"A software emulator of the UCSD P-machine was implemented for the P-857 minicomputer . . . and came up with correct results."

A UCSD Pascal P-Machine Implementation on a Minicomputer

by

Amante A. Mangaser*

INTRODUCTION

Pascal has become a widely accepted general purpose programming language since its introduction more than a decade ago. Because of its growing popularity, various implementations and extensions of Pascal exist in diverse computer types.

UCSD Pascal is a specific version of Pascal which can be easily ported to different computers. Its portability can be attributed to the virtual P-machine concept on which it is based. Using this concept, UCSD Pascal is ported to a new computer by writing a software equivalent of the virtual P-machine on the new computer.

This paper describes an implementation of the P-machine on the Philips P-857 minicomputer. It consists of five parts. The first part gives a short description of the UCSD Pascal version and an explanation of this version's portability. It also outlines the objectives and scope of this specific implementation. The second part deals with the UCSD Pascal P-machine – its architecture, data types, instruction set, and operation. The methodology used in the realization of the P-machine is discussed in the third part. Next, a description of the P-857 software that emulates the P-machine is given. Finally, the fifth part presents the conclusions with regard to this P-machine implementation.

THE UCSD PASCAL SYSTEM

UCSD Pascal is a particular implementation of Pascal specially suited to minicomputers and microcomputers. It is a complete Pascal programming environment for small computers which provides a compiler, a filer, and an editor, among other system programs. The system is written almost entirely in the Pascal programming language, and extended to provide for systems programming and for disk-based interactive applications. The UCSD Pascal system can be run on a typical hardware system that has 56 K-bytes of main memory, 8-bit or 16-bit word organization, standard floppy disk for secondary storage, ASCII keyboard, console display, and optional printer.

Why is the UCSD Pascal System Portable?

The UCSD Pascal system uses a P(ortable)–compiler to process Pascal programs. This compiler accepts a Pascal source program as input and emits intermediate code as output. The intermediate code is defined in terms of operations on

*Professor, Department of Electrical Engineering, U.P. College of Engineering.

a hypothetical stack machine. This virtual or pseudo-machine is often called a P-machine and its code is appropriately termed as P-code. With this approach, the P-compiler is not bound to any real computer. In contrast, a conventional compiler takes in a Pascal source language program and translates it into an equivalent machine or assembly language program of a particular computer. In this implementation, the compiler is inherently associated with the computer, i.e., it is machine-specific. The compiler produces object code which can be run only in identical computers. Thus, to implement Pascal in another type of computer, it is necessary to rewrite the compiler so that it produces the object program in the new computer's machine/assembly language.

While the machine language output of a conventional compiler can be subsequently loaded and run in a computer, the P-code output of the P-compiler still has to be processed further. Figure 1 shows the difference between the approaches used by a conventional compiler and the P-compiler. In essence, the UCSD system's compilation scheme is a two-part process. The first part employs the P-compiler to translate a Pascal source program into its P-code equivalent. The second part uses a P-machine emulator, also called an interpreter, to read and directly execute the P-code. (The interpreter is a program, usually written in the assembly language of a host computer, that emulates the abstract P-machine at run-time.) Thus, the conventional compiler uses a compile-run approach while the UCSD Pascal system uses a compile-interpret scheme.

The advantage of the compile-run type over that of the compile-interpret type is that of speed. A machine language program generated by the compile-run technique runs faster than an equivalent P-code program that is executed interpretively. In cases of big source programs, however, the size of the machine program becomes quite large compared to the P-code program. The reason is that the P-code is optimized for memory space. As such, a program in P-code is more compact than a corresponding assembly language program. Further, the compile-interpret implementation, which uses a two-part process, offers several other premiums. A decided advantage is portability. Because the P-compiler produces an intermediate code, the compiler can be written without regard to a particular machine. In this sense, the first part is machine-independent: a characteristic that enhances the portability of the compiler. So, to implement the high-level language on a machine, it is only necessary to write the second stage – the interpreter which directly executes the P-code. Another advantage is the convenience of being able to write the interpreter independently of the P-compiler.

Objective

The general objective of this endeavor was to implement the Pascal language on the Philips P-857 minicomputer. Although the UCSD Pascal system was already implemented on a number of minicomputers and microcomputers, it was then not yet available for the P-857. This fact did not pose insurmountable difficulties because most of the UCSD Pascal system was written in Pascal. It was only necessary to have an interpreter written in the assembly language of the P-857 which could execute the UCSD P-code. Thus, the particular purpose of this work was to come up with a UCSD P-code interpreter – a software emulator of the virtual P-machine – using the P-857 as the host computer.

Before presenting the methodology used in realizing the emulator, it is worthwhile to first discuss the P-machine.

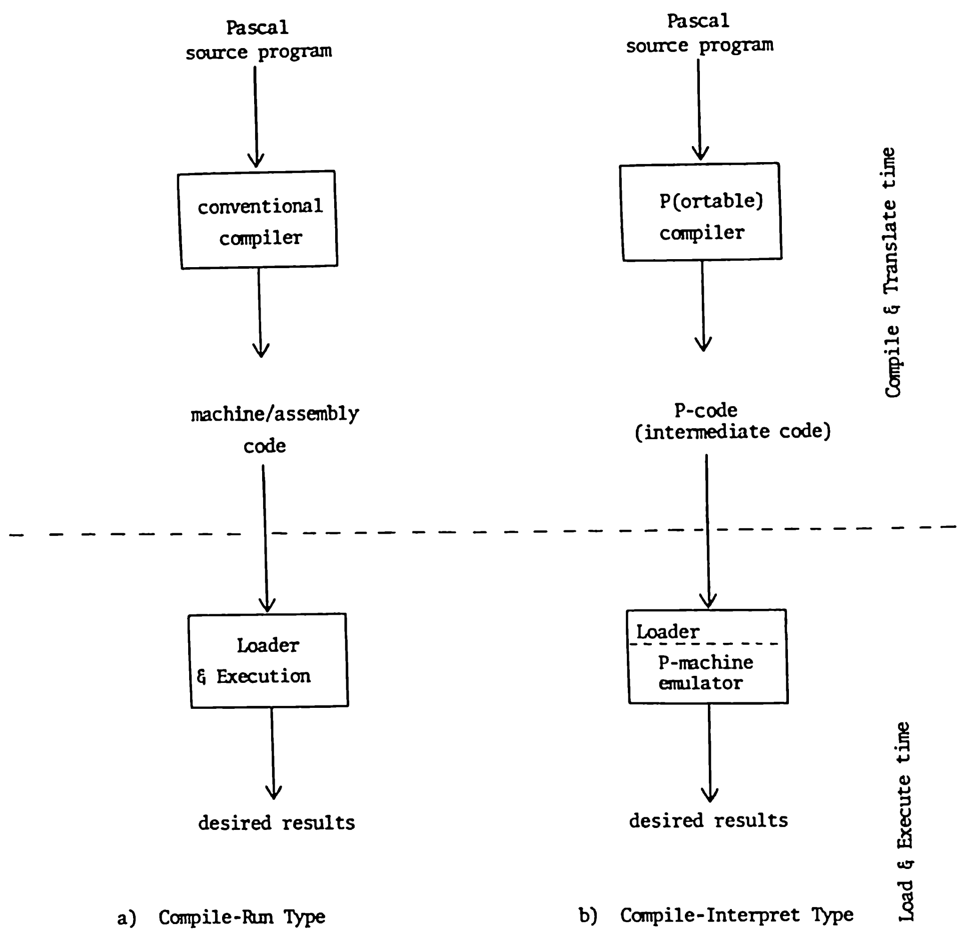


Figure 1. Two types of Compilation and Execution of Pascal Programs.

THE UCSD P-MACHINE

Architecture

The UCSD P-machine is a hypothetical machine designed for the Pascal programming language. It employs either an 8-bit or 16-bit word organization to suit a specific computer. To support Pascal's block structure and recursion capabilities, it takes on the basic form of a zero-address (stack) machine as shown in Figure 2. The P-machine uses a stack as a storage which contains not only the local data of many P-codes, but also code segments and procedure activation records. The stack is thus not limited to contain data alone but also P-codes (instructions) and information about procedure calls and returns. It also serves the purpose of passing parameters to procedures and returning function values. The stack is filled by loads and procedure calls; and decreased by stores, procedure returns, and execution of arithmetic instructions. It starts in high memory and grows toward low memory.

