1974

A methodology for evaluating compiler performance /

James Patrick Clancy
Lehigh University

A METHODOLOGY FOR EVALUATING COMPILER PERFORMANCE

by

James Patrick Clancy

A thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

1974

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

April 24, 1974
Date

M Wayne Shumaker
Professor in Charge

Chairman of the Department
of Industrial Engineering

ACKNOWLEDGEMENTS

The author wishes to express his appreciation to Dr. W. Shively, who served as facility advisor, for his advice and encouragement during the preperation and writing of this thesis.

Appreciation is also extended to Mr. G. E. Whitney and Dr. C. W. Cheng of Western Electric Company's Engineering Research Center for suggesting the topic and providing technical guidance and assistance throughout this project.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

The introduction of compilers and high-level (user oriented) programming languages has made the computer available as a tool to those with only limited programming knowledge. Consequently, a great deal of effort was and still is being devoted to developing techniques to improve the object code produced by compilers. However, the measurement of overall compiler performance in terms of the cost of both compiling and executing a wide range of programs has been neglected, primarily due to the variations in machine and program environments. A methodology is proposed which can be applied to the measurement of compiler performance. The methodology consists of three parts: 1) define an abstract computer which can mimic the machine environment by using parameters; 2) design a compiler for this abstract computer; 3) design an experiment that will allow the application of statistical methods to the measurement of performance for different machine and program environments. The methodology is tested using a subset of FORTRAN describing arithmetic operations and a simulated compiler. A simplified abstract computer is defined and a compiler is described which can generate code for this computer. Three optimization techniques for allocating registers in the compiler and the method of simulating

compilation are described. An experiment is described which can serve as a prototype for implementation of this methodology. Analyses of the experimental results using an analysis of variance test and Duncan's multiple range test are provided. The results indicate that this methodology can be a valuable tool in measuring compiler performance.

CHAPTER 1

1.0 Introduction

A translator is a program which accepts as input a program in one language called the source language and produces as output an equivalent program in another language called the object language. If the source language is a high level programming language and the object language is assembly or machine language, the translator is referred to as a compiler and the object program is referred to as object code [3]. Compilers, by allowing the use of high level programming languages, have made the computer available as a tool to those with only limited programming knowledge, such as engineers, scientists and businessmen. However, early compilers produced code which was not as efficient in terms of execution time and storage allocation as code written in assembly or machine language by experienced programmers.

A great deal of effort was and still is being devoted to developing techniques to optimize (a more accurate word is improve) the code generated by compilers. These techniques can generally be classified as machine dependent and machine independent. Those techniques which manipulate the source language are machine independent. The elimination of common subexpressions is an example of this type. Those techniques which manipulate the object

code are machine dependent. Register assignment algorithms are an example of this type. As a result of the work in this field, most present day compilers for the widely used programming languages can produce code which is as good as hand coding in a fraction of the time that hand coding requires. There remains, nevertheless, a large and growing number of special purpose programming languages which still require the design of compilers.

As in most design work, trade-offs must be made. The compiler designer must decide what level of code optimization is desired beyond the generation of basic executable code. The more optimization techniques included in the compiler, the more complex it becomes with resulting increases in storage requirements and compile time. The designer must trade-off these increases against the hoped for improvements in the storage requirements and/or execution time of the object code produced. These decisions require some method of determining performance levels of different compilers (or different versions of the same compiler).

1.1 Problem Description

If the performance measure is the cost, as yet undefined, for compilation and execution of a program, then, theoretically, a measure of performance could be written as

$$Ct = Cc + hCe$$

where: Ct is the total cost of compiling and executing.

Cc is the total cost of compiling.

Ce is the total cost of executing.

h is the number of executions.

If these costs were known, then a decision could be made as to the relative merits of different compilers as a function of the number of executions of the object code. However, since these cost elements vary for each compiler program combination, this approach is not very practical.

Another difficulty in measuring performance is the variation in computer environments. As an example, consider the differences between an educational and business environment. In an educational environment involved in the teaching of programming languages, the programs would typically be small, require little time to execute and require few repeated executions. In the business environment, the programs, which might involve payroll, inventory or accounting applications, would typically be large, require much execution time and are executed periodically. The best compiler for the educational environment would have little, if any, optimization, since compile time would probably exceed execution time. However, the business environment would require a very high level of optimization, since the

savings in repeated execution costs would more than offset the increased compile time. This variation in environments also extends to machines. Due to the machine dependence of many optimization techniques, the performance of a compiler including these techniques will vary depending upon the target machine.

Although there is an extensive literature on compiler optimization, there is none on the measurement of compiler performance. The typical paper [1,7,8] which describes a technique for optimization also defines a criteria to measure the performance of the technique. This criteria is usually the total number of object code instructions generated or the number of memory references required to perform a specific operation. Since the effect of optimization techniques on a specific compiler depend upon not only the existence of the condition which is to be improved but also the frequency with which this condition occurs in the compiler environment, a method of measuring overall performance which considers these factors is required.

1.2 Methodology

An approach to this problem is to develop a methodology for evaluating compiler performance. The methodology proposed consists of three parts:

1) Define an abstract computer which can mimic the machine environment by means of parameters.
2) Design a compiler (or compilers) for this abstract computer which can generate code which (with some intermediate processing) can be executed on the target machines.
3) Design an experiment that will allow the application of statistical methods to measure the performance of this compiler for different machine and program environments.

1.2.1 Abstract Computer Definition

The abstract computer developed is called the STAR ABSTRACT COMPUTER (SAC). Its definition requires the specification of both an architecture and instruction set. These are described in Chapter 2. Since the target machines, IBM 360 and PDP-11, differ markedly in instruction capability and architecture, in the interests of simplicity, the definition of SAC was biased towards the IBM 360. Figure 1 lists some of the characteristics of the target machines. One advantage of this abstract computer, quite apart from the subject of this thesis, is its tutorial uses in teaching machine concepts without the necessity of learning assembly language or pecularities of specific machines.

	PDP-11	IBM 360
ADDRESSING		
	Each operand consists of a register number and an address mode indicating how that register is to be used. There are eight address modes. The register mode indicates that the value of the operand is in the specified register. The index mode indicates that the contents of the register specified is to be added to the contents of the word following the instruction to form the address of the operand. Two modes are used to increment and decrement registers to facilitate stack operations. In addition, all four modes can be used to specify an indirect address where the address obtained from the instruction points to the address of the operand rather than the operand itself. The use of the program counter register in an operand address permits immediate addressing where the operand itself is in the word following the instruction in memory.	Base-displacement addressing is used. Register operands are addressed by their number. Operands in storage are addressed by the sum of a displacement included in the instruction and the contents of a base register whose number is included in the instruction. The address may also include the contents of an index register specified in the instruction. The differentiation between a register number used as a base or index register and a number used to address a register is made by the use of different instructions for the different locations of operands. Indirect addressing is not available. Immediate addressing is available but the operand contained in the instruction is limited to one byte in length.

FIGURE 1. SOME CHARACTERISTICS OF THE PDP-11 AND IBM 360 COMPUTERS.

PDP-11	IBM 360
INSTRUCTION SET	
Both single and double operand instructions are used. Operands are designated as either a source or destination operand. Results of an instruction are left in the address specified by the destination operand. Neither operand is required to be in a register. Instructions include register to register, register to storage and storage to storage operations. There is one set of instructions which are used for the various combinations of addressing modes. For example, the "MOV" instruction can move an operand from one register to another, from a storage location into a register, from a register into a storage location or from one location in memory to another, depending upon the address modes of the operands.	Double operand instructions are used. No single operand instructions are available. There are five types of instructions which are used depending upon the location of the operands. For example, RR type instructions are used when both operands are in registers, RS and RX type instructions are used when the first operand is in a register and the second operand is in memory, and SS type instructions are used when both operands are in memory. The SI type of instruction is used when one operand is in memory and the other operand is contained in the instruction (this is the immediate operand and is limited to one byte in length).
REGISTERS	
Eight General registers are available numbered 0-7. Register 6 is used as a stack pointer and has hardware functions associated with	Sixteen general registers are available numbered 0-15. Registers 0, 1, 13, 14 and 15 are reserved by convention for such uses as passing

FIGURE 1.(cont'd) SOME CHARACTERISTICS OF THE PDP-11 AND IBM 360 COMPUTERS.

PDP-11	IBM 360
REGISTERS (cont'd)	
interrupts and subroutine calls. Register 7 is the program counter register. (Some configurations can expand to 16 registers - 3 sets of 6 general purpose registers, 3 stack pointers and 1 program counter register.)	subroutine parameters. The program counter register is not addressable by the programmer.

FIGURE 1.(cont'd) SOME CHARACTERISTICS OF THE PDP-11 AND IBM 360 COMPUTERS.

1.2.2 Compiler Design

Since the design of an actual compiler is beyond the scope of this thesis, a compiler was simulated in FORTRAN. Chapter 3 describes the compiler and the methods used to simulate its various functions. A subset of FORTRAN describing arithmetic operations was chosen as the source language for the simulated compiler in order to allow the sampling of arithmetic statements from actual FORTRAN programs.

1.2.2.1 Source Language

This subset of FORTRAN consists of scalar variables and constants of type integer. The allowable symbols are the alphabetic characters, A-Z, and the numeric characters, 0-9. Variable names may be single alphabetic characters only. Although no execution of the code from the simulated compiler is performed, scalar constants are included for completeness and for possible future implementation. The operators allowed in this language are

+ - / *	Arithmetic operators
( )	Precedence operators
=	Assignment operator
;	End of statement operator

The operations of addition, subtraction, multiplication, division and unary minus are the only arithmetic operations which will be considered. Since the

same operator is used to represent subtraction and unary minus, the latter must be determined from the context of the arithmetic statement. This language contains no branches of any kind so that any section of code has only one path to enter and one path to exit. This type of code is referred to as straight-line code.

A program in the source language consists of a sequence of arithmetic statements which have the general form:

A = B;

where A is a variable, B is an expression, = is the assignment operator and ; is the end of statement operator. At execution time, the expression B is evaluated and the resultant value is assigned to the variable A. An expression is a sequence of scalar variables or constants of type integer seperated by arithmetic operators and parenthesis in accordance with mathematical convention and the following rules:

1. An expression may consist of a single basic element such as a scalar variable or constant.
2. Compound expressions may be formed by using the arithmetic operators to combine basic elements.
3. Compound expressions must be constructed according to the following rules:

a. Any expression may be enclosed in parenthesis and considered to be a basic element.

b. Expressions which are preceeded by a - sign are also expressions. (For convienence, leading + signs were not allowed)

c. If the precedence of operations is not given explicitely by parenthesis, it is understood to be the following (in order of decreasing precedence):

* and /	multiplication and division
+ and -	addition and subtraction or unary minus

In the case of operations of equal precedence, the calculations are performed from left to right.

d. No two arithmetic operators may appear in sequence.

Arithmetic expressions described by the above rules are said to be in infix notation.

1.2.3 Experiment

The last step of this methodology is to design an experiment and select apropriate statistical tests which will serve as a prototype for a systematic evaluation procedure. These tests should have the ability to detect differences among alternative compilers (or versions of

the same compiler) due to different machine and program environments as well as indicating preferred alternatives. Inherent in this step is the definition of a cost function to characterize the compiler performance. The costs dealt with are limited to simulated execution costs. The compilation costs were not simulated since these would depend upon the skill of the compiler designer rather than the execution time of the simulated compiler. Chapter 4 describes the cost function, the experiment and the statistical tests used.

CHAPTER 2

2.0 Definition of Star Abstract Computer

One of the reasons for using an abstract computer is to reduce the variability in the machine environment. If the common features of the target machines are identified and included in SAC, then optimization techniques which depend upon these features become "machine independent", at least for the machines of interest. Since the purpose of this thesis is to describe a methodology rather than develop the ideal SAC, to facilitate presentation, the architecture described below is biased towards the IBM 360, particularly in the method of addressing storage. Similiarly, the instruction set is limited to those operations which are required for integer arithmetic. However, in an actual implementation of this methodology, the desired instruction set would be the maximal intersection of the instruction sets of the target machines. Appendix A contains a list of symbols used in this and subsequent chapters.

