

Code Optimization of Arithmetic Expressions in Stack Environments

HELEN B. CARTER

*Information Systems Staff
Communication Sciences Division*

September 12, 1974



NAVAL RESEARCH LABORATORY
Washington, D.C.

Approved for public release; distribution unlimited.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NRL Report 7787	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) CODE OPTIMIZATION OF ARITHMETIC EXPRESSIONS IN STACK ENVIRONMENTS		5. TYPE OF REPORT & PERIOD COVERED Interim report on a continuing problem
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Helen B. Carter		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Research Laboratory Washington, D.C. 20375		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NRL Problem B02-06 Task No. WF 21-241-601
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Naval Air Systems Command Washington, D.C. 20361		12. REPORT DATE September 12, 1974
		13. NUMBER OF PAGES 50
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
Common subexpressions	Expression reordering	
Computers, addressable-register	Register allocation	
Computers, stack	Stack-manipulation operations	
Computers, zero-address	stack optimization techniques	
Defense mechanism		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)		
<p>The allocation of registers has played a major role in code optimization. The use of a hardware stack removes the possibility of register allocation. Little work has been done on optimizing code for machines whose arithmetic units use hardware stacks, perhaps because the inherent characteristics of stacks are considered superior to the characteristics of addressable registers. This thesis demonstrates the importance of code optimization techniques for stack machines.</p> <p>One addressable register and three stack architectures are defined. Instructions generated from sample expressions for each of the machines are compared to determine which (Continued)</p>		

20. ABSTRACT (Continued)

architectures benefit from optimization. When programs with repeated operands are considered, stack machines execute the programs as efficiently as the addressable-register machine only when optimization techniques are applied. Comparison of the three stack mechanisms demonstrates that stack machines should be designed with facilities that are able to take advantage of optimizing software.

Stack optimization techniques are governed by the fact that operations can only be performed on values residing on the top of the stack. Two methods which minimize the number of memory accesses by reusing common variables and subexpressions are discussed. The first method involves expression reordering. The second method involves the use of stack manipulation operations which position needed data elements to the top of the stack. A stack optimization algorithm based on the second approach is presented.

CONTENTS

I. INTRODUCTION	1
II. BACKGROUND	2
III. DESCRIPTION OF FOUR ARCHITECTURES	4
A. Addressable-Register Machine	5
B. Parenthetical Control Machine	6
C. Manipulative Stack Machine	7
D. Pure Stack Machine	8
IV. OPTIMIZATION TECHNIQUES	9
V. COMPARISON OF CODE GENERATED FOR EACH MACHINE	11
VI. OPTIMIZATION ON THE PURE STACK MACHINE	19
VII. CONCLUSION	24
ACKNOWLEDGMENT	24
BIBLIOGRAPHY	24
APPENDIX A—Sample Code Generated by the Algol and Fortran Compilers of the Burroughs B6700	27
APPENDIX B—Stack Optimization Algorithm	31
APPENDIX C—Examples of Code Generated by the Algorithm	46

CODE OPTIMIZATION OF ARITHMETIC EXPRESSIONS IN STACK ENVIRONMENTS

I. INTRODUCTION

“There are sceptical thoughts, which seem for the moment to uproot the firmest faith; there are blasphemous thoughts, which dart unbidden into the most reverent souls; there are unholy thoughts which torture, with their hateful presence, the fancy that would fain be pure. Against all these some mental work is a most helpful ally.”

Lewis Carroll

Code optimization for addressable-register machines such as the Univac 1108 has become a major part of compiler writing (12, p. 764-765). The existing compilers for the Burroughs B5700 and B6700 stack machines perform little optimization (Appendix A). Perhaps stack optimization techniques were not considered necessary because a hardware stack implicitly performs some functions that must be explicitly performed by software in addressable-register machines—for example, the storage of a temporary result. In order to clarify some of the architectural tradeoffs relevant to the choice between stack and addressable-register machines, stack optimization should be considered. This thesis examines optimization of arithmetic expressions in several different stack environments and compares optimized code generated for stack and general register machines.

As a basis for comparison, one addressable register and three stack architectures are defined. An effort was made to keep all features of the machine the same with the exception of the arithmetic mechanisms being compared. Since a machine design is an integrated combination of its features, the alteration of one feature influences the design of other machine components. The machine comparison is not intended to determine which of the four architectures is best. This is not possible because there are too many variables which are not considered. The comparison is used to demonstrate the need for optimization on stack machines.

Chapter II contains background material on stack architectures and discusses the inherent attributes of stack and addressable-register configurations. Chapter III defines the four machine architectures. Chapter IV discusses the optimization techniques used in our analysis. Objective comparison of the code generated for each architecture requires that optimization be performed at the same level. Our analysis is restricted to techniques which are locally performed on basic blocks* of code.

*A basic block is a segment of code which contains a single entry and exit point.

Note: Manuscript submitted June 20, 1974.

In Chapter V, instructions generated for each machine to evaluate representative expressions are compared. The comparison is based on the following parameters which measure the efficiency of representation and execution:

1. number of bits comprised by the instructions.
2. number of explicit data memory accesses.
3. number of stack or addressable registers utilized.

The machine comparisons are analyzed on two levels. The use of an addressable-register set is compared with the use of a stack and the three stack mechanisms are compared. Analysis of expressions containing repeated operands and common subexpressions demonstrates that the addressable-register machine is more efficient than the stack architectures in executing unoptimized code. However, when optimized expressions are considered, stack machines are, in general, as efficient. In particular, one stack architecture executes optimized code more efficiently than the addressable-register machine. This implies that improved software for stack machines should be developed to compete with the present register optimization techniques. The comparison among the stack mechanisms demonstrates that the application of optimization increases the efficiency on two of the three stack machines. The other stack architecture permits little compile-time optimization.

To substantiate the conclusion that stack machines can be optimized, Chapter VI defines a stack optimization algorithm for one of the stack architectures. The design of the algorithm is based on the circular rotation of the contents of the stack registers. The algorithm generates code for sample programs that executes as efficiently as hand-optimized code for conventional register machines. Different ways to implement the stack architecture for which the algorithm is defined are also discussed in this chapter.

II. BACKGROUND

“Contrariwise,” continued Tweedledee,
“if it was so, it might be; and if it were so,
it would be; but as it isn’t, it ain’t.
That’s logic.”

Lewis Carroll
Through the Looking Glass

An arithmetic stack is a first-in, last-out memory which functions as a temporary store for data values. Typically, a stack consists of a limited number of high-speed registers on its top with the rest of the stack extending into main memory. Restricting the maximum stack depth to the available number of registers prevents implicit memory accesses caused by the overflow into memory of the hardware portion of the stack. A stack with a limited number of hardware registers on its top may be compared to a general register set with a similar number of directly addressable registers. In stack machines when a register is needed, the contents of the bottom register of the stack are implicitly pushed into main memory; in general register machines, the contents of a selected register is explicitly stored into main memory.

While designing a stack architecture, one must consider the tradeoffs between the instruction stream* length and the number of memory accesses required to represent and execute a program. The implicit storage of values into stack memory tends to reduce the length of the instruction stream because explicit register address fields are not needed in the instructions. However, in a typical stack machine only the top one or two registers may be accessed. Values within the stack may be retrieved by performing operations that position the values at the stack's top. Specification of the stack-manipulation operations tends to increase the length of the instruction stream.

Stack-manipulation operations may be specified by separate instructions or by special codes in existing instructions. The first method requires the definition of new operation codes; the second method requires extending the length of the instruction word. When two operations are specified in a single instruction, one operation (e.g., load, add) is indicated by the operation code and the secondary operation (e.g., duplicate value on stack, store result in location specified by the address field), by the special code. Representation of a single stack-manipulation operation in a separate instruction requires more storage than its representation in an existing instruction. However, the reservation of a fixed number of bits in each instruction word for a frequently unused code may result in a total net increase of the instruction stream length.

The decision to use stack-manipulation operators at all is dependent upon the hardware stack depth. If there are enough registers in which values can be duplicated and saved, stack operations which position the values in the top-of-stack registers eliminate otherwise needed memory accesses.

The instruction stream of the Burroughs B6700 corresponds to the postfix notation of expressions (8, Section 3). The operand symbol corresponds to a load instruction which pushes the operand value or address specified in the instruction onto the stack. The operator symbol corresponds to an instruction consisting only of an operation code. The operation is performed on the top one or two elements of the stack. This type of computer is called a zero-address machine because the only instructions with address fields are those that load values onto the stack.

Zero-address stack machines are not the only existing stack computers* and are not necessarily the most efficient. Replacing several zero-address instructions of the B6700 by single-address instructions appears to increase the machine's efficiency. For example, storage of data values into main memory requires two B6700 instructions. A single-address load instruction places the memory address onto the stack; then, a zero-address store instruction stores the contents of the second-to-top-of-stack cell into the memory location specified by the top-of-stack cell. Inclusion of the store operation code and the memory address in the same instruction eliminates the need to temporarily save the address on the stack. Additional load instructions can be avoided if instructions performing arithmetic and logical operations contain address fields for their corresponding operands. These ideas are explored in Chapter III, where two of the three stack architectures defined have no zero-address instructions.

*Instruction stream is the collection of machine instructions used to perform some task within a program.

*The HP 3000 is a single-address stack machine (3).

III. DESCRIPTION OF FOUR ARCHITECTURES

“The thing can be done,” said the Butcher,
“I think.
The thing must be done. I am sure.
The thing shall be done! Bring me paper
and ink,
The best there is time to procure.”

Lewis Carroll
The Hunting of the Snark

The four machine architectures described in this chapter are called the Addressable-Register Machine, the Parenthetical Control Machine, the Manipulative Stack Machine, and the Pure Stack Machine. A general overview of the machine designs is first presented. This is followed by a more detailed description of each machine.

Each machine has eight general-purpose registers. In the Addressable-Register Machine, the registers are directly addressable and in the other three machines, the registers are the top of a stack whose bottom portion extends into main memory. Indexing in all machines is performed by the use of seven addressable index registers. The first three machines have fixed-length instructions of 32 bits each. The Pure Stack Machine has single-address 32-bit instructions and zero-address 8-bit instructions. The single-address instruction words of all machines contain a 16-bit memory address and an indirect bit.

Effective address calculation is the same for all four machines. When the index field is nonzero, the contents of the index register is added to the memory address to produce a modified address. If the indirect bit is zero, addressing is direct and the modified address is the effective address used in the execution of the instruction. If the indirect bit is one, addressing is indirect and another address word is retrieved from the location specified by the modified address already determined. Indexing and indirect addressing are performed on the new word in exactly the same manner. This process continues until some referenced location is found whose indirect bit is zero.

The instruction set may be divided into two classes: the class of instructions which alters the addressable or stack registers and the class which alters the index registers. In the first three machines, 9-bit operation codes represent the typical load, store, transfer, algebraic, logical and index register manipulation operations. In the Pure Stack Machine, 12-bit operation codes of the single-address instructions are used for loading and storing operands and manipulating index registers. The 8-bit zero-address instructions perform algebraic, logical, and stack manipulation operations.

