A MICRO-PASCAL INTERPRETER

MASTER OF SCIENCE (1977)
(Computation)

McMASTER UNIVERSITY
Hamilton, Ontario

TITLE: A Micro-Pascal Interpreter

AUTHOR: David R. Bandy, B.Sc. (McMaster)

SUPERVISOR: Dr. N. Solntseff

NUMBER OF PAGES: viii, 91

A MICRO-PASCAL
INTERPRETER

by

DAVID R. BANDY, B.Sc.

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University
September 1977

ABSTRACT

A discussion of portability is presented along with a description of the MICRO-PASCAL language. The program developed and documented in this project, accepts an intermediate abstract machine language (intcode) as input and executes these intcode programs on the HP2100. A description of the intcode instruction set and the microprograms used in the interpreter is given. The Micro-Pascal Machine design reflects the current trend to increase program portability.

ACKNOWLEDGEMENTS

I want to thank my supervisor Dr. N. Solntseff, Mark Green and Chris Bryce for their kind help during my project. An honourable mention is reserved for the secretaries of the Applied Mathematics Department whose good spirit is always refreshing. The typing was ably handled by Virginia Rakoczy.

INDEX

		Page
CHAPTER 1	INTRODUCTION	1
1.1	Portability	1
1.1.1	How Can Portability Be Achieved	3
1.1.2	Problems	4
1.2	Interpreters and Compilers	5
1.3	The Portable STAB System	6
CHAPTER 2	MICRO-PASCAL	
2.1	Design Philosophy	8
2.2	Basic Features	8
2.3	Micro-Pascal Machine	10
2.4	Portability and MICRO-PASCAL	11
2.5	Language Assessment	12
2.6	Interpreter Outline	13
CHAPTER 3	INTERPRETER DESCRIPTION	
3.1	Structural Design	15
3.2	The Micro-Pascal Machine	17
3.2.1	Data Representations	18
3.2.2	Procedure Structure	19
3.2.3	Addressing Modes	20
3.2.4	Instruction Set	21
3.3	Code Discussion	33
3.3.1	Main Program	33

		Page
3.3.2	Subroutine Design	37
3.3.3	Program Input	37
3.4	Using The Interpreter	39
3.4.1	Program Input	39
3.4.2	Output	40
3.4.3	Summary of Error Messages	40
3.4.4	Input/Output Routines	41
3.5	Instruction Testing	42
CHAPTER 4	MICROCODING	
4.1	Microprogramming On The HP2100	44
4.2	Microprograms For The Interpreter	45
4.2.1	INCST	45
4.2.2	DECST	47
4.2.3	GBYTE	47
CHAPTER 5	INTERPRETER TESTING	
5.1	Ideal Interpreter Test Program	50
5.2	Sample Programs	53
	Test Program One	54
	Test Program Two	61
CHAPTER 6	SUMMARY	67
	Conclusion	69
APPENDIX A	Running The Interpreter	72
APPENDIX B	Listing Of HP Microprograms	75
APPENDIX C	Test Programs	81
	ARITH	81

Page

BFUNC 82

LOAD + STORE 83

LOGIC 84

MANIP 86

PCALL 87

TRANS 88

Test of Error Messages 89

BIBLIOGRAPHY 91

REFERENCES 91

FIGURES

		Page
2.1	Data Types	9
2.2	MICRO-PASCAL Interpretation on HP2100	11
3.1	Table of Subroutines	15
3.2	Subroutine Map	16
3.3	Instruction Register	18
3.4	Stack Representation of String	18
3.5	Stack Representation of An Array Header	19
3.6	Procedure Header Format	20
3.7	Stack Set-Up for Load Using Relative Address	22
3.8	Stack Set-Up for Store Using Relative Address	23
3.9	Relative to Absolute Address Conversion	24
3.10	Two-Byte Integer Load	25
3.11	Two-Byte Integer Store	26
3.12	Two-Byte Integer Stack Configuration	27
3.13	Case Header Format	31
3.14	Flowchart of Main Program	35, 36
3.15	Input Stream Subgroups	37
3.16	Fix-Up Group Description	38
3.17	Stack Configuration for Procedure Test Program	43
4.1	Flowchart of INCST Microroutine	46
4.2	Flowchart of DECST Microroutine	47
4.3	Flowchart of GBYTE Microroutine	49

CHAPTER 1

INTRODUCTION

This project documents the development of an interpreter to execute intermediate code representations of MICRO-PASCAL programs on the Hewlett-Packard 2100. The interpreter is one part of a portable language system known as the Micro-Pascal Machine. This introductory chapter considers the portability question in general and documents my own experience with the portable STAB system.

1.1 Portability

Portability is a subjective measure of ease with which a program can be moved from one installation to another. A program is highly portable if the effort required to transfer it is significantly less than the original implementation effort. Adaptability is a related problem which measures the ease of program alterations needed to meet different system constraints. The difference in these two operations is that portability concerns environmentally governed changes in the algorithm.

The need for portability seems to arise in two situations. In the first case, programs should be portable over a whole machine range to permit moving to a larger machine or adding a smaller one in parallel. The second case concerns portability to and from alien machines. An installation with a program library that is highly portable is not as apt to be committed to a specific computer or manufacturer. Such an installation has a better bargaining position when in the market for new machinery. Manufacturers whose programs are portable are able to provide working software in short order to complement new hardware.

A program package written in a portable fashion is more attractive to other installations due to the relative ease in adaptation. Programmers can often save time and effort by adapting an existing program that does some or all of the desired task instead of designing a new one from scratch. As an illustration, academic and research people could move to other installations easily and exchange portably designed programs avoiding much wasteful duplication.

It has been said that programs should not be portable because they can be improved if they are rewritten. However if a program is portable the user has the option to allocate resources to improve it or rewrite the program. Even if the decision is made to rewrite, the portable version can be used during the rewrite period and as an aid in designing the new program.

In summary the advantages of portability lie in the minimization of development time and duplicate programming effort, the retained usefulness of older programs and the increased mobility of program packages.

1.1.1 How Can Portability Be Achieved?

Portability requires a program to be independent of special properties of the operating system or, more generally, requires that an appropriate program environment be provided on many current installations. Some people have suggested rigid standardization as a possible solution. This solution would permit greatly increased application portability. However, program packages like compilers and operating systems would still have to be installation dependent. In the past, standards have been incomplete, compromised because specific machine features were not exploited and several years behind current trends.

Another solution may lie in machine independent systems. Machine independence refers to program properties that isolate it from the details of the computer structure such as word length, addressing scheme and the number and kind of registers.

These two solutions to the portability problem are different yet not mutually exclusive. Both properties can be achieved by using a high-level programming language.

These programs are machine dependent only with respect to the accuracy of real arithmetic, the range of arithmetic values and the character set. The character set problem can be partially alleviated through the use of a standard 48 character set. Programs written in high-level languages are more portable if the use of Input/Output is restricted to the more standard sequential files.

Another solution suggests dividing a program into a data description segment and an algorithmic section. The algorithmic part would be as machine independent as possible while the data description would be adjusted to cope with the host machine. An abstract machine model is a mechanistic interpretation of the data and algorithm split up. The basic operations and data types are used to define a fictitious computer or abstract machine. An abstract machine model of this type can be used to construct a new high-level language like MICRO-PASCAL and the Micro-Pascal Machine.

1.1.2 Problems

One of the problems facing program portability is the lack of good current programming standards. Strict adherence to such standards will pave the way for more portable programs. Portability is hindered by the wide variation in machine codes and architectures currently available in the market place.

Historically tasks such as input/output and code generation have relied on machine-code programming. These habits must be broken if more portable programs are to be written.

It is often claimed that portable software is synonymous with inefficient software. Unfortunately this has frequently been the case. Inefficiency usually stems from data packing schemes that are not suited to fast access on the host computer, very complicated interfaces to the environment like the operating system and inefficient code for heavily repeated loops. The trade-off between portability and efficiency is a problem that will likely persist for some time. Its current solution lies in minimizing the inefficiency until machine architectures and therefore machine-codes become standardized.

1.2 Compiler - Interpreters and Compilers

Consider the differences in compiler - interpreter and compiler systems in the light of our portability discussion. A compiler - interpreter, as the name suggests, performs two functions. In the first phase, it analyzes the complete source program and translates it into an internal form. The second phase interprets or executes the internal code representation of the source program.

In compiler systems the source program is analyzed and translated into object code. Programs compiled into object code usually execute faster because this code is handled faster by system routines than internal code is by the interpreter. Compiler - interpreter systems tend to be more portable than compiler systems. The compilation phase is largely machine independent and can usually be lifted intact for a transfer. The execution segment is fairly straight-forward and can be written for the host machine without undue difficulty if the documentation is thorough.

Compiler systems generate machine dependent object code which makes them significantly less portable. However, if the system was written to be reasonably portable and modifiable the prospects for successful transfer are dramatically improved. The machine dependent areas such as code generation and input/output could be clearly marked so that modifications could proceed as smoothly as possible.

1.3 The Portable STAB System

Prior to this project I spent about six weeks working with the portable STAB system. The STAB machine is very similar to the Micro-Pascal Machine. STAB source code is compiled into an intermediate machine language which is subsequently interpreted.

STAB is a programming language spawned from BCPL and designed as a high-level language implementable on small machines as well as a straight-forward compiler writing tool. I was writing a new STAB compiler because the existing one was unstructured, unreadable and unmodifiable. As parts of the new compiler were written they were tested using the old compiler. This testing procedure was hampered considerably by numerous errors detected in my source code by the old compiler. Many of the errors were not sufficiently explained by the error messages or by the STAB programming language documentation. Numerous errors remained a mystery to both my supervisor, Dr. Solntseff and myself. Initially I was able to correct these errors by intuition and program re-organization. However, as time went on the situation deteriorated to the point where little real progress was being made on my compiler and its code was so significantly altered to facilitate a clean compile that it was inefficient. At this point the project was halted and this project started.

The portable STAB system failed on two accounts: a poor compiler and insufficient documentation. We can conclude that a portable system needs a readable, structured and easily modifiable compiler accompanied by a full and thorough documentation to be workable in a new surrounding.

CHAPTER 2

MICRO-PASCAL

2.1 Design Philosophy

MICRO-PASCAL is a high-level language to be implemented on micro or mini-computers like the HP2100. It is a language well suited for writing compilers in a readable, understandable and modifiable form.

2.2 Basic Features

MICRO-PASCAL, as its name would suggest, is modelled after the full PASCAL language. The quantities in a Micro-Pascal program are constants, simple variables, arrays, strings, procedures and the presently unimplemented functions. There are five types of declarations: labels, constants, variables, procedures and functions. Data types fall into three categories based on their size.

SIZE	NAME	DESCRIPTION
One Byte	Byte	integer
	Char	character
Two Byte	Integer	integer
N Byte	String	character string
	Array	integers, characters

figure 2.1 Data Types

Numbers in MICRO-PASCAL are represented by one or two byte integers.

Arithmetic operations available include addition, subtraction, multiplication and division on BYTE and INTEGER types. The basic boolean operations of EQ, NE, LT, LE, GT, GE, AND, OR and NOT exist for BYTEs and INTEGERs. MICRO-PASCAL has four basic input/output operations, READ-reads from the current input buffer, READLN-terminates reading from the current input buffer, WRITE-writes to the output buffer and WRITELN-dumps the output buffer to the desired output unit.

Valid statements include a compound form as well as the standard assignment. A compound statement consists of a group of statements surrounded by a BEGIN and an END. The Micro-Pascal control statements are limited to a GOTO.

an IF-THEN and IF-THEN-ELSE block, a CASE block and a WHILE-DO repeat block. Parameter passing on procedure calls is limited to call by value only.

Some of the PASCAL features removed from MICRO-PASCAL include sets, records, pointers, types, reals and the REPEAT and FOR statements. Fuller documentation on MICRO-PASCAL is available elsewhere.(GRE)

2.3 Micro-Pascal Machine

We refer to the Micro-Pascal package as the Micro-Pascal Machine. The Micro-Pascal source program is executed by a compiler - interpreter which works in two phases. Source code is first compiled into an intermediate abstract machine language referred to as intcode. The intcode is then executed on the host machine. My goal, in this project, is to develop and document an interpreter to execute intcode programs on the Hewlett-Packard 2100.