The P-machine also has a heap for dynamic variables. The heap starts from low memory and grows upward toward the stack. It is increased by calling the procedure "new" and decreased by calling the procedure "release." The heap is actually a stack itself.

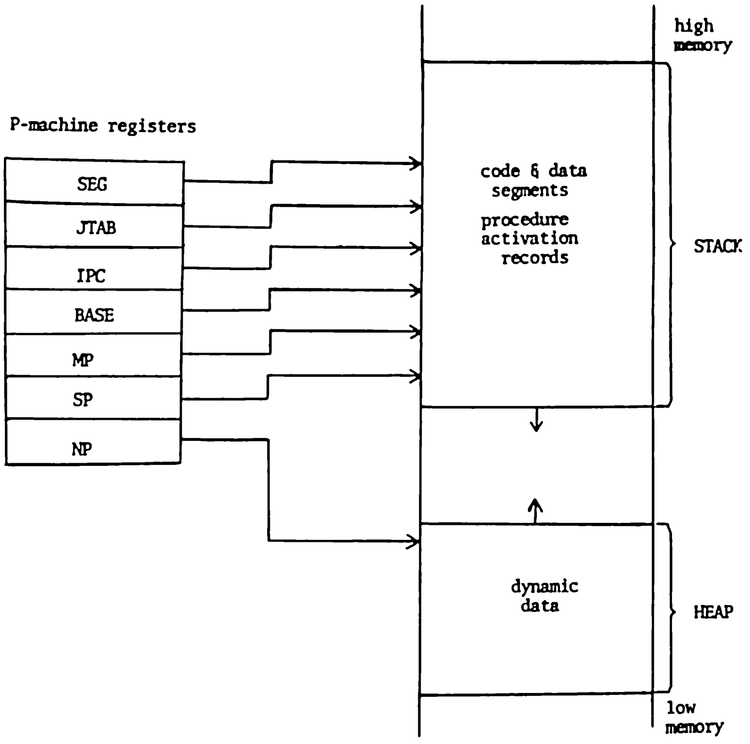


Figure 2. UCSD P-machine's Registers and Storage.
The P-machine is a Zero-Address (Stack) Machine.

The P-machine uses several registers as pointers to relevant areas in the stack and the heap as follows:

IPC	– the interpreter program counter, points to the next P-code instruction to be executed.
SP	– stack pointer, points to the top of the stack.
NP	– new pointer, points to the top of the dynamic heap.
JTAB	– jump table pointer, points to the procedure attribute table of the currently executing procedure.
SEG	– segment pointer, points to the procedure dictionary of the segment to which the currently executing procedure belongs.
MP	– most recent procedure pointer, points to the activation record (Mark Stack Control Word) of the currently executing procedure. Local variables are accessed by indexing off MP.
BASE	– base procedure pointer, points to the activation record of the base (lexical level 0) procedure. Global variables are accessed by indexing off BASE.

Instruction Set

The P-machine instructions are mostly zero address instructions that are designed to optimize memory space. As such, they are one or two bytes long followed by zero to four parameters. The instructions are actually divided into five main groups:

1) variable fetching, indexing, storing, and transferring
2) top-of-stack arithmetic and comparisons

3) jumps
4) procedure/function calls and returns
5) system program support procedures

The parameters of the instructions are of the following types and most of them specify one word of information:

UB	–	unsigned byte: high order byte is implicitly zero.
SB	–	signed byte: high order byte is the sign extension of bit 7.
DB	–	don't care byte: can be treated as UB or SB because its value is always in the range 0 . . . 127. (Note that the two periods between the numbers given above are read as "to." This is consistent with the notation for subranges in Pascal.)
B	–	big: this parameter can be one or two bytes long. It is one byte representing values in the range 0 . . . 127, two bytes for values in the range 128 . . . 32767. If bit 7 of the first byte is not set, then, it represents 0 . . . 127; otherwise, bit 7 of the byte is cleared and the byte is taken as the most significant byte. The following byte is the low order byte.
W	–	word: next two bytes, low byte first.

Figure 3 shows some typical P-machine instructions. Further, a complete list of instruction mnemonics is shown in Table 1. In this table the three numbers before a mnemonic are the equivalent P-codes in decimal, octal and hexadecimal,

```
Variable Fetching, Storing, Indexing and Transferring

      SLDC53                  Short load word constant
      STM    UB               Store multiple words
      IXA    B                Index array

Top of Stack Arithmetic and Comparisons

      LNOT                    Logical Not
      MPI                     Multiply Integers
      TNC                     Truncate Real
      EQUI                    Integer Equal Comparison

Jumps Instructions

      UJP                     Unconditional Jump
      XJP    W1,W2,W3
             <case table>     Case jump

Procedure/Function Calls and Returns

      CLP    UB               Call Local Procedure
      RBP    DB               Return from Base Procedure

System Program Support Procedures

      TRS                     Treesearch
      XIT                     Exit
      TIM                     Time
```

Figure 3. Typical P-machine Instructions.

Table 1. P-machine Instruction Mnemonics. (P-code)

0	000	00	SLDC	0										
1	001	01	SLDC	1										
	.													
	.													
	.													
126	176	7E	SLDC	126										
127	177	7F	SLCD	127										
128	200	80	ABI		171	253	AB	SRO		214	326	D6	XIT	
129	201	81	ABR		172	254	AC	XJP		215	327	D7	NOP	
130	202	82	ADI		173	255	AD	RNP		216	330	D8	SLDL	1
131	203	83	ADR		174	256	AE	CIP		217	331	D9	SLDL	2
132	204	84	AND		175	257	AF	EQU		218	332	DA	SLDL	3
133	205	85	DIF		176	260	B0	GEQ		219	333	DB	SLDL	4
134	206	86	DVI		177	261	B1	GTR		220	334	DC	SLDL	5
135	207	87	DVR		178	262	B2	LDA		221	335	DD	SLDL	6
136	210	88	CHK		179	263	B3	LDC		222	336	DE	SLDL	7
137	211	89	FLO		180	264	B4	LEQ		223	337	DF	SLDL	8
138	212	8A	FLT		181	265	B5	LES		224	340	E0	SLDL	9
139	213	8B	INN		182	266	B6	LOD		225	341	E1	SLDL	10
140	214	8C	INT		183	267	B7	NEQ		226	342	E2	SLDL	11
141	215	8D	IOR		184	270	B8	STR		227	343	E3	SLDL	12
142	216	8E	MOD		185	271	B9	UJP		228	344	E4	SLDL	13
143	217	8F	MPI		186	272	BA	LDP		229	345	E5	SLDL	14
144	220	90	MPR		187	273	BB	STP		230	346	E6	SLDL	15
145	221	91	NGI		188	274	BC	LDM		231	347	E7	SLDL	16
146	222	92	NGR		189	275	BD	STM		232	350	E8	SLDO	1
147	223	93	NOT		190	276	BE	LDB		233	351	E9	SLDO	2
148	224	94	SRS		191	277	BF	STB		234	352	EA	SLDO	3
149	225	95	SBI		192	300	C0	IXP		235	353	EB	SLDO	4
150	226	96	SBR		193	301	C1	RBP		236	354	EC	SLDO	5
151	227	97	SGS		194	302	C2	CBP		237	355	ED	SLDO	6
152	230	98	SQI		195	303	C3	EQUI		238	356	EE	SLDO	7
153	231	99	SQR		196	304	C4	GEQI		239	357	EF	SLDO	8
154	232	9A	STO		197	305	C5	GTRI		240	360	F0	SLDO	9
155	233	9B	IXS		198	306	C6	LLA		241	361	F1	SLDO	10
156	234	9C	UNI		199	307	C7	LDCI		242	362	F2	SLDO	11
157	235	9D	S2P		200	310	C8	LEQI		243	363	F3	SLDO	12
158	236	9E	CSP		201	311	C9	LESI		244	364	F4	SLDO	13
159	237	9F	LDCN		202	312	CA	LDL		245	365	F5	SLDO	14
160	240	A0	ADJ		203	313	CB	NEQI		246	366	F6	SLDO	15
161	240	A1	FJP		204	314	CC	STL		247	367	F7	SLDO	16
162	242	A2	INC		205	315	CD	CXP		248	370	F8	SIND	0
163	243	A3	IND		206	316	CE	CLP		249	371	F9	SIND	1
164	244	A4	IXA		207	317	CF	CGP		250	372	FA	SIND	2
165	245	A5	LAO		208	320	D0	SIP		251	373	FB	SIND	3
166	246	A6	LCA		209	321	D1	IXB		252	374	FC	SIND	4
167	247	A7	LDO		210	322	D2	BYT		253	375	FD	SIND	5
168	250	A8	MOV		211	323	D3	EFJ		254	376	FE	SIND	6
169	251	A9	MVB		212	324	D4	NFJ		255	377	FF	SIND	7
170	252	AA	SAS		213	325	D5	BPT						

decimal hexadecimal
octal

respectively. As mentioned above, these instructions may be followed by parameters.