The functions performed by one 3-bit field of the instruction demonstrate the particular characteristic in each of the machines that is being compared. The 3-bit field in the Addressable Register Machine addresses one of the eight general-purpose registers. On the Parenthetical Control Machine, the three bits control the order in which instructions are executed. An operand and operator are pushed onto the stack when the execution of that particular operation is to be deferred. The instruction stream corresponds to infix notation and the contents of the 3-bit field serves the same function as parentheses.

On the Manipulative Stack Machine, the 3-bit field specifies where operands are located and where the results of operations should be placed. An operand may be on the stack or may be referenced by the address field of the instruction. The result of an operation may be placed in one or both of the two top-of-stack registers or in the memory location specified in the address field. On the Pure Stack Machine an operation may only be performed on operands residing on the stack. The three bits are used to extend the operation code length so that the instruction set includes manipulative stack operators which perform such functions as deletion, exchange, duplication, and rotation of elements on the stack. In the Manipulative Stack Machine, stack manipulation is performed as part of the other instructions. In the Pure Stack Machine, stack-manipulation operators are separate instructions.

A. Addressable-Register Machine

The 32-bit instruction word of the Addressable-Register Machine contains the following fields:

- 9-bit operation code (GOP for general register opcode, XOP for index register opcode)
- 1 indirect bit
- 3-bit index register address (I1-I7 for seven index registers)
- 3-bit general register address (R0-R7 for eight general registers)
- 16-bit memory address (A)

The eight general registers are referenced as the first eight locations of memory. When the memory address field contains the values 0 through 7, in a non-immediate instruction, the value refers to a general register*.

The symbol @ preceding a memory address indicates that the indirect bit is set. Placing one of the index register mnemonics (I1-I7) in parentheses following the memory address indicates that the address is to be modified by the contents of the specified register. The two classes of instructions are represented by:

GOP @A(Ii),Rn
XOP @A,Ii

where $i = 1, \dots, 7$ and $n = 0, \dots, 7$. Immediate instructions are defined by operation codes with an "I" appended (e.g., GOPI).

*The use of the address field to reference a register wastes 13 bits of the instruction word. Many machines (e.g., IBM 360) have shorter length instructions for register-to-register operations. The use of variable-length instructions on the Addressable-Register Machine does not fit in with the general design of the four architectures. The unused 13 bits in register-to-register instructions will be considered in our final analysis.

B. Parenthetical Control Machine**

Instructions of the Parenthetical Control Machine are sequentially executed until a parenthesized term is encountered. The current operation is automatically deferred until the term is computed. The 3-bit field of each instruction controls sequencing of operations as follows:

<i>Code</i>	<i>Mnemonic</i>	<i>Action Taken</i>
0	None	Normal execution of instruction.
7	(Deferral of instruction. The operation code is first stored into the current top-of-stack register containing already computed arithmetic results. The operand is then stored into a new register which becomes the top of the stack. Processing on the next instruction begins.
1-6), . . . ,)))))	The operation specified by the current instruction is first performed and the result is treated as an operand for the most recently deferred operation. For each close parenthesis, the deferred operation specified in the second-to-top-of stack register is performed on the operands in the two top-of-stack registers.

Source programs written for the Parenthetical Control Machine are translated into object programs corresponding to infix notation. Translation involves removing the unnecessary explicit parentheses and inserting the needed implicit parentheses. A translation algorithm (14, p. 22-23) demonstrates that it is never necessary to associate more than one begin parenthesis with the operand specified in a single instruction.

To demonstrate the deferral mechanism, consider the expression $A+B*(C+D*E*F)$ which is translated to $A+(B*(C+(D*E*F)))$. The following table contains the instructions which are generated for the expression. The second column displays the stack contents after execution of each instruction. The symbol ',' separates the cells of the stack and the rightmost data item represents the contents of the top stack cell.

**This machine has been modeled after the Data Processing Element of the All Applications Digital Computer (AADC) which is currently being designed as the Navy's standard computer for the next generation (13). The stack mechanism of the Parenthetical Control Machine is very similar to the deferral mechanism (14) of the AADC.

<i>Instructions</i>	<i>Stack Contents</i>	<i>Sequence of Operations</i>
LOAD A	A	
ADD (B	A+; B	'+' is deferred
MULT (C	A+; B*; C	'*' is deferred
ADD (D	A+; B*; C+; D	'+' is deferred
MULT E	A+; B*; C+; D*E	'*' is performed
MULT F)))	A+; B*; C+; D*E*F	'*' is performed
	A+; B*; C+(D*E*F)	deferred '+' is performed
	A+; B*(C+(D*E*F))	deferred '*' is performed
	A+(B*(C+(D*E*F)))	deferred '+' is performed

The 32-bit instruction word of the Parenthetical Control Machine contains the same fields as the Addressable-Register Machine with the exception that the 3-bit general register field is replaced by the 3-bit parenthetical control field. Stack opcodes (SOP) take the place of the general register opcodes. An instruction is represented by one of the following forms:

SOP @A(i)

XOP @A,i

where $i = 1, \dots, 7$. A single left parenthesis or up to six right parentheses may be included with the operand name.

C. Manipulative Stack Machine

The instruction word format and mnemonic representation of the Manipulative Stack Machine* are the same as those of the Parenthetical Control Machine with the exception of the interpretation and representation of the 3-bit field. The use of the bits is shown in the following table. The normal mode of operation which uses the memory address as the source of one of the operands occurs when all bits are zero. When a bit is set, the corresponding mnemonic is appended to the operation name.

<i>Bit</i>	<i>Mnemonic</i>	<i>Action Taken When Bit Is Set</i>
1	S	Operation is performed on the top two stack elements.
2	M	Memory address is used as the destination of the operation.
3	D	Result of operation is duplicated on top of the stack.

The use of the bits is demonstrated by considering the addition operation. In the following table, assume E is the effective operand, T1 is the contents of the top-of-stack register, and T2 is the contents of the second-to-top-of-stack register.

*The design of this machine was motivated by Parnas (26).

<i>Opcode</i>	<i>Function</i>	<i>Description</i>
ADD	$E+T1 \rightarrow T1$	Addition is performed on the effective operand and the top stack element. The result is stored on the top of the stack.
ADDS	$T1+T2 \rightarrow T1$	Addition is performed on the top two stack elements. The result is stored on the top of the stack.
ADDM	$E+T1 \rightarrow E$	Addition is performed on the effective operand and the top stack element. The result is stored in the memory location of the effective operand.
ADDSM	$T1+T2 \rightarrow E$	Addition is performed on the top two stack elements. The result is stored in the memory location of the effective operand.
ADDD	$E+T1 \rightarrow T1$ $\rightarrow T2$	Addition is performed on the effective operand and the top stack element. The result is stored in the top two stack registers.
ADDSD	$T1+T2 \rightarrow T1$ $\rightarrow T2$	Addition is performed on the top two stack elements. The result is stored in the top two stack registers.
ADDMD	$E+T1 \rightarrow E$ $\rightarrow T1$	Addition is performed on the effective operand and the top stack element. The result is stored in the memory location of the effective operand and on the top of the stack.
ADDSMD	$T1+T2 \rightarrow E$ $\rightarrow T1$	Addition is performed on the top two stack elements. The result is stored in the memory location of the effective operand and on the top of the stack.

D. Pure Stack Machine

The Pure Stack Machine has two instruction formats. The 8-bit zero-address instructions are operations which are performed on elements of the stack. The 32-bit instructions load data onto the stack, store data from the stack into main memory, and alter the contents of the index registers.

The 32-bit instruction word has the following fields:

12-bit operation code (SOP for stack opcodes, XOP for index register opcodes)

1 indirect bit

3-bit index register address (I1-I7 for seven index registers)

16-bit memory address (A)

There are two store operation codes in the class of stack instructions. The STND (store nondestructive) instruction stores the value on top of the stack into the specified memory location without destroying the value on the stack. The STD (store destructive) instruction deletes the top stack element after storage is performed.

The 8-bit stack instructions include:

1. Logical and algebraic operations which are performed on the top one or two stack elements
2. The DUP instruction which duplicates the top element of the stack
3. The XCII instruction which exchanges the top two stack elements
4. The ROTD (rotate down) instruction which permutes the top eight operands of the stack by rotating the value on top of the stack to the bottom of the stack's register set
5. The ROTU (rotate up) instruction which permutes the top eight operands of the stack by rotating the value at the bottom of the stack's register set to the top of the stack

On a functional level, the rotate instructions perform multiregister circular shifts.

The mnemonic representation of the 32-bit instruction is the same as for the previously described stack machines. The 8-bit instructions are simply represented by their operation code mnemonics.

IV. OPTIMIZATION TECHNIQUES

“Can you do Addition?” the White Queen
asked.

“What’s one and one and one and one and
one and one and one and one and one and one?”

Lewis Carroll
Through the Looking Glass

This chapter discusses the optimization techniques used to generate code for the machines considered in our analysis. A good background on compiler optimization appears in Cocke and Schwartz (12). It should be emphasized that most optimization techniques have been developed for single-accumulator or multiregister machines.

Optimization procedures may be classified into two distinct categories: *machine-independent* techniques performed on the source program and *machine-dependent* techniques performed on the object language.

Machine-independent optimizations make use of a general set of transformations for a large class of machines. *Common subexpression elimination* prevents redundant calculations and memory fetches. *Dead variable elimination* removes statements that assign values to variables which are not used again in the program. *Constant propagation* performs calculations on operand values known at compile-time.

Machine-dependent optimizations are governed by the structure of the target machine. The effectiveness of these techniques in minimizing the time-storage cost function depends upon the register configuration and the operations permitted by the instruction set.

The sample expressions considered in our analysis are translated from the original source language into either an intermediate postfix or parenthesized infix form. In Chapter V instructions are generated from two translations: *pure translation* in which the operands appear in the same order in the original and translated expressions and *reordered translation** which involves reordering within and between expressions. We will assume that code generated from a pure translation is unoptimized and code generated from a reordered translation is optimized.

On the machine-independent level, reordering helps to identify common subexpressions. On the machine-dependent level, reordering has a different effect on each machine. The order that the operands appear in the translated program is the order that they are assigned to registers or are placed on the stack. On the Addressable-Register Machine, the order that values are assigned to registers is not critical because the contents of any register can be accessed at any time during execution. In fact, analysis of sample expressions shows that reordering does not increase the efficiency of the Addressable-Register Machine.

On the Manipulative Stack and Pure Stack Machines, the order that operands are placed on the stack is important because values may only be reused when they appear on top of the stack. As a result, the order of repeated operands in the translated program is critical to efficient execution on these machines.*

Optimal reordering of expressions is not as important in the Parenthetical Control Machine as in the other stack machines because the hardware of the Parenthetical Control Machine does not permit the application of optimization techniques which eliminate common subexpressions.** There is no facility to retain a value on the top of the stack after the value is used in a calculation. A value in one stack register cannot be duplicated into another stack register.

*Operands within an expression are reordered according to the mathematical rules of commutativity. We will ignore the noncommutativity of certain multiplication and addition operations caused by machine idiosyncrasies.