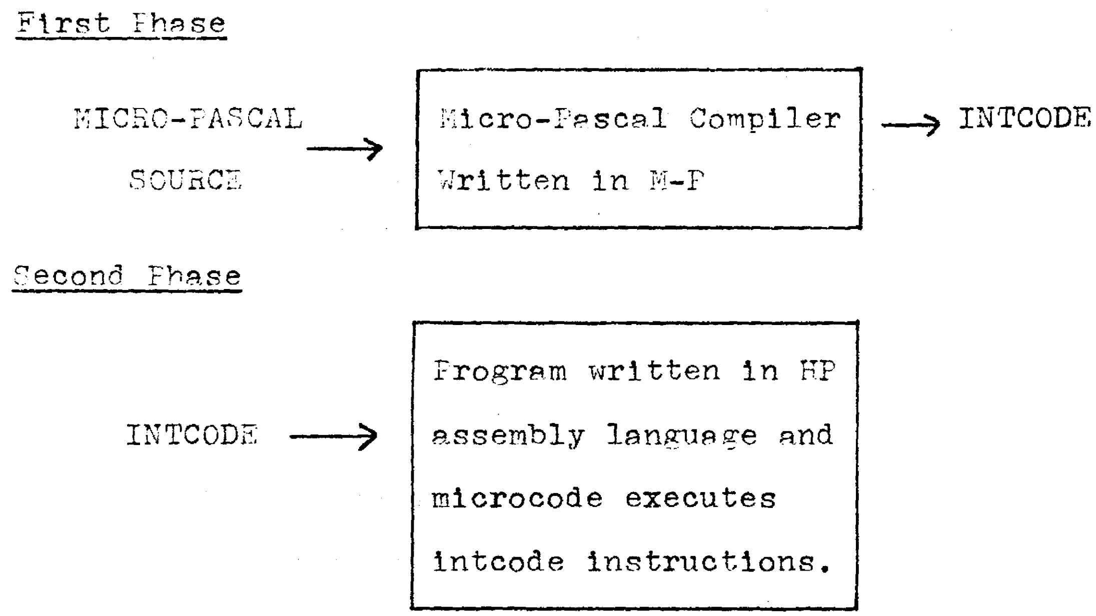


figure 2.2 Micro-Pascal Interpretation on HP2100

The Micro-Pascal Machine has already been implemented on the CDC 6400. This project will allow Micro-Pascal programs to be compiled into intcode on the CDC then executed on the HP2100. The full Micro-Pascal Machine will be realized on the HP2100 when the compiler is written in MICRO-PASCAL and interpreted into intcode.

2.4 Portability and MICRO-PASCAL

We can now consider the Micro-Pascal Machine with respect to the portability question. The compile phase of interpretation is fairly machine independent since it is written in MICRO-PASCAL and generates the standard abstract machine language as code. It could be used with intcode executors on various host machines requiring only minor alterations if any.

The execution phase of the interpreter is firmly rooted in the host machine. Its flexibility stems from the fact that a compile operation for any source language emitting compatible intcode programs could be used with it. The execution phase could theoretically be part of a number of different language machines or interpreters.

2.5 Language Assessment

Even though MICRO-PASCAL and its implementations are still in the experimental phase, some general language criticisms are worth considering. A Micro-Pascal program must be compiled then interpreted in order to run. This two phase operation contributes to a lengthy total run time as well as a slow execution speed. Programmers accustomed to the larger and more powerful high-level languages such as PASCAL will find MICRO-PASCAL restrictive, at least initially, due to the limited control statements. The size restriction on Micro-Pascal programs will depend on the available host machine memory and the efficiency of the interpreter implementation.

A strong argument for MICRO-PASCAL is the availability of a high-level language on a mini or micro-machine that has been previously restricted to lower-level languages. The mini or micro-machine programmer is given new freedom not present in assembler and machine languages. This freedom should contribute to a reduction in program development time. Since MICRO-PASCAL is a small and simple language its compiler can be written using time and space to maximum efficiency. In a small machine environment it is essential that resources be efficiently allocated.

The Micro-Pascal system is semi-portable since the compile phase emits a standard machine independent intcode. Once the execution phase has been adapted to the host machine to accept intcode, the standard compile program can be used. MICRO-PASCAL is intended as a modifiable system which is fairly flexible to local tampering unlike some systems such as PASCAL-S.

2.6 Interpreter Outline

The interpreter is written in Hewlett-Packard assembly language and microcode. It accepts and executes intcode programs which are read as a string of bytes. These programs can be read by the interpreter from any specified input device. As currently implemented, input during program execution is limited to cards and the keyboard while output can be routed to the line printer or the keyboard.

Instructions executed by the interpreter are at least one eight-bit byte long and this first byte is split into a group and level number. The main program decodes the group and level components of the current instruction and branches to the required group subroutine. The flow of control within the subroutine is based on the level number and when it is matched the required instruction is executed. Control is then returned to the main program where the instruction cycle is repeated until the stop command is encountered.

CHAPTER 3

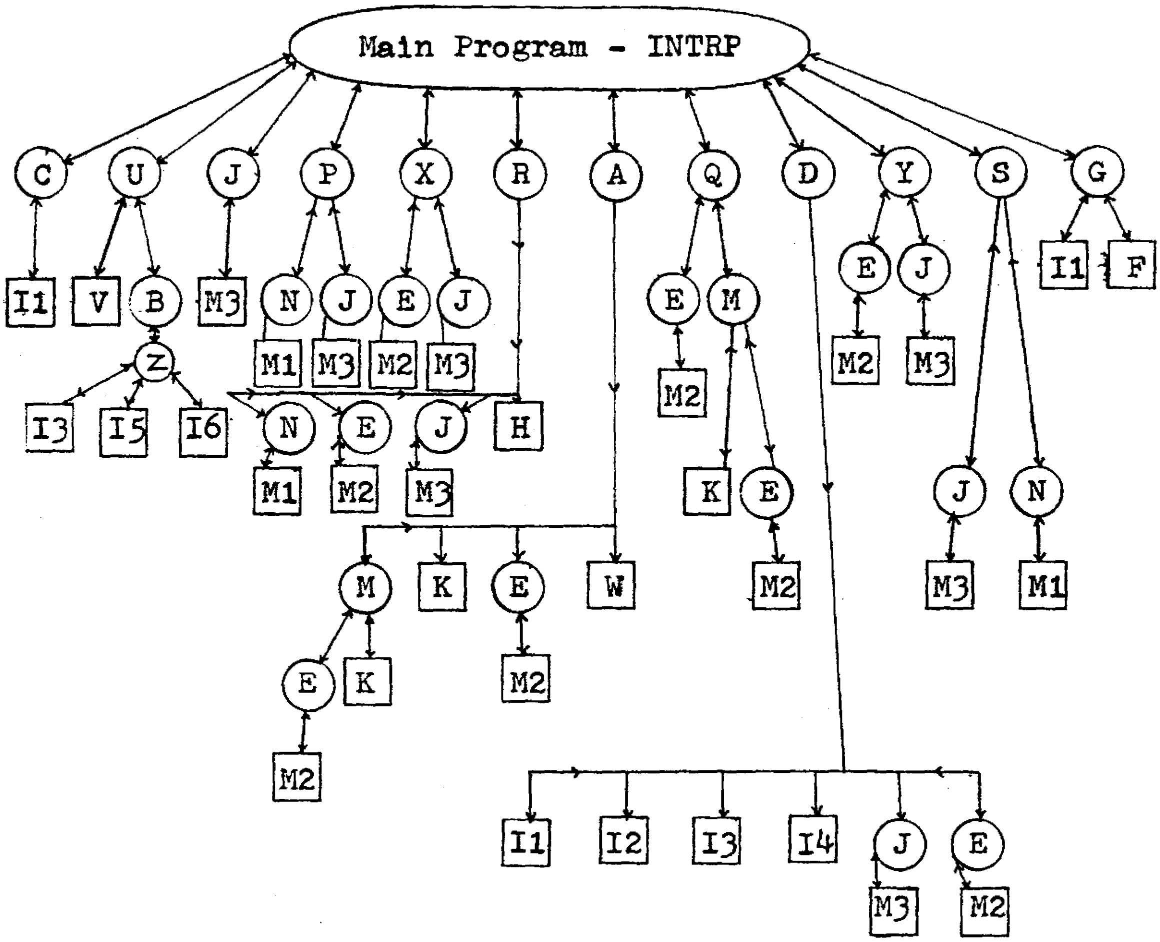
INTERPRETER DESCRIPTION

3.1 Structural Design

The interpreter consists of one main routine, three microroutines, twenty-five subroutines and six input/output subroutines. The heart of the interpreter is the main program, INTRP, along with the subroutines ARITH, BFUNC, LOAD, LOGIC, MANIP, PCALL, STORE and TRANS.

Assembler Subroutines		Microprogram Subroutines
A - ARITH	N - INCSK	
B - ASSBY	O - LLERR	M1 - INCST
C - BEGIN	P - LOAD	M2 - DECST
D - BFUNC	Q - LOGIC	M3 - GBYTE
E - DECSK	R - MANIP	
F - DIGIT	S - PCALL	I/O Subroutines
G - FINI	T - PUTBT	I1 - WRCRT
H - GETAD	U - READP	I2 - WRLP
I - GETBT	V - STBYT	I3 - RDCRD
J - GETBY	W - STINT	I4 - RDCRT
K - GETIN	X - STORE	I5 - RDPT
L - GPERR	Y - TRANS	I6 - RDDSC
M - GT2IN	Z - ZASSY	
N - INCSK		

figure 3.1 Table of Subroutines



Symbols

○ - represents subroutines that can call other subroutines

□ - represents subroutines that do not call other subroutines

figure 3.2 Subroutine Map

The subroutine map illustrates the overlaps that are avoided with a modular type design.

3.2 The Micro-Pascal Machine

The Micro-Pascal Machine consists of three main arrays: CODE, STACK and DSPLY and a few main pointers. Intcode generated by the Micro-Pascal compiler is stored two bytes per word in the CODE array. The instruction pointer, IP, points to the executable byte in the CODE array. The STACK array is used for run-time data storage and is referenced by all but a few instructions. SP is the stack pointer which indexes the top stack element. A zero SP value indexes the first stack element. Even though each stack element is two bytes large, only the lower byte is used in the stack operations. The level of procedure nesting is stored in LVL. DSPLY is a sixteen bit array used in addressing and LVL is the index of the most current entry. When a procedure is called, DSPLY(LVL + 1) is set to the current value of SP. After procedure execution the stack pointer can be reset to the proper value using the DSPLY array. Initial values for these variables are SP = 0, LVL = 0, DSPLY(LVL) = 0 and IP = 1.

All the interpreter instructions have an initial byte of the following form:

GROUP	LEVEL
4 bit	4 bit

figure 3.3 Instruction Register

The first four bits represent the group category of the instruction while the other four usually specify a particular instruction within a general group classification. There can be zero or more bytes following the initial one that are part of an instruction.

3.2.1 Data Representations

The data representations fall into three categories one, two and N-byte types. Characters and integers are stored in one byte, larger integers can be accomodated in two bytes and strings and arrays take an unspecified number of bytes. Strings are at least two bytes long and appear on the stack as follows:

Byte	1	2	3......	N
	PHYSICAL SIZE	LOGICAL SIZE	CHARACTERS	

figure 3.4 Stack Representation of String

Each string occupies two plus the physical string size bytes on the stack.

An array has a header that describes its structure and data. The header appears on the stack as illustrated below:

Byte	1	2	3	4	5	6
	elsize	dim #	lb1	size 1	lb2	size 2 ...

elsize - the total array size in bytes

dim # - the number of array dimensions

lb1 - lower bound of the first array index

size 1 - the range of the first array subscript
upper bound - lower bound + 1

.
.
.
etc.

figure 3.5 Stack Representation of An Array Header

3.2.2 Procedure Structure

When a procedure is called, a seven-byte header is created on top of the stack and an entry is made in the DSPLY array which points to the start of the header. The procedure call specifies the new value of LVL for the life of the procedure and the address of the procedure.

Byte	Contents
1	the old LVL value
2 + 3	the return address
4 + 5	address of the result for functions
6 + 7	entry in DSPLY for the old LVL

figure 3.6 Procedure Header Format

3.2.3 Addressing Modes

Instructions

All the instruction addresses are relative to the first element of the CODE array. The instruction pointer value, IP, that references the first byte in the CODE array is one.

Data

All data addresses are relative to byte zero of the STACK array. This starting location corresponds to a stack pointer or SP value of zero. Stack addressing can be relative or absolute. A relative address consists of three bytes. An element of the DSPLY array is specified by the index in the first byte and this value serves as the base address. The following two bytes form a positive sixteen bit displacement which is added to the base address giving the final address. An absolute stack address consists of two bytes which together specify a sixteen bit stack address.

3.2.4 Instruction Set

The instructions currently implemented in the interpreter are described below:

GROUP 0 - LOAD

This is a three-byte instruction in which the LEVEL component of the first byte points to an entry in the DSPLY array and this value acts as the base address. The second and third bytes of the instruction form a sixteen bit address, ADR. The byte at address DSPLY(LEVEL) + ADR is loaded onto the top of the stack.

GROUP 1 - STORE

STORE is a three-byte instruction where the LEVEL part of byte one specifies an element of the DSPLY array. Bytes two and three form a sixteen bit address, ADR. The byte on the top of the stack is stored at stack address DSPLY(LEVEL) + ADR.

GROUP 2 - STACK MANIPULATION

Level

0 - In this two-byte instruction byte two specifies the number of bytes by which the stack pointer is to be incremented.

1 - The second byte is stored on top of the stack.

2 - The string following the first byte is stored onto the stack. See figure 3.4 for string format.

GROUP 2 - STACK MANIPULATION cont'd

Level

3 - The second byte specifies the number of following bytes that are to be loaded onto the stack.