2.1 Architecture

2.1.1 Memory Structure

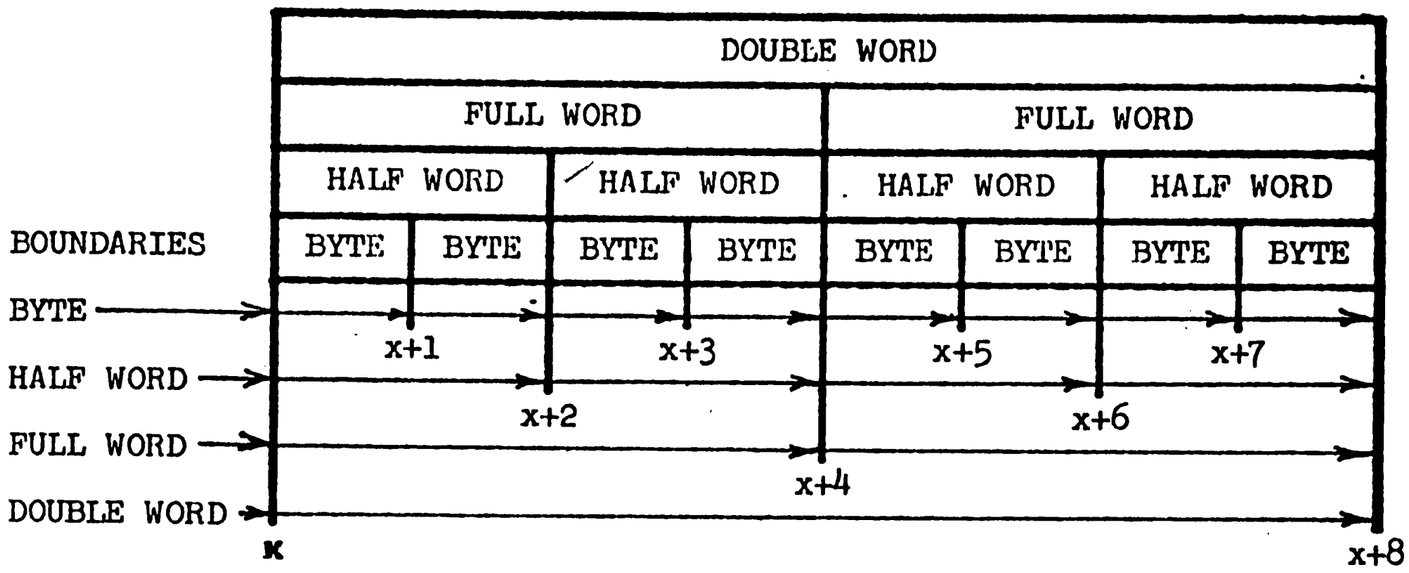
The basic unit of memory is a word although the word size in terms of bytes (one byte equals eight bits) will not be specified. The two computers of interest use the byte as the basic unit of storage (i.e. the smallest

directly addressable unit). However, their differences in word size makes a common ground difficult and would unnecessarily complicate the symbolic nature of SAC. The IBM 360 [10] uses four bytes per word (32 bits) and allows half-word, full-word and double-word references, while the PDP-11 [11] uses two byte (16 bit) words and allows only half-word and full-word references. Figure 2 shows the memory formats of the target machines. Each word in memory is numbered consecutively from zero to some upper limit which is dependent upon the computer configuration available. This number constitutes the actual address of each word.

2.1.2 Addressing

Each word in memory is referenced by a number which indicates its actual address. In order for an instruction to reference an address, it must be able to contain a number as large as the highest address available. This could require extremely large instructions (large in the sense of the number of bits required to express them). In order to make all memory locations addressable, while at the same time limiting the size of the instructions, a scheme, which is derived from the IBM 360, is used. This scheme uses a register to specify a base address (BA) and the instruction to specify a displacement constant (DC), which, when added to the base address, provides the actual

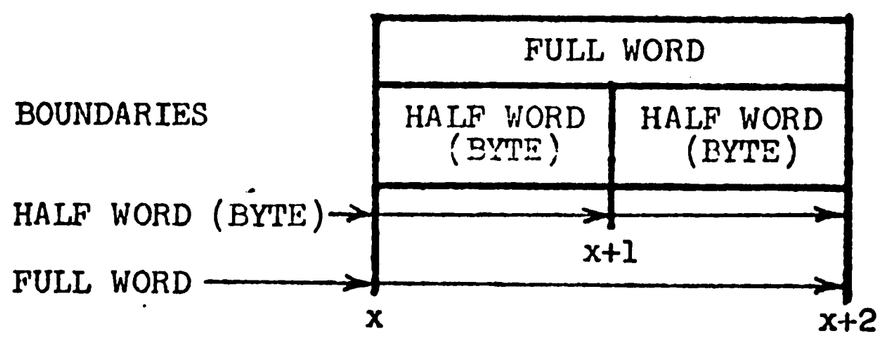
IBM 360 MEMORY FORMAT



ALLOWABLE ADDRESSES

BYTE	x = p
HALF WORD	x = p*2
FULL WORD	x = p*4
DOUBLE WORD	x = p*8

PDP-11 MEMORY FORMAT



ALLOWABLE ADDRESSES

HALF WORD(BYTE)	x = p
FULL WORD	x = p*2

x,...,x+1 ARE NUMERIC MEMORY ADDRESSES (ACTUAL ADDRESSES)
p = 0,1,2,...

FIGURE 2. MEMORY FORMATS OF THE IBM 360 AND PDP-11.

address required. Since registers are a full word in size, this provides the ability to reference the entire memory. (The IBM 360 requires only 24 of the 32 bits available for addressing). This method also uses the contents of another register called an index register. The use of an index register allows modification of an instruction address without permanently altering the instruction in memory by the use of arithmetic operations on the contents of this register. Thus, an address can be symbollically represented in the form k(i,j) where k is the displacement constant, i is the base register number and j is the index register number. The actual address specified is then

actual address = k + contents of base register i
+ contents of index register j

The actual address calculation is done in an address register, which is not addressable by the programmer, and neither k nor the contents of registers i or j are altered. By convention, if either i or j is zero, this means the number zero, rather than register zero. Thus k(0,0) addresses the k-th memory location. The instructions must be able to hold a displacement and two register numbers. The largest displacement allowed in SAC, K(=max(k)), is dependent upon the number of bits available in the instruction. In the IBM 360, there are 12 bits to

hold the displacement constant. Thus, 4096(=2^{12}) bytes or 1024 full-words can be addressed from a given base register. The PDP-11 has the ability of using a full-word for the displacement constant and therefore can address 65536(=2^{16}) bytes or 32768 full-words from a given base register. Since our basic memory unit is the word, K is a parameter of SAC. For the IBM 360, K = 1023 and for the PDP-11, K = 32767.

Another advantage of base-displacement addressing is relocatability. Since the available memory locations, which are unknown at compile time, are determined when the object code is loaded, the addresses used in the object code can be altered by changing the base address values. Thus, the compiled program can be made relocatable to any area of memory. This ability is required in a multi-programming environment. Although this scheme allows for the use of an index register, this need not be used when dealing only with scalar variables and straight-line code. Consequently, a short-hand notation will be used such that k(i) is equivalent to k(i,0).

In addition to memory locations, data may be stored in registers. The address of a register is indicated by a unique number assigned to each register. Confusion with memory addresses is avoided by the form with which register addresses are specified. The address of a

register will be indicated by Rn which addresses the register numbered n. (The PDP-11 indicates register addresses by specifying a register number and an address mode indicating that the operand is in that register. The IBM 360 uses different instructions depending upon whether the operands are in memory or registers.)

2.1.3 Registers

Registers are hardware devices used for the temporary storage of one or more words to facilitate arithmetic, logical or transferral operations in the computer. They are frequently referred to as "fast memory" because of their extremely fast access time.

Those registers which are used by the control unit of the computer to control the execution of the program are generally not addressable by the programmer. These include the program counter (PC) register, the instruction register and the address registers. The PC register holds the actual address of the instruction currently being executed. Upon completion of execution, this register is automatically incremented by 1, thereby pointing to the next instruction to be executed. The control unit uses this register to load the next instruction into the instruction register. This register controls the actual address calculations using the address registers and the execution of the operation code portion of the

instruction.

The registers that are available for the programmer's use are referred to as general purpose registers. The term general purpose register refers to those registers which can be used as accumulators to perform arithmetic operations or as base and index registers to address storage locations.

A minimum of two general purpose registers are required for arithmetic operations when base-displacement addressing is used. This can be understood from the situation when the value of one operand is in a register and the other operand requires a register to address its value in storage. However, the multiplication and division operations in both the IBM 360 and the PDP-11 require two consecutive registers. In multiplication of two full words, the result will be a double-word (i.e. 16 bits times 16 bits yields 32 bits) requiring two full-word registers. In division, one register is required to hold the quotient and another to hold the remainder. Since these are the target machines of SAC, a minimum of three general purpose registers will be assumed.

Since the source language is limited to scalar variables of type integer, the remainder in division is not required due to truncation. In multiplication, it is usually sufficient to retain only the low-order register.

However, if the high order register must also be retained, this requires scaling and arithmetic shift operations to save the contents of both registers in a full-word of storage. Since this requirement would unnecessarily complicate the instruction set, it is assumed that the low order register need only be saved and the high order register can be ignored. Even though these operations ignore the result of the additional register, it is still included to facilitate compatibility with the target machines. The maximum number of general purpose registers, N(=max(n)), is dependent upon the target machines and is a parameter of SAC.

In addition to the N general purpose registers, another register, with limited use, is required. This register is assigned the special function of holding the address of the first memory location available at execution time. This register can be used only as a base register and not for arithmetic operations. It is assigned the address (N+1), and, for convienence, will be referred to as the R-th register. The purpose of this register will be described in the next chapter. A diagram of the SAC memory structure is shown in Figure 3.

2.2 Instruction Set

The instruction set for SAC uses both single and double operand instructions with each instruction

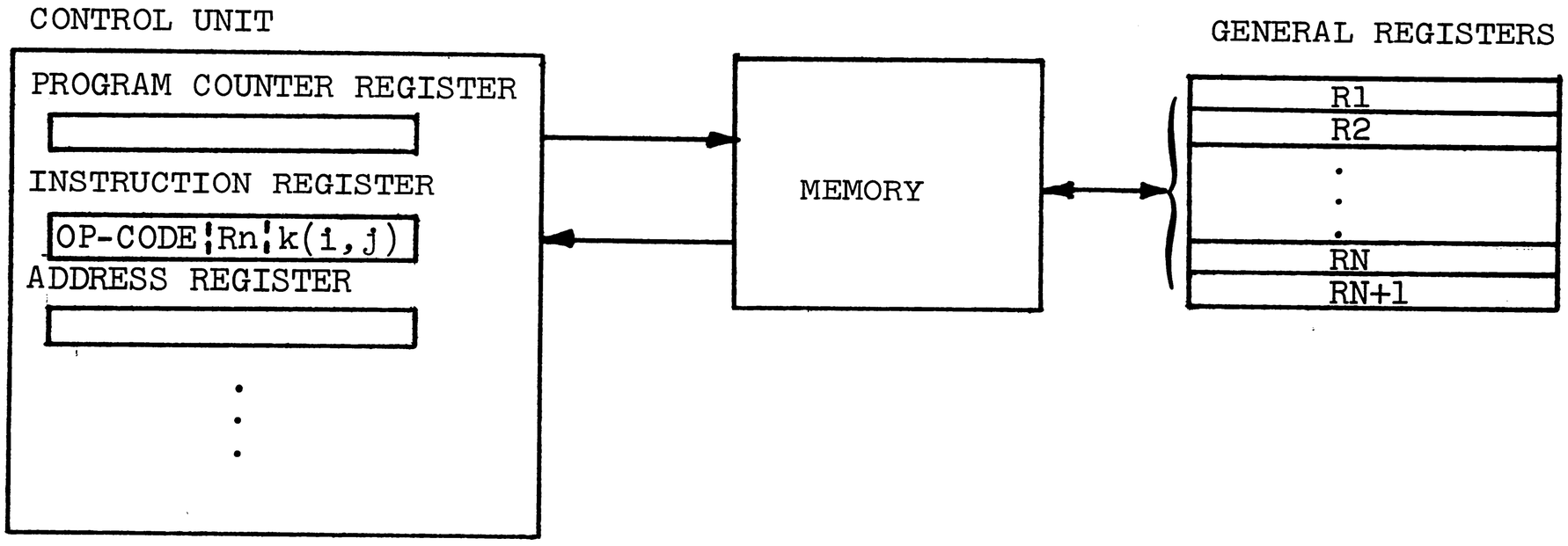


FIGURE 3. THE STRUCTURE OF STAR ABSTRACT COMPUTER (**SAC**)

requiring a full word of storage. register to register (R/R) operations as well as register to storage (R/S) operations are available with a single set of instructions used for both types of operations. (The IBM 360 uses different instructions for the same operation depending upon the location of the operands). Storage to storage operations are assumed not available. All operations require that the first operand be in a register. The following notations are used to explain the instructions:

@ Either a register, Rn, or a memory location, k(i). In the context of an instruction, "@" should be read as "the address specified by @". In the instructions which follow, @ may not be the same register as the first operand but may use that register as a base register. For example, the instruction LOAD R2,R2 is not allowed, but LOAD R2,k(2) is allowed. This is possible since the address calculation takes place before the operation code portion of the instruction is executed and does not change the contents of the specified base register.

[Rn,Rn+1] The concatenation of registers n and n+1. This means that the indicated register

pair functions as one register.

2.2.1 Single Operand Instructions

NEG Rn — Changes the sign of the contents of Rn.

2.2.2 Double Operand Instructions

LOAD Rn,@ — Places the contents of @ into Rn. The previous contents of Rn are lost and the contents of @ are unchanged.

STORE Rn,k(i) — Puts the contents of Rn into memory location k(i) (i may equal n). The contents of Rn are unchanged and the previous contents of k(i) are lost.

ADD Rn,@ — Adds the contents of @ to the contents of Rn. The contents of @ are unchanged and the previous contents of Rn are lost.

SUB Rn,@ — Subtracts the contents of @ from the contents of Rn. The contents of @ are unchanged and the previous contents of Rn are lost.