*On the Pure Stack Machine, use of the ROTU and ROTD operators which shift needed data values from within the stack to the top of the stack is an alternative to expression reordering. The stack optimization algorithm described in Chapter VI demonstrates how rotation of the stack registers increases program efficiency.

**The motivating factor behind the design of the deferral mechanism for the AADC was, in fact, to relieve the software burden (14).

Software on the Manipulative Stack and Pure Stack Machines decides if the storage of an element into memory destroys its value on the top of the stack. The Parenthetical Control Machine does not have the facility to make the decision. The value stored into memory always remains on the top of the stack.

V. COMPARISON OF CODE GENERATED FOR EACH MACHINE

“Explain all that,” said the Mock Turtle.
 “No, No! The adventures first,” said the
 Gryphon in an impatient tone: “Explanations
 take such a dreadful time.”

Lewis Carroll
Alice's Adventures in Wonderland

Three basic blocks are considered in the machine comparison. The first example consists of one expression. Code for the expression is generated from the pure translation only, because there are no common subexpressions or repeated variables. The second example is a single expression containing a common subexpression. The third example is a set of expressions that contain several repeated variables. In the latter two examples, code generated from both the pure and reordered translations is examined.

Example 1: $Z = (A(J)+B)*(C+D)$

This example demonstrates the use of index registers. The code generated for each of the machines is displayed in Figure 1. Since there are no repeated operands, optimization techniques to eliminate common subexpressions are not applicable. The machines execute the expression at approximately the same cost. The Parenthetical Control and Manipulative Stack Machines both require one less instruction than the Addressable-Register Machine. The deferral mechanism of the Parenthetical Control Machine tends to reduce the number of loads since operands associated with deferred operators are automatically loaded onto the stack. In the Manipulative Stack Machine, a store is avoided because the address field of the instruction performing the final operation contains the destination of the expression. The Pure Stack Machine requires more program storage because all loads onto the stack are explicit.

Example 2: $A = B*C+D*5+B*C*E$

Code generated from the following pure translated forms appears in Figure 2:

postfix	$BC*D5*+BC*E+A: =$
infix	$A: = B*C+(D*5) + (B*C*E)$

The Addressable-Register Machine executes the expression most efficiently because the subexpression $B*C$ is in a register when it is needed the second time. In the stack machines, $B*C$ has already been used in calculation. As a result, the values of B and C are redundantly fetched from memory and multiplied.

		BITS	MEMORY ACCESSES	REGISTERS
XLOAD	J,I1	224	6	2
LOAD	A(I1),R0			
ADD	B,R0			
LOAD	C,R1			
ADD	D,R1			
MULT	R0,R1			
STOR	Z,R1			
(A) ADDRESSABLE-REGISTER MACHINE				
XLOAD	J,I1	192	6	2
LOAD	A(I1)			
ADD	B			
MULT	(C			
ADD	D)			
STOR	Z			
(B) PARENTHETICAL CONTROL MACHINE				
XLOAD	J,I1	192	6	2
LOAD	A(I1)			
ADD	B			
LOAD	C			
ADD	D			
MULTM	Z			
(C) MANIPULATIVE STACK MACHINE				
XLOAD	J,I1	216	6	3
LOAD	A(I1)			
LOAD	B			
ADD				
LOAD	C			
LOAD	D			
ADD				
MULT				
STD	Z			
(D) PURE STACK MACHINE				

Figure 1. Pure translation of Example 1.

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B,R0	B IN R0	256	5	2
MULT C,R0	B*C IN R0			
LOAD D,R1	D IN R1			
MULTI 5,R1	D*5 IN R1			
ADD R0,R1	B*C+D*5 IN R1			
MULT E,R0	B*C*E IN R0			
ADD R1,R0	B*C+D*5+B*C*E IN R0			
STOR A,R0				

(A) ADDRESSABLE-REGISTER MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	256	7	2
MULT C	B*C			
ADD (D	B*C+; D			
MULTI 5)	B*C+; D*5			
	B*C+D*5			
ADD (B	B*C+D*5+; B			
MULT C	B*C+D*5+; B*C			
MULT E)	B*C+D*5+; B*C*E			
	B*C+D*5+B*C*E			
*STOR A	A			

*ALL STORES ON THIS MACHINE ARE NONDESTRUCTIVE.

(B) PARENTHETICAL CONTROL MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	288	7	2
MULT C	B*C			
LOAD D	B*C; D			
MULTI 5	B*C; D*5			
ADDS	B*C+D*5			
LOAD B	B*C+D*5; B			
MULT C	B*C+D*5; B*C			
MULT E	B*C+D*5; B*C*E			
ADDM A	B*C+D*5+B*C*E			
	EMPTY			

(C) MANIPULATIVE STACK MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	304	7	3
LOAD C	B; C			
MULT	B*C			
LOAD D	B*C; D			
LOADI 5	B*C; D; 5			
MULT	B*C; D*5			
ADD	B*C+D*5			
LOAD B	B*C+D*5; B			
LOAD C	B*C+D*5; C			
MULT	B*C+D*5; B*C			
LOAD E	B*C+D*5; B*C; E			
MULT	B*C+D*5; B*C*E			
ADD	B*C+D*5+B*C*E			
STD A	EMPTY			

(D) PURE STACK MACHINE

Figure 2. Pure translation of Example 2.

One possible reordering reverses the terms $B*C*E$ and $D*5$. Figure 3 displays code generated for each of the machines from the following reordered translations:

```

postfix    BC*BC*E*+D5*+A =
infix      A: = B*C+(B*C*E) + (D*5)

```

The code generated for the Addressable-Register Machine resulting from the reordered translation involves one more instruction and one more register than the code generated from the pure translation. The result of the first addition operation in Figure 2(a) is placed into R1, and, as a result, the value $B*C$ is preserved in R0. In Figure 3(a) reuse of $B*C$ requires loading its value into a second register.

The code generated from the reordered translation for the Parenthetical Control Machine has the same efficiency as code generated from the pure translation. Since the contents of the stack is heavily controlled by hardware, software is not permitted to duplicate repeated variables or common subexpressions within an expression.

On the remaining two stack machines, the code resulting from the reordered translation is more efficient than the code resulting from the pure translation. On the Manipulative Stack Machine, the use of the 3-bit field to duplicate $B*C$ on the stack eliminates two instructions and two memory accesses. On the Pure Stack Machine, an 8-bit duplication instruction similarly eliminates the redundant calculation of $B*C$.

In a recent study of Fortran programs (21), the number of occurrences of easily recognizable syntactic constructions was counted. It was determined from the data that most Fortran statements used in practice are quite simple in form. A static analysis of the programs showed that 68% of 250,000 statements were trivial replacements of the form $A = B$ where no arithmetic operations were present and 24% were of the form $A = BopC$ where a single operation was present. In many cases, the first variable on the right was the same as the variable on the left ($A = AopB$).

The following basic block is representative of existing Fortran programs:

```

Example:3: A: = A+1
           B: = A
           C: = B*2
           D: = D+A
           E: = D

```

Code generated for the pure translations appears in Figure 4. The Addressable-Register Machine requires one less memory fetch than the other machines since the value of A is retained in a register.

Reordering does not alter the efficiency in the first two machines. However, changing the order of the operands in expressions $A: = A+1$ and $D: = D+A$ reduces the number of instructions and memory accesses required by the Manipulative Stack and Pure Stack Machines to execute the statements. Code resulting from the reordered translated program $1A+A: = AB: = B2*C: =AD+D: =DE: =$ for the Manipulative Stack and Pure Stack Machines is displayed in Figure 5.

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B,R0	B IN R0	288	5	3
MULT C,R0	B*C IN R0			
LOAD R0,R1	B*C IN R1			
MULT E,R1	B*C*E IN R1			
ADD R0,R1	B*C+B*C*E IN R1			
LOAD D,R2	D IN R2			
MULTI 5,R2	D*5 IN R2			
ADD R1,R2	B*C+B*C*E+D*5 IN R2			
STOR A,R2				

(A) ADDRESSABLE-REGISTER MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	256	7	2
MULT C	B*C			
ADD (B	B*C+; B			
MULT C	B*C+; B*C			
MULT E)	B*C+; B*C*E			
	B*C+B*C*E			
ADD (D	B*C+B*C*E+; D			
MULTI 5)	B*C+B*C*E+; D*5			
	B*C+B*C*E+D*5			
STOR A	A			

(B) PARENTHETICAL CONTROL MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	224	5	2
MULT D C	B*C; B*C			
MULT E	B*C; B*C*E			
ADDS	B*C+B*C*E			
LOAD D	B*C+B*C*E; D			
MULTI 5	B*C+B*C*E; D*5			
ADDSM A	EMPTY			

(C) MANIPULATIVE STACK MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	240	5	3
LOAD C	B; C			
MULT	B*C			
DUP	B*C; B*C			
LOAD E	B*C; B*C; E			
MULT	B*C; B*C*E			
ADD	B*C+B*C*E			
LOAD D	B*C+B*C*E; D			
LOAD 5	B*C+B*C*E; D; 5			
MULT	B*C+B*C*E; D*5			
ADD	B*C+B*C*E+D*5			
STD A	EMPTY			

(D) PURE STACK MACHINE

Figure 3. Reordered translation of Example 2.

HELEN B. CARTER

		BITS	MEMORY ACCESSES	REGISTERS
LOAD A,R0	A IN R0	320	7	2
ADD 1,R0	A+1 IN R0			
STOR A,R0				
STOR B,R0				
LOADI 2,R1	2 IN R1			
MULT R0,R1	2*B IN R1			
STOR C,R1				
ADD D,R0	D+A IN R0			
STOR D,R0				
STOR E,R0				

(A) ADDRESSABLE-REGISTER MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD A	A	320	8	1
ADDI 1	A+1			
STOR A	A			
STOR B	B			
MULTI 2	B*2			
STOR C	C			
LOAD D	D			
ADD A	D+A			
STOR D	D			
STOR E	E			

(B) PARENTHETICAL CONTROL MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD A	A	320	8	1
ADDI 1	A+1			
STORD A	A			
STORD B	B			
MULTI 2	B*2			
STOR C	EMPTY			
LOAD D	D			
ADD A	D+A			
STORD D	D			
STOR E	EMPTY			

(C) MANIPULATIVE STACK MACHINE

		BITS	MEMORY ACCESSES	REGISTERS
LOAD A	A	354	8	2
LOADI 1	A; 1			
ADD	A+1			
STND A	A			
STND B	B			
LOADI 2	B; 2			
MULT	B*2			
STD C	EMPTY			
LOAD D	D			
LOAD A	D; A			
ADD	D+A			
STND D	D			
STD E	EMPTY			

(D) PURE STACK MACHINE

Figure 4. Pure translation of Example 3.