4 - The top three bytes on the stack specify an index into the DSPLY array and a sixteen bit address, ADR. The byte at DSPLY(LVL) + ADR is loaded onto the stack

Low byte of address	SP
High byte of address	SP - 1
LVL	SP - 2

figure 3.7 Stack Set-Up for Load Using Relative Stack Address

GROUP 2 - STACK MANIPULATION cont'd

Level

5 - The top stack element is stored at the address DSPLY(LVL) + ADR.

Top Stack Element	SP
Low byte of address, ADR	SP - 1
High byte of address, ADR	SP - 2
LVL	SP - 3

figure 3.8 Stack Set-Up for Store Using Relative Stack Address

GROUP 2 - STACK MANIPULATION cont'd

Level

6 - The three-byte relative stack address on the top of the stack is converted to a two-byte absolute stack address preceded by a zero byte.

Stack Before		Stack After
Low byte of relative address	SP	Low byte of absolute address
High byte of relative address	SP - 1	High byte of absolute address
LVL	SP - 2	0

figure 3.9 Relative to Absolute Stack Address Conversion

7 - The second byte specifies the number of bytes by which the stack pointer is to be decremented.

GROUP 2 - STACK MANIPULATION cont'd

Level

8 - This instruction determines a relative stack address and loads a two-byte integer onto the stack using that address. The top three stack elements form the relative stack address, ADR. The high byte of an integer stored at ADR is loaded onto the stack at SP - 2 while the low byte of the integer is loaded from ADR + 1 onto the stack at SP - 1.

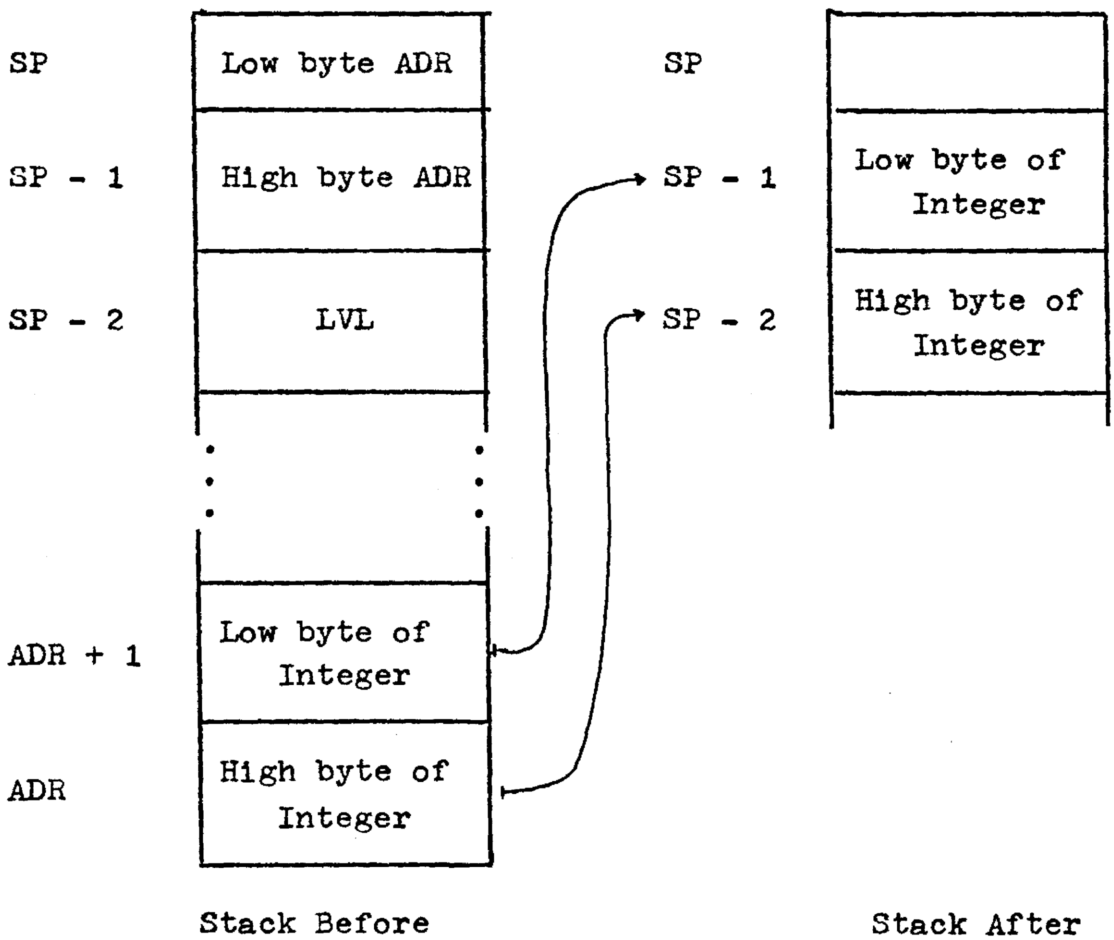


figure 3.10 Two-byte Integer Load

GROUP 2 - STACK MANIPULATION cont'd

Level

9 - This instruction stores the two-byte integer on top of the stack at the relative stack address, ADR, specified by the three bytes below it.

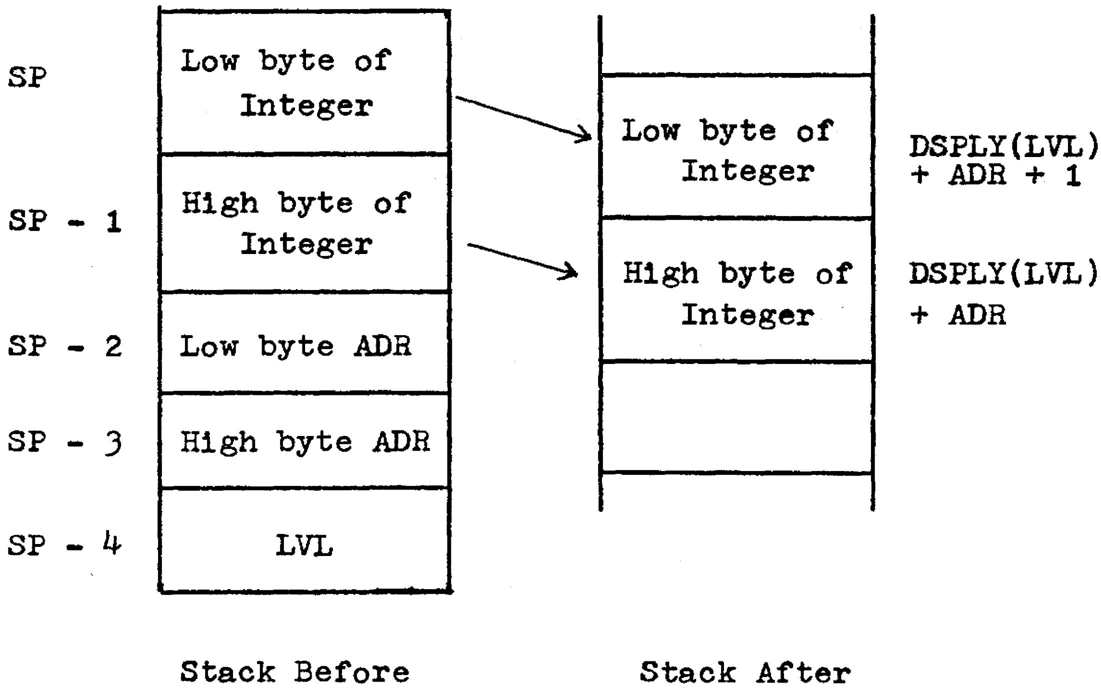


figure 3.11 Two-byte Integer Store

GROUP 3 - ARITHMETIC OPERATIONS

Level

0 - The top stack element is negated in two's complement form.

1 - The two bytes on top of the stack are added together and stored at stack position SP - 1.

2 - The byte at STACK(SP), the top stack element, is subtracted from STACK(SP - 1) and the result is stored at stack position SP - 1.

3 - The byte at stack position SP is multiplied by the byte at SP - 1 and the result is stored at SP - 1.

4 - The byte at stack position SP - 1 is integer divided by the byte at SP with the result stored at SP - 1.

The next five operations work with two-byte integers that appear on the stack as follows:

SP	Low byte of Integer	referred to as I2
SP - 1	High byte of Integer	
SP - 2	Low byte of Integer	referred to as I1
SP - 3	High byte of Integer	

figure 3.12 Two-byte Integer Stack Configuration

GROUP 3 - ARITHMETIC OPERATIONS cont'd

Level

5 - The top two-byte integer, I2, is negated in two's complement form.

6 - Add the integers I2 and I1 storing the low byte of the integer result at SP - 2 and the high byte at SP - 3.

7 - Subtract I2 from I1 and store the result at SP - 2 and SP - 3.

8 - Multiply I2 by I1 and store the result at SP - 2 and SP - 3.

9 - Divide the integer I1 by the integer I2 storing the result at SP - 2 and SP - 3.

GROUP 4 - LOGICAL OPERATIONS

The top two stack elements are compared according to the specified logical relation. If the relation holds a one is placed at stack location SP - 1. Otherwise a zero is placed at SP - 1. The logical operation is performed as illustrated below.

STACK(SP - 1)   Logical Operator   STACK(SP)

GROUP 4 - LOGICAL OPERATIONS cont'd

Level

0 - equal

1 - not equal

2 - less than

3 - less than or equal to

4 - greater than

5 - greater than or equal to

The next six operations repeat the same tests on two-byte integers

SP	Low byte of Integer	referred to as I2
SP - 1	High byte of Integer	
SP - 2	Low byte of Integer	referred to as I1
SP - 3	High byte of Integer	

I1 Logical Operator I2

6 - equal

7 - not equal

8 - less than

9 - less than or equal to

10 - greater than

11 - greater than or equal to

GROUP 5 - BUILT-IN FUNCTIONS

Level

0 - This instruction terminates the current output line. (For those familiar with PASCAL this corresponds to a WRITELN).

1 - This instruction terminates reading from the current input line. If the input is coming from cards the rest of the card is skipped and a further read will use the next input card. (READLN).

2 - The character on the top of the stack is written to the output device specified by the byte at stack position SP - 1. (WRITE).

The Input/Output units currently implemented on the HP2100 are illustrated below:

Available for	INPUT	OUTPUT
Execution and	0 - CRT	0 - CRT
Input	1 - CARDS	1 - LINE PRINTER
Intcode Input	PAPER TAPE	
	DISC	

GROUP 6 - TRANSFER OPERATIONS

Level

0 - This is an unconditional jump instruction. The second and third bytes specify a relative instruction address which is transferred to immediately.

1 - This is a conditional jump instruction. The second and third bytes provide a relative instruction address which is transferred to if the top stack element is zero. If the top stack element is non-zero then the next instruction is executed.

2 - This instruction specifies a case statement on the top stack element. The GROUP - LEVEL byte is followed by a number of case elements which consist of a case header and code for the case element. The case header format is

Byte	1	2 + 3
	item	address

figure 3.13 Case Header Format

where: item - the case label for the case element

address - the relative instruction address of the next case header

GROUP 6 - TRANSFER OPERATIONS cont'd

Level

2 - If the address value is zero then the case statement has ended. Therefore the final case element consists of an undefined item value and a zero address. When the case statement is executed, the case header items are searched and compared with the top stack item. If a match is made, the code corresponding to the particular case item is executed before continuing with the next instruction after the case. When no match is made with top stack element control passes directly to the instruction after the case.

3 - This instruction is a procedure return which uses the procedure header, created on call, to recreate stack condition as they were before the call. These activities include resetting LVL, DSPLY(LVL), the stack pointer, SP, and the instruction pointer, IP, to the return address.

4 - This is the stop instruction which terminates program execution.

GROUP 9 - PROCEDURE CALL

The two bytes after the GROUP - LEVEL byte specify the relative instruction address of a procedure to be called. A procedure header is created on top of the stack as described in section 3. The base of the procedure header is pointed to by the DSPLY(LEVEL) entry and control is transferred to the procedure.

3.3 Code Discussion

3.3.1 Main Program

The main program performs the following five basic functions:

1) Declares and initializes program variables and error messages.
2) Introduces the interpreter to the user.
3) Reads in the intermediate code.
4) Executes the code.
5) Prints summary information related to interpreter execution.

Two of the more prominent variables initialized at the start of interpretation are STLIM, the maximum value of stack pointer, and CODMX, the maximum number of bytes that can be stored in the CODE array.

The program execution begins with some descriptive information that is sent to the CRT via the BEGIN routine. This routine flags the start of interpretation, summarizes the input and output numbers and lets the user force program output to the CRT or the line printer.

The intcode program is read into the CODE array by the READP routine. A program pause occurs so the user can specify the input device number. When this is completed the program reads characters one at a time from the input stream assembling instruction bytes and storing them in the CODE array. Once the termination header is encountered the reading process is over.

The transfer of control in the main execution loop is based entirely on the group number of the current instruction. The loop begins with a call to GETBY which recovers the current instruction byte pointed to by IP. This byte is split into the GROUP and LEVEL components and the GROUP number tested against the valid possibilities. (An error message for an invalid group number is emitted if the matching attempt is unsuccessful.) If a match is made, then the appropriate group subroutine is called to execute the instruction. After execution, control returns to the start of the loop where an end flag is inspected. A stop instruction sets this flag and brings the main loop to an end.