Data Types

The P-machine has several data types. All of these data types are represented in the stack so that they are always aligned on word boundaries. The least significant bit is bit 0 and the most significant bit is bit 15. The data types and their descriptions are:

BOOLEAN	One word. The least significant bit indicates the value (0 for false, 1 for true) and is the only bit used by boolean comparisons. Other boolean operators like AND, IOR, and NOT use all 16 bits.

INTEGER	:	One word, two's complement representation having values in the range – 32768 . . . 32767.
SCALAR	:	User defined. One word, in the range 0 . . . 32767.
CHAR	:	One word, with the least significant byte containing the character. Values ranging from 0 . . . 127 represent the standard ASCII set, while values in the range 128 . . . 255 represent a user-defined character set.
REAL		Two words, format implementation dependent. The interpreter is the only one that has to know the detailed internal format of REALs.
POINTER		One or three words, depending on pointer type. Pascal pointer, internal word pointer: one word, containing a word address. Internal byte pointer: one word, containing a byte address. Internal Packed Field Pointer: three words – word 2: word pointer to where word field is in word 1: field width in bits word 0: right bit number of field.
SET		0 to 255 words in data segment, 1 to 256 words on stack. Sets are implemented as bit vectors. When a set is in a data segment, all allocated words contain valid information. When it is on a stack, the first word contains the length followed by that number of words all containing valid information.
RECORDs and ARRAYs	:	Any number of words (up to 16384 words in one dimension). Arrays are stored in row-major order, and always have a lower index of zero.
STRINGs	:	1 . . . 128 words. Strings are flexible versions of packed array of char. A string of length (n) contains (n div. 2) + 1 words. Byte 0 contains length of string, and bytes 1. . . n contain valid characters.
CONSTANTs	:	The following have special formats because they can be embedded in the instruction stream: All scalars (excluding reals) not in the range 0 . . . 127: two bytes, low byte first. Strings: All string literals take length + 1 bytes, and are byte-aligned. The first byte gives the length, the rest are actual characters. Reals and sets: word-aligned, in reverse word order.

Figure 4 presents the more common data types in diagram form.

The P-machine's Operation

To clarify the use of the P-machine's registers and storage, consider the illustrative program shown in Figure 5. It is a typical Pascal program which when compiled to an equivalent P-code program can be run in the P-machine. The program shows the block structure of Pascal. Although the figure does not explicitly show recursive procedure calls, they are available in Pascal. To support these two Pascal properties, the P-machine adopts a dynamic storage allocation scheme that uses a

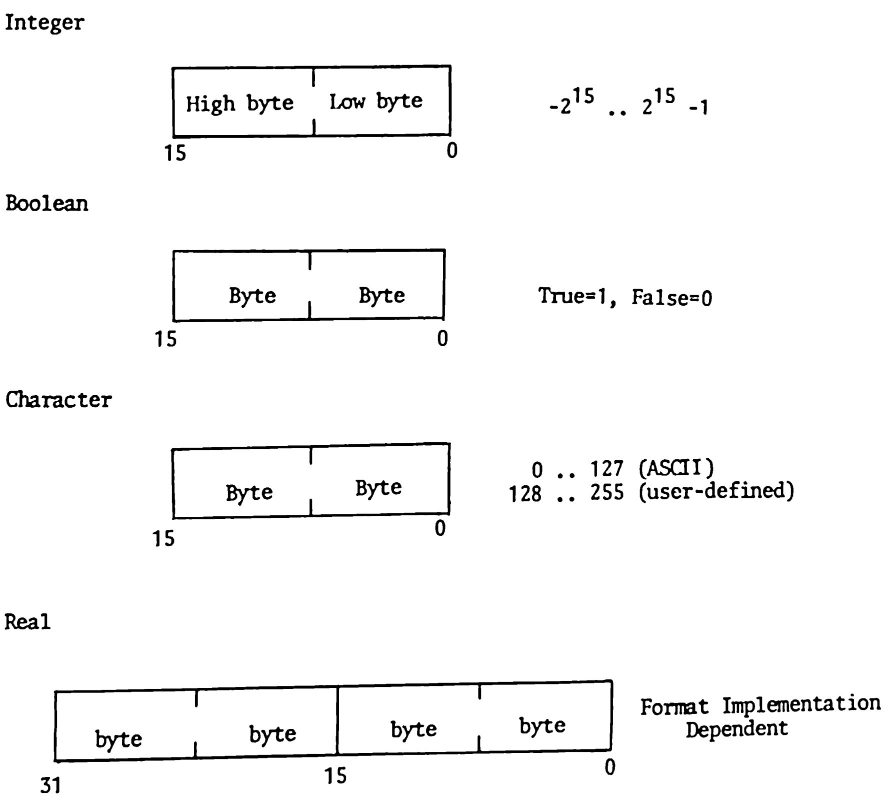


Figure 4. Some Common P-machine Data Types.

stack. With this scheme, entry to a procedure entails allocation of memory space for the procedure's local variables at the top of the stack, and corresponding exit from the procedure frees the allocated space. It is noteworthy to point out that the stack contains ail the variables (data segment) and instructions (code segment) of the currently active procedures as well as information about procedure calls and returns (mark stack).

Taking the illustrative program as an example, the memory mapping of the P-machine's stack during procedure B's execution is given in Figure 6. This figure also shows the interpreter, the heap, and the relative positions of the P-machine's register-pointers. Figure 7 shows a more detailed view of the code segment for the user program. This segment contains the instruction codes of the main program and its three procedures. At the time of B's execution, SEG points to the start of the user program's segment code; JTAB to the start of the procedure B's code section; and IPC to the P-code in B which is currently being interpreted. This is further depicted in Figure 8 which is a magnified view of B's P-code section. Other relevant information about procedure B is also included in its code section.

As a result of the P-machine's dynamic storage allocation scheme, effective addresses cannot be determined at compile time. Hence, they are assigned at run time. Thus, the P-compiler assigns only displacements from a datum within the data segment. This datum is within the Mark Stack Control Word (MSCW) of a given procedure for the P-machine. Figure 6 shows the MSCW (mark stack) and data segment of procedure B during its execution. A more detailed view of this is given in Figure 9. The datum of the currently executing procedure, B in this case

```
Program  PASCALSYSTEM;
    Var  SYSCOM:SYSCOMREC;
         CH:Char;

    Segment Procedure USERPROGRAM;

        Procedure MAINPROGRAM;
            Var X,Y,Z:Integer;

            Procedure A;
                Var I,J,K:Integer;

                Procedure C

                Begin
                .
                .
                .
                End; (*C*)

            Begin
            .
            .
            .
            End; (*A*)

            Procedure B;
                Var T,F:Boolean;

            Begin
            .
            .
            .
            End; (*B*)

        Begin
            A;B
        End; (*MAINPROGRAM*)

    Begin
        MAINPROGRAM
    END; (*USERPROGRAM*)

    Segment Procedure COMPILER;

    Begin
    .
    .
    .
    End; (*COMPILER*)
```

Figure 5.a. Illustrative Program.

is pointed to by the MP register of the P-machine. Local variables are just indexed off MP. Global variables, on the other hand, are based on a datum in the MSCW of the main program (Figure 6) which is pointed to by BASE. Global variables are indexed off BASE. If the variable is from an encompassing procedure, it is obtained by searching down the data segments through a static chain. The static chain contains information that reflects the address environment or the static hierarchichal structure of the procedures. Each of the currently-active procedures has a static link (MSSTAT in Figure 9) in its data segment. The static link of a procedure points to the datum of the data segment belonging to the immediately enclosing procedure. Thus, the P-compiler generates an address couple – the static level difference and relative displacement from the datum – for all variables, with the exception of local and global variables.

