses in Variable Length Addressing," Communications of the ACM **19**(6) pp. 335-338 (June 1976).

Appendix A

The following is a description of all the intermediate control operators used by the cross compiler. All control operators are listed. Those unique (or modified for) the M68000 compiler are marked with an asterisk (*).

* **TITLE** name

is always the first pseudo-op generated by pass 0 of the compiler and initializes the assembly language file with a predefined set of assembler directives. The name, representing the name of the 'C' source file being compiled, will be used in a title directive to the assembler.

* **LINE** line

initiates the printing of line number comments in the assembly language file. The line is an integer that signifies a line number change in the 'C' source file.

EOFC

signals the end of the intermediate file.

TEXT

causes subsequent code to be compiled into the text section (which contains program instructions).

DATA

causes subsequent code to be compiled into the data section (which contains initialized static data).

BSS

causes subsequent code to be compiled into the bss section (which contains uninitialized static data).

BDATA flag data ...

causes the generation of initialized byte data into the assembly language file. When the integer flag is non-zero, it indicates that the integer data field contains a valid 8-bit quantity of initialized data. A flag value of zero signifies the end of the list.

* **SDATA** flag data ...

signifies the beginning of string data. It is used with the **SDATA1** and **SDATA2** pseudo-ops to generate a complete ascii string in the assembly language file. The flag is used in the same manner as described above. Each integer of data transmits one 8-bit byte that is to be printed in an ascii representation. Only the first 35 to 40 characters of the string are transmitted via this pseudo-op.

* **SDATA1** flag data ...

is identical to the **SDATA** directive except that it represents the intermediate characters of a string. Only 35 to 40 characters are transmitted per single **SDATA1** pseudo-op. Multiple occurrences of the directive are used to complete the transmission of the entire string.

* **SDATA2**

signifies the end of string data.

WDATA flag data ...

is identical to the **BDATA** pseudo-op except that the integer data represents a 16-bit quantity.

* **LDATA** flag ldata ...

is identical to the **BDATA** pseudo-op except that the long integer ldata represents a 32-bit quantity.

CSPACE name size

signifies that the string name references an uninitialized external data area of size bytes in length.

SSPACE size

signifies that size bytes are to be reserved in the current code section for some uninitialized data. It is preceded by a label when it defines static data areas.

EVEN

causes storage to be aligned on an even boundary.

SYMDEF name

signifies that the local symbol represented by the string name will be made globally available to other modules.

* **SBTTL** name

causes a fixed set of assembler listing directives to be output to the assembly language file. Name represents the name of a new 'C' function that will be used in a subtitle directive to the assembler.

* **BGNFUNC** fnumber

is responsible for the generation of the code that handles function prologues. The integer fnumber represents the current function number and is used to create function specific labels.

* **ENDFUNC** fnumber regmask stack

is responsible for the generation of the code that handles function epilogues. The integer fnumber represents the current function number and is used to create function specific labels. The regmask is an encoded integer that represents the registers that were allocated for local variables and temporaries in the function. The stack parameter is the number of bytes of stack space required to hold all user defined automatic and register variables in the current stack frame.

* **SINFO** name number

initiates the printing of a comment into the assembly language file which describes the allocation of a static variable in the subsequent function. The name represents the 'C' variable that will be referenced at

the assembly language level by the label associated with number.

* **AINFO** name number

initiates the printing of a comment into the assembly language file which describes the allocation of a automatic variable in the subsequent function. The name represents the 'C' variable that will be referenced at the assembly language level by the number displacement from the function frame pointer.

* **RINFO** name number

initiates the printing of a comment into the assembly language file which describes the allocation of a register variable in the subsequent function. The name represents the 'C' variable that will be referenced at the assembly language level by the register denoted by the encoded number.

* **SETREG** regmask

redefines the current maximum number of available data and address registers that can be used by the code generator. Regmask is an encoded integer representing these maximum values.

LABEL number

causes generation of an internal label represented by the integer number. All references to the label will be specified using number.

NLABEL name

causes a label represented by name to be printed in the assembly language file. It is generally used to precede the definition of initialized data.

RLABEL name

causes a label represented by name to be printed in the assembly language file. It is generally used to mark the entry point of a function.

EXPR line

causes the the current intermediate code tree to be compiled. It involves handing the current expression tree to the expression optimizer for analysis and then to the code generator for output of assembly code. Line represents the current line number and is used for error diagnostic output.

CBRANCH number cond line

causes the current intermediate code tree to be compiled. It involves handing the current expression tree to the expression optimizer for analysis and then to the code generator for output of assembly code. This directive differs from **EXPR** in that code for a conditional branch to the label represented by the integer number is produced if the truth value of the expression is identical to that of cond. (Thus, if cond is zero and the expression evaluates to false, then a branch to the label is performed). Line represents the current

line number and is used for error diagnostic output.

SWIT default line swlabel swvalue ...

causes immediate generation of code for a **switch** statement. Default is the label number for the default case and line is the current line number that is used in the generation of error diagnostics. As long as swlabel is non-zero, pairs of labels and switch values (swvalue) are gathered together so that a switch table can be generated.

BRANCH number

causes an unconditional branch to the internal label represented by the integer number.

Appendix B

The following is a list of all expression operators that are used in the intermediate code of the cross compiler.

NAME sclass type name

NAME sclass type number

represents a name node in an expression. The sclass represents the storage class for the variable of type type. If the storage class is external, the first representation is used and name represents the ascii name of the external variable. The second representation is used for static, automatic and register variables where the number describes label number, stack frame offset or encoded register respectively.