		BITS	MEMORY ACCÈSSES	REGISTERS
LOAD 1	1	256	6	1
ADDMD A	1+A			
STORD B	B			
MULTI 2	B*2			
STOR C	EMPTY			
LOAD A	A			
ADDMD D	A+D			
STOR E	EMPTY			

(A) MANIPULATIVE STACK MACHINE

LOAD 1	1	320	7	2
LOAD A	1; A			
ADD	A			
STND A	A			
DUP	A; A			
STND B	A; B			
LOADI 2	A; B; 2			
MULT	A; B*2			
STD C	A			
LOAD D	A; D			
ADD	A+D			
STND D	D			
STD E	EMPTY			

(B) PURE STACK MACHINE

Figure 5. Reordered translation of Example 3.

The most efficient code for the Addressable-Register Machine appears in Figure 4(a) and most efficient code for the Manipulative Stack Machine appears in Figure 5(a). The Manipulative Stack Machine requires one less memory fetch and 20% fewer bits to execute the expressions.

Our conclusions are based on the analysis of the instruction stream length and the number of memory accesses. Little emphasis has been placed on the number of registers because the code generated from our sample expressions does not use the maximum number of addressable or stack registers. We are primarily concerned with the execution of basic blocks containing subexpressions and repeated variables because execution of expressions with no repeated operands has approximately the same time and storage cost on each machine.

The results are summarized in Table 1.

Example	Machine*	Pure Translation Memory			Reordered Translation Memory		
		Bits	Accesses	Registers	Bits	Accesses	Registers
1	1	224(208†)	6	2			
	2	192	6	2			
	3	192	6	2			
	4	216	6	3			
2	1	256(224†)	5	2	288(240†)	5	3
	2	256	7	2	256	7	2
	3	288	7	2	224	5	2
	4	304	7	3	240	5	3
3	1	320(304†)	7	2	320(304†)	7	2
	2	320	8	1	320	8	1
	3	320	8	1	256	6	1
	4	354	8	2	320	7	2

*Machine code:

1. Addressable-Register Machine
2. Parenthetical Control Machine
3. Manipulative Stack Machine
4. Pure Stack Machine

†These values for the Addressable-Register Machine Assume that the register-to-register operations are performed in 16-bit instructions, similar to the IBM 360.

We will first compare the Addressable-Register Machine with the stack machines. When code is generated from a pure translation, the Addressable-Register Machine executes the code most efficiently. When code is generated from a reordered translation, the Manipulative Stack Machine executes the code most efficiently. The parameter that distinguishes the Manipulative Stack Machine from the Addressable-Register Machine is the instruction stream length. The Manipulative Stack Machine requires the fewest number of bits to represent its object code because of the compact representation of its register-to-register transfer (duplicate) and register-to-memory transfer operations.

The Addressable-Register and Parenthetical Control Machines are similar in the sense that expression reordering does not increase the efficiency of either machine. However, the unoptimized code for the Addressable Register Machine is more efficient than the unoptimized code for the Parenthetical Control Machine because the latter does not have the facility to eliminate redundant memory accesses. Expression reordering on the Pure Stack Machine results in code that executes as efficiently as the Addressable-Register Machine.

Comparison of unoptimized code for the three stack machines shows the Pure Stack Machine to perform least efficiently because its explicit load instructions increase the length of the instruction stream. Comparison of optimized code shows the Parenthetical Control Machine to perform least efficiently because its hardware does not permit the application of optimization. Optimization techniques on the Manipulative and Pure Stack Machines utilize the specific capabilities of each machine to increase efficiency. To substantiate the conclusion that different stack architectures can take advantage of optimization techniques, an optimization algorithm for the Pure Stack Machine is defined in the next chapter.

VI. OPTIMIZATION ON THE PURE STACK MACHINE

“Would you tell me please, which way I
ought to go from here?”
“That depends a good deal on where you
want to get to,” said the Cat.
“I don’t much care where-” said Alice.
“Then it doesn’t matter which way you go,”
said the Cat.
“-so long as I get somewhere,” Alice
added as an explanation.
“Oh, you’re sure to do that,” said the Cat,
“if you only walk long enough.”

Lewis Carroll
Alice’s Adventures in Wonderland

Additional optimization techniques that may be applied to the Pure Stack Machine are defined in this chapter. Analysis of representative programs in the preceding chapter shows that the maximum stack depth never exceeds three. As a result, we assume that the Pure Stack Machine has a fixed-length eight-register stack. The cost of overflow of the hardware portion of the stack into main memory is not considered. In addition to

defining an optimization algorithm, this chapter discusses different ways that the registers might be used to configure the stack.

An alternative to expression reordering is rotating data values from within the stack to the top of the stack when they are needed. The stack optimization algorithm* (Appendix B) eliminates all unnecessary memory fetches by taking advantage of the stack operations which exchange and rotate data elements on the stack. The XCH, ROTD and ROTU instructions are defined in Chapter III. The algorithm is divided into two functional sections. The first section which identifies common subexpressions and repeated variables performs the following sequence of operations:

1. Read source program in postfix form.
2. Reorder terms of commutative operations so that common subexpressions can be identified.
3. Identify the largest common subexpressions.
4. Reformat the postfix program to specify the occurrence of the common subexpressions. The first occurrence of a subexpression is followed by an identifying symbol. Subsequent occurrences of the subexpression are replaced by the symbol.
5. Obtain a count of the number of times each unique variable or subexpression appears in the program. (The variables which comprise the subexpressions are not counted.)

The second section is the code generation phase which reads one symbol of the modified postfix program at a time:

1. If the symbol is a variable which is encountered the first time in the program, its value is loaded from memory onto the stack. If the variable is repeated later in the program, its value is duplicated on top of the stack.
2. If the symbol identifies the first occurrence of a common subexpression, the value of the subexpression is duplicated on top of the stack.
3. If the symbol is a constant, the value is loaded onto the stack.
4. If the symbol is a variable or subexpression whose value is located in some stack register, one of two actions is taken. If the operand is repeated later in the program (it has already been repeated two or more times), it is shifted to the top of the stack by the minimal number of ROTU or ROTD instructions and is then duplicated. If the operand is not repeated, it remains where it is and will be shifted to the stack's top when the operator symbol is encountered.
5. If the symbol is an assignment operator, a store nondestructive instruction is generated if the variable whose value is being altered is repeated later in the program; otherwise, a store destructive instruction is generated.
6. If the symbol is a dyadic operator, both operands are already on the stack. The fewest possible XCH, ROTD and ROTU instructions are generated to position the operands at the top of the stack.

*The algorithm is written in the language SIMPL-T (5).

7. If the symbol is a monadic operator and if the operand is not on the top of the stack, the fewest necessary ROTU or ROTD instructions are generated to bring it there.

To demonstrate how the algorithm works, the code generated for the expressions

$$B: = B * C + D$$

$$E: = D * B + C$$

appears in Figure 6(a). Additional examples of code generated by the algorithm appear in Appendix C.

The code resulting from the expression reordering technique used in the machine comparisons is not as efficient as the code generated by the stack optimization algorithm. The best reordered translation of the previous example is $BC * D + B: = BD * C + E: =$. Figure 6(b) displays code generated for the Pure Stack Machine from the reordered translation. No matter what reordering of the expression is used, at least two of the three repeated variables must be fetched twice from memory.

The code in Figure 6(a) compares favorably with hand-optimized code for the Addressable-Register Machine which appears in Figure 6(c). In both the Pure Stack and Addressable Register Machines, the storage and time cost functions are approximately the same.

Each of the eight stack registers of the Pure Stack Machine has a bit indicating whether the register is full or empty. The number of data items that are actually moved during execution of the duplicate, exchange, and rotate instructions depends upon the quantity and locations of the registers that are marked full. Figure 7 demonstrates the necessary data movements caused by execution of a ROTD instruction. The stack registers are designated by SR_n where $n = 0, \dots, 7$. SR_0 is the top stack register. In Example 1, data item A is moved, SR_0 is marked empty, and SR_6 is marked full. In Example 2 data items A, D, and E are moved; SR_0 is marked empty; and SR_5 is marked full.

The relative times required to execute the duplicate, exchange, and rotate instructions depend on how the registers are used to configure the stack. The smallest numbered register that is marked full is the top element of the stack. Rotate operations minimally require the movement of one data item and the resetting of two "full-empty" bits. A duplication operation requires the shortest amount of time when there is an empty register that is numbered smaller than the current top-of-stack register. The number of data movements required to duplicate the top stack element is minimized if data elements are loaded into the largest numbered register such that the registers above it are marked empty. For example, assume registers SR_0 through SR_3 are empty and SR_4 through SR_7 are full. A LOAD is followed by a DUP. If the value to be loaded is placed in SR_0 , then the DUP causes the data item to be shifted from SR_0 to SR_1 and then duplicated. If the value is placed in SR_3 , no data movements are necessary prior to duplicating the item in SR_2 .

		BITS	MEMORY ACCESSES	REGISTERS
LOAD B	B	232	5	4
LOAD C	B; C			
DUP	B; C; C			
ROTD	C; B; C			
MULT	C; B*C			
LOAD D	C; B*C; D			
DUP	C; B*C; D; D			
ROTD	D; C; B*C; D			
ADD	D; C; B*C+D			
STND B	D; C; B			
ROTU	C; B; D			
MULT	C; B*D			
ADD	C+B*D			
STD E	EMPTY			

(A) STACK OPTIMIZATION ALGORITHM

LOAD B	B	256	7	2
LOAD C	B; C			
MULT	B*C			
LOAD D	B*C; D			
ADD	B*C+D			
STND B	B			
LOAD D	B; D			
MULT	B*D			
LOAD C	B*D; C			
ADD	B*D+C			
STD E	EMPTY			

(B) REORDERED TRANSLATION FOR THE PURE STACK MACHINE

LOAD C,R0	C IN R0	288(224*)	5	3
LOAD R0,R1	C IN R1			
MULT B,R1	B*C IN R1			
LOAD D,R2	D IN R2			
ADD R2,R1	B*C+D IN R1, B IN R1			
MULT R1,R2	B*D IN R2			
ADD R0,R2	B*D+C IN R2, E IN R2			
STOR B, R1				
STOR E,R2				

*WHEN REGISTER-TO-REGISTER INSTRUCTIONS ARE 16 BITS LONG.

(C) HAND-OPTIMIZED CODE FOR THE ADDRESSABLE-REGISTER MACHINE

Figure 6. Code generated for sample expressions.

EXAMPLE 1		EXAMPLE 2			
	BEFORE	AFTER		BEFORE	AFTER
SR0	A		SR0	A	
SR1	B	B	SR1	B	B
SR2	C	C	SR2	C	C
SR3			SR3		
SR4	D	D	SR4		
SR5	E	E	SR5		D
SR6		A	SR6	D	E
SR7			SR7	E	A

Figure 7. Data movements caused by execution of a ROTD instruction.

The above discussion about where in the set of empty registers values should be loaded is rhetorical since the hardware usually determines the register configuration and the implicit data moves between registers. It would be nice if the software could specify loading the value in SR3, knowing that the next instruction was a DUP. In Example 2 of Figure 7, the data items D, E, and A could have been moved into registers SR3, SR4, and SR5 as easily as they were moved into SR5, SR6, and SR7. If the software were able to control the movement, the number of implicit data moves between registers could be minimized by anticipating what would occur during execution of the instructions which follow ROTD. For example, if several LOAD instructions followed, it would be best to move D, E, and A to the bottom of the register set initially to avoid moving them later on. If another ROTD instruction followed, the initial positioning of the items SR3 through SR5 would eliminate the need to move the items a second time. It should again be noted that these options are not available in any existing stack machine. Adding these capabilities to the existing stack functions combines the advantages of stack and addressable-register machines but imposes the requirement that the software keep track of which registers are empty and which are full.