Procedures in Pascal can call one another, in fact, even recursively. It is therefore necessary to keep track of the procedure activations so that when a called pro-

```
            Segment Procedure EDITOR;

            Begin
            .
            .
            .
            End; (*EDITOR*)



      Begin
          Repeat    Read (CH);

                    Case CH of
                         C:COMPILER;
                         E:EDITOR;
                         .
                         .
                         .
                         U:USERPROGRAM
                    End (*Case*)

          Until CH = 'H'
      End. (*PASCALSYSTEM*)
```

Figure 5.b. Illustrative Program.

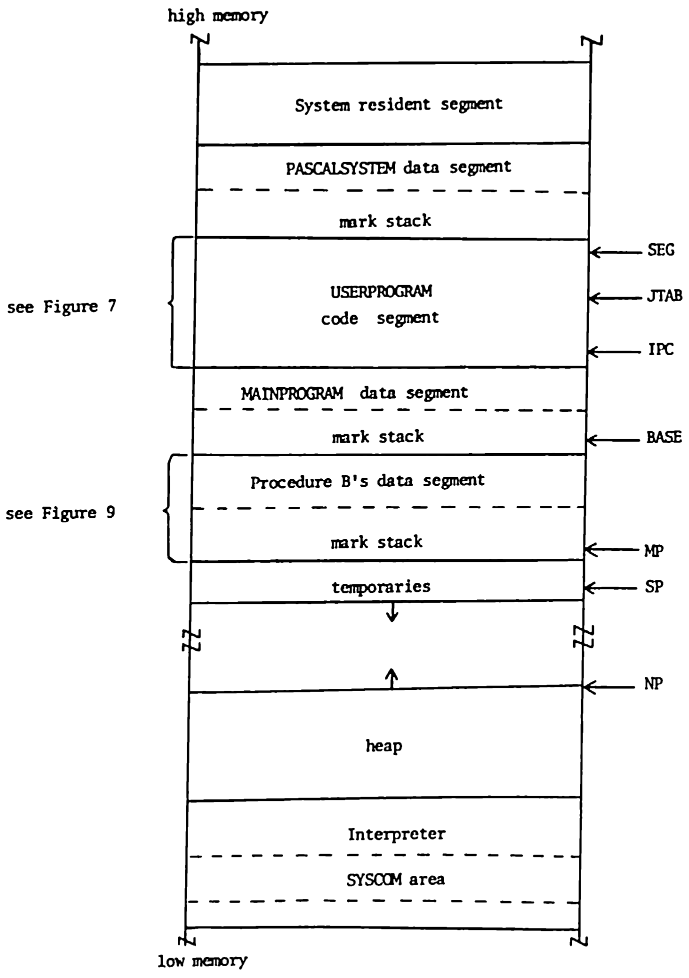


Figure 6. Memory Map During Procedure B's Execution.

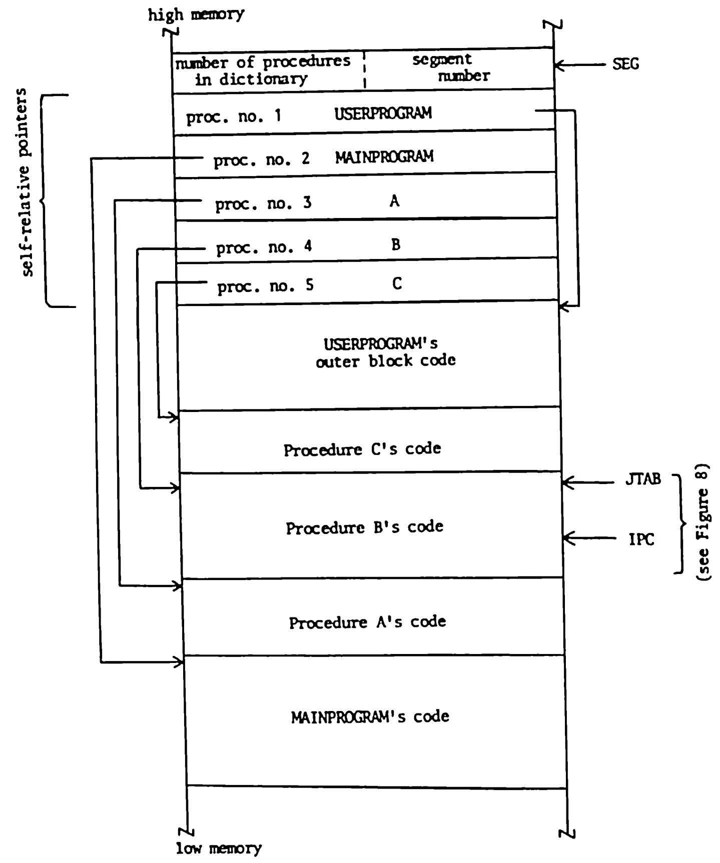


Figure 7. Code Segment of USERPROGRAM. P-machine pointers SEG, JTAB & IPC Shown During B's Execution.

cedure finishes, it has the correct return address of the procedure which called it. It is also necessary to restore the P-machine's registers' contents to their values before the call was made and executed. Therefore, to ensure correct procedure linkage, additional storage is allocated on the stack for each called procedure. The MSCW is used for this purpose. It contains the static link and other information for procedure linkage. (See Figure 9.) MSSP contains the stack pointer's value of the calling procedure at the time of the call. MSJTAB contains JTAB pointer of the calling procedure. MSIPC contains the interpreter's program counter at the time of the call, which is in essence, the program return address. MSDYN contains the dynamic link which is a pointer to the datum of the calling procedure's data segment. The dynamic links also form a chain called the dynamic chain. This chain reflects the procedure activation history. Thus, the dynamic link of a given data segment points to the datum of the procedure's data segment which called it.

STRATEGY IN REALIZING THE P-MACHINE

The implementation of the P-machine on the Philips P-857 minicomputer was planned to be done in three phases. Of the three phases, two were completed but the third was dropped due to lack of time. The result of the first phase was a prototype interpreter (P-machine emulator) written in Pascal. The prototype was made to serve as a model for the implementation of the P-machine on the P-857 minicomputer. So, after it was tested and debugged the next phase was started. In the second phase, the Pascal version of the interpreter was manually translated into the P-857 assembly language. Subsequently, the P-857 version of the interpreter was tested, debugged, and run under the Disk Operating Monitor (DOM) of the P-857 minicomputer. The third and final phase would have been the bootstrapping of the UCSD Pascal system into the P-857 with the use of the interpreter. Because of time constraints, however, this phase was not done.

A High Level Language Prototype of the P-Machine

In the planning stage of the implementation of the P-machine, two possibilities were considered in writing the software emulator using the P-857 assembly language. One was to first come up with a flow chart of the emulator and based on this, write the corresponding assembly language program. The other possibility considered was to first implement a prototype of the P-machine emulator using a high level language, specifically Pascal. After the prototype was tested and debugged, the assembly language version would then be written based on it.