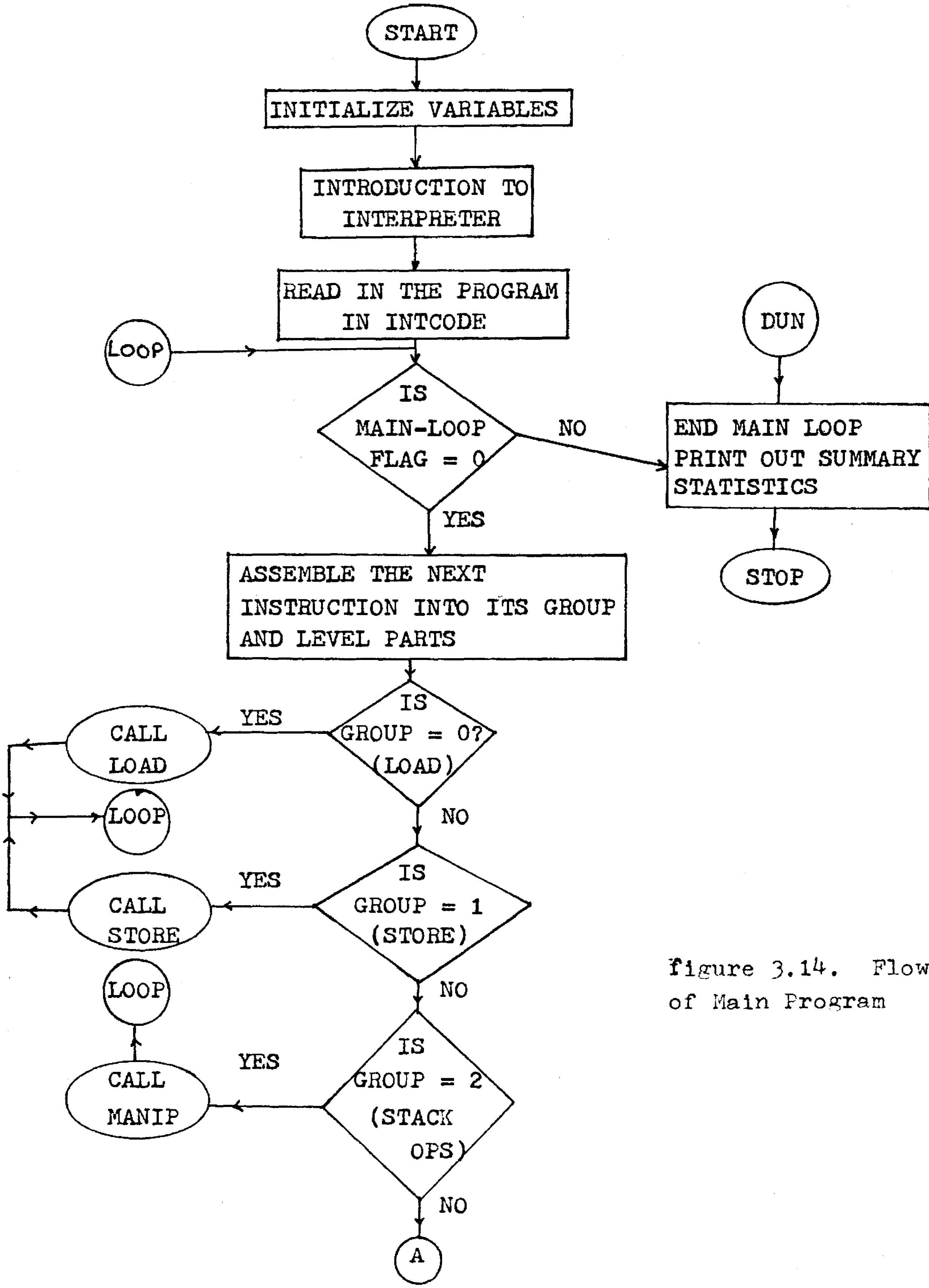


figure 3.14. Flowchart of Main Program

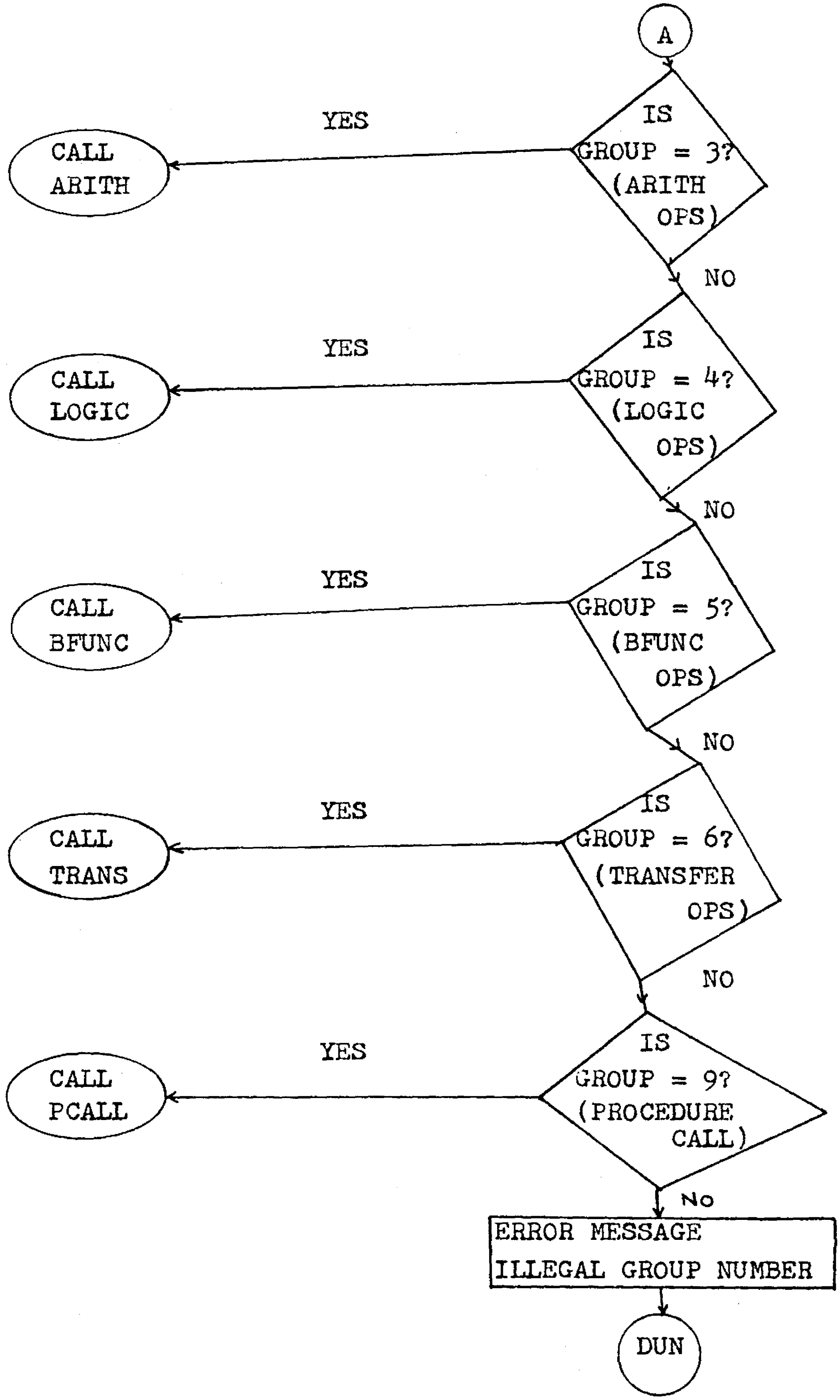


figure 3.14 Flowchart of Main Program

3.3.2 Subroutine Design

The main subroutines, ARITH, BFUNC, LOGIC, MANIP and TRANS were written in as structured and readable a manner as possible in assembly language. The transfer of control within these subroutines is based upon the LEVEL number of the current instruction. A valid LEVEL value uniquely defines an instruction within a GROUP subroutine. These subroutines are equipped with error exits which flag the occurrence of unexpected high LEVEL values after the legal ones have been checked.

3.3.3 Program Input

The interpreter accepts a string of octal bytes as input. Within the input stream there can be occurrences of three subgroups. One group consists of code bytes, another of fix-up bytes to edit addresses and the other is an end of information marker.

TYPE OF INPUT	1st BYTE	2nd BYTE	FOLLOWING BYTES
CODE	001	n_1	n_1 bytes
FIX - UP	002	n_2	n_2 bytes
END OF INFO	143	000	

figure 3.15. Input Stream Subgroups

Each form has a header consisting of two bytes. The first byte indicates the type of input to follow. The second byte of the code block indicates the number of bytes that follow and are to be placed in the CODE array. A code block must be the first form appearing in the input. It can reappear anywhere in the input stream except as the last block which is always the end of information. The fix-up block can appear after any code block. The number of bytes in the block, specified by the second byte, is a multiple of four since each fix-up instruction uses four bytes. The Micro-Pascal compiler uses a one-pass approach so that labels are often left unspecified until they are encountered later on in the program scan. A zero address is generated by the compiler when a label with an unknown address is scanned. When the label is determined a fix-up entry is prepared to replace the zero address.

			Fix-Up Address High byte	Fix-Up Address Low byte
Byte Number	1	2	3	4
	form 16-bit relative instruction address - ADR		load at ADR	load at ADR + 1

figure 3.16 Fix-Up Group Description

The first two bytes of the fix-up group specify a sixteen bit relative instruction address within the CODE array where the third byte will be loaded followed by the fourth byte in the next location. Together, these final two bytes form the previously undefined label address. The end of information block occurs only once at the end of the input stream.

3.4 Using The Interpreter

3.4.1 Program Input

The intcode program can be read from disc, cards and paper tape. Input is expected to be in the form of octal bytes separated by some delimiter. Illegal characters, those that are not octal digits, are completely ignored. A sample program is illustrated below.

```
001 144
021 000 010 140 000 051 143 040 000 040
001 044 021 000 012 040 024 040 001 040
001 001 000 012 041 012 064 021 000 014
001 000 014 041 000 101 141 000 000 040
007 041 001 041 000 041 014 046 047 012
221 000 153 140 000 234 001 000 012 001
000 014 041 012 063 062 041 060 061 021
000 013 000 000 025 001 000 013 122 143
040 000 040 001 040 001 040 001 040 001
041 001 021 000 010 001 000 010 041 012

002 054
000 120 000 142 000 140 000 147 000 060
000 152 000 153 040 002 000 211 000 234
001 040 001 223 000 337 001 226 000 312
001 242 000 027 001 243 001 302 002 005
002 043 002 143
143 000
```

3.4.2 Output

Program output is normally routed to the unit specified in the program code. It can also be directed to the CRT or line printer by the user from the CRT.

3.4.3 Summary of Error Messages

After each error message printout we get the normal program termination output which includes the number of instructions executed, the maximum stack pointer value during program execution and the program size in bytes. The error messages are summarized below.

1) Group number of current instruction is not valid.
2) Level number of current instruction is too large.
3) CODE array is too small to accomodate the program.
4) The index of the DSPLY array is greater than fifteen and therefore references an out of bounds element.
5) The stack is too small for the current program, stack overflow.
6) The stack pointer has been decremented below zero, stack underflow.
7) An input block with an illegal type header, not 1, 2, or 143, has been encountered.

If error three is encountered the size of the CODE array can be increased by changing source lines 84 and 219. Similarly an occurrence of error five may require an increased stack size which can be made by changing source lines 91 and 213. The dimension of the DSPLY array limits the level of procedure nesting to fifteen.

3.4.4 Input/Output Routines

The input/output routines used by the interpreter provide the interface between the Hewlett-Packard DOS-M operating system and the Micro-Pascal Machine. This group of routines makes available all of the peripheral devices related to the Hewlett-Packard 2100 at McMaster. The routines are summarized below:

1) Line - Printer
2) Paper - Tape Punch
3) Paper - Tape Reader
4) CRT - Input
5) CRT - Output
6) Card - Reader

} All of these routines are equipped to handle single character and buffered information.

7) Read from Disc
8) Write to Disc
9) Write to Job Binary Area

} These routines work with one binary byte.

10) Write a record to Job Binary Area
11) Display time of day

These routines were written originally for use in the STAB system by my supervisor, Dr. N. Solntseff. The routines currently used by the interpreter are 1, 3, 4, 5, 6 and 7.

3.5 Instruction Testing

Programs were written to verify each group of instructions. The test program created the conditions where the operation of each instruction could be sequentially checked. This checking procedure was possible using a debug package loaded with the program. Stack conditions including the stack pointer were sampled before and after the instruction execution. The actual stack configurations were checked against the expected values which were calculated by hand before execution. All of the test programs are included in Appendix C. The testing process is illustrated here for the procedure call and return program.