Both target machines require that multiplication and division be performed in a consecutive even-odd pair of registers. Therefore, the register, Rn, shown as the first operand in the subsequent instructions indicates the even-odd pair, [Rn,Rn+1]. (The PDP-11 also allows the use of an odd register alone for multiplication, but restricts the result to that register.)

MULT Rn,@ — Multiplies the contents of Rn+1 by the contents of @ and leaves the result in [Rn,Rn+1]. (Rn+1 is assumed to hold the only meaningful result.) The contents of @ are unchanged and the previous contents of [Rn,Rn+1] are lost.

DIV Rn,@ — Divides the contents of [Rn,Rn+1] by the contents of @ and leaves the quotient in Rn+1 and the remainder in Rn. (The PDP-11 reverses the quotient and remainder.) The contents of @ are unchanged and the previous contents of [Rn,Rn+1] are lost.

CHAPTER 3

3.0 Introduction

A compiler can be described in two phases. The first is an analysis of the source program which decomposes the source language into its constituent parts. The second is a synthesis of the object program using code generated from these basic parts. Figure 4 shows a diagram of a compiler. As information is gained at the local level in the analysis routines, it is stored in tables to make it available to all phases of the compiler.

3.1 Compiler Description

3.1.1 Tables of Information

The number and types of tables is determined by both the preferences of the compiler designer and the amount of information required for code generation. One table which is common to all compilers is the symbol or name table. This table holds the variable names of the source program and their attributes which include such things as type, precision, symbolic address and any additional information required for code generation. The compiler used in this thesis requires two additional tables. One is a constant table which holds the value, type, symbolic address, precision and other pertinent information about the constant. The other is a temporary variable table. Temporary variables are those used by the compiler to

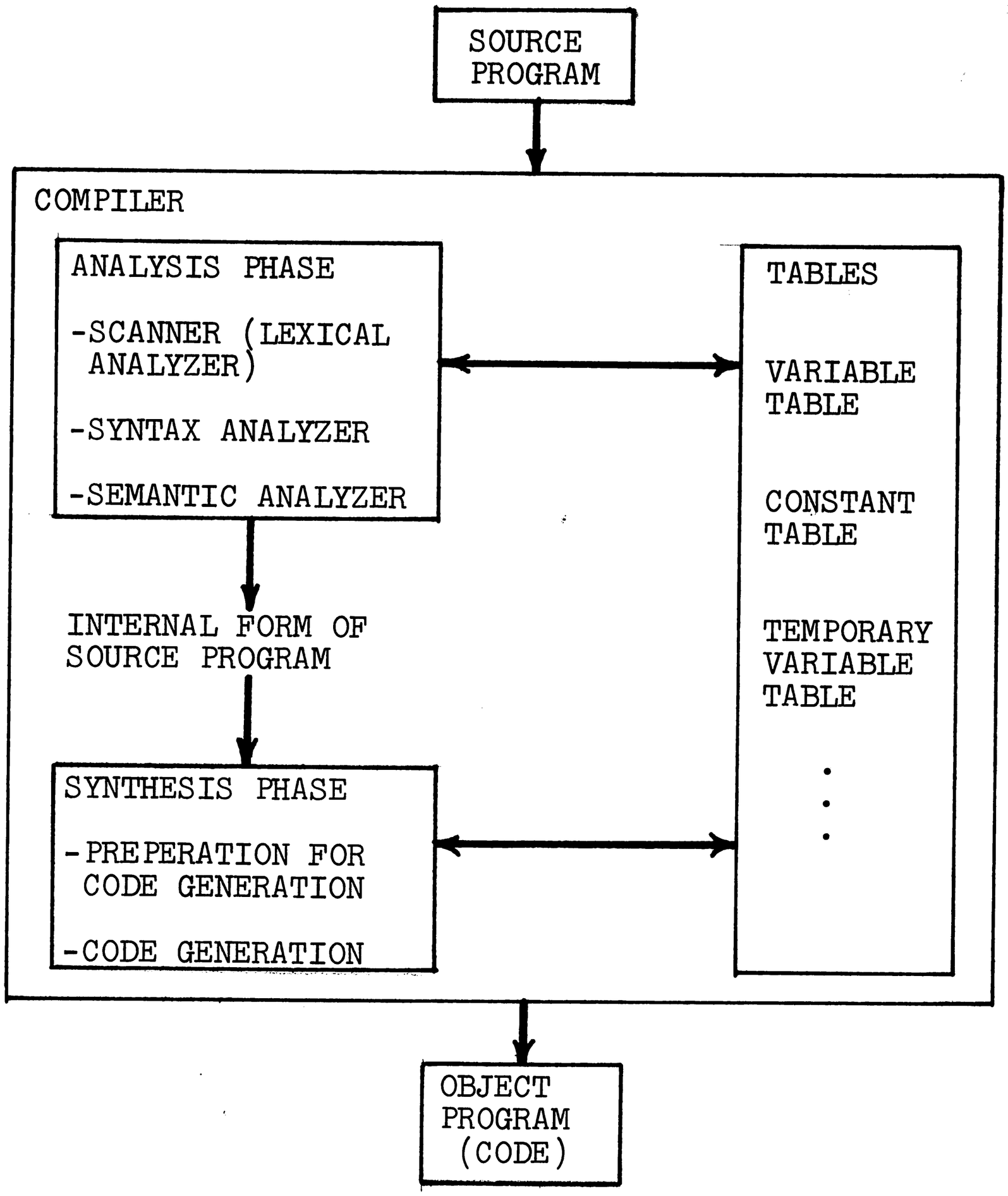


FIGURE 4. CONCEPTUAL DIAGRAM OF A COMPILER.

represent the intermediate results generated when evaluating arithmetic expressions. The temporary variable table holds the type, symbolic address, precision and any other information required for code generation.

Since the source language allows variables and constants to have representations of varying length, this compiler replaces variables and constants by unique fixed length designations consisting of two integers. The first refers to a particular table and the second refers to an entry in that table. This form is also used for temporary variables. Thus variables, constants and temporary variables, all of which are called operands, can be denoted as follows:

(1, Pointer to variable table entry)	(1,v)
(2, Pointer to constant table entry)	(2,c)
(3, Pointer to temporary table entry)	(3,t)

where the form on the right is a shorthand notation for these operands.

3.1.2 Symbolic Addresses

Since the memory locations that will be allocated at execution time are not known at compile time, symbolic addresses are assigned to variables, constants and temporary variables. These addresses are kept in the tables and used during code generation. Each address is represented by an ordered pair of integers, (d,k), where d

is the number of a data area and k is the displacement within that data area.

SAC memory is logically divided into contiguous blocks of words called data areas. The number of words in a data area is determined by the maximum displacement, K, allowed in an instruction. For example, the IBM 360 allows displacements up to 1023 full words. Thus, a data area for the IBM 360 consists of 1024 consecutive words of storage. The maximum number of data areas available, D', is determined by the memory capacity of the target machine and the size of the displacement allowed in that machines instruction set. Thus, if the IBM 360 configuration had a memory capacity of 131,072 full words, then the maximum number of data areas would be 121(=131072/1024). Data areas are numbered from zero to (D-1) where D is the number of data areas required for a given program.

Then D(K+1) is the size of the memory needed for this program (where K+1 is the number of words in a data area). If a contiguous memory block of size D(K+1) is allocated at execution time and the actual address of the first word in memory is denoted by T, then the actual address corresponding to the symbolic address, (d,k), would be

$$\text{actual address} = T + d(K+1) + k$$

The quantity, (T + d(K+1)), is the address of the first word of data area d. This quantity is the base

address of data area d and must be in a register in order to implement the addressing scheme described in Chapter 2. In order to implement base-displacement addressing, the first D words in data area 0 are reserved. Each of these D words contains the address of the data area whose number corresponds to the displacement of the word in data area 0. (i.e. the word whose symbolic address is (0,d) holds the base address of data area d) When the memory space is allocated, the operating system loads these addresses into the reserved area in data area 0 and also loads the starting address, T, into the R-th register (recall that the R-th register is the (N+1)-st register). This allows non-contiguous memory blocks to be allocated to different data areas. Base addresses can be loaded into any register, Rn, by using the R-th register as a base register and the data area number as a displacement with the following instruction

LOAD Rn,d(R)

3.1.3 Analysis Phase

The analysis phase is made up of the scanner, syntax and semantic analyzers. The scanner reads the source program character by character and builds the symbols (tokens) of the source language. These symbols might consist of variables, constants, key words, single and double character operators, etc. The scanner also stores

variables and constants in their appropriate table, replaces them with their fixed length form, identifies operators and replaces them with their internal form (usually an integer number). The syntax analyzer checks the symbols for syntactic correctness, enters their attributes in the tables and, when higher level syntax entities, such as arithmetic expressions, are encountered, calls the semantic routines to convert them to an internal form.

3.1.4 Internal Form of the Source Program

The internal form of the source program depends primarily upon the preferences of the compiler designer. However, some optimization techniques can be implemented more efficiently with particular internal forms.[6] For the SAC compiler, the infix notation of the source language is converted to quadruple notation which was selected because its simplicity aids the tutorial uses of SAC. Quadruple notation consists of a sequence of quadruples of the form

(operator,first operand,second operand,result)

which, when taken in sequence, describe the operations needed to evaluate an arithmetic expression. The correspondence between the arithmetic and assignment operators and the quadruples which represent them are as follows:

A+B	(+,a,b,t)
A-B	(-,a,b,t)
A*B	(*,a,b,t)
A/B	(/,a,b,t)
-B	(-,b, ,t)
A=B	(=,b,a,)

where upper case letters indicate symbols of the source program and lower case letters indicate their internal form. The operators themselves are used instead of their internal form for convienence and t represents the internal form of a temporary variable. An example of this notation is shown below for the arithmetic statement: A=B*C+D;

```
(*, b, c,t1)
(+,t1, d,t2)
(=,t2, a,  )
```

The notation ti indicates the i-th entry in the temporary table. These are assigned sequentially to quadruples beginning with t1 for each arithmetic statement processed. When code has been executed for an arithmetic statement (a sequence of quadruples the last one of which must be an assignment quadruple), the temporary variables used for that statement have no further use. Thus, space is allocated in the compiler for the temporary variable table based upon the maximum number of temporary variables that one statement can generate. This same principle is

applied to allocating storage for temporary variables at execution time by reserving a sufficient number of words in data area 0. The temporary variable table contains preassigned symbolic addresses. A temporary variable is then made addressable by obtaining the displacement from its entry in the temporary variable table and using the R-th register as a base register. A procedure is described in Appendix B which will convert an arithmetic statement in infix notation to a sequence of quadruples based upon the precedence rules described in Chapter 1.

3.1.5 Synthesis Phase

The synthesis phase consists of preperation for code generation and code generation itself. Applying optimization techniques to the source program (in its internal form - quadruples) is an important part of the preperation for code generation. These optimization techniques are machine independent. They will not be used nor discussed further in this thesis. This is followed by the code generation routines which generate target code from the internal form of the source program. Optimization techniques may be applied during or subsequent to the generation of object code. These are machine dependent techniques. Frequently a compiler will combine both techniques. The compiler used in this thesis converts the internal form directly to object code. Several compiler

techniques for generating code are discussed below. These techniques are not intended to be unique but rather to provide several distinct levels of object code for comparison.

3.2 Code Generation Techniques

Code generation for a quadruple can be divided into three steps. The first step generates code to make the operands addressable. This consists of insuring that either the operand itself or its base address is in a register. The second step generates code to load one of the operands into a register. This consists of insuring that the first operand is in a register. The third step generates code to perform the required operation. The implementation of these steps can be very simple or very complex depending upon the design of the compiler.

A code generation procedure, which follows these three steps, is discussed in the next section. This procedure requires six variables. Four variables, OPER, OPND1, OPND2, and RSLT, are designated for the elements of quadruples where OPER is the operator, OPND1 and OPND2 are the first and second operand respectively and RSLT is the temporary variable identifying the result of the operation. Two other variables, ADDR1 and ADDR2, are used to hold the addresses of the first and second operand respectively. These two variables may hold addresses of

the form Rn, k(i) or blank. All six variables are global since they must be available to all procedures of the compiler.

3.2.1 Basic Executable Code

This procedure assumes that any operand or base address required in a register must always be loaded from memory. The following steps describe the procedure and are implemented sequentially unless otherwise indicated.

Step 1 initializes the address variables and reads the next quadruple.

1. Initialize ADDR1=" " and ADDR2=" " and read the first (next) quadruple. If there are no remaining quadruples, code generation for the program is complete.

Step 2 generates code to make the first operand addressable.

2. Generate "LOAD R3,d1(R)" (where d1 is the data area number of the first operand) and set ADDR1="k1(3)" (where k1 is the displacement of the first operand).

Step 3 generates code to make the second operand addressable.

3. If OPND2 equals " ", go to step 4. Otherwise generate "LOAD R1,d2(R)" (where d2 is the data area number of the second operand) and set

ADDR2="k2(1)" (where k2 is the displacement of the second operand).

Step 4 generates code to place the first operand in a register.