There are pros and cons for each of these two approaches. Writing the emulator directly in assembly language with the aid of a flowchart seems to be fast and efficient. However, testing such an emulator can be done only after the emulator is written in its entirety. The second approach, on the other hand, seems to entail a lot of work, i.e., the interpreter has to be written twice. However, having a verified and working prototype decreases the possibility of failures in the assembly language version and increases the confidence of the implementor. Further, it may

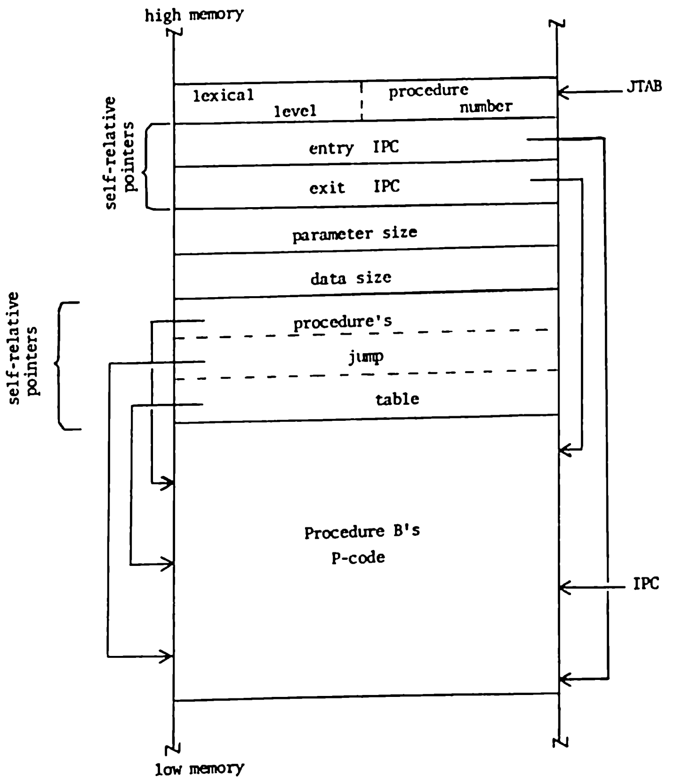
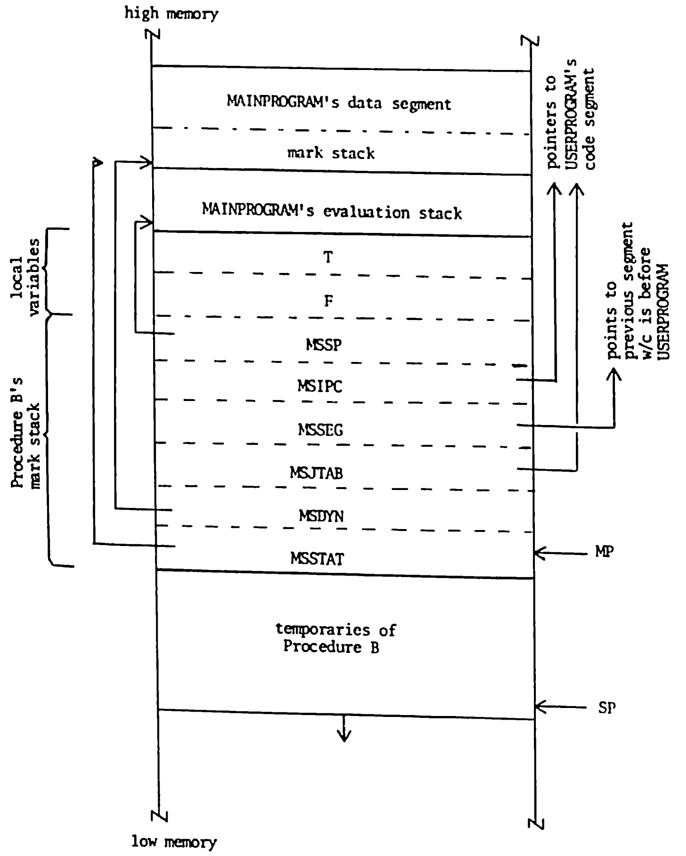


Figure 8. Procedure B's P-code Section. P-machine Pointers JTAB & IPC Shown during B's Execution.

be argued that writing a prototype in Pascal obviates the need for flowcharts. The reason is that the algorithm used in the prototype of the emulator, which would have been represented in flowcharts, are inherently expressed and embodied by the Pascal program describing the emulator. Therefore, the amount of work exerted when a prototype is employed is roughly the same as that required when flowcharts are used. Thus, based on the arguments given above, the second option was taken and a Pascal prototype of the P-machine was realized. Appendix A gives the listing of the prototype P-machine in Pascal.

Two Types of Interpreter Implementation

Central to the implementation of the P-machine in a real computer is the P-code interpreter. So, before any attempt at writing the interpreter was done, two types of interpreter implementation were first investigated. The usual interpreter implementation (Figure 10) has a routine which fetches a code from a sequence, decodes it, and transfers control to the appropriate routine which executes the code. After execution, control is transferred back to the fetch and decode routine. Threaded code implementation, on the other hand, dispenses with the fetch and decode routine used by a conventional interpreter. Instead of having a sequence of codes which are fetched, decoded and executed, the threaded code implementation has the sequence of addresses of the routines which execute the desired operations. Because of this change, the threaded code interpreter runs faster than a conventional interpreter.



high memory
MAINPROGRAM's data segment
mark stack
MAINPROGRAM's evaluation stack
T
F
MSSP
MSIPC
MSSEG
MSJTAB
MSDYN
MSSTAT
temporaries of Procedure B
low memory
local variables
Procedure B's mark stack
pointers to USERPROGRAM's code segment
points to previous segment w/c is before USERPROGRAM
MP
SP

Figure 9. Procedure B's Data Segment & Markstack.

A fast executing threaded code implementation is very attractive. But in the case of UCSD Pascal, it could not be easily implemented – it would require extensive rewriting of the P-compiler to produce addresses instead of P-codes. For this reason, it was decided to use a conventional fetch and decode interpreter for the P-codes.

THE P-MACHINE REALIZATION FOR THE P-857 MINICOMPUTER

The P-machine realization for the P-857 minicomputer is based on the Pascal prototype of the P-machine. Using the prototype as a model, the P-857 software emulator for the P-machine was hand-coded entirely in the P-857 assembly language. (A short description of the P-857 registers is given in Appendix B.)

The software emulator is subdivided into modules. (The motive behind this modularization is the consequent facility in coding, editing, and testing each of the modules separately before they are linked together to perform the ultimate goal.) The fourteen different modules and one macro file which constitute the software emulator are shown in Table 2. The functions of these modules and file are discussed below. In the discussions, it might be necessary to refer to the assembly language listing of the software emulator given in Appendix C.

The following discussion of the modules and definition file is arranged according to function. The first group centers on the mapping of the P-machine registers to the P-857 minicomputer registers. The second discusses the P-code interpreter organization and operation. The third is about the routines that execute the P-machine instructions. The last group focuses on the auxiliary and contingency routines.

Mapping of the P-machine Registers

Macrodefinitions for the P-machine

This module is a file consisting of four macrodefinitions that are shared globally by the other modules. It has the macrodefinition named REGASG that maps the P-857 registers onto the registers needed by the P-machine such as BASE, IPC, MP, SP, NP, GOBACK, and RETURN. Of these seven P-machine registers, the uses of the first five have already been discussed in the section on the P-machine. The last two – GOBACK and RETURN – are actually auxiliary registers in which the labels of subsequent assembly language instructions are saved for complicated P-code operations.

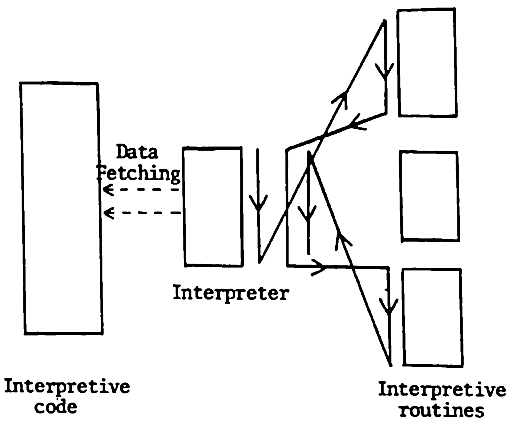
This file also has the macrodefinition SEGFOR which defines the constant offsets from the JTAB pointer of a currently executing procedure. They are ENTRIC, EXITIC, PARMSZ and DATASZ. Actually, the contents of the location pointed to by JTAB + ENTRIC is a pointer to the first P-code instruction of the aforementioned procedure. JTAB + EXITIC points to the address of the P-code instruction which causes exit from the procedure. JTAB + PARMSZ and JTAB + DATASZ point to the pointers of the procedure's parameter size and data size in bytes, respectively.