Pre-Calculated Results

1) Load 1 onto the stack, SP ← 1

2) Load 2 onto the stack, SP ← 2

3) Procedure call - return address 12,
SP ← 11

4) Load 4 onto the stack, SP ← 12

5) Procedure return,
Set IP ← 12
LVL ← 0
SP ← 2

6) Load 377 onto the stack, SP ← 3

7) Stop

Stack		SP
4		12
0		11
0		10
0		7
0		6
7		5
0		4
0	377	3
2		2
1		1
		0

figure 3.17 Stack Configuration for Procedure Test Program

Octal Representation of Program

041	001	041	002	220	000
012	041	377	144	041	004
143					

CHAPTER 4

MICROCODING

4.1 Microprogramming On The HP2100

The Hewlett-Packard 2100 used in this project is equipped with microprogramming facilities. It has four writable control store modules where the micro-instructions are stored. Module 0 contains the basic 2100 instruction set, module 1 the floating point instructions and the remaining modules 2 and 3 are available for programmer use. A microprogram is a program-structured sequence of commands residing in the hardware or writable control store. When a microprogram is executed it is translated into hardware actions by hardware controls. This hardware translation means fast and efficient execution. Microprograms are usually more difficult to write because they work on such a primitive level. Further information on Hewlett-Packard Microprogramming can be found in the HP2100 Microprogramming Guide and Microprogramming Software Handbook.

The three most frequently called subroutines: DECSK-stack decrementation, INCSK-stack incrementation and GETBY-extraction of next instruction byte were chosen for microcoding because they would maximize the increase in interpreter execution speed and could be implemented fairly quickly.

4.2 Microprograms For The Interpreter

This section summarizes the microroutines with brief program descriptions and flowcharts. The microprogram listings appear in Appendix E.

4.2.1 INCST

This microroutine increments the stack pointer, SP, by a specified value, checks for stack overflow and retains the maximum stack pointer value. It has four arguments: the stack increment in the B register, the stack pointer address, the address of maximum allowable SP value - STLIM and the address of maximum SP value to date - MSTCK. The value returned in the B register on microprogram termination indicates a stack overflow condition with a one and a normal exit with a zero value.

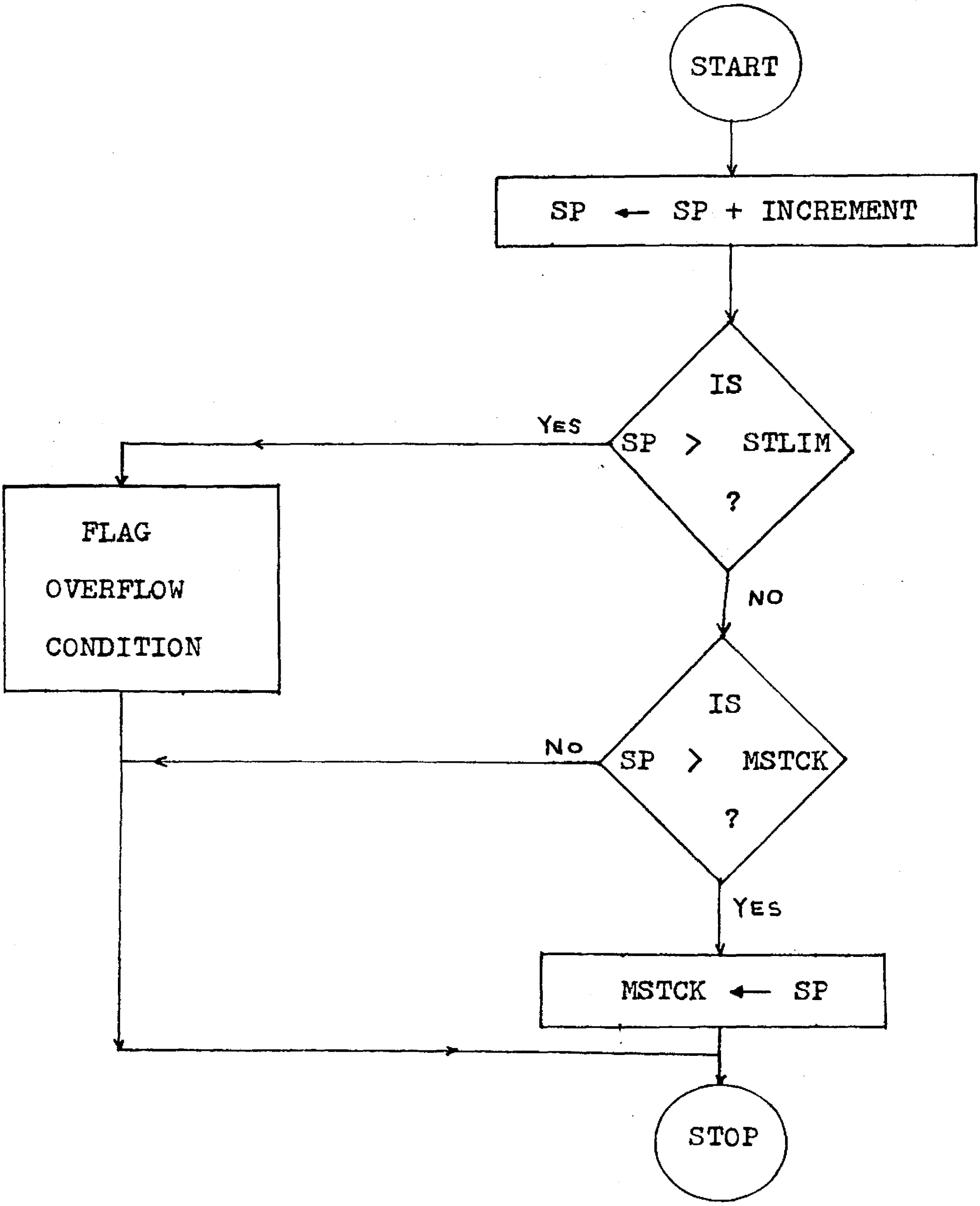


figure 4.1 INCST

4.2.2 DECST

This routine decrements the stack pointer, SP, by a specified value and checks for stack underflow. It has two arguments, the stack pointer address and the stack pointer decrement in the B register. On microprogram exit an A register value of one indicates stack underflow while a zero value flags a normal termination.

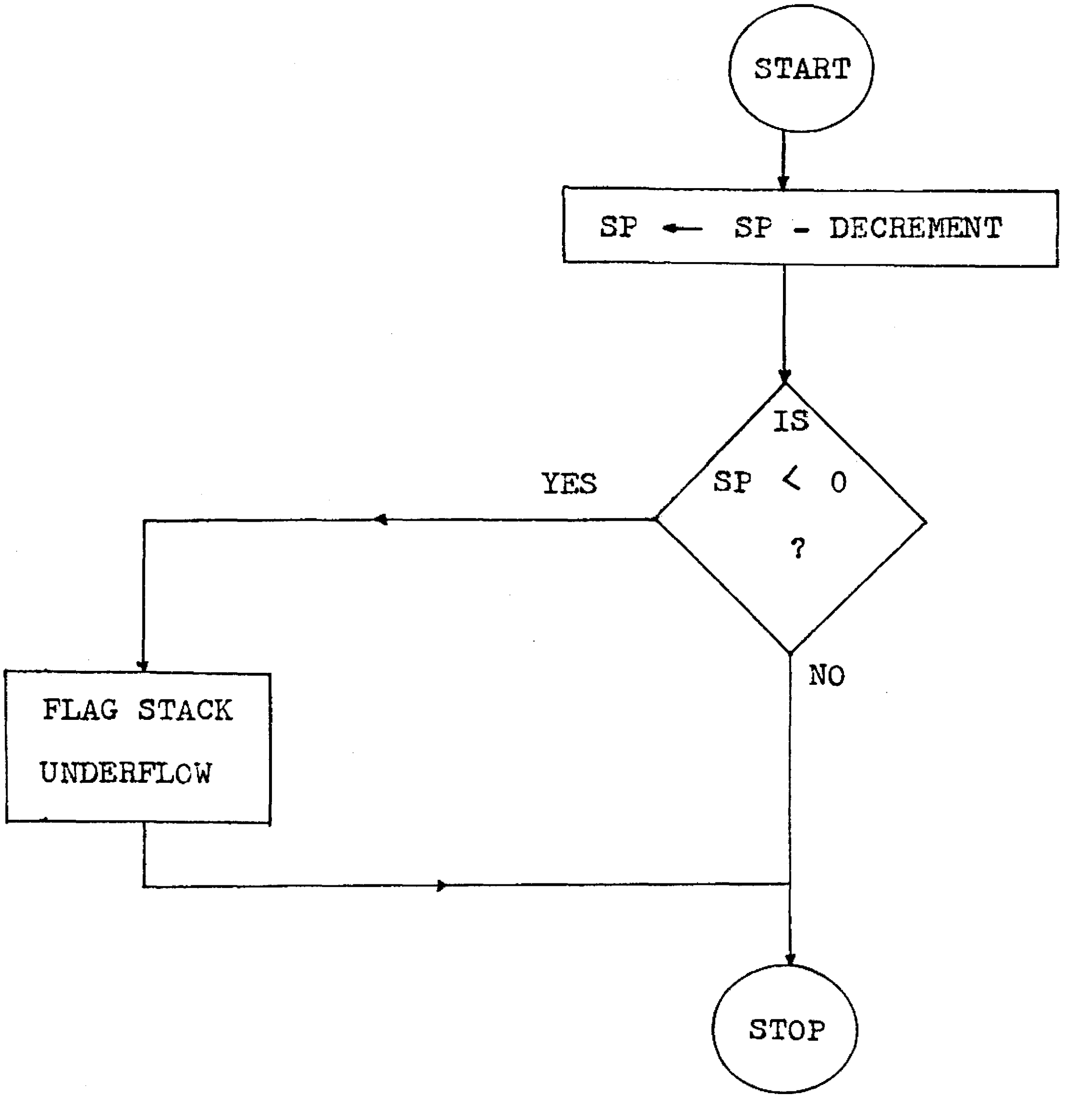


figure 4.2 DECST

4.2.3 GBYTE

This routine extracts a specified byte out of an array of two byte words. It has two arguments, the starting address of the array and the index of the desired byte in the B register. The index specifies a particular byte within an array word. This array word is determined from the index value and the byte is extracted from it. The byte is returned in the A register.

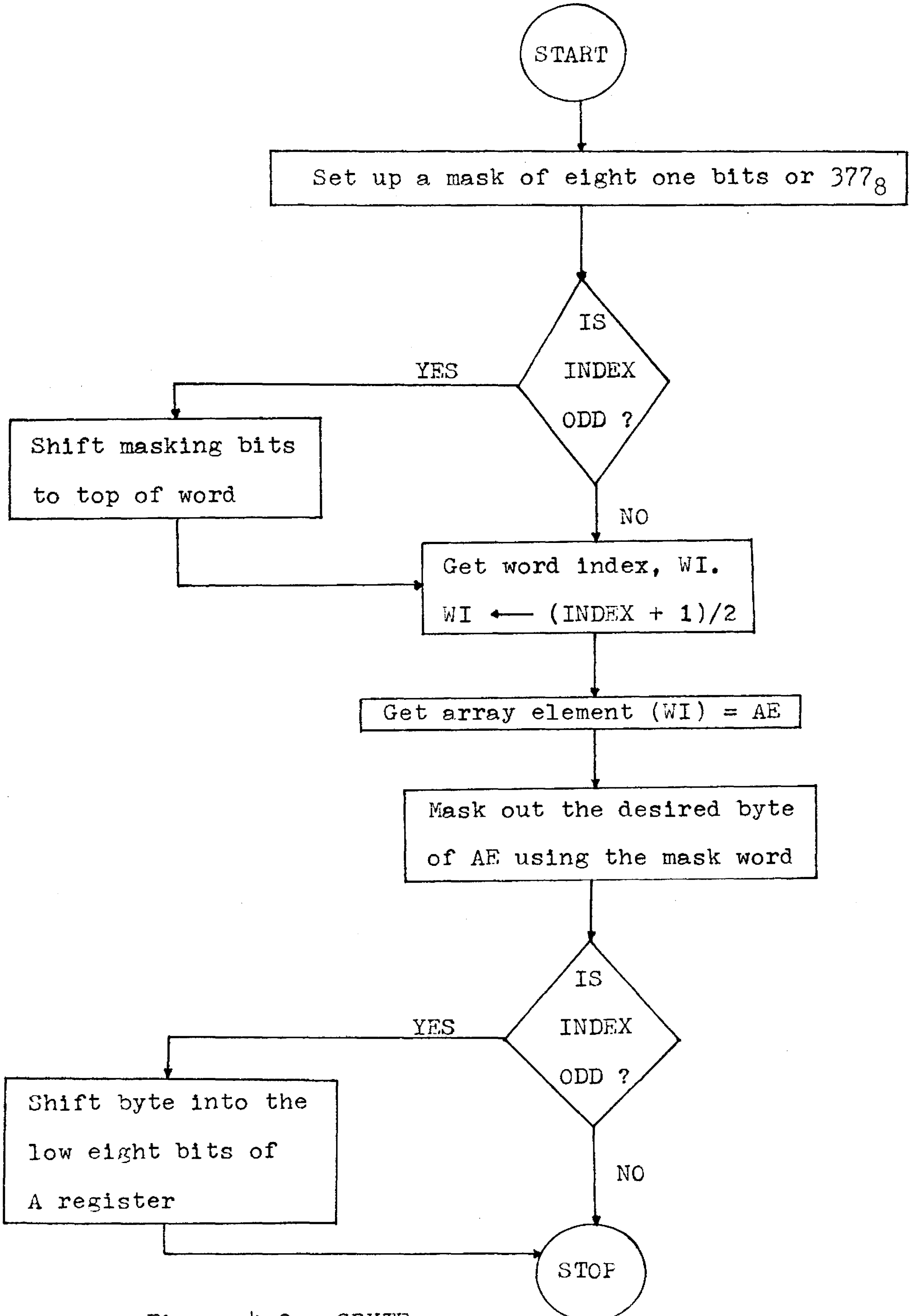


Figure 4.3 GBYTE

CHAPTER 5

INTERPRETER TESTING

The interpreter testing process is intended to verify the instruction set and monitor its performance on complete programs. Two Micro-Pascal test programs included in this chapter illustrate the latter quality. The interpreter instruction set was verified using the test programs documented in Appendix C. In the following section a more comprehensive possible test program is considered.