Another method of implementing the stack is to have an additional register, R_p , pointing to the stack register which is effectively the top of the stack. For example, if R_p contains the value 6, SR6 contains the top element of the stack, SR7 contains the second-to-top element, SR0 contains the third, and so on. Execution of the rotation operation only involves altering the value of R_p . This requires less time than rotating the contents of the registers. Duplication of a value in the top stack register can only be performed if the register above it is empty. In certain cases, the use of R_p minimizes the execution time. For example, assume SR0 is marked full and SR7 is marked empty. If R_p points to SR0, the contents of SR0 is copied into SR7 and R_p is altered to point to SR7. In the previously described stack configuration when R_p is not used, it is necessary to move data items before the contents of SR1 is duplicated in SR0. The number of data elements moved depends on which registers are full and empty. If SR3 is the first empty register closest to the top of the stack, then three data shifts are necessary before duplication can occur. We can conclude that the use of a top-of-stack pointer would often improve the execution speed in stack machines.

VII. CONCLUSION

"A slow sort of country!" said the Queen.
"Now, here, you see, it takes all the
running you can do, to keep in the same place.
If you want to get somewhere else, you must run
at least twice as fast as that."

Lewis Carroll
Through the Looking Glass

It has been demonstrated that optimization techniques on various stack architectures are as important as optimization on general register machines. On two of the three machines defined, optimization has a critical effect on the efficiency of execution of sample programs.

Stack machines have the advantage of implicitly storing temporary results. Since addressing is implicit, desired values within the stack are not immediately accessible. To compensate for this disadvantage, stack architectures should be designed with facilities to permit optimizing software.

Two methods of making elements on the stack easier to access have been discussed. The first approach is compile-time reordering of expressions. Values are placed on the stack in such an order that the maximum number of desired data elements are on the top of the stack when they are needed. The second approach involves storing values on the stack in any order. When elements within the stack are needed, they are positioned at the stack's top by the minimal number of operations.

In addition to software optimization, efficient execution of programs on stack machines is dependent upon how the registers are used to configure the top portion of the stack. Thus, both hardware and software designs for stack architectures should be improved to compete with the latest designs for conventional register machines.

ACKNOWLEDGMENT

I wish to express my thanks to my thesis adviser, Richard Hamlet, for his patient understanding and his counsel throughout my graduate studies at the University of Maryland.

I also wish to thank John Shore for suggesting I write this thesis and my other colleagues at the Naval Research Laboratory for their assistance.

BIBLIOGRAPHY

1. Allmark, R. H., and Lucking, J. R., "Design of an Arithmetic Unit Incorporating a Nesting Store." *Proc. IFIP Congress 1962*, p. 694-698 (reprinted in 6).
2. Anderson, James P. "A Computer for Direct Execution of Algorithmic Languages." *Proc. AFIPS 1961 FJCC*, p. 184-193.

3. Bartlett, Joel F., "The HP3000 Computer Systems, Proc. Symposium on High-Level-Language Computer Architecture," University of Maryland, College Park, Md., Nov. 1973.
4. Barton, R. S., "A New Approach to the Functional Design of a Digital Computer." *Proc. 1961 WJCC*, p. 393-396.
5. Basili, V., and Turner, A. J., "SIMPL-T. A Structured Programming Language," Computer Science Center, University of Maryland, College Park, Md. Jan. 1974.
6. Bell, C. Gordon, and Newell, Allen (ed.), *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971, p. 62-63, 257-273.
7. Bingham, Harvey W., and Kauffman, S. Bruce, Analysis of Static Object Code Produced by Algol and Cobol Compilers for the Burroughs B5500, Burroughs Corp. (TR-69-1), Paoli, Pa., Feb. 1969.
8. Burroughs B6500 Information Processing Systems Reference Manual, Burroughs Corps., Detroit, Mich., 1969.
9. A Narrative Description of the Burroughs B5500 Disk File Master Control Program, Business Machine Group, Burroughs Corp., Detroit, Mich., 1966.
10. Carlson, C. B., "The Mechanization of a Pushdown Stack." *Proc. AFIPS 1963 FJCC*, p. 243-250.
11. Church, Charles C., "Computer Instruction Repertoire—Time for a Change." *Proc. AFIPS 1970 SJCC*, p. 343-349.
12. Cocke, John, and Schwartz, J. T., "Programming Languages and Their Compilers, Preliminary Notes," Courant Institute of Mathematical Sciences, New York University, New York, April 1970.
13. Deerfield, A., et al., "Final Report for AADC Arithmetic and Control Logic Design Study," Part I-III, Raytheon Company (BR-7162), Bedford, Mass., 1972.
14. Deerfield, A., "Instruction Deferral Sequencing Mechanism," Proc. Symposium on Programming and machine Organization, IEEE Computer Society, Mid-Eastern and N. J. Coast Chapters, April 1971.
15. Gries, David., *Compiler Construction for Digital Computers*, Wiley, New York, 1971.
16. Haley, A. C. D., "The KDF9 Computer System." *Proc. AFIPS 1962 FJCC*, p. 108-120.
17. Hamblin, C. L., "Translation to and from Polish Notation." *Computer Journal* 5, Oct. 1962, p. 210-213.
18. Hauck, E. A., and Dent, B. A., "Burroughs B6500/B7500 Stack Mechanism. *Proc. AFIPS 1968 SJCC*, p. 245-251.
19. Keeler, F. S., et al., "Computer Architecture Study, Information and Communication Applications (SAMSO-TR-420)." Silver Spring, Md. Oct. 1970.
20. Kildall, Gary Arlen, "Global Expression Optimization During Compilation," Ph.D. thesis, Computer Science Group (TR72-06-02), University of Washington, Seattle, Wash., June 1972.

21. Knuth, D. E., "An Empirical Study of FORTRAN Programs." *Software Practice and Experience*, 1971.
22. Lawson, H. W., Jr., "Programming Language-Oriented Instruction Streams." *IEEE Transactions on Computers* C-17, May 1968, p. 476-485.
23. Lonergan, William, and King, Paul, "Design of the B5000 System." *Datamation*, May 1961, p. 28-32, (reprinted in 6).
24. Miller, James, et al., "Engineering Study From the Functional Design of a Multi-processor System, Intermetrics (TR14-72)," Cambridge, Mass., Sept. 1972, p. 82-87.
25. Organick, Elliott I., *Computer System Organization. The B5700/B6700 Series*, Academic Press, New York, 1973.
26. Parnas, David L., "Changing the AADC ISP Level Design to a Postfix Machine," Naval Research Laboratory, Washington, D.C., 1972.
27. Randell, B., and Russell, L. J., *Algol 60 Implementation*, Academic Press, New York, 1964.
28. Sneck, Paul B., "A Survey of Compiler Optimization Techniques," Goddard Institute for Space Studies (NTIS N73-30148), 1973.
29. Wersan, S. J., et al., "Architectural Study for Advanced Guidance Computers, Part 2," Cirad (SAMSO-TR-71-6), Claremont, Calif., Feb. 1971.

APPENDIX A

SAMPLE CODE GENERATED BY THE ALGOL AND FORTRAN
COMPILERS OF THE BURROUGHS B6700

BURROUGHS B6700 ALGOL COMPILER, VERSION 2.5.000,

TUESDAY, 12/11/73

A L C O L / C O D E

BEGIN
B.0000
SEGMENT DESCRIPTOR

ARRAY H[1:20];
H
INTEGER A,B,C,D,E;
A
B
C
D
E

A:=B+A;

003:0000:0	VALC	(02,00004)	1004
003:0000:2	VALC	(02,00003)	1003
003:0000:4	ADD		80
003:0000:5	NTGR		87
003:0001:0	NAMC	(02,00003)	5003
003:0001:2	STOD		B8

C:=C+1;

003:0001:3	VALC	(02,00005)	1005
003:0001:5	ONE		B1
003:0002:0	ADD		80
003:0002:1	NTGR		87
003:0002:2	NAMC	(02,00005)	5005
003:0002:4	STOD		B8

C:=C+B;

003:0002:4	STON		B9
003:0002:5	VALC	(02,00004)	1004
003:0003:1	ADD		80
003:0003:2	NTGR		87
003:0003:3	NAMC	(02,00005)	5005
003:0003:5	STOD		B8

HELEN B. CARTER

D := (D+1)*(C+B)*(C+B+D);

003:0004:0	VALC	(02,00006)	1006
003:0004:2	ONE		B1
003:0004:3	ADD		80
003:0004:4	VALC	(02,00005)	1005
003:0005:0	VALC	(02,00004)	1004
003:0005:2	ADD		80
003:0005:3	MULT		82
003:0005:4	VALC	(02,00005)	1005
003:0006:0	VALC	(02,00004)	1004
003:0006:0	ADD		80
003:0006:3	VALC	(02,00006)	1006
003:0006:5	ADD		60
003:0007:0	MULT		82
003:0007:1	NTGR		87
003:0007:2	NAMC	(02,00006)	5006
003:0007:4	STOD		B8

E := D;

003:0007:4	STON		B9
003:0007:5	NTGR		87
003:0008:0	NAMC	(02,00007)	5007
003:0008:2	STOD		B8

A := E;

003:0008:2	STON		B9
003:0008:3	NTGR		87
003:0008:4	NAMC	(02,00003)	5003
003:0009:0	STOD		B8

D := 2+2;

003:0009:1	LTS		B204
003:0009:3	NAMC	(02,00006)	5006
003:0009:5	STOD		B8

D := 5+3;

003:000A:0	LTS		B20F
003:000A:2	NAMC	(02,00006)	5006
003:000A:4	STOD		B8

H[D] := H[D]+1;

003:000A:4	STON		B9
003:000A:5	ONE		B1
003:000B:0	SUBT		81

B6700/57700

FORTRAN COMPILATION MARK

2.5.000

SET OPT = -1

DIMENSION H(20)