4. Generate "LOAD R3,ADDR1". If OPER equals either "*" or "/", set I=2. Otherwise set I=3.

Step 5 generates code to perform the required operation using the procedure INSTR(I) where I is a register number specified by the calling procedure. This procedure will generate the appropriate machine language instruction for each quadruple based upon the operator involved. A detailed description of this procedure is given in Appendix C.

5. Call INSTR(I).

Step 6 generates code to save the result of this quadruple and is required by the basic nature of this algorithm.

6. If OPER equals "=", go to step 1. Otherwise generate "STORE R3,kt(R)" (where kt is the address in the temporary variable table entry pointed to by RSLT) and go to step 1.

Even though the number of available registers is a parameter of SAC, the registers used by this algorithm are explicitely stated. In this procedure the compiler does not keep track of the contents of the registers.

Therefore, the results of each quadruple must be stored before the next quadruple is encountered. One result of this is the lack of change in the object code if additional registers are available. This algorithm will generate the same code for three registers as for thirty registers. This lack of information about the contents of the registers prohibits the compiler from making a decision as to which register to chose when one is needed.

3.2.2 Register Map [3]

It is intuitively appealing to leave certain frequently used values in registers, thus eliminating some LOAD and STORE instructions. In order to accomplish this, the compiler must keep a running record of the contents of the registers, a so-called register map, by using an array named RV (Register Value). When the n-th element of this array, RV(n), contains an operand, this means that the value of that operand is in register n. We have already defined operands to describe variables, constants and temporary variables. Since the registers can also hold data area addresses, we define an additional operand as follows:

(4,Data area number) (4,d)

If RV(n) contains (4,d), this indicates that the address of data area d is in register n. Thus RV(N+1) always contains (4,0). Each time code is generated which

changes the contents of a register, say Rn, the array element RV(n) must be updated. This technique makes it unnecessary to store the result of each quadruple since the result will be identified by an operand of the form (3,t) in RV.

A register can now be classified as free if it may be used without the necessity of saving its contents. Those registers which are empty are obviously free. Those which hold the value of operands of type (1,v), (2,c) and (4,d) are also free since these values are always available in memory. The only operand which must be stored when the register holding its value is required is of type (3,t). Thus Rn is free if it does not hold the value of a temporary variable or, equivalently, if RV(n) does not hold an operand of type (3,t).

When a register is required, this technique chooses the first (lowest numbered) free register and avoids storing temporary variables unless no free registers are available. When a temporary variable in register n is to be stored, the storage address is obtained from its entry in the temporary variable table.

When implementing a register map, care must be taken to preserve the addressability of operands. If the n-th register were used to address the first operand (either by holding the value of the operand or by holding the base

address of the operand), and there were no other free registers, then, if the address of the second operand required a free register, the addressability of the first operand would be destroyed. To avoid this problem a register must be tested to insure that it is not referenced by either one of the address variables, ADDR1 and ADDR2, before it is reused or its contents stored.

3.2.3 Register Status [3]

The previous method of selecting a free register does not differentiate between a register that is empty or one that contains variables or constants. Another technique which is used in combination with the register map provides this capability by using an array called RS (Register Status) to keep track of the importance of the contents of each register. The possible values of the n-th element of this array, RS(n), are:

0 Register n is empty and may be used without storing its contents.

1 Register n holds either a constant, variable or data area address whose value is in memory and may be reused without storing its contents.

2 Register n holds a temporary variable whose value is not in memory and must be stored before being reused.

Each time the contents of a register are changed not

only is the RV array updated but also the RS array. When this technique requires a free register, it first searches for a register with status 0, then status 1 and finally status 2. Thus it will show a preference for using empty registers, thereby leaving constants, variables and base addresses in registers. This increases the liklihood that these values can be reused in subsequent operations without the necessity of loading them from memory.

3.2.4 Assigned Base Register

Since symbolic addresses are assigned sequentially as the variables and constants are encountered in the source program, variables and constants occuring in adjacent areas tend to be stored in the same data area. This technique takes advantage of this fact by assigning a specific register, RN, to be used as a base register. This register is loaded with the base address of the predominant data area and is not used for arithmetic operations. The method of determining the predominant data area for a particular block of coding and loading its address into the assigned base register is beyond the scope of this paper. This technique has the advantage of reducing the requirements for loading data area addresses but has the disadvantage of reducing the number of registers available for arithmetic operations. Because of this disadvantage, the minimum number of registers

required when using this technique is four.

3.3 Procedure Descriptions

Providing the compiler with the ability to make decisions about register usage increases its complexity. Consequently, the procedure which implements the three steps of code generation using the techniques described above, is best described (and implemented) using five seperate procedures: FREANY, FREODD, FIXAD, PTINRG, and INSTR. The first two procedures are used to find free registers. The remaining three are used to perform the three steps of code generation. The INSTR procedure has been previously described. Functional descriptions of the remaining four procedures are given below and detailed logical descriptions are provided in Appendix C.

FREANY(I) This procedure is used to find a free register consistent with the address constraints. If none are found, code is generated to store the contents of the lowest numbered register that is not referenced by the address variables. The number of the free register is returned in I.

FREODD(I) This procedure is used to find a free even-odd register pair. Since this procedure is used only when loading OPND1 into a register, the two registers involved may be referenced by

ADDR1. If no free pair can be found, code is generated to free a pair of registers and preserve addressability of the operands. The number of the odd register is returned in I.

FIXAD(X,ADDR) This procedure is used to construct (fix) the addresses of the operands. X is either OPND1 or OPND2 and ADDR is the corresponding address variable (ADDR1 or ADDR2). This procedure first determines if either X or its base address is in a register, in which case, no code is required and the address variable is set to the appropriate value. If neither is in a register, the procedure FREANY(I) is called and code is generated to load the base address of X into RI. The address variable is then set to the appropriate value.

PTINRG(I) This procedure is used to place OPND1 in a register. If OPER is either "*" or "/" (which requires OPND1 to be in an odd register), this procedure calls FREODD(J) and generates code, if necessary, to load OPND1 into the odd register, RJ. I returns the number of the preceeding even register (J-1). If OPER is not "*" or "/", this procedure first determines if OPND1 is in a register, in which case, no code

is generated and I returns the number of that register. If it is not, FREANY(J) is called and code is generated to load OPND1 into RJ. I then returns register number J.

The procedure for code generation can now be described using the procedures FIXAD, PTINRG and INSTR.

1. Initialize ADDR1=" " and ADDR2=" " and read the first (next) quadruple. If there are no remaining quadruples, code generation for the program is complete.
2. Call FIXAD(OPND1,ADDR1)
3. Call FIXAD(OPND2,ADDR2)
4. Call PTINRG(I)
5. Call INSTR(I)
6. Go to step 1.

Both the register status and assigned base register techniques require knowledge of the contents of registers. Consequently both these techniques require a register map for implementation. Four possible combinations of these techniques will be considered.

a. Register map.
b. Register map with register status.
c. Register map and an assigned base register.
d. Register map with register status and an assigned base register.

The differences between these four levels of code generation appear when implementing the logical steps described in Appendix C.

One difference occurs in the method used to find a free register. When using a register map without register status, the RV array is searched in numerical order (from 1 to N) for the first entry which is not type (3,t). If register status is used, then the RS array is searched in numerical order (from 1 to N) for the first entry whose value is 0. If none are found, then RS is again searched for the first entry whose value is 1. When an assigned base register is used with either of the above techniques, the range of the search is limited to the array entries from 1 to N-1, since the N-th register is used as the assigned base register.

The other differences consist of the requirements for updating the RV and RS arrays when code is generated which changes the value or status of the contents of a register. Register to register LOAD instructions of the form LOAD Rn,Rm require the values of RV(n) and RS(n) to be set equal to the values of RV(m) and RS(m) respectively. No changes are required in the values of RV(m) and RS(m). Storage to register LOAD instructions of the form LOAD Rn,d(R) require the value of RV(n) to be set equal to (4,d) and the value of RS(n) to be set equal to 1. STORE

instructions (except in the INSTR procedure) and storage to register LOAD instructions of the form LOAD Rn,k(i) do not require any changes in either RV or RS because they are always followed by an instruction which causes the necessary changes to be made.

The requirements for the procedure INSTR(I) depend upon the instruction generated. The ADD, SUB and NEG instructions require RV(I) to be set equal to RSLT and RS(I) to be set equal to 2. The MULT and DIV instructions require RV(I) to be cleared, RV(I+1) to be set equal to RSLT, RS(I) to be set equal to 0 and RS(I+1) to be set equal to 2. The STORE instruction requires RV(I) to be set equal to OPND2 and RS(I) to be set equal to 1. All entries in RV of the form (3,t) must be cleared and all entries in RS that equal 2 must be changed to 0.

3.4 Compiler Simulation

The simplification obtained by using a simulated compiler occurs primarily in the analysis phase. The limited scope of the source language and the fact that execution is only simulated eliminates the need for tables. All variables are assumed to be stored in data area 1. Since displacements were not required for the simulation, the variable or temporary character was used in the address variable in place of displacement. This made printed code more readable.

Since all variables and operators are single characters, the scanner need not build any symbols. Differentiating between variables and operators is accomplished by testing the sign of the character since alphabetic characters have negative values and operators have positive values. Temporary variables were denoted by literal numeric characters which also have positive values. Syntax and semantic checking was not included because of the simple nature of the source language. The source language statements which were converted from FORTRAN were visually checked for correctness.

CHAPTER 4

4.1 Cost Model

The purpose of evaluating compiler performance is to provide a means of differentiating between different compilers (or levels of the same compiler) in terms of some cost function. The ideal cost function would be the one used by the computer center where the compiler is to be implemented. The cost function used in this thesis is relatively simple and an element of many computer billing schemes [2]. The cost is based on the time intergrated storage requirements of a program. The storage or space requirements of a program consist of two elements: the instruction space, which is required for the object code, and the variable space, which is required for the values of variables, constants and temporary variables. Since the space requirements of the variables used in the source language are not typical due to the limited number of variables allowed, the cost function was confined to the requirements of the instruction space. The cost function is

$$Ct = R\left(\sum_{i} n(i)T(i)\right)\left(\sum_{i} n(i)S(i)\right)$$

where T(i) is the execution time in seconds of the i-th type of instruction.

S(i) is the storage requirement in kilo-bytes of the i-th type of instruction.

R is the rate charged per kilo-byte-second.

n(i) is the number of times the i-th type of instruction is executed.

Ct is the "total" cost of executing a program.

This model shows no preference between space or time considerations. If, in a particular computer environment, either space or time were critical, than a different model would be used which would allow the use of weighting functions.

The different types of instructions refer to the instructions described in Chapter 2 but differentiate between register to register (R/R) and register to storage (R/S) operations. Since the source language is limited to straight line code, the number of times an instruction is executed is equal to the number of times that instruction is generated in the compiler. Thus, the simulated execution consisted of counting the number of times each type of instruction was generated by the simulated compiler. Execution times for each type of instruction were derived from typical execution times for similiar instructions published for the PDP-11 with the first operand in the register mode and the second operand in either the register mode for R/R instructions or index mode for R/S instructions.[11] The space requirements were obtained from the space required by the IBM 360 for R/R

and R/S instructions.[10] These are also the requirements for the PDP-11. The data used is shown in Table 1.

4.2 Experiment

Two factors were selected for analysis: the level of optimization and the number of registers. Five levels of optimization were used:

1) Basic executable code.
2) Register map.
3) Register map with register status.
4) Register map and an assigned base register.
5) Register map with register status and an assigned base register.

The number of registers was chosen to reflect the target machines. The IBM 360 has 16 registers numbered 0-15. However, registers 0, 1, 13, 14 and 15 are usually reserved by convention for special uses, leaving registers 2-12 for arithmetic operations. The PDP-11 has 8 registers numbered 0-7. However, register 6 is used as a stack pointer and register 7 is the program counter, leaving registers 0-5 for arithmetic operations. Therefore, the number of registers used were 6 and 11. Since the PDP-11 considers registers 0 and 1 to form an even-odd pair, both register configurations begin with even-odd pairs. The simulated compiler was modified to ignore register 1 in order to simulate this same register

INSTRUCTION	TYPE	i	STORAGE REQUIREMENTS (BYTES) S(i)	EXECUTION TIME (MICROSECONDS) T(i)
NEG	-	1	2	1.28
STORE	R/S	2	4	2.90
LOAD	R/S	3	4	2.63
LOAD	R/R	4	2	0.90
ADD	R/S	5	4	2.78
ADD	R/R	6	2	0.90
SUB	R/S	7	4	2.78
SUB	R/R	8	2	0.90
MULT	R/S	9	4	5.56
MULT	R/R	10	2	3.83
DIV	R/S	11	4	10.19
DIV	R/R	12	2	8.46

TABLE 1. STORAGE REQUIREMENTS AND EXECUTION TIMES OF INSTRUCTIONS USED FOR SIMULATED EXECUTION.

configuration. Registers 2-12 were used for the IBM 360 and registers 2-7 were used for the PDP-11.[10,11]

Since cost is a function of program size (particularly with straight line code), the statistic chosen to characterize performance was the total cost of a program divided by the square of the number of quadruples which generated the cost. Since both the total space and total time requirements are each proportional to the number of quadruples in a program, dividing by the square of the number of quadruples should effectively eliminate program size as a variable. In order to provide convienent numbers for analysis, the rate charged per kilo-byte second was chosen as 10^8 (the units are irrelevant). Thus, the statistic used is

$$X = (Ct/q^2)10^8$$

where X is the observed statistic for each event

Ct is the total cost of the event

q is the number of quadruples comprising the event.