5.1 Ideal Interpreter Test Program

I would like to outline an approach which could be used to write a comprehensive interpreter test program. First we must determine what such a program will do. It should test almost all of the instruction set. An indication of the instruction being tested should be followed by an error report if an error is detected or the next instruction if the instruction worked as expected. An error message will provide as many details as possible about the machine environment at the time of detection. Documentation of some sort will probably be required to fully interpret the error messages.

The value of our test program lies in the quick debugging and verification possible of a new interpreter. When a malfunctioning instruction is detected, the test program will indicate the particular machine parameters that are in error. These parameters include the stack, instruction pointer and stack elements

A subset of the basic instruction set will have to be verified by other means for use in the program. This subset could be checked, using a debug package and programs like those in Appendix C, and assumed correct for use in the test program. Instructions likely to be members of this set are stack load operations, some logical tests, some transfer operations and keyboard input/output operations.

The design of an instruction test will be as follows.

1) The test conditions are set up by loading values onto the stack.

2) The instruction is executed.

3) The results of the instruction are tested against the expected results. This means we check the values of the stack pointer, instruction pointer and stack contents. If an error is located we,

   a) indicate the source of error by displaying a meaningful symbol (ex. SP for stack pointer and IP for instruction pointer)

   b) display the values of pertinent variables at the time of error to aid in debugging

and c) cause the program to pause so the user can assess the error before deciding to continue error checking or abort.

These three steps are repeated until all of the instructions have been considered.

5.2 Sample Programs

In this segment two test programs are presented to illustrate the execution of Micro-Pascal code. The source listing is followed by the intermediate code output from the compiler and ended by the results of program execution.

Test 1

This program illustrates some of the arithmetic operations available in MICRO-PASCAL. The program input consists of two positive integers separated by a non-numeric character. Each integer is less than 256 since this is the maximum value storable in eight bits and is input from the keyboard. These input values are then used in four arithmetic operations.

```
TEST PROGRAM ONE

THIS PROGRAM ILLUSTRATES THE MATHEMATICAL
OPERATIONS IN MICRO-PASCAL

TWO INTEGER NUMBERS ARE REQUESTED AS INPUT.
THE BASIC ARITHMETIC OPERATIONS OF ADDITION,
SUBTRACTION, MULTIPLICATION AND DIVISION ARE
APPLIED TO THE TWO INPUT VALUES.

.
VAR
  EQUAL : BYTE ;
  DIV : BYTE ;
  MULT : BYTE ;
  MINUS : BYTE ;
  PLUS : BYTE ;
  ZERO : BYTE ;
  BLANK : BYTE ;
  UNIT1 : BYTE ;
  UNIT2 : BYTE ;
  NUM1 : BYTE ;
  NUM2 : BYTE ;
  TEMP : BYTE ;
  NUMBER : BYTE ;
.
PROCEDURE TO ASSEMBLE AN INTEGER VALUE IN NUMBER.
INTEGER DIGITS ARE READ UNTIL A NON-INTEGER
CHARACTER IS DETECTED.
.
PROCEDURE REDNUM ;
VAR
  L : BYTE ;
  M : BYTE ;
BEGIN
  M := 0 ;
  NUMBER := 0 ;
  WHILE M EQ 0 DO
    BEGIN
      READ(UNIT1,L) ;
      L := L - 48 ;
      IF (L GE 0) AND (L LE 9) THEN
        NUMBER := NUMBER * 10 + L ELSE M := 1 ;
    END ;
END ;
.
THIS PROCEDURE WRITES OUT THE INTEGER VALUE THAT
IS STORED IN THE ARGUMENT N ON DEVICE SPECIFIED
BY UNIT2.  (IN THIS CASE THE LINE PRINTER)
```

```
.
PROCEDURE WRTNUM(N : BYTE) ;
VAR
  L : BYTE ;
  M : BYTE ;
BEGIN
  M := N / 10 ;
  IF ( M NE 0 )  THEN  WRTNUM(M) ;
  L := N - (M * 10) + 48 ;
  WRITE(UNIT2,L) ;
END ;
BEGIN
.

ASCII CODES FOR CHARACTERS ARE SET UP FOR
PRINTING PURPOSES.
.
  MULT := 42 ;
  EQUAL := 61 ;
  MINUS := 45 ;
  DIV := 47 ;
  BLANK := 32 ;
  ZERO := 48 ;
  PLUS := 43 ;
.
UNIT1 IS THE KEYBOARD OR TERMINAL AND UNIT2 IS
THE LINE PRINTER.
.
  UNIT1 := 0 ;
  UNIT2 := 1 ;
.

READ IN THE TWO NUMBERS NUM1 AND NUM2.
.
  READLN(UNIT1) ;
  REDNUM ;
  NUM1 := NUMBER ;
  REDNUM ;
  NUM2 : = NUMBER ;
.

ADDITION SEQUENCE
.
  WRITE(UNIT2,BLANK) ;
  WRITELN(UNIT2) ;
  WRTNUM(NUM1) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,PLUS) ;
  WRITE(UNIT2,BLANK) ;
  WRTNUM(NUM2) ;
  WRITE(UNIT2,BLANK) ;
```

```
  WRITE(UNIT2,EQUAL) ;
  WRITE(UNIT2,BLANK) ;
  TEMP := NUM1 + NUM2 ;
  WRTNUM(TEMP) ;
  WRITELN(UNIT2) ;
.
SUBTRACTION SEQUENCE
.
  WRITE(UNIT2,BLANK) ;
  WRITELN(UNIT2) ;
  WRTNUM(NUM1) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,MINUS) ;
  WRITE(UNIT2,BLANK) ;
  WRTNUM(NUM2) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,EQUAL) ;
  WRITE(UNIT2,BLANK) ;
  TEMP := NUM1 - NUM2 ;
  WRTNUM(TEMP) ;
  WRITELN(UNIT2) ;
.
MULTIPLICATION SEQUENCE
.
  WRITE(UNIT2,BLANK) ;
  WRITELN(UNIT2) ;
  WRTNUM(NUM1) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,MULT) ;
  WRITE(UNIT2,BLANK) ;
  WRTNUM(NUM2) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,EQUAL) ;
  WRITE(UNIT2,BLANK) ;
  TEMP := NUM1 * NUM2 ;
  WRTNUM(TEMP) ;
  WRITELN(UNIT2) ;
.
DIVISION SEQUENCE
.
  WRITE(UNIT2,BLANK) ;
  WRITELN(UNIT2) ;
  WRTNUM(NUM1) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,DIV) ;
  WRITE(UNIT2,BLANK) ;
  WRTNUM(NUM2) ;
  WRITE(UNIT2,BLANK) ;
  WRITE(UNIT2,EQUAL) ;
```

```
  WRITE(UNIT2,BLANK) ;
  TEMP := NUM1 / NUM2 ;
  WRINUM(TEMP) ;
  WRITELN(UNIT2) ;
END.
```

INTCODE Listing of Test Program One

```
001 144
040 005 040 001 040 001 040 001 040 001
040 001 040 001 040 001 040 001 040 001
040 001 040 001 040 001 040 001 140 000
000 040 000 040 001 040 001 041 000 021
000 010 041 000 020 000 021 001 000 010
041 000 100 141 000 000 000 000 014 123
021 000 007 001 000 007 041 060 062 021
000 007 001 000 007 041 000 105 001 000
007 041 011 103 063 141 000 000 000 000
021 041 012 063 001 000 007 061 020 000
```

```
001 144
021 140 000 150 041 001 021 000 010 140
000 057 143 040 000 040 001 044 021 000
012 040 004 040 001 040 001 001 000 012
041 012 064 021 000 014 001 000 014 041
000 101 141 000 000 040 007 041 001 041
000 041 014 046 047 012 221 000 161 140
000 242 001 000 012 001 000 014 041 012
063 062 041 060 061 021 000 013 000 000
015 001 000 013 122 143 041 052 020 000
007 041 075 020 000 005 041 055 020 000
```

```
001 144
010 041 057 020 000 006 041 040 020 000
013 041 060 020 000 012 041 053 020 000
011 041 000 020 000 014 041 001 020 000
015 000 000 014 121 221 000 037 000 000
021 020 000 016 221 000 037 000 000 021
020 000 017 000 000 015 000 000 013 122
000 000 015 120 040 007 041 000 041 000
041 016 046 047 012 221 000 161 000 000
015 000 000 013 122 000 000 015 000 000
011 122 000 000 015 000 000 013 122 040
```

```
001 144
007 041 000 041 000 041 017 046 047 012
221 000 161 000 000 015 000 000 013 122
000 000 015 000 000 005 122 000 000 015
000 000 013 122 000 000 016 000 000 017
061 020 000 020 040 007 041 000 041 000
041 020 046 047 012 221 000 161 000 000
015 120 000 000 015 000 000 013 122 000
000 015 120 040 007 041 000 041 000 041
016 046 047 012 221 000 161 000 000 015
000 000 013 122 000 000 015 000 000 010
```

```
001 144
122 000 000 015 000 000 013 122 040 007
041 000 041 000 041 017 046 047 012 221
000 161 000 000 015 000 000 013 122 000
000 015 000 000 005 122 000 000 015 000
000 013 122 000 000 016 000 000 017 062
020 000 020 040 007 041 000 041 000 041
020 046 047 012 221 000 161 000 000 015
120 000 000 015 000 000 013 122 000 000
015 120 040 007 041 000 041 000 041 016
046 047 012 221 000 161 000 000 015 000

001 144
000 013 122 000 000 015 000 000 007 122
000 000 015 000 000 013 122 040 007 041
000 041 000 041 017 046 047 012 221 000
161 000 000 015 000 000 013 122 000 000
015 000 000 005 122 000 000 015 000 000
013 122 000 000 016 000 000 017 063 020
000 020 040 007 041 000 041 000 041 020
046 047 012 221 000 161 000 000 015 120
000 000 015 000 000 013 122 000 000 015
120 040 007 041 000 041 000 041 016 046

001 132
047 012 221 000 161 000 000 015 000 000
013 122 000 000 015 000 000 006 122 000
000 015 000 000 013 122 040 007 041 000
041 000 041 017 046 047 012 221 000 161
000 000 015 000 000 013 122 000 000 015
000 000 005 122 000 000 015 000 000 013
122 000 000 016 000 000 017 064 020 000
020 040 007 041 000 041 000 041 020 046
047 012 221 000 161 000 000 015 120 144

002 030
000 126 000 150 000 146 000 155 000 066
000 160 000 161 040 002 000 217 000 242
000 035 000 272

143 000
```