A=B+A

0000:0	VALC	(3,003) = B	3003
0000:2	VALC	(3,002) = A	3002
0000:4	ADD		80

NRL REPORT 7787

		0000:5	NAMC	(3,002) = A	7002
		0001:1	STOD	(3,002) = A	B8
C=C+1					
C=C+1		0001:2	VALC	(3,004) = C	3004
		0001:4	ONE		B1
		0001:5	ADD		80
		0002:0	NAMC	(3,004) = C	7004
		0002:2	STOD		B8
C=C+B					
	*	0002:2	STON		B9
		0002:3	VALC	(3,003) = B	3003
		0002:5	ADD		80
		0003:0	NAMC	(3,004) = C	7004
		0003:2	STOD		B8
D=(D+1)*(C+B)*(C+B+D)					
		0003:3	VALC	(3,005) = D	3005
		0003:5	ONE		B1
		0004:0	ADD		80
		0004:1	VALC	(3,004) = C	3004
		0004:3	VALC	(3,003) = B	3003
		0004:5	ADD		80
		0005:0	MULT		82
		0005:1	VALC	(3,004) = C	3004
		0005:3	VALC	(3,003) = B	3003
		0005:5	ADD		80
		0006:0	VALC	(3,005) = D	3005
		0006:2	ADD		80
		0006:3	MULT		82
		0006:4	NAMC	(3,005) = D	7005
		0007:0	STOD		B8
E=D					
	*	0007:0	STON		B9
		0007:1	NAMC	(3,006) = E	7006
		0007:3	STOD		B8
A=E					
	*	0007:3	STON		B9
		0007:4	NAMC	(3,002) = A	7002
		0008:0	STOD		B8
D=2+2					
		0008:1	LT8	2	B202
		0008:3	LT8	2	B202
		0008:5	ADD		80
		0009:0	NAMC	(3,005) = D	7005
		0009:2	STOD		B8
D=5*3					
		0009:3	LT8	5	B205
		0009:5	LT8	3	B203
		000A:1	MULT		82
		000A:2	NAMC	(3,005) = D	7005
		000A:4	STOD		B8

HELEN B. CARTER

H(D)=H(D)+1

*	000A:4	STON		B9
	000A:5	ONE		B1
	000B:0	SUBT		81
	000B:1	NAMC	(3,007) = H	7007
	000B:3	INDX		A6
	000B:4	VALC	(3,005) = D	3005
	000C:0	ONE		B1
	000C:1	SUBT		81
	000C:2	NAMC	(3,007) = H	7007
	000C:4	NXLV		AD
	000C:5	ONE		B1
	000D:0	ADD		80
	000D:1	STOD		B8

H(1)=H(1)+1

	000D:2	ONE		B1
	000D:3	ONE		B1
	000D:4	SUBT		81
	000D:5	NAMC	(3,007) = H	7007
	000E:1	INDX		A6
	000E:2	ONE		B1

APPENDIX B

STACK OPTIMIZATION ALGORITHM

```

/*****STACK OPTIMIZATION ALGORITHM*****/
/*
/*GENERATES OPTIMAL CODE TO EVALUATE ALGEBRAIC EXPRES-
SIONS ON A STACK COMPRISED OF EIGHT HARDWARE REGIS-
TERS. THE NUMBER OF MEMORY ACCESSES ARE MINIMIZED
BY DUPLICATING, EXCHANGING AND ROTATING ELEMENTS ON
THE STACK.
/*
/*
/*****/
CHAR ARRAY POSTFIX(100)
/*USER INPUT POSTFIX NOTATION OF ALGEBRAIC EXPRESSIONS CONSIST-
ING OF VARIABLES, CONSTANTS AND OPERATORS. FOR SIMPLICITY,
ALL VARIABLE NAMES ARE SINGLE LETTERS AND ALL CONSTANTS ARE
DIGITS 0-9. */
STRING PTFX[100]
STRING A[50]
ARRAY NUMPOST(100)
/*POSTFIX NOTATION OF ALGEBRAIC EXPRESSIONS CONSISTING OF NU-
MERICAL VALUES WHICH CORRESPOND TO VARIABLES, CONSTANTS,
OPERATORS AND SUBEXPRESSIONS. COMMON VARIABLES WHOSE VALUES
ARE NOT ALTERED HAVE THE SAME NUMERICAL VALUE. NUMERICAL
VALUES 1-6 DENOTE OPERATORS(# IS NEGATION,= IS REPLACEMENT),
7-39 DENOTE VARIABLES, CONSTANTS AND INTERMEDIATE CALCULA-
TIONS, AND 40-50 DENOTE SUBEXPRESSIONS OCCURRING MORE THAN
ONCE. THE FIRST OCCURRENCE OF A COMMON SUBEXPRESSION IN
NUMPOST IS FOLLOWED BY THE NEGATION OF THE CORRESPONDING NU-
MERICAL VALUE. ALL SUBSEQUENT OCCURRENCES OF THAT SUBEXPRES-
SION ARE REPLACED BY THE VALUE. */
CHAR ARRAY ALPH(50)=(' ','*', '/', '+', '-', '#', '=', ' ' (33), '$' (10))
/*ALPH(I) CONTAINS THE OPERATOR OR OPERAND CORRESPONDING TO
THE I-TH NUMERICAL VALUE. FOR SUBEXPRESSIONS, ALPH(I) CON-
TAINS '$'. */
ARRAY COUNT(50)=(0(50))
/*COUNT(I) CONTAINS THE NUMBER OF TIMES THE OPERAND CORRESPOND-
ING TO THE I-TH NUMERICAL VALUE OCCURS IN THE SET OF ALGE-
BRAIC EXPRESSIONS. */
ARRAY ONSTK(50)=(0(50))
/*ONSTK(I) HAS THE VALUE 1 IF THE OPERAND IS ON THE STACK. */
ARRAY STACK(8)=(0(8))
/*STACK STRUCTURE CONTAINING THE CONTENTS OF THE 8 STACK REG-
ISTERS. IS POINTS TO THE TOP OF THE STACK. IF IS=2,

```

HELEN B. CARTER

```

STACK(2) CONTAINS THE TOP STACK ELEMENT AND STACK(1), THE
BOTTOM STACK ELEMENT. */
ARRAY SAVE(8)=(0(8))
/*STACK STRUCTURE CONTAINING NUMERICAL VALUES OF OPERANDS TO
BE USED IN SUBSEQUENT CALCULATIONS */
ARRAY KEEP(8)=(0(8))
/*STACK STRUCTURE CONTAINING STARTING POSITIONS IN ARRAY NUM-
POST OF OPERANDS WHOSE OPERATOR HAS NOT AS YET BEEN
ENCOUNTERED */
ARRAY FST(50)=(0(50))
/*FST(I) CONTAINS THE POSITION IN ARRAY NUMPOST OF THE FIRST
SYMBOL OF THE I-TH SUBEXPRESSION. */
ARRAY LEN(50)=(0(50))
/*LEN(I) CONTAINS THE LENGTH OF THE I-TH SUBEXPRESSION. */
ARRAY REP(50)= (0(50))
/*REP(I) GETS THE NUMERICAL VALUE ASSOCIATED WITH THE I-TH
SUBEXPRESSION IF IT OCCURS MORE THAN ONCE IN THE SET OF
ALGEBRAIC EXPRESSIONS; OTHERWISE, IT GETS THE VALUE OF -1. */
ARRAY COM(10)=(0(10))
/*CONTAINS POSITIONS IN ARRAYS FST, LEN AND REP OF SUBEXPRES-
SIONS OF THE SAME LENGTH */
INT IMAX=6,
/*CONTAINS THE LAST NUMERICAL VALUE ASSIGNED TO VARIABLES,
CONSTANTS AND INTERMEDIATE RESULTS */
SEMAX=39,
/*CONTAINS THE LAST NUMERICAL VALUE ASSIGNED TO COMMON
SUBEXPRESSION */
IP=0, /*POINTER INTO ARRAYS POSTFIX AND NUMPOST; LENGTH OF POSTFIX
INSTRUCTION STREAM */
FLG=0, /*SET TO 1 WHEN ALL OF ARRAY NUMPOST HAS BEEN SCANNED */
ISCAN=0, /*POINTER INTO ARRAY NUMPOST */
IS=0, /*STACK POINTER INTO ARRAY STACK */
IK=0, /*STACK POINTER INTO ARRAY KEEP */
IV=0, /*STACK POINTER INTO ARRAY SAVE */
X=0, /*POINTER INTO ARRAYS FST, LEN AND REP; NUMBER OF SUBEXPRES-
SIONS */
CURR, /*CURRENT SYMBOL OF ARRAY NUMPOST */
NXT1, /*NEXT SYMBOL OF ARRAY NUMPOST */
TEMP,
PT, /*OPERATOR I.D. */
FND,
M, /*NUMBER OF SUBEXPRESSIONS OF THE SAME LENGTH BEING COMPARED */
LONG, /*LENGTH OF CURRENT SET OF SUBEXPRESSIONS BEING COMPARED */
ISE, /*POINTER INTO ARRAYS FST, LEN AND REP OF FIRST SUBEXPRESSION
BEING COMPARED */
JSE, /*POINTER INTO ARRAYS FST, LEN AND REP OF SECOND SUBEXPRESSION
BEING COMPARED */
Y, /*POSITION OF FIRST SUBEXPRESSION BEING COMPARED IN ARRAY
NUMPOST */

```

```

Z,      /*POSITION OF SECOND SUBEXPRESSION BEING COMPARED IN ARRAY
        NUMPOST */
ORD,    /*0 IF OPERATOR IS COMMUTATIVE; OTHERWISE,1 */
POS1,POS2 /*POSITIONS IN STACK OF OPERANDS TO BE SHIFTED TO THE TOP
          OF THE STACK */
/* */
PROC MAIN
  CALL ASSIGNVAL      /*ASSIGNS NUMERICAL VALUES TO OPERATORS AND
                      OPERANDS*/
  CALL SUBEXP        /*IDENTIFIES COMMON SUBEXPRESSIONS*/
  CALL CODEGEN       /*DRIVER PROCEDURE FOR CODE GENERATION*/
  RETURN
/* */
PROC ASSIGNVAL
  /*SCANS INPUT POSTFIX INSTRUCTION STEAM: ASSIGNS NUMERICAL
  VALUES TO OPERATORS AND OPERANDS*/
  READC(POSTFIX)
  CALL PACK(POSTFIX,PTFX)
  WHILE 1 DO
    IF POSTFIX(IP)=# THEN /*DONE READING INPUT*/
      IP:=IP-1
      EXIT
    END
    PT:=MATCH('* /+-#=',PTFX[IP+1,1])
    /*IF SYMBOL IS AN OPERATOR, PT GETS CORRESPONDING NUMERICAL
    VALUE; OTHERWISE, PT GETS 0 */
    IF PT THEN
      CALL GETOPTR
    ELSE
      CALL GETOPRND
    END
    IP:=IP+1
  END
  RETURN
/* */
PROC SUBEXP
  /*SEARCHES LIST OF SUBEXPRESSIONS FOR COMMON TERMS, ASSIGNS
  NUMERICAL VALUES TO COMMON SUBEXPRESSIONS AND ADJUSTS ARRAY
  NUMPOST*/
  WHILE 1 DO
    CALL COMLEN
    IF LONG=0 THEN EXIT END /*ALL SUBEXP. HAVE BEEN EXAMINED*/
    IF M=1 THEN /*ONLY ONE SUBEXP. HAS LENGTH LONG*/
      REP(COM(1)):= -1
    ELSE
      CALL COMPARE
    END
  END
  END
  RETURN