Eighty programs were randomly selected from a list of 836 FORTRAN programs provided by the local time sharing center. These 80 programs yielded 3751 arithmetic statements which were converted to the source language. A description of the procedure used to extract and convert arithmetic statements from these programs to the source language is given in Appendix D and some analysis of this

data is shown in Appendix E.

Seperate experiments were performed for program sizes of 17, 46 and 118 arithmetic statements. These sizes were selected as the 50%, 70% and 90% points respectively of the cumulative frequency distribution of the number of arithmetic statements per FORTRAN program. Each event consisted of randomly selecting the appropriate number of statements from the 3751 available statements and simulating compilation with a particular level of optimization and a particular number of registers available. The experiment used was a factorial design which combined all levels of one factor (levels of optimization) with all levels of the other factor (number of registers). An experiment of this design can be easily extended to several factors. The number of replications (observations taken for each combination of factors) was four.

A two factor analysis of variance (ANOVA) test was used to analyze the results. The mathematical model for the ANOVA test is

$$Xijk = u + Li + Rj + LRij + \varepsilon k(ij)$$

where Xijk is the k-th observation (k=1,...,4) of the i-th level of optimization (i=1,...,5) combined with the j-th number of registers available (j=6,11).

u is a common effect for the whole experiment.

Li is the effect of the i-th level of optimization.

Rj is the effect of the j-th level of available registers.

LRij is the interaction effect of Li and Rj.

$\varepsilon_{k(ij)}$ represents the random error within the cell i,j.

Those factors which appeared significant in the ANOVA test were further analyzed by Duncan's multiple range test. This test orders the means of the factor being tested from low to high and then tests the means for significant differences.

A detailed explanation of both these tests can be found in Hicks[4]. Both tests were conducted at the 95% significance level, which means that there is a .05 probability that a factor which appears significant is, in fact, not significant. Those factors which were significant in the ANOVA test at the 99% significance level were noted. A diagram of the experimental procedure is shown in Figure 5.

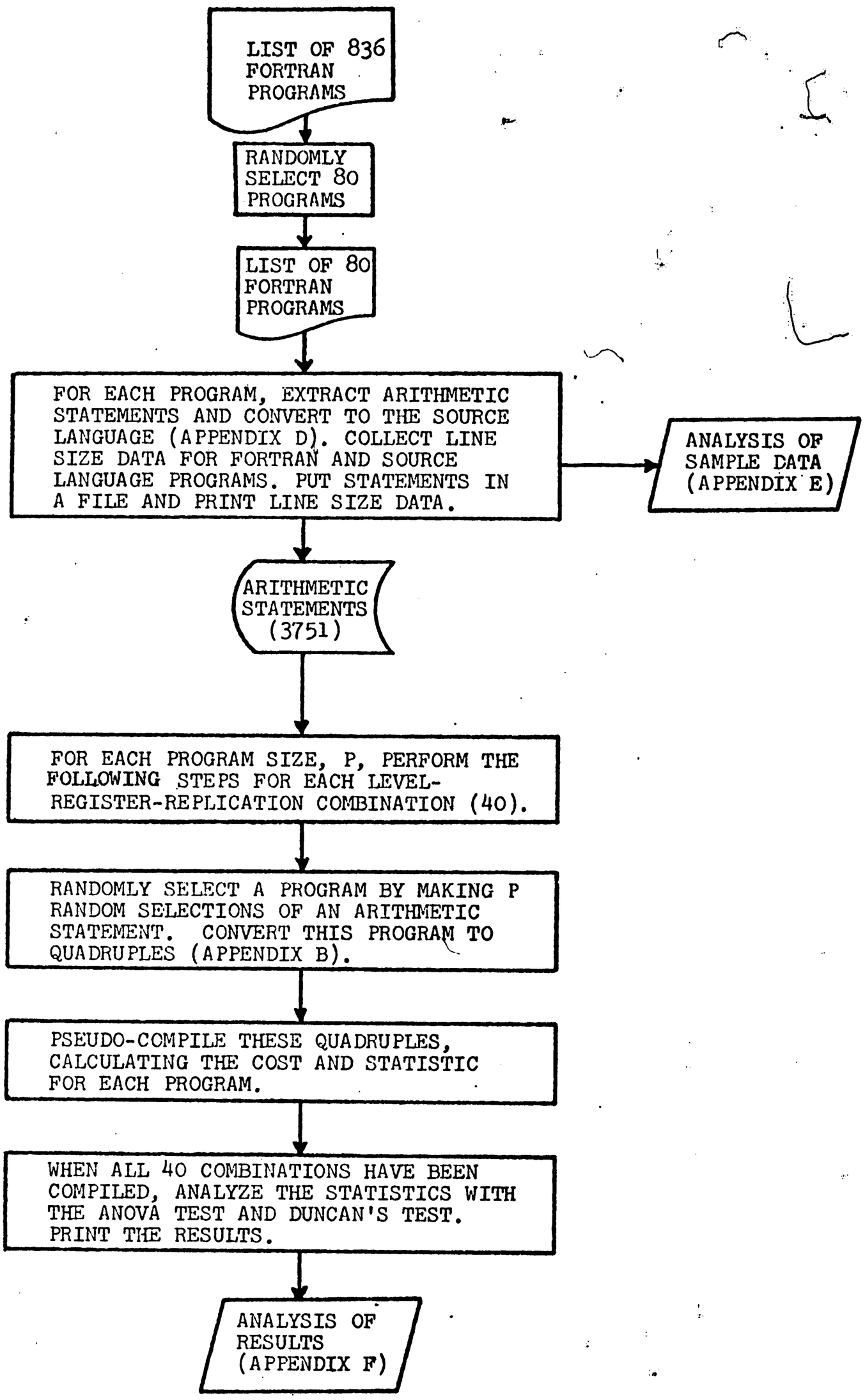


FIGURE 5. DIAGRAM OF EXPERIMENTAL PROCEDURE

CHAPTER 5

5.1 Results

Tables 2 thru 4 show the results and analyses of three experiments using program sizes of 17, 46 and 118 arithmetic statements respectively. Tables 5 thru 7 show the results and analyses when the data for level 1 is excluded.

The multiple range test orders the means of each factor found significant from lowest to highest (left to right). The actual means are not shown but are calculated from the 8 values in a column (for levels), from the 20 values in a row (for registers) or from the 4 values in a column-row intersection (for level-register combinations). Those level, register or level-register combination indices which are underlined indicate that their means are not significantly different (i.e. they could have come from the same population). Conversely, those that are not underlined are statistically different. Thus, in Table 3, levels 4 and 2 are significantly different from level 1. In this same table, levels 5, 3 and 4 are significantly different from levels 4 and 2. The overlap of level 4 can be interpreted as having two distinct population distributions (i.e. different means) which have some overlap. Levels 2, 3 and 5 belong to either one or the other population, but, because of the overlap, level 4

LEVELS	1	2	3	4	5
REGISTERS 6	23.17 22.34 21.08 24.01	3.66 4.14 3.79 4.75	3.45 3.86 3.68 3.34	3.06 3.57 3.04 4.50	3.46 3.46 3.95 3.97
REGISTERS 11	20.16 23.24 23.68 22.10	4.41 4.03 4.22 3.90	3.61 2.82 3.29 4.11	3.82 3.88 4.35 3.82	3.13 3.22 3.17 3.24

ANOVA TABLE

SOURCE	DF	SS	MS	EMS	SIG
LEVEL (Li).....	4	2255.09	563.77	1065.26	99%
REGISTERS (Rj).....	1	0.11	0.11	0.20	---
L X R INTERACTION (LRij).	4	1.08	0.27	0.51	---
ERROR (εk(ij)).....	30	15.88	0.53	1.00	
TOTALS	39	2272.15			

MULTIPLE RANGE TEST (95%)

LEVEL	5	3	4	2	1

TABLE 2. RESULTS AND ANALYSIS OF EXPERIMENT WITH FIVE LEVELS OF OPTIMIZATION FOR SMALL PROGRAM ENVIRONMENT (17 ARITHMETIC STATEMENTS PER PROGRAM).

LEVELS	1	2	3	4	5
REGISTERS 6	21.96 20.73 23.11 23.61	4.37 4.14 3.53 4.05	3.24 3.34 3.76 3.26	3.52 3.90 3.59 3.56	4.20 3.62 3.41 3.26
REGISTERS 11	24.18 22.58 22.66 22.05	3.65 4.12 3.84 3.76	3.21 3.15 3.00 2.94	3.83 3.78 3.47 3.76	2.61 2.84 3.09 2.78

ANOVA TABLE

SOURCE	DF	SS	MS	EMS	SIG
LEVEL (Li).....	4	2335.89	583.97	1951.46	99%
REGISTERS (Rj).....	1	0.21	0.21	0.69	---
L X R INTERACTION (LRij).	4	1.88	0.47	1.57	---
ERROR (εk(ij)).....	30	8.98	0.30	1.00	
TOTALS	39	2346.95			

MULTIPLE RANGE TEST (95%)

LEVEL	5	3	4	2	1

TABLE 3. RESULTS AND ANALYSIS OF EXPERIMENT WITH FIVE LEVELS OF OPTIMIZATION FOR MEDIUM PROGRAM ENVIRONMENT (46 ARITHMETIC STATEMENTS PER PROGRAM).

LEVELS	1	2	3	4	5
REGISTERS 6	23.65 21.66 22.74 22.72	3.94 3.96 4.07 3.86	3.61 3.64 3.49 3.61	3.94 3.83 3.46 3.59	3.09 3.49 3.28 3.31
REGISTERS 11	23.45 23.51 21.47 24.10	3.71 3.66 3.62 3.56	2.84 2.65 2.73 2.92	3.65 3.19 3.56 3.70	2.84 2.80 2.75 2.92

ANOVA TABLE

SOURCE	DF	SS	MS	EMS	SIG
LEVEL (Li).....	4	2435.78	608.94	2839.97	99%
REGISTERS (Rj).....	1	0.70	0.70	3.28	---
L X R INTERACTION (LRij).	4	1.68	0.42	1.96	---
ERROR (εk(ij)).....	30	6.43	0.21	1.00	
TOTALS	39	2444.59			

MULTIPLE RANGE TEST (95%)

LEVEL	5	3	4	2	1

TABLE 4. RESULTS AND ANALYSIS OF EXPERIMENT WITH FIVE LEVELS OF OPTIMIZATION FOR LARGE PROGRAM ENVIRONMENT (118 ARITHMETIC STATEMENTS PER PROGRAM).

LEVELS	2	3	4	5
REGISTERS 6	3.66 4.14 3.79 4.75	3.45 3.86 3.68 3.34	3.06 3.57 3.04 4.50	3.46 3.46 3.95 3.97
REGISTERS 11	4.41 4.03 4.22 3.90	3.61 2.82 3.29 4.11	3.82 3.88 4.35 3.82	3.13 3.22 3.17 3.24

ANOVA TABLE

SOURCE	DF	SS	MS	EMS	SIG
LEVEL (Li).....	3	2.14	0.71	4.53	95%
REGISTERS (Rj).....	1	0.01	0.01	0.09	---
L X R INTERACTION (LRij).	3	0.92	0.31	1.94	---
ERROR ($\varepsilon_{k(ij)}$).....	24	3.77	0.16	1.00	
TOTALS	31	6.84			

MULTIPLE RANGE TEST (95%)

LEVEL	5	3	4	2
			———	———
	———	———		

TABLE 5. RESULTS AND ANALYSIS OF EXPERIMENT WITH FOUR LEVELS OF OPTIMIZATION FOR SMALL PROGRAM ENVIRONMENT (17 ARITHMETIC STATEMENTS PER PROGRAM).

LEVELS	2	3	4	5
REGISTERS 6	4.37 4.14 3.53 4.05	3.24 3.34 3.76 3.26	3.52 3.90 3.59 3.56	4.20 3.62 3.41 3.26
REGISTERS 11	3.65 4.12 3.84 3.76	3.21 3.15 3.00 2.94	3.83 3.78 3.47 3.76	2.61 2.84 3.09 2.78

ANOVA TABLE

SOURCE	DF	SS	MS	EMS	SIG
LEVEL (Li).....	3	2.88	0.96	15.05	99%
REGISTERS (Rj).....	1	0.76	0.76	11.92	99%
L X R INTERACTION (LRij).	3	0.79	0.26	4.11	95%
ERROR ($\varepsilon_{k(ij)}$).....	24	1.53	0.06	1.00	
TOTALS	31	5.96			

MULTIPLE RANGE TEST (95%)

LEVEL	5	3	4	2
REGISTER	11	6		

LEVEL	5	3	3	5	4	4	2	2
REGISTER	11	11	6	6	6	11	11	6

TABLE 6. RESULTS AND ANALYSIS OF EXPERIMENT WITH FOUR LEVELS OF OPTIMIZATION FOR MEDIUM PROGRAM ENVIRONMENT (46 ARITHMETIC STATEMENTS PER PROGRAM).