RESULTS

24 + 12 = 36

24 - 12 = 12

24 * 12 = 288

24 / 12 = 2

Test 2

This program performs an arithmetic sort on ten positive integers. These ten values are less than 256, separated by a non-numeric character and input from the keyboard. The program echo-prints the ten input values and follows with a sorted display of the numbers in ascending order.

```
TEST PROGRAM TWO

THIS PROGRAM SORTS INTEGER VALUES IN ASCENDING
ORDER.
THE USER PROVIDES THE TEN INTEGER VALUES TO BE
SORTED.
.
VAR

  BLANK : BYTE ;
  NOS : ARRAY (1..10) OF BYTE ;
  UNIT1 : BYTE ;
  UNIT2 : BYTE ;
  IJ : BYTE ;
  CH : BYTE ;
.

THIS PROCEDURE READS IN INDIVIDUAL INTEGER
DIGITS UNTIL A NON-INTEGER IS DETECTED.
THE INTEGER NUMBER IS ASSEMBLED IN LOCATION CH.

.
PROCEDURE REDNUM ;
VAR
  L : BYTE ;
  M : BYTE ;
BEGIN
  M := 0 ;
  CH := 0 ;
  WHILE M EQ 0 DO
    BEGIN
      READ(UNIT1,L) ;
      L := L - 48 ;
      IF (L GE 0) AND (L LE 9) THEN CH := CH * 10 + L ELSE
        M := 1 ;
    END ;
END ;
.

THIS PROCEDURE WRITES OUT THE INTEGER VALUE
THAT IS STORED IN LOCATION N.
THE NUMBER IS WRITTEN TO THE DEVICE CORRESPONDING
TO UNIT2.

.
PROCEDURE WRTNUM(N : BYTE) ;
VAR
  L : BYTE ;
  M : BYTE ;
BEGIN
  M := N / 10 ;
  IF (M NE 0) THEN WRTNUM(M) ;
  L := N - (M * 10) + 48 ;
  WRITE(UNIT2,L) ;
END ;
.
```

```
THIS PROCEDURE SORTS THE INTEGERS IN ARRAY NOS
INTO ASCENDING SEQUENCE.
THE CURRENT IMPLEMENTATION SORTS 10 NUMBERS.
.
PROCEDURE SORT ;
  VAR
    K : BYTE ;
    J : BYTE ;
    I : BYTE ;
    TEMP : BYTE ;
  BEGIN
    J:=1;
    WHILE J LT 10 DO
      BEGIN
        I := J ;
        K := I ;
        WHILE K LT 10 DO
          BEGIN
            K := K + 1 ;
            IF NOS(I) GT NOS(K) THEN
              BEGIN
                TEMP := NOS(K) ;
                NOS(K) := NOS(I) ;
                NOS(I) := TEMP ;
              END ;
          END ;
        J := J + 1 ;
      END ;
  END ;
BEGIN
.
UNIT1 CORRESPONDS TO THE KEYBOARD OR TERMINAL
AND UNIT2 CORRESPONDS TO THE LINE PRINTER.
.
  UNIT1 := 0 ;
  UNIT2 := 1 ;
  BLANK:=32;
  READLN(UNIT1) ;
  IJ := 1 ;
.
THIS LOOP READS IN THE TEN INPUT INTEGERS AND STORES
THEM IN THE NOS ARRAY.
.
```

```
   WHILE IJ LE 10 DO
     BEGIN
       REDNUM ;
       NOS(IJ) := CH ;
       IJ := IJ + 1 ;
       WRITE(UNIT2,BLANK);
       WRTNUM(CH) ;
     END ;
   WRITELN(UNIT2) ;
 .
WRITE OUT THE INITIAL CONFIGURATION
OF THE INPUT VALUES.
 .
   WRITE(UNIT2,BLANK) ;
   WRITELN(UNIT2) ;
 .
USE THE SORT PROCEDURE TO ORDER THE NOS
ARRAY ELEMENTS.
 .
   SORT ;
   IJ := 1 ;
 .
WRITE OUT THE SORTED ARRAY.
 .
   WHILE IJ LE 10 DO
     BEGIN
       WRITE(UNIT2,BLANK) ;
       CH := NOS(IJ) ;
       WRTNUM(CH) ;
       IJ := IJ + 1 ;
     END;
WRITELN(UNIT2) ;
END.
```

INTCODE Listing of Test Program Two

```
001 144
040 005 040 001 041 001 041 001 041 001
041 012 040 012 040 001 040 001 040 001
040 001 140 000 000 040 000 040 001 040
001 041 000 021 000 010 041 000 020 000
027 001 000 010 041 000 100 141 000 000
000 000 024 123 021 000 007 001 000 007
041 060 062 021 000 007 001 000 007 041
000 105 001 000 007 041 011 103 063 141
000 000 000 000 027 041 012 063 001 000
007 061 020 000 027 140 000 142 041 001

001 144
021 000 010 140 000 051 143 040 000 040
001 044 021 000 012 040 004 040 001 040
001 001 000 012 041 012 064 021 000 014
001 000 014 041 000 101 141 000 000 040
007 041 001 041 000 041 014 046 047 012
221 000 153 140 000 234 001 000 012 001
000 014 041 012 063 062 041 060 061 021
000 013 000 000 025 001 000 013 122 143
040 000 040 001 040 001 040 001 040 001
041 001 021 000 010 001 000 010 041 012

001 144
102 141 000 000 001 000 010 021 000 011
001 000 011 021 000 007 001 000 007 041
012 102 141 000 000 001 000 007 041 001
061 021 000 007 041 000 001 000 011 124
041 001 124 067 000 000 006 124 070 041
000 041 004 066 041 000 041 006 066 044
041 000 001 000 007 124 041 001 124 067
000 000 006 124 070 041 000 041 004 066
041 000 041 006 066 044 104 141 000 000
041 000 001 000 007 124 041 001 124 067

001 144
000 000 006 124 070 041 000 041 004 066
041 000 041 006 056 044 021 000 012 041
000 001 000 007 124 041 001 124 057 000
000 006 124 070 041 000 041 004 066 041
000 041 006 066 041 000 001 000 011 124
041 001 124 067 000 000 006 124 070 041
000 041 004 066 041 000 041 006 066 044
045 041 000 001 000 011 124 041 001 124
067 000 000 006 124 070 041 000 041 004
066 041 000 041 006 066 001 000 012 045
```

```
001 144
140 001 223 140 000 330 001 000 010 041
001 061 021 000 010 140 000 303 143 041
000 020 000 024 041 001 020 000 025 041
040 020 000 005 000 000 024 121 041 001
020 000 026 000 000 026 041 012 103 141
000 000 221 000 031 041 000 000 000 026
124 041 001 124 067 000 000 006 124 070
041 000 041 004 066 041 000 041 006 066
000 000 027 045 000 000 026 041 001 061
020 000 026 000 000 025 000 000 005 122

001 144
040 007 041 000 041 000 041 027 046 047
012 221 000 153 140 001 273 000 000 025
120 000 000 025 000 000 005 122 000 000
025 120 221 000 264 041 001 020 000 026
000 000 026 041 012 103 141 000 000 000
000 025 000 000 005 122 041 000 000 000
026 124 041 001 124 067 000 000 006 124
070 041 000 041 004 066 041 000 041 006
066 044 020 000 027 040 007 041 000 041
000 041 027 046 047 012 221 000 153 000

001 020
000 026 041 001 061 020 000 026 140 002
034 000 000 025 120 144

002 054
000 120 000 142 000 140 000 147 000 060
000 152 000 153 040 002 000 211 000 234
001 040 001 223 000 337 001 226 000 312
001 242 000 027 001 243 001 302 002 005
002 043 002 143

143 000
```

RESULTS

9 8 7 10 43 52 3 1 734 2

1 2 3 7 8 9 10 43 52 734

CHAPTER 6

SUMMARY

Project work proceded smoothly while the Hewlett-Packard machine was operating properly. Disc problems arose on two occasions delaying progress in the middle stages of the project. My implementation of the interpreter on the HP2100 closely followed the initial writing and debugging of the Micro-Pascal Machine by Mark Green. During this time instructions were understandably added and changed as problems in the compiler were solved. A minor problem arose due to the incomplete documentation of the Micro-Pascal Machine. The exact nature of the input stream was left unspecified in the interpreter description. This oversight was not recognized until the latter stages of the project. Input routines had to be redesigned and implemented so the interpreter would accept programs in the form output by the compiler.

Since the Micro-Pascal Machine is a rather new concept, test programs are currently in short supply. The intermediate code instructions were individually tested but the interpreter performance on larger programs was not extensively probed. A number of compiler errors were uncovered during the debugging of the two test programs found in chapter five.

There are about four hundred words allocated to literal messages and summaries. Many of these messages could be pared down to representative numbers and some descriptive statements concerning the interpreter excised entirely if the interpreter size had to be decreased. The current input capabilities are limited to keyboard, paper tape and card access while output is set up for the keyboard and the line printer. Other features in the input/output package such as disc reads and writes could easily be added when desired.

The size of an HP2100 word is sixteen bits or two bytes. In my stack implementation there are two bytes allocated for each stack item where in the ideal case one byte would do. By doing this I can bend the rules of a byte stack item so that negative values can be represented in a normal HP fashion using sixteen bits the first of which is a sign bit. Positive byte items are able to use all eight bits and no special handling of negative byte values is necessary. This was done only to permit convenient implementation on the HP2100 and might not be appropriate for other machines. In addition my experience with the interpreter performance so far indicates that a stack size of two hundred words would easily handle program executions. This would mean a maximum of one hundred wasted words. In the light of these arguments the implementation of a one byte stack is not yet warranted.

The execution speed could be increased by partially or entirely microcoding other interpreter routines. Instruction group categories and specific instructions within given groups can be easily added. The maximum number of groups and instructions per group are each limited to fifteen.

Conclusion

The interpreter is the key piece of software in the Micro-Pascal compiler-interpreter system. Once the compiler is written in MICRO-PASCAL it can be self-compiled into intermediate code. When an interpreter is established on a host machine the Micro-Pascal system is readily available.

MICRO-PASCAL was designed as a basic high-level language for use on mini and micro processors. Once MICRO-PASCAL is implemented and debugged there are plans to make it concurrent.

MICRO-PASCAL, as currently implemented, has a very limited source language instruction set. Compiler implementation has not reached the stage where strings can be successfully handled. Their complete implementation will contribute greatly to the usefulness of the language. The availability of more and varied control statements would improve language flexibility and ease of use.

The interpreter code is fairly readable using comments liberally to indicate the flow of control. Subroutines invoke a modular design which contributes to an efficient program with minimized redundancy. If errors are detected during interpreter execution the user is provided with valuable information not contained in an error number alone.

Work on this project has certainly given me a better understanding of portable languages. My experience with the STAB system brought to light the importance of good language and program documentation. If programs are to be portable they must be precisely described in order to be understandable and modifiable.

I have also acquired a better understanding of terminal programming, editting and file management. During the disc problem phase of the project I learnt the hard way that files should always be well backed up for security against machine failure and other mishaps. The documentation process was a good disciplinary exercise illustrating an important yet unglamorous aspect of a complete program.

APPENDIX A

Running The Interpreter

This appendix describes how to run the interpreter with microprograms. The commands input by the user are underlined and described where appropriate. The statements in capital letters are system comments.

:JOB,DB04

:PR,DBTST

- begin interpreter execution

BEGIN 'DEBUG' OPERATION

M,10400

- base address for loading of program in core

R

- run the program

MICRO-DEBUG EDITOR

COMMAND? LOAD

- load the microprogram

ENTER FILE NAME DBMP1

- name of file where microprograms are stored

COMMAND? E,0

- begin to execute the program from the beginning

START OF MICRO-PASCAL INTERPRETER

THE UNIT NUMBERS FOR I/O ARE

- these unit numbers refer to i/o during program execution

INPUT 0 - CRT

1 - CARDS

OUTPUT 0 - CRT

1 - PRINTER

OUTPUT CAN BE DIRECTED TO

CRT - ENTER A 1 BELOW OR

PRINTER - ENTER A 2 BELOW OR

AS SPECIFIED - ENTER A 0 BELOW

IN PROGRAM

> 0 or 1 or 2 DBTST SUSP

- the interpreter is now waiting for the user to specify the origin of source program input. The current implementation permits three options:

2 - USER DISC

5 - CARD READER

10 - PAPER TAPE (default option if unit number is not 2 or 5)

:GO, unit number

program execution

summary of program execution

- this includes the number of instructions executed, the maximum stack size and the program size in bytes.