MSCWFORM is a macrodefinition for the constant offsets within the Mark Stack Control Word (MSCW). (MSCW, also referred to as the activation record of the called procedure, was discussed in connection with the P-machine.) To illustrate how these offsets are used, MSIPC is taken as an example. MP + MSIPC is in reality a pointer to that word in the MSCW which contains the interpreter program counter of the calling procedure, at the time of the call. The other offsets are used in the same manner.

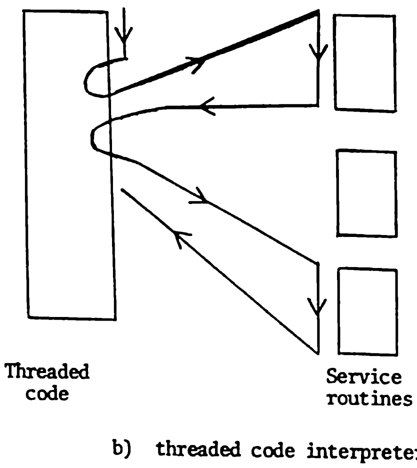
INTCONST is a macrodefinition of two interpreter constants. NIL contains the address of the location which contains zero. It is used as one of the values for a pointer type variable. MSDLTA is the required offset from BASE or MP to point to the first variable of a data segment. These macrodefinitions are made available to the other fourteen modules by using their names in macrocalls.

Syscom area and other P-machine pointers

This module (MODL02) contains the initialization and reservation of memory locations for two other P-machine pointers and the P-machine System Communication area. The P-machine pointers were discussed in the section on the P-machine. The System Communication area consists of locations in which some of the more important P-machine registers are stored.



a) conventional interpreter



b) threaded code interpreter

Figure 10. Flow of Control: a) Conventional Interpreter and b) Threaded Code Interpreter.

Table 2. P-machine Emulator Modules and Functions.

MODULE NAME	FUNCTIONS
Macrodefinition file	Register Assignment, code segment pointers, mark stack offsets, and interpreter constants
INITLZ	P-machine initialization
PRINT 1	Print utility for ASCII codes
MODL01	Transfer Table
MODL02	Syscom Area, Other P-machine pointers
MODL03	Integer Arithmetic
MODL04	Integer Compare
MODL05	Boolean Operations
MODL06	Fetch and Decode, Constant-One-Word Loads, Most Common P-codes
MODL07	Jumps
MODL08	Parameter Fetch, Local and Global Stores and Loads
MODL09	Indirect Store, Intermediate Stores and Loads
MODL10	Procedure Returns
MODL11	Procedure Calls
MODL12	Contingency Routines, Exit and No Operation, Not Yet Implemented P-codes

P-Code Interpreter Organization and Operation

INITLZ (P-machine Initialization)

This module initializes the P-machine registers and pointers before starting the interpretation process. It also prints a header consisting of the phrase 'START P-CODE INTERPRETER' to indicate the impending start of the emulation of the P-machine. Lastly, it transfers control to the fetch and decode routine of the interpreter which begins at the label BACK.

Fetch and Decode Routine

MODL06 is a very important module because it contains the interpreter routine that fetches and decodes all P-code instructions. This routine, which starts at the label BACK, determines whether the instruction is a short load constant instruction or not. If it is, the control is transferred to SLDC, otherwise the P-code is transformed to become an index of the transfer table XFRTBL. (This table contains the labels of routines that correspondingly simulate all of the remaining P-code instructions.) Observe that when the fetch and decode routine obtains a P-code from the code segment, it automatically increments IPC by one.

The inclusion of SLDC in the fetch and decode routine might seem unusual. However, this is not so. Because SLDC is the most common P-code instruction, its attachment to the fetch and decode routine is really an optimization for speed. (Note that MODL06 contains not only the P-code interpreter and the SLDC routine but also routines for other commonly used P-code instructions. These latter routines are discussed further under the P-machine instructions.)

Transfer Table

MODL01 is a data module used by the interpreter as a transfer table. It contains the labels of the corresponding assembly language portions which carry out the desired P-code operations.

Termination

MODL12 contains the XIT routine that terminates the P-code interpreter execution. The routine for XIT transfers the control from the interpreter program to the Disk Operating Monitor. It does so, when either a P-machine error occurs or the interpreter has finished executing all the P-codes. This routine prints 'END P-CODE INTERPRETER' to indicate the transfer of control.

P-Machine Instructions

Data Transfer Instructions

A. Constant-One-Word Loads, Short-Load Local and Global, and Short Index and Local Word

MODL06 contains the P-code interpreter's fetch and decode routine as well as the routines for commonly used P-codes. These P-code instruction routines are the constant-one-word loads, short-load locals and globals, and short index and load word. The constant-one-word loads LDCI and LDCN are self-explanatory. The other load instructions, however, need some explanation. The routines for these remaining instructions make use of P-857 register A10 to receive the P-code value. After the transfer of control from the fetch and decode routine, the corresponding load routine calculates the effective offset from the value of the P-code in A10. A look at the listing of MODL06 will clarify this action.

B. Local and Global Stores and Loads

MODL08 has the routines for local and global stores and loads. All of these instructions use the routine GETBIG to obtain the needed displacement from the MP or BASE pointer. GETBIG actually tests whether the requested displacement is contained in a byte or a word. This displacement is then obtained by GETBIG and is passed to the routine which needs it via register A10. The transfer of control from a given routine to GETBIG and back is achieved by using the P-machine auxiliary register RETURN. Before a routine passes control to GETBIG, the routine saves its return label in RETURN. After GETBIG performs its task, the P-857 program counter, P, is loaded with the contents of RETURN. This enables the correct continuation of the given routine's execution. (This technique is more efficient than the call/return mechanism of the P-857.)

The local loads and store actually index off MP to access the required local variable. The global loads and stores, on the other hand, index off BASE to access global variables.

The P-codes STL and SRO call procedure PRINT (an auxiliary routine to be discussed later) to print the hexadecimal value which is to be stored in a particular data segment.

C. Indirect Store, and Intermediate Store and Loads

MODL09 contains the routines of the P-codes which perform indirect store and intermediate store and loads. STO first obtains the word and the address consecutively from the stack. Next, it stores the word in the given address, thus executing an indirect store.

The intermediate loads and store P-codes need two parameters which are embedded in the instruction stream. The first portions of these P-code routines get a byte which contains the number of levels to link down the static chain in order to get the correct data segment where the required variable is located. These routines also use GETBIG to obtain the second parameter which is a displacement from the datum of the correct data segment. The remaining portions of these routines complete the tasks.

Routines for P-codes STO and STR, like other store instructions, call the procedures PRINT.

Data Operation Instructions

A. Top-of-stack Integer Arithmetic

The routines in MODL03 are those which perform the P-machine's integer top-of-stack arithmetic. The results of these routines are pushed into the stack in place of the operands which produced them. The unary operators are ABI, NGI, and SQI. ABI takes the absolute value of the integer on the top of the stack (tos), NGI negates it, whereas SQI takes the square of the integer at tos. CHK performs range checking of an index. Its operands are the top three stack positions. ADI and SBI are the dyadic operators which adds and subtracts, respectively, the integer at tos from the integer at tos-1. MOD, DVI, and MPI are the dyadic operators modulo, divide, and multiply. They are special in the sense that they make use of the P-857's multiply and divide instructions. Among these instructions, ABI, MPI, and SQI provide for overflow indication.

B. Integer Comparisons

MODL04 contains the integer comparison routines of the P-machine. They include equal, not equal, less or equal, less than, greater or equal, and greater than comparisons. All of them work on the top two elements of the stack, i.e., tos-1 is compared with tos. The boolean result of the comparisons replaces the two operands in the stack.

C. Boolean Arithmetic and Comparisons

The routines in MODL05 emulate the P-machine's top-of-stack boolean operations and boolean comparisons. The logical operators AND, IOR and NOT perform the logical functions conjunction, disjunction, and negation, respectively. These logical operators operate on all 16 bits, although the logical value is represented only by the least significant bit.

This module also consists of the boolean comparisons which are grouped with other types of comparisons, e.g., those for sets, strings, reals and so forth. Because of this, they are implemented so that they make use of the comparison type table CMPTBL. Routine COMPAR transfers control to the proper comparison type routines. The boolean comparisons actually make use of MODL04 – the integer comparisons. The other types of complex comparisons are not yet implemented.