LEVELS	2	3	4	5
REGISTERS 6	3.94 3.96 4.07 3.86	3.61 3.64 3.49 3.61	3.94 3.83 3.46 3.59	3.09 3.49 3.28 3.31
REGISTERS 11	3.71 3.66 3.62 3.56	2.84 2.65 2.73 2.92	3.65 3.19 3.56 3.70	2.84 2.80 2.75 2.92

ANOVA TABLE

SOURCE	DF	SS	MS	EMS	SIG
LEVEL (Li).....	3	2.92	0.97	48.05	99%
REGISTERS (Rj).....	1	1.56	1.56	77.07	99%
L X R INTERACTION (LRij).	3	0.43	0.14	7.13	99%
ERROR (εk(ij)).....	24	0.49	0.02	1.00	
TOTALS	31	5.40			

MULTIPLE RANGE TEST (95%)

LEVEL	5	3	4	2
REGISTER	11	6		

LEVEL	3	5	5	4	3	2	4	2
REGISTER	11	11	6	11	6	11	6	6

TABLE 7. RESULTS AND ANALYSIS OF EXPERIMENT WITH FOUR LEVELS OF OPTIMIZATION FOR LARGE PROGRAM ENVIRONMENT (118 ARITHMETIC STATEMENTS PER PROGRAM).

could be in either of the two populations. This overlap has the effect of weakening the differences identified.

When all five levels of optimization were included in the analyses (Tables 2 thru 4), the ANOVA test indicated a very strong level effect and Duncan's test indicated little, if any, difference in levels 2 thru 5. This results from the very high cost of level 1 compared to the other 4 levels. To offset this effect and gain additional information about levels 2 thru 5, the same data was reanalyzed with level 1 excluded. These results are shown in Tables 5 thru 7. The differences (and lack of differences) between the levels, registers and level-register combinations are more apparent in this analysis. The increase in the significance of the level factor as well as the inclusion of additional factors as the program size increases results from the increase in opportunities for improvement in larger programs.

Even though the different levels of optimization were used only as illustrations, care should be exercised when interpreting the results of these experiments. The assumption that all variables are allocated to data area 1 favors the use of the assigned base register. This assumption may not be fully met in actual practice. Also the use of only 26 variables (A-Z) increases the

opportunities for improvements obtained by leaving variables in registers. These opportunities may be significantly reduced in practice where a much larger number of variables are used.

The interpretation of these results would indicate that the addition of a register map is highly desirable in all cases due to the large improvement over basic executable code. The additional techniques which give the best performance tend to depend upon the number of registers available and the size of the programs. In a small program environment (17 lines), any of the levels 3 thru 5 provide the same improvement when added to the register map regardless of the number of registers. In the medium program environment (46 lines) with 6 registers, any of the levels 3 thru 5 provide essentially the same improvement when added to the register map. With 11 registers, adding an assigned base register provides no significant improvement over the register map. If used with a status array, the improvement is significant but no different than the improvement achieved with the status array alone. In the large program environment (118 lines) with 6 registers, a status array or an assigned base register both provide the same significant improvement. An additional improvement can be obtained by combining these techniques. With 11 registers available, the

results are the same as the medium program environment with 11 registers.

The above interpretations are based on empirical results relating specifically to a subset of FORTRAN describing arithmetic statements. However, this does not restrict the application of the methodology to other languages or to more complete versions of FORTRAN. If, for example, the optimization levels were tested for a language which made little use of arithmetic statements and, as a result, the analysis of experimental data indicated no significant differences in performance for all levels of optimization, then, the designer would still have gained significant information. He would have learned that the least complicated technique could be used in the compiler without affecting the object code produced.

5.2 Conclusions

The results of the experiments indicate that the methodology proposed in this thesis can be a valuable tool for measuring performance in the design and evaluation of compilers (and compiler optimization techniques).

The definition of an abstract computer is feasible in many machine environments (particularly those consisting of similiar computers). Even in those environments consisting of very dissimiliar computers, SAC can be

useful for special purpose user oriented languages of limited complexity.

The design of a compiler for SAC requires no new technology since a compiler, like any other program, uses parameters, whether initialized inside the program or initialized externally at execution time. Thus the parameters of SAC do not restrict the design of a suitable compiler for SAC.

The factorial experiment used can be extended to any number of factors, both quantitative and qualitative, and the ANOVA test provides a method for evaluating the effects and interactions of these factors. Duncan's test provides the ability to select a preferred alternative among those being tested.

5.3 Recommendations For Further Study

Several areas of further study are indicated by this proposed methodology. The most obvious is the enrichment of the definition of SAC. The instruction set of SAC could be expanded to include instructions which are common to most computers, and, with additional effort, could be extended to various combinations of computers. Included in this effort would be the determination of the best method of implementing in SAC the various addressing schemes used by different computers. A very pertinent area of study would be to determine whether this problem

is best handled by the intermediate processing which SAC code requires before execution, or the use of an addressing scheme in SAC which would reduce to various addressing schemes as special cases or, possibly, some combination of both approaches.

Another area of study is the actual design and implementation of a SAC compiler, particularly for languages more complex than that treated in this thesis. This effort would involve not only the difficulties inherent in compiler design but also the additional task of incorporating in the compiler the ability to gather information which is required for statistical analysis.

A third area of investigation would be the development of a simplified SAC definition and several models of program environments which would provide standards for relative evaluations of various compiler techniques. These standards need not be all encompassing in terms of machine and program environments to be used as effective indices of performance in the publications concerning compiler performance.

Bibliography

1. Beatty, J. C. "Register Assignment Algorithm for Generation of Highly Optimized Object Code" IBM J. Res. Develop. (Jan. 1974), 20-39

2. Bookman, P. G. "Make Your Users Pay The Price" Computer Decisions 9 (Sept. 1972), 28-31

3. Gries, D. Compiler Construction for Digital Computers. New York, New York: John Wiley & Sons, Inc., 1971

4. Hicks, C. R. Fundamental Concepts in the Design of Experiments. New York, New York: Holt, Rinehart and Winston, 1964

5. Knuth, D. E. "An Empirical Study of FORTRAN Programs" Software Practice & Experience 1 (1971), 105

6. Mendicino, S. F., et al. "The LRLTRAN Compiler" Comm. ACM 11 (Nov. 1968), 747-755

7. Sethi, R., and Ullman, J. D. "The Generation of Optimal Code for Arithmetic Expressions" J. ACM 9 (Sept. 1972), 715-728

8. Sethi, R., Ullman, J. D., and Aho, A. V. "A Formal Approach to Code Optimization" Proc. Symposium on Compiler Optimization, ACM, (1970), 86-100

9. Sippl, C. J. and Sippl, C. P. Computer Dictionary and Handbook 2nd Ed. Indianapolis, Indiana: Howard W. Sams & Co., Inc. 1972

10. Struble, G. Assembler Language Programming: The IBM System/360. Reading, Massachusetts: Addison-Wesley Publishing Co., 1969

11. _____ PDP-11/45 Processor Handbook. Digital Equipment Corp., 1973

APPENDIX A

List of Symbols

a-z a variable in internal form used by the compiler

A-Z a variable in the source language

c a pointer to an entry in the constant table

d a data area number

D the number of data areas required for a particular program

D' the maximum number of data areas available

i the number of a register used to hold a base address (a base register)

j the number of a register used to hold the index portion of an address (an index register)

k a displacement constant (frequently called an offset)

K the maximum displacement that an instruction may hold (K=max(k))

k(i) the address obtained by adding the displacement constant, k, to the contents of base register i. In the context of an instruction, "k(i)" should be read as "the contents of the address specified by k(i)". Equivalent to k(i,0) where 0 is the number zero rather than the register numbered zero.

k(i,j) the address obtained by adding the displacement constant, k, to the sum of the contents of base register i and index register j.

m or n the number of a general purpose register available for arithmetic operations as well as addressing operations

N the maximum number of general purpose registers available (N=max(n)).

R the (N+1)-st register

Rm or Rn the register numbered m or n. In the context of an instruction, "Rm" or "Rn" should be read as "the contents of the register numbered m or n".

[Rn,Rn+1] the concatenation (uniting in a series) of the registers numbered n and n+1

t a pointer to an entry in the temporary variable table

T the actual address of the first word in memory allocated at execution time

v a pointer to an entry in the variable table

@ either a register, Rn, or a memory location, k(i). In the context of an instruction, "@" should be read as "the address specified by @".

APPENDIX B

Quadruple Generation

Quadruples are generated from the source language described in Chapter 1. The method described uses two procedures and is based upon the precedence relations between operators.[3] The main procedure reads and analyses each arithmetic statement and then calls the QUAD procedure to generate the appropriate quadruples. The precedence relationships between operators are defined by a function, P(operator), as follows:

$$P(=) > P(*) = P(/) > P(+) = P(-) > P(() = P()) > P(;)$$

This method requires two LIFO (last-in-first-out) stacks. A stack is a storage device into which one stores data. However, data can only be entered at the "top", thus "pushing down" the data already in it. Accordingly one can only reference or change the top (or the top few) elements. When no longer required, the top elements are deleted, thus "popping up" the ones below. The usual method of implementing a stack is to use an array S, and a counter V. If V=0 the stack is empty. If V=m, where m is greater than zero, the stack contains S(1), S(2),..., S(m) where S(m) is the top stack element.[3] One of the stacks required is an operator stack called OPS which holds only operators. The other stack is an operand stack called OPANDS which holds both operators and operands. The i-th element of the stacks, counting the top element as 1, will be denoted by OPS(i) and OPANDS(i). These stacks are global since they must be available to both procedures.

Temporary variables are sequentially assigned to the quadruples beginning with each arithmetic statement. A counter, t, which is a global variable, is initialized to zero when each arithmetic statement is read and incremented by 1 each time a quadruple is generated. The symbol, t, should be interpreted as the internal form of a temporary variable, (3,t), which points to the t-th entry in the temporary variable table.

The main procedure is described by the following rules which are applied sequentially unless otherwise indicated.

1. Read the first (next) arithmetic statement and initialize the temporary variable counter, t,to zero. If there are no remaining statements, conversion to quadruples for the program is complete.

2. Put into the variable CHAR the first (next) input character by scanning the input string from left to right.

3. If CHAR is an operator, go to step 5.

4. Push CHAR on OPANDS and go to step 2.

5. If OPS(1) does not equal "=", go to step 8.

6. If CHAR equals ";", call QUAD and go to step 1.

7. Push CHAR on OPS and go to step 4.

8. If OPS(1) equals " " (i.e. the operator stack is empty), go to step 7.

9. If P(CHAR) > P(OPS(1)), go to step 7.

10. If CHAR equals "(", go to step 7.

11. If CHAR does not equal ")", call QUAD and go to step 5.

12. If OPS(1) equals "(", go to step 14.

13. Call QUAD and go to step 12.

14. Let SAVE=OPANDS(1), pop OPANDS(1), OPANDS(2) and OPS(1) from the stacks, push SAVE on OPANDS and go to step 2.

The procedure QUAD is defined by the following rules.

1. Increment temporary variable counter: t=t+1.

2. If OPS(1) equals "+", "*" or "/", go to step 5.

3. If OPS(1) equals "=", go to step 6.

4. If OPS(1) equals "-" and OPANDS(3) equals either "=" or "(", go to step 7.

5. Generate (OPANDS(2),OPANDS(3),OPANDS(1),t). Pop OPANDS(1), OPANDS(2), OPANDS(3) and OPS(1) from the stacks. Push t on OPANDS and return.

6. Generate (OPANDS(2),OPANDS(1),OPANDS(3),"blank"). Pop OPANDS(1), OPANDS(2), OPANDS(3) and OPS(1) from the stacks and return.

7. Generate (OPANDS(2),OPANDS(1),"blank",t). Pop OPANDS(1), OPANDS(2) and OPS(1) from the stacks. Push t on OPANDS and return.

An example is shown in Figure 6 for the following arithmetic statement: a=-b+c-(d+(-e)/f);.

a=-b+c-(d+(-e)/f);

OPANDS	OPS	CHAR	QUADRUPLE
		a	
a		=	
a=	=	-	
a=-	=-	b	
a=-b	=-	+	(-,b, ,1)
a=1	=	+	
a=1+	=+	c	
a=1+c	=+	-	(+,1,c,2)
a=2	=	-	
a=2-	=-	(	
a=2-(	=-(	d	
a=2-(d	=-(	+	
a=2-(d+	=-(+	(	
a=2-(d+(	=-(+(	-	
a=2-(d+(-	=-(+(-	e	
a=2-(d+(-e	=-(+(-	)	(-,e, ,3)
a=2-(d+(3	=-(+(	)	
a=2-(d+3	=-(+	/	
a=2-(d+3/	=-(+/	f	
a=2-(d+3/f	=-(+/	)	(/,3,f,4)
a=2-(d+4	=-(+	)	(+,d,4,5)
a=2-(5	=-(	)	
a=2-5	=-	;	(-,2,5,6)
a=6	=	;	(=,6,a,)

FIGURE 6. EXAMPLE OF QUADRUPLE GENERATION.

APPENDIX C

Logical Procedure Descriptions

In the descriptions which follow, all steps are executed sequentially unless otherwise indicated. In all except the INSTR procedure, the letter t, when used in the STORE instruction should be interpreted as the displacement of a temporary variable obtained from the entry in the temporary variable table pointed to by an operand of type (3,t) in the RV array. The variables and arguements used are described in Chapter 3.