CON type value

describes a 16-bit signed or unsigned constant of the specified type and value that is to be used in the current expression.

LCON type lval

describes a 32-bit constant of the specified type and value (lval) that is to be used in the current expression.

FCON type acon

describes a 4 word floating point constant of the specified type in PDP-11 floating point notation. The value of the constant is encoded in ascii (acon).

SFCON type value

describes a special floating point constant of the specified type that is correctly represented by its high order 16 bits (value).

binary-operator type

The **binary operator** represents any one of a multitude of binary operators such as PLUS, MINUS, etc. The integer type represents the type of the expression after evaluation.

unary-operator type

The **unary operator** represents any one of a multitude of unary operators such as NEG, ITOL, etc. The integer type represents the type of the expression after evaluation.

Appendix C

The following is an example of the code generation steps and final assembler output for a sample 'C' program.

'C' Program:

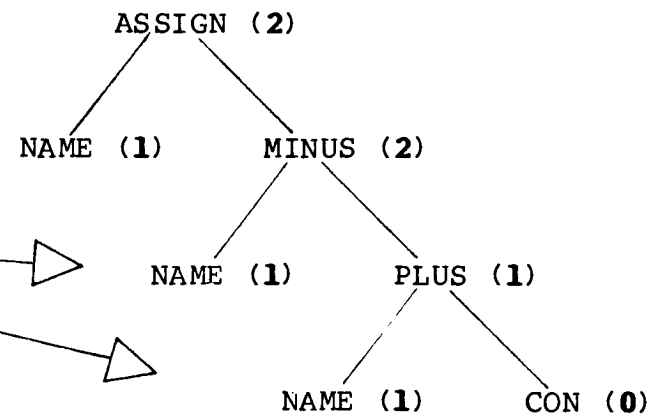
```
main()
{
        int a, b, c;

        a = b - (c + 3);
}
```

Intermediate Code:

```
        TITLE   example.c
        LINE    2
        SBTTL   _main
        SYMDEF  _main
        TEXT
        RLABEL  _main
        BGNFUNC 1
        SETREG  1544
        LINE    3
        AINFO   _a -2
        AINFO   _b -4
        AINFO   _c -6
        LINE    4
        LINE    5
        NAME    11 0 -2
        NAME    11 0 -4
        NAME    11 0 -6
        CON     0 3
        PLUS    0
        MINUS   0
        ASSIGN  0
      * EXPR    5
        LINE    6
        LABEL   1
        ENDFUNC 1 1544 6
        LINE    7
        EOFC
```

Pass 1 Code Tree for EXPR (*)



In the Code Tree on the right, each node is marked with its corresponding Sethi-Ullman number (number of scratch registers required for computation of the node without temporary stores).

The following is an example of the code generation process for the code tree. Levels of indent will represent levels of recursion in the code generator. The current data register is **d0** and the next data register is **d1**.

1. Use efftab to generate code for ASSIGN. The following subtable is matched:

```
%a,n
        S
        IBl     R,Al
```

2. Expand the **S** macro, placing the result of the right subtree in **d0**. Use dregtab to generate code for MINUS. The following subtable is matched:

```
%n,e
        F
        Sl
        IDl     Rl,R
```

3. Expand the **F** macro, placing the result in **d0**. Code is generated as is shown on line 16 of the assembler listing on the next page.

4. Expand the **Sl** macro, placing the result of the right subtree in **d1**. Use dregtab to generate code for PLUS. The following subtable is matched:

```
%n,q
        F
        I'Dl    A2,R
```

5. Expand the **F** macro, placing the result of the left subtree in **d1**. Code is generated as is shown on line 17 of the assembler listing.

6. Expand I'Dl A2,R as is shown on line 18 of the assembler listing.

7. Expand IDl Rl,R as is shown on line 19 of the assembler listing.

8. Expand IBl R,Al as is shown on line 20 of the assembler listing.

Assembler Output:

```
example.c          m68000/mas v02.03 Tue May 12 18:55:27 1981 page  2
Function: main()

       5                                sbttl   Function: main()
       6                                globl   _main
       7                                text
       8 00000000               _main:
       9 00000000  4e56                 link    a6,#LST1
                   fffa
      10 00000004  48ee                 movem.l #LRM1,LRG1(a6)
                   0000
                   0000
      11                        *                                       ; 3
      12                        *       a       = -2(a6)
      13                        *       b       = -4(a6)
      14                        *       c       = -6(a6)
      15                        *                                       ; 4
      16 0000000a  302e                 move.w  -4(a6),d0
                   fffc
      17 0000000e  322e                 move.w  -6(a6),d1
                   fffa
      18 00000012  5641                 addq.w  #3,d1
      19 00000014  9041                 sub.w   d1,d0
      20 00000016  3d40                 move.w  d0,-2(a6)
                   fffe
      21                        *                                       ; 6
      22 0000001a               L1:
      23 0000001a  4e5e                 unlk    a6
      24 0000001c  4e75                 rts
      25           0000         LRM1    equ     $0
                   0000
      26           ffff         LST1    equ     -6
                   fffa
      27           0000         LRG1    equ     0
                   0000
      28
      29                        *                                       ; 7
      30                                data
      31                                nolist  bex
      32                                end

example.c          m68000/mas v02.03 Tue May 12 18:55:27 1981 page 3
user symbol table

L1          000018 T      LRM1        000000        _main       000000 TG
LRG1        000000        LST1        00fffa

Text Segment Size:      1e =      30. bytes
Data Segment Size:       0 =       0. bytes
Bss  Segment Size:       0 =       0. bytes
Total Module Size:      1e =      30. bytes

Errors Detected: 0
```