```

```

/* */
PROC CODEGEN
    /*SCANS ADJUSTED POSTFIX NOTATION OF NUMERICAL VALUES AND
    GENERATES OPTIMIZING CODE*/
    INT SCURR
    CALL PACK(ALPH,A)
    WHILE 1 DO
        CALL SCAN
        IF FLG THEN EXIT END
        SCURR:=CURR
        IF SCURR>0.AND.SCURR<7 THEN CALL OPTR END
        IF SCURR>6 THEN CALL OPRAND END
    END
    RETURN
/* */
PROC GETOPTR
    /*BUILDS LIST OF SUBEXPRESSIONS, AND REORDERS OPERANDS OF
    COMMUTATIVE OPERATIONS INTO LEXICAL ORDER*/
    NUMPOST(IP):=PT
    CASE PT OF
    \1\ \2\ \3\ \4\      /*BINARY OPERATOR*/
        X:=X+1
        FST(X):=KEEP(IK-1)
        LEN(X):=IP-KEEP(IK-1)+1
        IF (PT=1.OR,PT=3).AND.      /*COMMUTATIVE OPERATOR*/
            (NUMPOST(KEEP(IK))<NUMPOST(KEEP(IK-1))) THEN CALL REORDER
        END
        IK:=IK-1
    \5\      /*UNARY OPERATOR*/
        X:=X+1
        FST(X):=KEEP(IK)
        LEN(X):=IP-KEEP(IK)+1
    END
    RETURN
/* */
PROC REORDER
    /*REORDERS TERMS OF COMMUTATIVE OPERATIONS SUCH THAT THE NU-
    MERICAL VALUES CORRESPONDING TO THE VARIABLES AND CONSTANTS
    ARE IN ASCENDING ORDER*/
    INK K,L
    L:=KEEP(IK)-KEEP(IK-1)
    WHILE L>0 DO
        L:=L-1
        TEMP:=NUMPOST(KEEP(IK-1))
        K:=KEEP(IK-1)
        WHILE K<IP-1 DO
            K:=K+1
            NUMPOST(K-1):=NUMPOST(K)
        END
        NUMPOST(IP-1):=TEMP
    END

```

```

END
RETURN
/* */
PROC GETOPRND
    /*ASSIGNS OR DETERMINES PREVIOUSLY ASSIGNED NUMERICAL VALUES
    FOR VARIABLES AND CONSTANTS*/
    IF POSTFIX(IP+1)="=" Then    /*ALTERED VARIABLE GETS NEWLY ASSIGNEE
                                NUMERICAL VALUE*/
        IK:=IK-1    /*DELETE LAST TERM OF ARRAY SAVE*/
        CALL NEWOPRND
    ELSE
        IK:=IK+1    /*SAVE POSITION OF OPERAND*/
        KEEP(IK):=IP
        CALL SEARCH
        IF .NOT.FND THEN
            CALL NEWOPRND
        END
    END
END
RETURN
/* */
PROC NEWOPRND
    /*ASSIGNS NEW NUMERICAL VALUE TO OPERAND*/
    IMAX:=IMAX+1
    ALPH(IMAX):=POSTFIX(IP)
    COUNT(IMAX):=1
    NUMPOST(IP):-IMAX
    RETURN
/* */
PROC SEARCH
    /*DETERMINES IF CURRENT OPERAND HAS PREVIOUSLY BEEN ENCOUNTERED
    IN POSTFIX STREAM*/
    INT K
    FND:=0
    K:=IMAX+1
    WHILE K>7 DO
        K:=K-1
        IF POSTFIX(IP)=ALPH(K) THEN    /*MATCH IS FOUND*/
            NUMPOST(IP):=K
            COUNT(K):=COUNT(K)+1
            FND:=1
            EXIT
        END
    END
END
RETURN
/* */
PROC COMLEN
    /*BUILDS LIST OF SUBEXPRESSIONS OF THE SAME LENGTH*/
    INT K
    M:=0
    K:=0

```

```

LONG:=0
WHILE K<X DO
  K:=K+1
  IF REP(K)=0 THEN
    IF LEN(K)=LONG THEN /*ADD ANOTHER SUBEXP. OF SAME LENGTH
                        TO ARRAY COM*/
      M:=M+1
      COM(M):=K
    END
    IF LEN(K)>LONG THEN /*FIND SUBEXP. OF A LONGER LENGTH
                       AND REINITIALIZE ARRAY COM*/
      LONG:=LEN(K)
      M:=1
      COM(1):=K
    END
  END
END
END
RETURN
/* */
PROC COMPARE
  /*STEP THROUGH LIST OF SUBEXPRESSIONS OF THE SAME LENGTH FOR
  COMPARISON*/
  INT I,J
  I:=0
  WHILE I<M-1 DO
    I:=I+1
    ISE:=COM(I)
    IF REP(ISE)=0 THEN
      J:=I
      WHILE J<M DO
        J:=J+1
        JSE:=COM(J)
        IF REP(JSE)=0 THEN
          Z:=FST(JSE)
          Y:=FST(ISE)
          CALL MATCHUP
        END
      END
    END
    IF REP(ISE)=0 THEN REP(ISE):=-1 END
  END
  IF REP(COM(M))=0 THEN REP(COM(M)):-1 END
  RETURN
/* */
PROC MATCHUP
  /*DETERMINES IF TWO SUBEXP. OF THE SAME LENGTH ARE IDENTICAL*/
  INT ICNT,NOMATCH
  ICNT:=0
  NOMATCH:=0
  WHILE ICNT<LONG DO /*COMPARE SUBEXP. SYMBOL-BY-SYMBOL*/

```

```

    ICNT:=ICNT+1
    IF NUMPOST(Y)<>NUMPOST(Z) THEN /*NOT IDENTICAL*/
        NOMATCH:=1
        EXIT
    END
    Y:=Y+1
    Z:=Z+1
END
IF .NOT.NOMATCH THEN /*COMMON SUBEXP. HAVE BEEN IDENTIFIED*/
    IF REP(ISE)=0 THEN
        CALL COMMON1
    END
    CALL COMMON2
END
RETURN
/* */
PROC COMMON1
    /*PERFORMS NECESSARY PROCESSING FOR FIRST OCCURRENCE OF COM-
    MON SUBEXPRESSION*/
    INT K,L
    SEMAX:=SEMAX+1 /*ASSIGN NEW NUMERICAL VALUE*/
    REP(ISE):=SEMAX
    CALL ELIM(ISE)
    IP:=IP+1
    K:=IP
    L:=FST(ISE)+LEN(ISE)
    WHILE K>L DO /*INSERT NEGATION OF NUMERICAL VALUE INTO POSTFIX
        STREAM AFTER THE FIRST OCCURRENCE OF THE COMMON
        SUBEXPRESSION*/
        NUMPOST(K):=NUMPOST(K-1)
        K:=K-1
    END
    NUMPOST(L):=-SEMAX
    K:=0
    WHILE K<X DO /*ADJUST VALUES IN ARRAY FST OF SUBEXP. WHOSE
        STARTING POSITIONS WERE ALTERED*/
        K:=K+1
        IF FST(K)>=L THEN
            FST(K):=FST(K)+1
        END
    END
    RETURN
/* */
PROC COMMON2
    /*PERFORMS NECESSARY PROCESSING FOR SUBSEQUENT OCCURRENCES OF
    THE COMMON SUBEXPRESSION*/
    INT K,L
    K:=0
    WHILE K<LEN(JSE)-1 DO /*REPLACE SUBEXP. BY NUMERICAL VALUE IN
        ARRAY NUMPOST*/

```

HELEN B. CARTER

```

      K:=K+1
      L:=NUMPOST(FST(JSE)+K-1)
      IF L>6 THEN
        COUNT(L):=COUNT(L)-1      /*REDUCE COUNT ENTRY OF OPERANDS
                                   COMPRISING SUBEXPRESSION*/
      END
    END
    REP(JSE):=SEMAX
    CALL ELIM(JSE)
    NUMPOST(FST(JSE)):=SEMAX
    COUNT(SEMAX):=COUNT(SEMAX)+1
    K:=FST(JSE)+1
    L:=K+LEN(JSE)-1
    WHILE L<=IP DO
      NUMPOST(K):=NUMPOST(L)
      K:=K+1
      L:=L+1
    END
    IP:=IP-LEN(JSE)+1
    K:=0
    WHILE K<X DO      /*ADJUST STARTING POSITIONS OF SUBEXP. IN ARRAY FST*/
      K:=K+1
      IF FST(K)>FST(JSE) THEN
        FST(K):=FST(K)-LEN(JSE)+1
      END
    END
  END
  RETURN
/* */
PROC ELIM(INT K)
  /*ELIMINATES THOSE SUBEXPRESSIONS CONTAINED WITHIN THE K-TH
  SUBEXPRESSION FROM THE LIST OF THE SUBEXP. BEING COMPARED*/
  INT L
  L:=0
  WHILE L<K-1 DO
    L:=L+1
    IF REP(L)=0 THEN
      IF FST(L)>=FST(K).AND.FST(L)<FST(K)+LEN(K) THEN
        REP(L):=-1
      END
    END
  END
  RETURN
/* */
PROC SCAN
  /*POSITIONS POINTER ISCAN TO NEXT SYMBOL OF ARRAY NUMPOST*/
  IF ISCAN>IP THEN
    FLG:=1
    RETURN
  END
  CURR:=NUMPOST(ISCAN)

```

```

NXT1:=NUMPOST(ISCAN+1)
ISCAN:=ISCAN+1
RETURN
/* */
PROC OPTR
    /*PROCESSES ALL OPERATORS EXCEPT THE REPLACEMENT OPERATORS*/
    IF CURR=5 THEN /*UNARY OPERATOR*/
        CALL GET1(SAVE(IV))
    ELSE
        IF CURR=1.OR.CURR=3 THEN ORD:=0 ELSE ORD:=1 END
        /*DETERMINES IF OPERATOR IS COMMUTATIVE*/
        CALL GET2(SAVE(IV),SAVE(IV-1))
        IV:=IV-1 /*COMBINE 2 OPERANDS INTO A SINGLE INTERMEDIATE RESULT*/
        IS:=IS-1
    END
    CASE CURR OF /*GENERATE CODE TO PERFORM OPERATIONS*/
        \1\ WRITEL ('MULT')
        \2\ WRITEL ('DIV')
        \3\ WRITEL ('ADD')
        \4\ WRITEL ('SUB')
        \5\ WRITEL ('NEG')
    END
    IF NXT1<0 THEN /*FIRST OCCURRENCE OF COMMON SUBEXP.*/
        SAVE(IV):=NXT1
        STACK(IS):=-NXT1
        ONSTK(-NXT1):=1
        CALL SCAN
        CALL DUP
    ELSE /*ASSIGN NEW NUMERICAL VALUE TO INTERMEDIATE RESULT*/
        IMAX:=IMAX+1
        SAVE(IV):=IMAX
        STACK(IS):=IMAX
    END
    RETURN
/* */
PROC OPRAND
    /*PROCESS VARIABLES, CONSTANTS AND COMMON SUBEXPRESSIONS*/
    IF NXT1=6 THEN /*VARIABLE CURR IS TO BE ALTERED*/
        CALL STOREVAL
    ELSE
        IF .NOT.ONSTK(CURR) THEN /*OPERAND NOT ON STACK*/
            CALL LOADVAL
        ELSE /*ON STACK*/
            CALL SHIFTVAL
        END
        IV:=IV+1
        SAVE(IV):=CURR
    END
    RETURN