FREANY(I)

1. If a register, Rn, is free and is not referenced by either address variable, set I=n and return.

2. Find the lowest numbered register, Rn, that is not referenced by either address variable. Generate "STORE Rn,t(R)", set I=n and return.

In the FREODD procedure, all references to an odd register refer only to odd registers numbered three or greater.

FREODD(I)

1. If OPND1 is in an odd register, Rn, go to step 5.

2. If an odd register, Rn, is not free, go to step 7.

3. If ADDR2 references Rn, go to step 9.

4. If ADDR1 does not reference Rn, go to step 8.

5. If ADDR2 does not reference Rn-1, go to step 14.

6. If a register, Rm, other than Rn and Rn-1, is free, go to step 19. Otherwise set m=1, generate "STORE Rm,t(R)" and go to step 19.

7. If an odd register, Rn, which is not referenced by ADDR2, is available, generate "STORE Rn,t(R)" and go to step 8. Otherwise find the first register, Rn, that is not referenced by ADDR1 (either R1 or R2) and go to step 20.

8. If ADDR2 references Rn-1, go to step 15. Otherwise go to step 13.

9. If an odd register, Rm, other than Rn is free, set n=m and go to step 13.

10. If a register, Rm, other than Rn and Rn-1, which is not referenced by ADDR1, is not available, set n=2 and go to step 20.

11. If Rm is not free, generate "STORE Rm,t(R)".

12. Generate "LOAD Rm,Rn", change the reference to n in ADDR2 to m.

13. If ADDR1 references Rn-1, set I=n and return.

14. If Rn-1 is free, set I=n and return. Otherwise generate "STORE Rn-1,t(R)", set I=n and return.

15. If a register, Rm, other than Rn and Rn-1, is free, go to step 17. Otherwise set m=1.

16. If ADDR1 references Rm, go to step 18. Otherwise generate "STORE Rm,t(R)" and go to step 19.

17. If ADDR1 references Rm, go to step 18. Otherwise go to step 19.

18. Generate "LOAD Rn,Rm", change the reference to m in ADDR1 to n.

19. Generate "LOAD Rm,Rn-1", change the reference to n-1 in ADDR2 to m, set I=n and return.

20. If Rn is free, go to step 22. Otherwise generate "STORE Rn,t(R)".

21. If n equals 1, go to step 22. Otherwise generate "LOAD R2,R1" and change the reference to 1 in ADDR1 to 2.

22. Generate "LOAD R1,R3", change the reference to 3 in ADDR2 to 1, set I=3 and return.

FIXAD(X,ADDR)

1. If X equals " ", return.

2. If the value of X is in a register, Rn, set ADDR=Rn and return. Otherwise form the operand, OP=(4,d) (where d is the data area number for X).

3. If the value of OP is in a register, Rn, set ADDR=k(n) (where k is the displacement constant for X) and return. Otherwise call FREANY(I), generate "LOAD RI,d(R)", set ADDR=k(I) and return.

PTINRG(I)

1. If OPER equals "*" or "/", call FREODD(J) and go to step 3.

2. If the value of OPND1 is in a register, Rn, set I=n and return. Otherwise call FREANY(J), generate "LOAD RJ,ADDR1", set I=J and return.

3. If the value of OPND1 is in a register J, set I=J-1 and return. Otherwise generate "LOAD RJ,ADDR1", set I=J-1 and return.

In the INSTR procedure, since the characters themselves are used to represent their internal form, the unary minus operator must be determined from the context of the quadruple (i.e. there will be no second operand).

INSTR(I)

1. If OPER equals "+", generate "ADD RI,ADDR2" and return.

2. If OPER does not equal "-", go to step 4.

3. If ADDR2 equals " ", generate "NEG RI" and return. Otherwise generate "SUB RI,ADDR2" and return.

4. If OPER equals "*", generate "MULT RI,ADDR2" and return.

5. If OPER equals "/", generate "DIV RI,ADDR2" and return.

6. If OPER equals "=", generate "STORE RI,ADDR2 and return.

APPENDIX D

FORTRAN Conversion Procedure

The procedure described below was used to extract arithmetic statements from FORTRAN programs and reduce these statements to the single letter variables, (A-Z), and operators described in Chapter 1. Operators which are not included in the source language were replaced with the following substitutions.

Operator	Substitution
.AND.,.XOR.,.EQV.,.OR.,.EQ.,.NE.	+
.NOT.,.GT.,.GE.,.LT.,.LE.	-
.NOT. (when preceeded by any of the above operators)	blank
**	/

This procedure uses a sequential input file which holds the FORTRAN program to be processed and a random access output file which holds the reduced arithmetic statements. Since the record length of these files was limited to 126 characters and a ";" must be added to each record, any arithmetic statement whose length exceeded 125 characters was rejected. The following steps are performed sequentially unless otherwise indicated.

1. Read the first (next) line in the input file into the array variable, ICOL(i), for i=1,...,72. If there are no remaining lines, go to step 14.

2. If ICOL(1) indicates a comment line, go to step 1.

3. If ICOL(6) indicates a continuation line, go to step 1.

4. If ICOL(i) does not have an "=" at the zero parenthesis level for i=7,...,72, go to step 1.

5. If ICOL(i) does have a "," at the zero parenthesis level for i=7,...,72, go to step 1.

6. Place the non-blank characters in ICOL(i) for i=7,...,72 into the array variable, KCOL(j), for j=1,...,L where L is the index of the last non-blank character.

7. Read the next line in the input file into the array ICOL(i) for i=1,...,72. If there are no remaining lines, go to step 12.

8. If ICOL(1) indicates a comment line, go to step 12.

9. If ICOL(6) does not indicate a continuation line, go to step 13.

10. Place the non-blank characters in ICOL(i) for i=7,...,72 into KCOL(j) for j=L+1,...,M where M is the index of the last non-blank character. Set L=M.

11. If L exceeds 125, go to step 1. Otherwise go to step 7.

12. Set KCOL(L+1)=";". Write KCOL(j) on the output file for j=1,...,L+1 and go to step 1.

13. Set KCOL(L+1)=";". Write KCOL(j) on the output file for j=1,...,L+1 and go to step 4.

14. Substitute single letter variables for all constants, multiple character variables, array elements and function calls. Substitute for logical, relational and exponential operators.

15. Conversion is complete.

APPENDIX E

Analysis of Sample Data

A list of the names and locations of 836 FORTRAN programs (distributed among 144 programmers) was provided by the local time-sharing center. From this list 80 programs (distributed among 44 programmers) were randomly selected. This sample of 80 programs consisted of 12994 lines (card images). The cumulative frequency distribution of the number of lines per program as well as the cumulative distribution of the total number of lines is tabulated in Table 8. In the interest of space, only non-zero frequencies are shown in this and subsequent tables.

Using the procedure described in Appendix D, 3751 arithmetic statements were extracted from these programs. The cumulative frequency distribution of the number of arithmetic statements per program is tabulated in Table 9. The program sizes of 17, 46 and 118 arithmetic statements, which were selected for the experiment, are the 50%, 70% and 90% points respectively of this distribution. Also shown in Table 9 is the cumulative distribution of the total number of arithmetic statements.

The frequency distribution of the ratio of the number of arithmetic statements obtained from a program to the size (number of lines) of that program is shown in Figure 6.

The 3751 statements used in the experiment contain a total of 6493 arithmetic and assignment operators (an average of 1.7 operators per statement) distributed as follows:

OPERATION	NO. OF OCCURRENCES	% OF TOTAL
Assignment(=)	3751	57.8
Addition(+)	795	12.2
Multiplication(*)	699	10.8
Division(/)	672	10.3
Subtraction(-)	457	7.0
Unary Minus(-)	119	1.8

The frequency distribution of the number of operators per statement is tabulated in Table 10. This distribution is very similiar to the results obtained by Knuth[5].

PROGRAM SIZE (LINES)	CUMULATIVE OCCURRENCES		CUMULATIVE LINES	
	NO.	%	NO.	%
5	1	1.3	5	0.0
6	2	2.5	11	0.1
8	3	3.8	19	0.1
9	5	6.3	37	0.3
11	7	8.8	59	0.5
12	8	10.0	71	0.5
13	9	11.2	84	0.6
16	10	12.5	100	0.8
18	12	15.0	136	1.0
19	15	18.8	193	1.5
22	17	21.3	237	1.8
23	19	23.8	283	2.2
24	20	25.0	307	2.4
25	24	30.0	407	3.1
32	26	32.5	471	3.6
33	28	35.0	537	4.1
34	30	37.5	605	4.7
37	31	38.7	642	4.9
38	32	40.0	680	5.2
39	33	41.3	719	5.5
40	34	42.5	759	5.8
45	35	43.8	804	6.2
47	36	45.0	851	6.5
49	37	46.2	900	6.9
50	38	47.5	950	7.3
53	40	50.0	1056	8.1
57	41	51.3	1113	8.6
61	42	52.5	1174	9.0
72	43	53.8	1246	9.6
75	44	55.0	1321	10.2
79	46	57.5	1479	11.4
82	47	58.7	1561	12.0
90	48	60.0	1651	12.7
95	49	61.2	1746	13.4
106	50	62.5	1852	14.3
113	51	63.8	1965	15.1
114	52	65.0	2079	16.0
115	53	66.3	2194	16.9
141	55	68.8	2476	19.1
145	56	70.0	2621	20.2
158	57	71.2	2779	21.4
163	58	72.5	2942	22.6
169	59	73.7	3111	23.9

TABLE 8. CUMULATIVE FREQUENCY DISTRIBUTIONS OF THE NUMBER OF LINES PER SAMPLED FORTRAN PROGRAM AND THE TOTAL NUMBER OF LINES SAMPLED.

PROGRAM SIZE (LINES)	CUMULATIVE OCCURRENCES NO.	CUMULATIVE OCCURRENCES %	CUMULATIVE LINES NO.	CUMULATIVE LINES %
176	60	75.0	3287	25.3
184	61	76.3	3471	26.7
197	62	77.5	3668	28.2
212	63	78.8	3880	29.9
221	64	80.0	4101	31.6
223	65	81.3	4324	33.3
247	66	82.5	4571	35.2
270	67	83.7	4841	37.3
338	68	85.0	5179	39.9
366	69	86.2	5545	42.7
371	70	87.5	5916	45.5
395	71	88.8	6311	48.6
415	72	90.0	6726	51.8
438	73	91.3	7164	55.1
450	74	92.5	7614	58.6
490	75	93.8	8104	62.4
534	76	95.0	8638	66.5
541	77	96.2	9179	70.6
557	78	97.5	9736	74.9
1186	79	98.7	10922	84.1
2072	80	100.0	12994	100.0

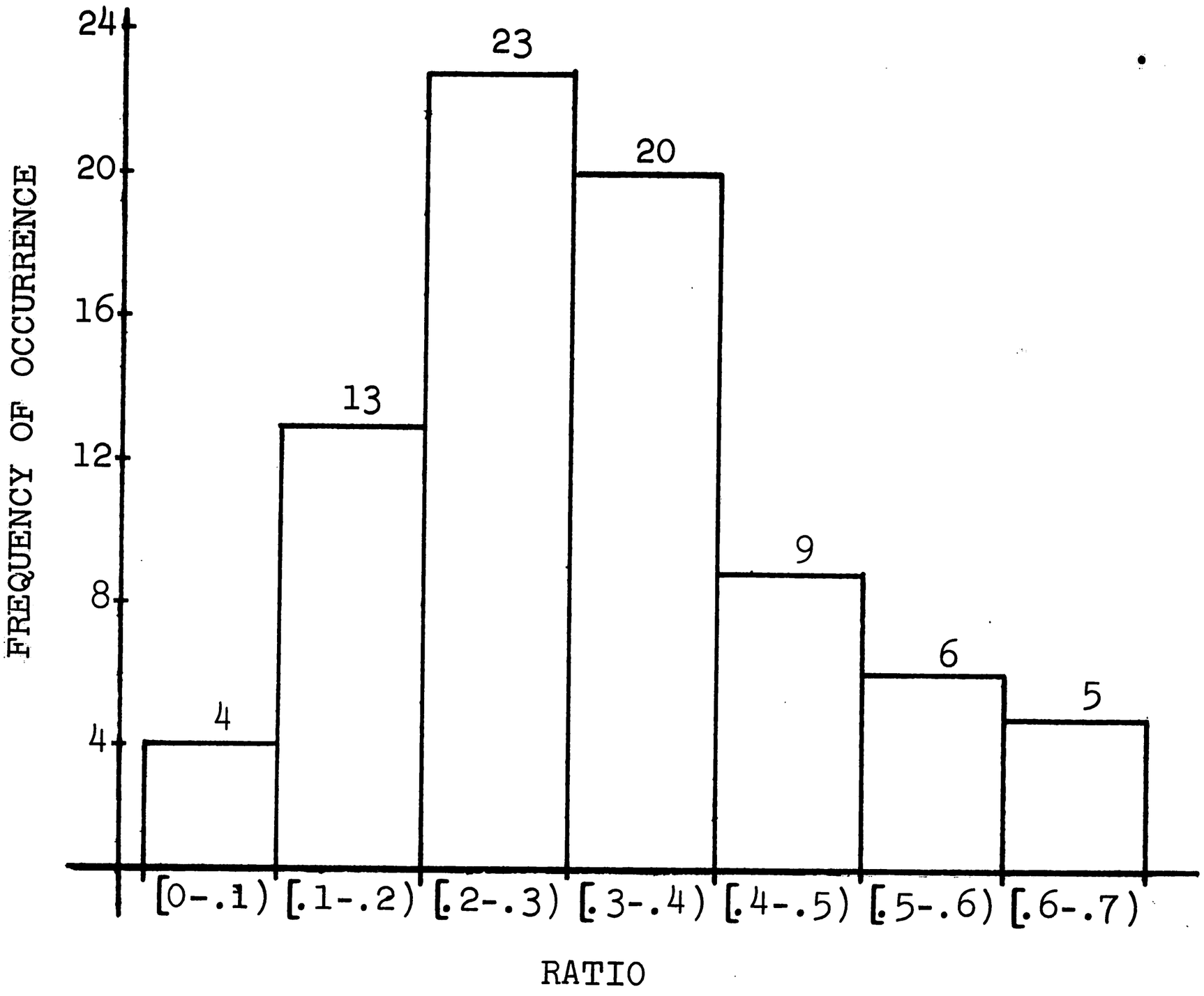
TABLE 8.(cont'd) CUMULATIVE FREQUENCY DISTRIBUTIONS OF THE NUMBER OF LINES PER SAMPLED FORTRAN PROGRAM AND THE TOTAL NUMBER OF LINES SAMPLED.