Control Instructions

A. Jumps

MODL07 contains the routines for the P-machine's jump instructions. EFJ, NFJ, FJP and UJP are all two-byte P-code instructions. The first byte is the op-code and the last byte is the jump offset. EFJ and NFJ compare the top two elements of the stack for equality and depending on the result, will either perform a jump or continue with the next P-code instruction. EFJ performs the jump when the two top-of-stack elements are not equal whereas NFJ performs the jump when they are equal. FJP performs the jump when the top-of-stack element is false. UJP does an unconditional jump. If the jump offset byte is positive, it is added to IPC when the condition for the given type of jump is satisfied. If the offset is negative, and a jump has to be performed, this offset is taken to be a self-relative pointer within the jump table of the currently active procedure. Thus the negative jump offset is employed when either a backward jump or a jump of more than 127 bytes is required.

XJP performs a case or index jump. It actually first tests whether the given index is within the case index range or not. If the test fails, control is then given to the else jump location. (Note that the UCSD Pascal case statement has an else which is not present in "Standard" Pascal.)

B. Procedure Calls

MODL11 consists of the routines for the P-machine's procedure calls. CLP – call local procedure – does several things to perform proper procedure/function linkage. First, it saves the SEG pointer in the Syscom area. Second, it fetches a byte from code which is the called procedure's number. It makes use of this byte to obtain the correct JTAB pointer. Third, it performs a check to see what the procedure number is. This is done because a procedure number equal to zero indicates that the procedure's code is not written in P-code but in the assembly/machine language of the host machine (the P-857 in this case). Presently, this is not yet implemented. Fourth, it gets the data segment size, not including the space for MSCW, and reserves stack space for it. It also checks at this point if there is still enough space in the P-machine's stack. Fifth, it builds the MSCW, i.e., the activation record of the called procedure. Sixth, it obtains the parameters size, with the aid of JTAB and an offset. It copies the parameters, if there are any, from the calling procedure's top-of-stack to the data segment of the called procedure. Lastly, it sets the IPC to point to the first byte of P-code, and transfers control to the interpreter's fetch and decode routine.

The other procedure calls–CGP, CBP and CIP–do all of these things that CLP does, and with some additional tasks. CGP–call global procedure–corrects the static link in the MSCW, so that it points to BASE. CBP–call base procedure–adds an extra space in the MSCW to save the previous BASE pointer. It also sets the new BASE pointer and stores this in the Syscom area. CIP–call intermediate procedure–checks the lexical level of the called procedure to see if it is a base procedure or not. If it is, CIP then transfers control to CBP. Otherwise, it uses the lexical level to search down the dynamic chain to arrive at the desired data segment of the caller of the currently executing procedure. Thus, it obtains the desired MP pointer.

C. Procedure Returns

MODL10 has the routines for P-codes RNP and RBP. RNP–return from normal procedure–does three things. First, it checks whether a procedure is returning to the same segment or not. (In the present P-machine realization, all procedures come from the same segment.) Next, it checks whether there are parameters to be returned, i.e., when the return is being executed by a function. If there are words to be returned, then, the words are copied to the top of the stack of the calling procedure. Thirdly, the routine restores all of the P-machine registers to their state before the procedure or function call.

RBP–return base procedure–does exactly what RNP does. In addition, it restores the BASE register to its value immediately before the procedure/function call was made.

Auxiliary and Contingency Routines

PRINT 1 (Hexadecimal to ASCII Conversion and Printing)

This module contains a procedure named PRINT which is called by the routines for P-codes STL, SRO, STR, and STO (store instructions). This procedure converts a hexadecimal number to its equivalent ASCII representation and subsequently prints it. For the present P-machine realization, this is the only facility which can be used to determine the interpreter's state.

Contingency Routines

This module contains the P-codes which are not yet implemented. They are P-codes which operate on reals, sets, strings, byte arrays, records and arrays and a number of system support procedures.

This module also has the contingency routines which take control when some abnormal interpreter situation occurs. They indicate the situation by printing out appropriate messages. These routines are:

NOTIMP	–	not yet implemented P-codes,
STKOVF	–	P-machine stack overflow,
OVRFLW	–	integer register overflow, and
INVNDX	–	invalid integer index.

RESULTS and CONCLUSION

Testing the P-machine Emulator

To ensure that the P-machine emulator performs according to expectation, a sample test program was used. The Pascal source program is shown in Figure 11 and its equivalent in P-code is given in Figure 12. (This program contains a procedure for multiplying two integers. The procedure effectively carries out the multiplication by using register shifts–integer multiplication and division by 2.) A look at the source program indicates that the only data types involved are integers and booleans. However, the program is constructed so that it involves procedure call and return which are essential in block structure and recursive languages.

The P-code equivalent of the Pascal source was used to test the P-857 software emulator of the P-machine. In Figure 12, it can be noticed that the P-code program was prepared as a data module named PCODE1. It was prepared as a data module so that it can be linked with the P-857 P-machine emulator and subsequently loaded before its execution.

The result of the execution of the P-code program is shown in Figure 13. The result is actually a trace of the data fetches and stores made by the P-machine while it was executing the P-code program. The output trace is bracketed by the phrases 'START P-CODE INTERPRETER' and 'END P-CODE INTERPRETER'. The values in the trace are in hexadecimal. Note that these values correspond with the expected results in decimal as shown in Figure 14. (Figure 14 is the output trace of the Pascal prototype of the P-857 software emulator.)

Conclusion

A software emulator of the UCSD P-machine was implemented for the P-857 minicomputer. The emulator was tested with a representative P-code program and came up with correct results. It can execute a workable subset of the UCSD Pascal P-codes. The subset (Appendix D) represents instruction types including one-word fetching, storing and transferring; integer and boolean top-of-stack arithmetic and comparisons; procedure/function calls and returns; jumps and some systems program support routines.

There was not enough time to extend the P-machine implemetation to include the other data types and structures (i.e., aside from integer and boolean) and to

```
Program EXAMPLE;

    VAR X,Y,Z: Integer;

    Procedure MULTIPLY

        VAR A,B:Integer;

    Begin
        A:=X;
        B:=Y;
        Z:=0;
        While B > 0 Do
        Begin
            If B Mod 2 = 1  Then Z:=Z + A;
            A:= 2*A;
            B:= B Div 2
        End
    End; (*MULTIPLY*)

Begin
    X:=7;
    Y:=85;
    MULTIPLY
End.  (*EXAMPLE*)
```

Figure 11. Equivalent Pascal Source Program of the P-code Test Program.

consequently bootstrap the whole UCSD Pascal system into the P-857 minicomputer. However, the emulator has surely paved the way toward ultimately porting UCSD Pascal to the P-857 minicomputer.

ACKNOWLEDGMENTS

The author wishes to thank Bong Abaya, Louie Alarilla, and Rolly Dayco who read the draft of this paper and gave helpful suggestions to improve its readability. Thanks must also go to Sonny Viray who brought up the idea of writing this paper, and to Medy Santiago for her skillful typing of the text.