APPENDIX B

Listing of HP Microprograms

```
$DEBUG

*
*-------------------------------------------------------------------
*
*
*    MICROPROGRAMS FOR MICRO-PASCAL INTERPRETER.
*
*    WRITTEN BY DAVE BANDY
*    DATE - AUGUST 1977.
*
*    SUPERVISOR : DR. N. SOLNTSEFF
*
*
*-------------------------------------------------------------------
*

$ORIGIN=1006

                JMP        INCST
                JMP        DECST
                JMP        GBYTE

$ORIGIN=1020
```

```
*
*
*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*    INCST
*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*
*
*    INPUT
*      B REGISTER - AMOUNT OF STACK INCREMENTATION TO STACK
*                   POINTER, SP.
*
*    OUTPUT
*      B REGISTER - 0 IF NO STACK OVERFLOW,
*                 - 1 IF STACK OVERFLOW
*
*
*    THIS ROUTINE INCREMENTS THE STACK POINTER, SP,
*    BY THE VALUE SPECIFIED IN THE B REGISTER. IT THEN CHECKS
*    FOR STACK OVERFLOW BY COMPARING SP WITH STLIM, THE MAXIMUM
*    ALLOWABLE STACK VALUE.  THE VARIABLE MSTCK IS ALWAYS SET TO
*    THE LARGEST STACK POINTER VALUE THAT HAS BEEN ENCOUNTERED.
*
*
INCST      P     IOR  M     RW          GET SP ADDRESS
           P     INC  P                 INC INST POINTER
           T     IOR  Q                 SP ADDRESS IN Q
      Q    RRS   IOR  M     RW          GET SP VALUE
           T     IOR  S2
      B    S2    ADD  S1                SP+INCREMENT IN S1
                 IOR
      Q    RRS   IOR  M     CW    UNC
                 IOR
           S1    IOR  T                 STORE NEW SP VALUE
           P     IOR  M     RW          GET STLIM ADDRESS
           P     INC  P                 IP = IP + 1
           T     IOR  S2
           S2    IOR  M     RW
           T     IOR  A                 STLIM IN A REG.
      A    S1    SUB              NEG   STLIM - SP
                 JMP        INC1        NO OVERFLOW
```

```
*
*    STACK OVERFLOW.
*
              CR    IOR   B     1              OVERFLOW SET B=1
                    JMP         INC2
INC1          P     IOR   M     RW             GET MSTCK ADDRESS
                    IOR   B                    CLEAR B REGISTER
              P     INC   P                    IP = IP + 1
              T     IOR   Q                    MSTCK ADDRESS IN Q
        Q     RRS   IOR   M     RW             GET MSTCK VALUE
              T     IOR   A                    MSTCK IN A REG.
        A     S1    SUB               NEG      MSTACK - SP
                    JMP         INC2           DO NOT RESET MSTCK
        Q     RRS   IOR   M     CW    UNC
                    IOR
              S1    IOR   T                    MSTCK = SP
INC2                IOR               EOP
                    IOR
```

```
*
*
*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*    DECST
*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*
*
*    INPUT
*      B REGISTER - AMOUNT OF STACK DECREMENT
*
*    OUTPUT
*      A REGISTER - 0 - NO STACK UNDERFLOW
*                   1 - STACK UNDERFLOW
*
*
DECST       P    IOR  M    RW         GET SP ADDRESS
            P    INC  P               INC INST. POINTER
            T    IOR  Q               ADDR. SP IN Q
      B    RRS   IOR  S1              STACK DECR. IN S1
                 IOR  A               CLEAR A REGISTER
      Q    RRS   IOR  M    RW         GET VALUE OF SP
           T     IOR  B               SP IN B REGISTER
      B    S1    ADD  B               SP - VALUE(S1)
      Q    RRS   IOR  M    CW    UNC  STORE NEW SP VALUE
                 IOR
      B    RRS   IOR  T          NEG  CHECK FOR UNDERFLO
                 JMP       ENDDC
*
*    STACK UNDERFLOW
*
      A          INC  A               A = 1
                 JMP       ENDDC
ENDDC            IOR             EOP
                 IOR
```

```
*
*
*
*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*   GBYTE
*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
*
*
*   INPUT
*     B REGISTER - INDEX OF DESIRED ARRAY ELEMENT
*
*   OUTPUT
*     A REGISTER - DESIRED BYTE
*
*
*
GBYTE       CR    CLO   A     377         MASK WORD IN A REG
      B           IOR   B           ODD   IS INDEX ODD
                  JMP         NXT         INDEX IS EVEN
            CR    SOV   IR    27          SET OVF,ALF IN IR
      A           IOR   A     SRG2        ALF
      A           IOR   A     SRG2        ALF
NXT   A           IOR   Q
                  IOR   A
      B           INC   B                 INDEX + 1
      B           CRS   B     R1          (INDEX + 1)/2
*
*   LOCATE THE STARTING ADDRESS OF THE ARRAY.
*
            P     IOR   M     RW          GET ARRAY ADDRESS
            P     INC   P                 INC INSTR. POINTER
            T     IOR   S1                ADDRESS GIVES VALU
            S1    IOR   M     RW
            T     IOR   S2
      B     S2    ADD   S3
            S3    IOR   M     RW
            T     IOR   S4
      Q     S4    AND   A           OVF
                  JMP         ENDGT
*
*   MOVE BYTE TO THE LOWER PORTION OF A REGISTER.
*
            CR    IOR   IR    27          ASSEMBLE ALF INST.
      A           IOR   A     SRG2        ALF
      A           IOR   A     SRG2        ALF
ENDGT             IOR               EOP
                  IOR

$END
```

APPENDIX C

This appendix contains test programs that can be used to verify the correct operation of the interpreter instruction set. Each program tests an instruction group or groups. Most of the programs have a step by step description of their operations included. It is assumed these test programs will be used with some type of debugging package so that stack contents and relevant variables can be periodically sampled. This may not be elegant but it is painfully precise.

ARITH

001	051								
041	004	060	041	007	061	041	001	062	041
002	063	041	002	064	041	000	041	003	065
041	000	041	011	066	041	000	041	002	067
041	000	041	006	070	041	000	041	004	071
144									
143	000								

1 -	Load 4 onto the stack and negate it	060
2 -	Load 7 onto the stack and add it to -4 to give 3	061
3 -	Load 1 onto the stack and subtract it from 3 to give 2	062

4 - Load 2 onto the stack and multiply it by 2 to give 4 — 063

5 - Load 2 onto the stack and divide 4 by it to give 2 — 064

6 - Load the two-byte integer 3 and negate it — 065

7 - Add the two-byte integer 11 to -3 to give 6 — 066

8 - Load 2, subtract to give 4 — 067

9 - Load 6, multiply to give 30 — 070

10 - Divide by 4 to give 6 — 071

11 - Stop

BFUNC

001	026								
041	000	123	041	000	041	102	122	041	001
123	041	001	041	104	122	120	121	041	001
123	041	377	124	144					
143	000								

C

E

1 - Load a zero unit number onto the stack, read in a character from the terminal and place it on top of the stack — 123

2 - Place a zero unit number on the stack, load a 102 and write this character to the terminal — 122

3 - Load a 1 and read a character from unit 1 (in this case the card reader) 123

4 - Load 1 and 104 and send 104 or the ASCII character D to the line printer 122

5 - The current output line is terminated and D is printed on the line printer 120

6 - Terminate reading from the current line and reset the read pointers 121

7 - Load 1 and read a character from the card reader to the top of the stack 123

8 - Load 377 onto the stack then move 377 to SP + 1 while a zero byte is inserted at SP 124

9 - Stop 144

LOAD + STORE

001 011

<u>041</u> 004 <u>000</u> 000 001 <u>020</u> 000 000 144

143 000

1 - Load a 4 onto the stack at location one, STACK(1) = 4 041

2 - Load the element stored at STACK(1) onto the stack top, STACK(2) = 4 000

3 - Store the top stack element at STACK(0) 020

4 - Stop 144

LOGIC

001	310								
041	005	041	005	100	041	005	100	041	005
041	005	101	041	005	101	041	003	102	041
004	041	002	102	041	000	102	047	002	041
005	041	005	103	041	005	103	041	000	103
041	005	041	005	104	041	003	104	041	004
041	002	104	041	001	105	041	000	105	041
003	105	047	002	041	000	041	001	041	000
041	001	106	041	000	041	001	041	000	041
002	106	041	000	041	002	041	000	041	001
107	041	111	041	112	041	111	041	112	107
041	000	041	002	041	000	041	004	110	041
000	041	002	041	000	041	001	110	041	000
041	002	041	000	041	002	110	041	000	041
002	041	000	041	004	111	041	000	041	002
041	000	041	001	111	041	000	041	002	041
000	041	004	112	041	000	041	002	041	000
041	001	112	041	000	041	002	041	000	041
002	112	041	000	041	002	041	000	041	004
113	041	000	041	002	041	000	041	001	113
041	000	041	002	041	000	041	002	113	144
143	000								

Testing One-Byte Commodities

1 - Test 5 EQ 5 which results in a 1 (true). 100
and test 1 EQ 5 which results in 0 (false).

2 - Test 5 NE 5 - false and test 0 NE 5 - true. 101

3 - Test 1 LT 3 - true, test 4 LT 2 - false 102
and test 0 LT 0 - false.

4 - Decrement the stack pointer, SP by 3. 047

5 - Test 5 LE 5 - true, test 1 LE 5 - true 103
and test 1 LE 0 - false.

6 - Test 5 GT 5 - false, test 0 GT 3 - false 104
and test 4 GT 2 - true.

7 - Test 0 GE 1 - false, test 0 GE 0 - true 105
and test 1 GE 0 - true.

8 - Decrement SP by 2. 047

Testing Two-Byte Commodities

9 - Test 1 EQ 1 - true and test 1 EQ 2 - true. 106

10 - Test 2 NE 1 - true and test 107
44512 NE 44512 - false.

11 - Test 2 LT 4 - true, test 2 LT 1 - false 110
and test 2 LT 2 - false.

12 - Test 2 LE 4 - true, test 2 LE 1 - false 111
and test 2 LE 2 - true.

13 - Test 2 GT 4 - false, test 2 GT 1 - true 112
and test 2 GT 2 - false.

14 - Test 2 GE 4 - false, test 2 GE 1 - true 113
and test 2 GE 2 - true.

MANIP

001	052								
040	004	041	377	042	004	004	101	102	103
104	043	003	000	000	006	044	043	004	000
000	005	077	045	043	003	001	000	005	046
047	003	043	006	000	000	002	000	000	005
050	051	144							
143	000								

1 - Increment the stack pointer by 4 to 4. 040

2 - Load a 377 onto the stack at SP = 5. 041

3 - Store the character string ABCD onto the stack, that is 4, 4, 101, 102, 103 and 104. 042

4 - Load 3 bytes onto the stack 0, 0 and 6. 043

5 - Load the byte at SP = 6 onto the stack, 4. 044

6 - Load 4 bytes onto the stack 0, 0, 5 and 77. Store 77 at stack location 5. 045

7 - Load 3 bytes 1, 0 and 5. Convert this three-byte relative stack address to a two-byte absolute stack address preceded by a zero byte. 046

8 - Decrement the stack pointer by 2. 047

9 - Load 0, 0, 2, 0, 0 and 5. Load the contents of stack element 5 at SP - 2 and stack element 6 at SP - 1. 050

10 - Store 4 at stack location 3 and 77 at stack location 2. 051

11 - Stop 144

<u>PCALL</u>

001 015

041 001 041 002 <u>220</u> 000 012 <u>041</u> 377 144

<u>041</u> 004 <u>143</u>

143 000

1 - Load 1 and 2 onto the stack. Call a procedure and thereby create a seven-byte header. 220

Byte 1	Old LVL = 0
Bytes 2 + 3	Return Address = 6
Bytes 4 + 5	Add. of Ftn Result = 0
Bytes 6 + 7	DSPLY For Old LVL = 0

Also set DSPLY(1) = 2, set LVL to 1 and instruction pointer IP = 11 (which will be immediately incremented to 12)

2 - Load 4 onto the stack to show we have reached the right spot. 041

3 - End the procedure which 143

Restores LVL to DSPLY(LVL - 1) = 0

Resets IP to 6

Resets the stack pointer, SP to 2.

4 - Load 377 onto the stack to show the procedure has concluded correctly. 041

5 - Stop 144

TRANS

001	122								
140	000	003	041	000	141	000	012	041	002
041	001	141	000	021	041	002	041	001	041
003	142	001	000	036	041	061	140	000	061
002	000	046	041	062	140	000	061	003	000
056	041	063	140	000	061	377	000	000	041
004	142	001	000	074	041	061	140	000	117
002	000	104	041	062	140	000	117	003	000
114	041	063	140	000	117	377	000	000	041
077	144								
143	000								

1 - Pass control to instruction 3 in that IP is set to 2 before incrementation. 140

2 - Load a zero (false) onto the stack. Execute a conditional jump on the top stack element. Since this element is false control jumps to IP = 12 so that 041 002 will not be executed. 141

3 - Load a one (true) onto the stack. In this case the conditional test fails so that a 2 should be loaded onto the stack. 141

4 - Load 1 and 3 onto the stack. Perform a case statement based on the top stack element 3. There are 3 case elements 1, 2 and 3 so there should be a match with the third element. This will cause a load of 63 to occur. 142

5 - Load 4 onto the stack. Perform the same case statement on 4. There should be no match with the case elements 1, 2 or 3. The next instruction to be executed should be a load of 077 onto the stack. 142

6 - Stop 144

Note - 143 is a procedure return and is tested with the procedure call in PCALL.

Test Of Error Messages

1 - Illegal Level Number

001 003

117 000 144

143 000

2 - Illegal Group Number

001 003

160 000 144

143 000

3 - Code Array Overflow

Four sets of the following 001 group if the code size = 400.

001 144

{041 001 041 001 041 001 041 001 041 001
{Nines lines like the above.

001 002

041 001

143 000

4 - Attempt to access a non-existent DSPLY element i.e. element 16

001 010

041 021 041 000 041 001 044 144

143 000

5 - Stack Overflow - tried to set SP←128_{10}

001 005

041 001 040 200 144

143 000

6 - Stack Underflow - tried to set SP ← -8

001 005

040 001 047 011 144

143 000

7 - Illegal Input Table Type

003 000

BIBLIOGRAPHY

Boon, C. High Level Languages. Berkshire, England: Infotech Info Ltd, 1972.

Goos, G., and Hartmanis. Lecture Notes In Computer Science, Software Engineering - An Advanced Course. New York: Springer - Verlag, 1973.

Gries, David. Compiler Construction For Digital Computers. Toronto: John Wiley and Sons, 1971.

Hewlett-Packard. Assembler Reference Manual. Cupertino, California: Hewlett-Packard Company, 1972.

Hewlett-Packard. Microprogramming Guide For HP2100. Cupertino, California: Hewlett-Packard Company, 1972.

Hewlett-Packard. Microprogramming Software For HP2100. Cupertino, California: Hewlett-Packard Company, 1973.

Solntseff, Dr. N.. A Pascal 6000 Primer. Hamilton: Applied Mathematics Department, 1975.

Wirth, Niklaus. Algorithms and Data Structures = Programs. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1976.

REFERENCES

GRE, Private Communication with Mark Green, 1977.