ARITHMETIC STATEMENTS	CUMULATIVE OCCURRENCES		CUMULATIVE STATEMENTS	
	NO.	%	NO.	%
1	1	1.3	1	0.0
2	8	10.0	15	0.4
3	11	13.7	24	0.6
4	15	18.8	40	1.1
5	18	22.5	55	1.5
6	22	27.5	79	2.1
7	26	32.5	107	2.9
8	30	37.5	139	3.7
9	31	38.7	148	3.9
10	32	40.0	158	4.2
12	34	42.5	182	4.9
13	35	43.8	195	5.2
14	36	45.0	209	5.6
15	39	48.8	254	6.8
17	40	50.0	271	7.2
19	41	51.3	290	7.7
20	42	52.5	310	8.3
21	43	53.8	331	8.8
23	45	56.3	377	10.1
28	46	57.5	405	10.8
29	47	58.7	434	11.6
30	48	60.0	464	12.4
31	49	61.2	495	13.2
35	50	62.5	530	14.1
37	51	63.8	567	15.1
41	52	65.0	608	16.2
42	53	66.3	650	17.3
44	54	67.5	694	18.5
45	55	68.8	739	19.7
46	56	70.0	785	20.9
52	57	71.2	837	22.3
54	58	72.5	891	23.8
55	61	76.3	1056	28.2
58	63	78.8	1172	31.2
62	64	80.0	1234	32.9
72	65	81.3	1306	34.8
74	66	82.5	1380	36.8
78	67	83.7	1458	38.9
79	68	85.0	1537	41.0
91	69	86.2	1628	43.4
97	70	87.5	1725	46.0
110	71	88.8	1835	48.9

TABLE 9. CUMULATIVE FREQUENCY DISTRIBUTIONS OF THE NUMBER OF ARITHMETIC STATEMENTS PER SAMPLED FORTRAN PROGRAM AND THE TOTAL NUMBER OF ARITHMETIC STATEMENTS.

ARITHMETIC STATEMENTS	CUMULATIVE OCCURRENCES		CUMULATIVE STATEMENTS	
	NO.	%	NO.	%
118	72	90.0	1953	52.1
125	73	91.3	2078	55.4
129	74	92.5	2207	58.8
131	75	93.8	2338	62.3
189	76	95.0	2527	67.4
200	77	96.2	2727	72.7
278	78	97.5	3005	80.1
281	79	98.7	3286	87.6
465	80	100.0	3751	100.0

TABLE 9. (cont'd) CUMULATIVE FREQUENCY DISTRIBUTIONS OF THE NUMBER OF ARITHMETIC STATEMENTS PER SAMPLED FORTRAN PROGRAM AND THE TOTAL NUMBER OF ARITHMETIC STATEMENTS.



MEAN.....	.31
MEDIAN....	.30
STD DEV...	.15
RANGE.....	.65
MINIMUM...	.05
MAXIMUM...	.70

FIGURE 7. FREQUENCY DISTRIBUTION OF THE RATIO OF ARITHMETIC STATEMENTS TO PROGRAM SIZE (LINES).

QUADRUPLES PER STATEMENT	FREQUENCY OF OCCURRENCES	CUMULATIVE OCCURRENCES NO.	%
1	2195	2195	58.5
2	1082	3277	87.4
3	254	3531	94.1
4	103	3634	96.9
5	41	3675	98.0
6	20	3695	98.5
7	12	3707	98.8
8	10	3717	99.1
9	5	3722	99.2
10	5	3727	99.4
11	4	3731	99.5
12	3	3734	99.5
13	3	3737	99.6
14	3	3740	99.7
15	3	3743	99.8
16	1	3744	99.8
17	1	3745	99.8
21	2	3747	99.9
22	1	3748	99.9
23	1	3749	99.9
24	1	3750	100.0
26	1	3751	100.0

TABLE 10. FREQUENCY DISTRIBUTION OF THE NUMBER OF QUADRUPLES PER ARITHMETIC STATEMENT.

APPENDIX F

Glossary of Terms[9]

access time - 1. The time interval between the instant at which information is called from storage and the instant at which delivery is completed (the read time). 2. The time interval between the instant at which data are ready for storage and the instant at which storage is completed (the write time).

accumulator - A register in which are formed algebraic sums and other arithmetic and logical results.

address - 1. A label, name or number identifying a register, location or unit where information is stored. 2. To call a specific piece of information from the memory or to put it in the memory.

address, actual - The real or designed address built into the computer by the manufacturer as a storage location or register. Adjacent addresses usually have adjacent numbers.

address, base - A number used in symbolic coding in conjunction with a relative address. The address of the first storage location in a data area, thus the address of a data area. Also an address used as a reference for a group of related addresses.

addressing, base-displacement - A system that uses a base address plus a displacement to designate all core-storage locations and provides abilities to: (1) easily relocate a program at load time, (2) address a very large amount of storage with relatively few address bits in each instruction and (3) convienently address three dimensional arrays.

addressing, direct - A procedure for specifically citing an operand in the instruction by the operand's location in storage. The direct address is the number representing the storage location.

addressing, indirect - Addressing in which the address part of an instruction specifies a location containing an address.

address, symbolic - A label, alphabetic or alphanumeric, used to specify a storage location in the context of a particular program. Often programs are

written first using symbolic addresses in some convienent code which is then translated into actual addresses by an assembly program.

bit - An abbreviation of binary digit. A unit of information capacity of a storage device. A unit of data in binary notation (0 or 1).

branch - 1. To depart from the normal sequence of executing instructions in a computer. 2. A machine instruction that can cause such a departure.

byte - 1. A measureable portion of consecutive binary digits (8 bits). 2. A sequence of adjacent binary digits operated upon as a unit and usually shorter than a word.

code, object - The code produced by a compiler or special assembler which can be executed on a target machine.

code, target - The machine language code that is the final output of a coding system.

compile time - The time required to compile a program. Also the time at which a program is being compiled.

computer - A device capable of accepting information, applying prescribed processes to the information, and supplying the results of these processes. It usually consists of input and output devices, storage, arithmetic and logical units and a control unit.

concatenation - Uniting in a series, linking together, chaining. For example, when referring to a pair of 16 bit registers, their concatenation is considered to function as one 32 bit register.

constant - Any number that does not change from one execution of a program to the next.

constant, displacement - The address of a variable relative to the beginning of the data area in which it is stored. Also called offset or displacement.

execution time - The time required to execute a program. Also the time at which a program is being executed.

high order - Pertaining to the weight or significance

assigned to the digits of a number. The high order position is the leftmost position in a number or word.

infix notation - A method of forming one dimensional expressions (arithmetic, logical, etc.) by alternating single operands and operators. Any operator performs its indicated function upon its adjacent terms which are defined subject to the rules of operator precedence and grouping brackets which eliminates ambiguity.

instruction - A set of characters together with one or more addresses (or no addresses), that define an operation and which, as a unit, causes the computer to operate accordingly on the indicated quantities.

instruction set - The set of instructions defining the operations that a given computer is capable of performing.

instruction space - A part of storage allocated to receive and store the group of instructions to be executed. The storage locations used to store the program. Also instruction area.

language - A defined set of characters that is used to form symbols, words, etc., and the rules for combining these into meaningful communications.

language, assembly - The machine oriented programming language belonging to an assembly system.

language, FORTRAN - Programs are written directly as algebraic expressions and arithmetic statements. Various symbols are used to signify equality, addition, subtraction, exponentiation, etc. Additional statements are provided to permit control over how the algebraic expressions and arithmetic statements are to be processed. These include transfer, decision, indexing and input/output statements.

language, high-level programming - A computer programming language that is less dependent on the limitations of a specific computer; for instance, pseudo-languages; problem oriented languages; languages common to most computer systems, such as ALGOL, FORTRAN and COBOL; and user oriented languages.

language, machine - A language for expressing information that is intelligible to a specific machine (i.e. a computer). Such a language may include instructions that define and direct machine operations, and information to be recorded or acted upon by these machine operations.

language, object - A language which is the output of an automatic coding routine. Usually object language and machine language are the same. However, a series of steps in an automatic coding system may involve the object language of one step serving as a source language for the next step, and so forth. See object code.

language, source - The original form in which a program is prepared prior to processing by the machine.

language, target - The language into which some other language is to be translated.

low order - Pertaining to the weight or significance assigned to the digits of a number. The low order position is the rightmost position in a number or word.

machine, target - The computer which accepts the object program to execute the instructions, as contrasted to a computer that might be used to merely compile the object program from the source program.

memory - Any device into which a unit of information can be copied, which will hold this information and from which the information can be obtained at a later time. Synonomous with storage.

name - A term of one or more words or symbols to identify one of a general class of items, e.g. machine component, operation code, variable, etc.

operand - A piece of data upon which an operation is performed. The address or name portion of an operation. Any one of the quantities entering into or arising from an operation.

parenthesis level - The number by which left parenthesis exceed right parenthesis in an arithmetic statement when counting from left to right.

pointer - The address of (or a reference to) another

value. For example, an index of an array references (points to) the value contained in that array element.

pointer, stack - The address of the location at the top of a stack is often called the stack pointer and is held in a pre-assigned register.

procedure - A precise step-by-step method for effecting a solution to a problem.

program - A set of instructions or steps that tells the computer exactly how to handle a computer problem.

program counter - See program counter register.

register - A device for the temporary storage of one or more words to facilitate arithmetic, logical and transferral operations. Frequently referred to as "fast memory" due to the speed with which it can be accessed.

register, address - A register that is used by the control unit to calculate and hold addresses.

register, base - An index register which holds the value of a base address.

register, index - A register that permits automatic modification of an instruction address without permanently altering the instruction in memory.

register, program - Register in the control unit that stores the current instruction of the program and controls computer operation during the execution of the program. Synonomous with instruction register.

register, program counter - A register in which the address of the current instruction is stored. Synonomous with instruction counter.

routines - A sequence of machine instructions that carry out a well defined function (analogous to subroutines in FORTRAN).

storage - See memory.

symbol - A substitute or representation of characteristics, relationships or transformations of ideas or things.

token - A distinguishable unit in a sequence of characters.

variable - A symbol whose numeric value changes from one repetition of a program to the next, or changes within each repetition of a program.

variable, global - A variable whose name is known to a main program and all its subroutines (analogous to COMMON variables in FORTRAN).

variable space - A part of storage allocated to receive and store the variables, constants, temporary variables and parameters of a program.

variable, temporary - A variable which is used to represent intermediate or partial results which occur when evaluating an arithmetic expression.

word - A set of characteristics that occupies one storage location and is treated by the computer as a unit and transported as such. Ordinarily a word is treated by the control unit as an instruction and by the arithmetic unit as a quantity.

VITA

Personal History

Name: James Patrick Clancy
Date of Birth: December 13, 1943
Place of Birth: Brooklyn, New York
Parents: Joseph E. and Rita G. Clancy

Educational Background

Xaverian High School — Graduated - 1961
Brooklyn, New York

Manhattan College — Attended - 1961-1963
Riverdale, New York

Polytechnic Institute of Brooklyn — 1963-1971
Brooklyn, New York

Bachelor of Science
in Electrical Engineering

Lehigh University — 1972-1974
Bethlehem, Pennsylvania

Candidate for Master
of Science in Industrial
Engineering

Professional Experience

Consolidated Edison — 1963-1965
New York, New York
Technician

Western Electric Company, Inc. — 1965-1970
Newark, New Jersey
Engineering Associate

Western Electric Company, Inc. — 1970-1972
New York, New York
Pricing Specialist

Western Electric Company, Inc. — 1972-1974
Princeton, New Jersey
Development Engineer