```

```

/* */
PROC STOREVAL
    /*DETERMINES IF STORED VALUE SHOULD REMAIN ON STACK, AND GEN-
    ERATES APPROPRIATE STORE INSTRUCTION*/
    COUNT(CURR):=COUNT(CURR)-1
    CALL GET1(SAVE(IV))
    IF COUNT(CURR)>0 THEN /*STORE NON-DESTRUCTIVE*/
        WRITEL('STND '.CON.A[CURR+1,1])
        IV:=IV-1 /*DELETE STORED VALUE FROM ARRAY SAVE*/
        STACK(IS):=CURR
        ONSTK(CURR):=1
        COUNT(CURR):=COUNT(CURR)-1
    ELSE /*STORE DESTRUCTIVE*/
        WRITEL('STD '.CON.A[CURR+1,1])
        IV:=IV-1 /*DELETE STORED VALUES FROM ARRAYS STACK AND SAVE*/
        IS:=IS-1
    END
    CALL SCAN /*BYPASS REPLACEMENT OPERATOR*/
    RETURN
/* */
PROC LOADVAL
    /*LOAD VALUE ONTO STACK AND DUPLICATE IF VALUE IS TO BE REUSED*/
    WRITEL('LOAN '.CON.A[CURR+1,1])
    IS:=IS+1
    STACK(IS):=CURR
    COUNT(CURR):=COUNT(CURR)-1
    IF .NOT.DIGIT(ALPH(CURR)) THEN /*NOT A CONSTANT*/
        ONSTK(CURR):=1
        IF COUNT(CURR)>0 THEN
            CALL DUP
        END
    END
    RETURN
/* */
PROC SHIFTVAL
    /*SHIFTS 1 OR 2 OPERANDS TO TOP OF STACK IF CURRENT VALUE IS TO
    BE DUPLICATED*/
    IF .NOT.DIGIT(ALPH(CURR)) THEN /*NOT A CONSTANT*/
        IF COUNT(CURR)>0 THEN
            IF NXT1>0.AND.NXT1<5 THEN /*BINARY OPERATION*/
                ORD:=1
                CALL GET2(CURR,SAVE(IV))
            ELSE /*UNARY OPERATION*/
                CALL GET1(CURR)
            END
            CALL DUP
        END
    END
    RETURN

```

/* */

PROC DUP

```

    /*GENERATES CODE TO DUPLICATE VALUE ON TOP OF STACK*/
    INT PTR
    IS:=IS+1
    IF CURR<0 THEN      /*FIRST OCCURRENCE OF SUBEXPRESSION*/
        PTR:=-CURR
    ELSE                /*SUBSEQUENT OCCURRENCE OF SUBEXP. OR OTHER OPERAND*/
        PTR:=CURR
    END
    STACK(IS):=PTR
    COUNT(PTR):=COUNT(PTR)-1
    WRITEL('DUP')
    RETURN

```

/* */

PROC ROTU

```

    /*GENERATES CODE TO PERFORM ROTATE UP, AND ADJUSTS CONTENTS OF
    STACK*/
    INT J
    TEMP:=STACK(1)
    J:=1
    WHILE J<IS DO
        STACK(J):=STACK(J+1)
        J:=J+1
    END
    STACK(IS):=TEMP
    WRITEL('ROTU')
    RETURN

```

/* */

PROC ROTD

```

    /*GENERATES CODE TO PERFORM ROTATE DOWN, AND ADJUSTS CONTENTS
    OF STACK*/
    INT J
    TEMP:=STACK(IS)
    J:=IS
    WHILE J>1 DO
        STACK(J):=STACK(J-1)
        J:=J-1
    END
    STACK(1):=TEMP
    WRITEL('ROTD')
    RETURN

```

/* */

PROC XCH

```

    /*GENERATES CODE TO EXCHANGE 2 TOP STACK ELEMENTS AND ADJUSTS
    CONTENTS OF STACK*/
    IF STACK(IS)<>STACK(IS-1) THEN
        TEMP:=STACK(IS)
        STACK(IS):=STACK(IS-1)
        STACK(IS-1):=TEMP
    WRITEL('XCH')

```

```

END
RETURN
/* */
PROC GET2(INT TP,INT STP)
  /*TP AND STP ARE THE NUMERICAL VALUES OF THE 2 TOP STACK
  ELEMENTS WHICH ARE TO BE POSITIONED INTO THE TOP 2 REGISTERS
  OF THE 8 REGISTER SET*/
  CALL LOCATE(TP,STP)
  WHILE 1 DO
    IF POS2-POS1=1 THEN      /*THE DESIRED DATA ITEMS ARE CONTIGUOUS
                              ON THE STACK*/
      CALL ROTATE
      EXIT
    END
    IF POS1=1 THEN
      IF POS2=IS THEN
        CALL ADJUST(1,'3')
        EXIT
      END
      IF POS2=IS-1 THEN
        IF IS=3 THEN
          CALL ADJUST(1,'2')
        ELSE
          CALL ADJUST(2,'13')
        END
        EXIT
      END
      IF POS2=3 THEN
        CALL ADJUST(4,'3313')
        EXIT
      END
      IF POS2=IS-2 THEN
        CALL ADJUST(5,'31212')
        EXIT
      END
      IF POS2=4 THEN
        CALL ADJUST(6,'331213')
        EXIT
      END
      IF POS2=5 THEN
        CALL ADJUST(7,'3131313')
        EXIT
      END
    END
    IF POS1=IS-2 THEN
      IF POS2=IS-1 THEN
        CALL ADJUST(1,'2')
      END
      IF POS2=IS THEN
        CALL ADJUST(2,'12')
      END
    END
  END

```

```
END
EXIT
END
IF POS1=IS-3.AND.POS2=IS-1 THEN
  CALL ADJUST(3,'212')
  EXIT
END
IF POS1=2 THEN
  IF POS2=IS THEN
    CALL ADJUST(3,'313')
  END
  IF POS2=IS-1 THEN
    CALL ADJUST(4,'1313')
  END
  EXIT
END
IF POS1=IS-4.AND.POS2=IS-2 THEN
  CALL ADJUST(4,'2212')
  EXIT
END
IF POS1=IS-3.AND.POS2=IS THEN
  CALL ADJUST(4,'1212')
  EXIT
END
IF POS1=IS-4.AND.POS2=IS THEN
  CALL ADJUST(5,'31313')
  EXIT
END
IF POS1=2.AND.POS2=5 THEN
  CALL ADJUST(7,'3331313')
  EXIT
END
IF POS1=IS-5.AND.POS2=IS-3 THEN
  CALL ADJUST(5,'22212')
  EXIT
END
IF POS1=IS-4.AND.POS2=IS-1 THEN
  CALL ADJUST(5,'21212')
  EXIT
END
IF POS1=3.AND.POS2=7 THEN
  CALL ADJUST(6,'131313')
  EXIT
END
IF POS1=3.AND.POS2=6 THEN
  CALL ADJUST(6,'221212')
  EXIT
END
IF POS1=4.AND.POS2=8 THEN
  CALL ADJUST(6,'121212')
```

HELEN B. CARTER

```

        EXIT
    END
    IF POS1=2.AND.POS2=6 THEN
        CALL ADJUST(8,'33121212')
        EXIT
    END
    IF POS1=2.AND.POS2=4 THEN
        CALL ADJUST(5,'33313')
        EXIT
    END
END
END
IF ORD THEN /*OPERATION IS NOT COMMUTATIVE*/
    IF STACK(IS)=STP THEN
        CALL XCH
    END
END
RETURN
/* */
PROC LOCATE(INT TP,INT STP)
    /*LOCATES DATA ITEMS TP AND STP ON THE STACK*/
    INT J,K
    FND:=0
    K:=0
    WHILE 1 DO
        K:=K+1
        J:=IS
        WHILE J>K DO
            IF (STACK(J)=TP.AND.STACK(J-K)=STP).OR.
                (STACK(J)=STP.AND.STACK(J-K)=TP) THEN
                POS2:=J
                POS1:=J-K
                FND:=1
                EXIT
            END
            J:=J-1
        END
        IF FND THEN EXIT END
    END
RETURN
/* */
PROC ROTATE
    /*DETERMINES THE MINIMAL NUMBER OF ROTATE INSTRUCTIONS NEEDED
    TO POSITION THE 2 CONTIGUOUS DATA OPERANDS TO THE TOP OF
    THE STACK*/
    INT DWNDIF,UPDIF
    DWNDIF:=IS-POS2
    UPDIF:-POS2
    IF DWNDIF<=UPDIF THEN
        WHILE DWNDIF DO
            CALL ROTD

```

```

        DWNDIF:=DWNDIF-1
    END
ELSE
    WHILE UPDIF DO
        CALL ROTU
        UPDIF:=UPDIF-1
    END
END
RETURN
/* */
PROC ADJUST(INT NO,STRING OP)
    /*GENERATES NO INSTRUCTIONS SPECIFIED BY THE CONTENTS OF THE
    STRING OP*/
    INT J
    J:=1
    WHILE J<=NO DO
        PT:=INTF(OP[J,1])    /*CONVERT CHAR TO INTEGER*/
        CASE PT OF
            \1\ CALL XCH
            \2\ CALL ROTD
            \3\ CALL ROTU
        END
        J:=J+1
    END
    RETURN
/* */
PROC GET1(INT TP)
    /*TP IS THE NUMERICAL VALUE OF THE STACK ELEMENT WHICH IS TO
    BE POSITIONED TO THE TOP OF THE STACK*/
    INT J
    J:=IS+1
    WHILE 1 DO
        J:=J-1
        IF STACK(J)=TP THEN
            POS2:=J
            EXIT
        END
    END
    CALL ROTATE
    RETURN
START MAIN

```

APPENDIX C

EXAMPLES OF CODE GENERATED BY THE ALGORITHM

1) A:=5+B
 B:=A+5

postfix 5B+A:=A5+B:=

 LOADI 5
 LOAD B
 ADD
 STND A
 LOADI 5
 ADD
 STD B

2) C:=(A+B)*(C+B)*(C+A)
 D:=C
 E:=A+B

postfix AB+CB+*CA+*C:=CD:=AB+E:=

 LOAD A
 DUP
 LOAD B
 DUP
 ROTD
 ADD
 DUP
 LOAD C
 DUP
 ROTU
 ADD
 XCH
 ROTD
 MULT
 ROTD
 ROTD
 ADD
 MULT
 STND C
 STD D
 STD E

3) C:=A*B+A*B

postfix AB*AB**C:=

LOAD A
 LOAD B
 MULT
 DUP
 ADD
 STD C

4) A:=B*C-D
 M:=E*F+B*C
 C:=E*F*G

postfix BC*D-A:=EF*BC**M:=EF*G*C:=

LOAD B
 LOAD C
 MULT
 DUP
 LOAD D
 SUB
 STD A
 LOAD E
 LOAD F
 MULT
 DUP
 ROTD
 ADD
 STD M
 LOAD G
 MULT
 STD C