The work, on which this paper is based, was done at the Philips Research Laboratories in connection with the author's studies at the Philips International Institute in the Netherlands.

```
          IDENT PCODE1
          ENTRY     INITSP.INIIPC.MEMTOP.INITNP

INITNP    EQU       •                  NP INITIALLY POINTS HERE
          RES       10000              MEMORY SPACE FOR STACK AND HEAP
INITSP    EQU       •                  THIS IS WHERE SP INITIALLY POINTS

          DATA      /E8CC              SLDO1: STL        PROCEDURE MULTIPLY:
          DATA      /01E9              1: SLDO2:          BEGIN A::=X:
          DATA      /CC02              STL 2:                B::=Y:
          DATA      /00AB              SLDC0: SRO            Z::=0:
          DATA      /03D9              3: SLDL?:             WHILE B>0 DO
          DATA      /00C5              SLDC0: GTRI:             BEGIN
          DATA      /A118              FJP 24:                     IF B MOD 2 = 1 THEN
          DATA      /D902              SLDL2: SLDC2:
          DATA      /8E01              MOD: SLDC1:
          DATA      /C3A1              EQUI: FJP
          DATA      /05EA              5: SLDO3:                   Z::=Z+A:
          DATA      /D882              SLDL1: ADI:
          DATA      /A803              SRO 3:
          DATA      /02D8              SLDC2: SLDL1:               A::=2•A:
          DATA      /8FCC              MPI: STL
          DATA      /01D9              1: SLDL2:                   B::=B DIV 2:
          DATA      /0286              SLDC2: DVI:
          DATA      /CC02              STL 2:
          DATA      /B9F6              UJP -10:                 END
          DATA      /AD00              RNP 0:              END: (• MULTIPLY •)
•
•
•
INIIPC    DATA      /C201              COP 1:
          DATA      /0607              XIT: SLDC7:        PROGRAM EXAMPLE:
          DATA      /AB01              SRO 1:              BEGIN X::=M: (• M=7 •)
          DATA      /55AB              SLDC85: SRO           Y::=N: (• N=85 •)
          DATA      /02CE              2: CLP                MULTIPLY
          DATA      /02C1              2: RBP              END. (• EXAMPLE •)
          DATA      /0007              0: NOP
•
•
•
          DATA      /002D              JUMP TABLE OF MULTIPLY
          DATA      /0004              DATA SEGMENT SIZE
          DATA      /0000              PARAMETER SIZE
          DATA      /0016              EXITIC OF MULTIPLY
          DATA      /003E              ENTRIC OF MULTIPLY
          DATA      /0102              LEX LEV: PROC # OF MULT
          DATA      /0006              DATA SEGMENT SIZE
          DATA      /0000              PARAMETER SIZE
          DATA      /0013              EXITIC OF EXAMPLE
          DATA      /001D              ENTRIC OF EXAMPLE
          DATA      /0001              LEX LEV: PROC # OF EXMPL
          DATA      /000C              POINTER TO MULTIPLY
          DATA      /0004              POINTER TO EXAMPLE
          DATA      /0201              # OF PROC: SEG #
MEMTOP    DATA      /0201
•
•
•
          END
```

Figure 12. P-code Test Program.

```
EXIT CODE = 85
RUN PROG01
DATE  80 /04 /28    TIME 14H-54M-15S-

LABEL = UAUMANS              DATE =  29 02 80        PACK NBR =  000     AMANTE
START P-CODE INTERPRETER
0007
0055
0007
0055
0000
0007
000E
002A
001C
0015
0023
0038
000A
0070
0005
0093
'00E0
0002
01C0
0001
0253
0380
0000
END P-CODE INTERPRETER
PROG ELAPSED TIME:  00H-00M-07S-220MS-
```

Figure 13. P-machine Output Trace for P-code Test Program. (P-857 Software Emulator of P-machine.)

```
START P-CODE INTERPRETER
            7
           85
            7
           85
            0
            7
           14
           42
           28
           21
           35
           56
           10
          112
            5
          147
          224
            2
          448
            1
          595
          896
            0
 END P-CODE INTERPRETER
```

Figure 14. P-machine Output Trace for P-code Test Program. (Pascal Prototype of P-machine.)

REFERENCES

Pascal:

BOWLES, K.L. (1978). "UCSD Pascal: A (Nearly) Machine Independent Software System," *BYTE* May 1978, pp. 46, 170-173.

JENSEN, K. and Wirth, N. (1975). *Pascal User Manual and Report*, Springer-Verlag, N.Y., U.S.A.

UCSD Pascal Manual Version II.0, Institute for Information Systems, UC San Diego, California, U.S.A. (1978).

Compilers and Interpreters:

BAUER, F.L. and Eickel, J. (1974). *Compiler Construction – An Advanced Course*, Springer-Verlag, N.Y., U.S.A.

BELL, J.R. (1973). "Threaded Code," *Comm. ACM*, 16:6 370-372.

CHUNG, K.M. and Yuen, A. (1978a). "A 'Tiny' Pascal Compiler – Part 1: The P-code Interpreter," *BYTE* September 1978, pp. 58, 60, 62-65, 148-155.

________, (1978b). "A 'Tiny' Pascal Compiler – Part 2: The P-Compiler," *BYTE* October 1978, pp. 34, 36, 38, 40, 42, 44, 46, 48, 50, 52.

________, (1978c). "A 'Tiny' Pascal Compiler – Part 3: P-code to 8080 Conversion," *BYTE* November 1978, p. 182, 184-192.

DEWAR, B.K. (1975). "Indirect Threaded Code," *Comm. ACM*, 18:6 330-331.

ELSWORTH, E.F. (1979). "Compilation via an Intermediate Language," *The Computer Journal* 22:3, 226-233.

EUWE, W.J.G. (1976). *P.I.I. Survey: Program Translation Techniques*, N.V. Philips Gloeilampenfabrieken, Eindhoven, The Netherlands.

FORSYTH. C.H. and Howard, R.J. (1978). "Compilation and Pascal on the New Microprocessors," *BYTE* August 1978, pp. 50, 52, 54, 56, 58, 60-61.

GRIES, D. (1971). *Compiler Construction for Digital Computers*, Wiley, N.Y., U.S.A.

MANGASER, A.A. (1980). *Design and Implementation of a UCSD Pascal P-code Interpreter*, Philips Research Laboratories Technical Note No. 92/80, Eindhoven, The Netherlands.

PASMAN, W.J.A. (1979). "Implementation of Pascal on an 8080 Microcomputer," *EUROMICRO Journal* 5:6, 363-369.

P-4 Assembler and Interpreter, ETH, Zurich, Switzerland (1976).

UCSD Pascal Interpreter for Zilog's Z-80/Intel's 8080A, Institute for Information Systems, UC San Diego, California, U.S.A. (1978).

UCSD Pascal System and Interpreter for PDP-11's, Institute for Information Systems, UC San Diego, California, U.S.A. (1978).

WIRTH, N. (1976). *Algorithms + Data Structures = Programs*, Prentice-Hall, New Jersey, U.S.A.

P857 minicomputer:

P800 Macroprocessor – Element Performance Specification, N.V. Philips Gloeilampenfabrieken, Eindhoven, The Netherlands (1975).

P800M Programmer's Guide 1, 2, and 3 Volume II: Instruction Set, Philips Data Systems B.V., Apeldoorn, The Netherlands (1978).

P800M Programmer's Guide 2 Volume I: DOM, Philips Data Systems B.V., Apeldoorn, The Netherlands (1976).

P852M Systems Handbook, Philips Electrologica B.V., Apeldoorn, The Netherlands (1974).

APPENDIX A. The Interpreter in Pascal

```
PROGRAM PCODEINTERPRETER(INPUT,OUTPUT);
     .

  (* A.A. MANGATER  22 FEBRUARY 1980   PHILIPS NAT LAB *)
  (* 22/02/80  WORKS ONLY WITH INTEGERS AND BOOLEANS    *)


LABEL 99;

CONST PBYTEMAX=255;   BYTEMIN=-128;   BYTEMAX=127;
      PWORDMAX=65535;   WORDMIN=-32768;   WORDMAX=32767;
        STKSIZE=4000;   CODESIZE=2000;
        OVERS=4001;   OVERC=2001;

TYPE  BIT=0..1;
        TIPE=(STRUCT,ABSLT,SIGNED);
        BYTE= RECORD CASE TIPE OF
                STRUCT: (BB:PACKED ARRAY[0..7] OF BIT);
                ABSLT : (UB:0..PBYTEMAX);
                SIGNED: (SB:BYTEMIN..BYTEMAX)
              END;
        WORD= RECORD CASE TIPE OF
                STRUCT: (BW:PACKED ARRAY[0..1] OF BYTE);
                ABSLT : (UW:0..PWORDMAX);
              , SIGNED: (SW:WORDMIN..WORDMAX)
              END;
      DATATYPE=(UNDEF,INT,BOOL,ONEWORD,ADDR);
      MESSAGE=PACKED ARRAY[1..25] OF CHAR;

VAR   STACK: ARRAY[0..STKSIZE] OF
              RECORD CASE DATATYPE OF
                INT: (VI:INTEGER);
                BOOL: (VB:BOOLEAN);
                ONEWORD: (VW:WORD);
                ADDR: (VA:INTEGER)
              END;
      CODE:ARRAY[0..CODESIZE] OF BYTE;
        IPC: -1..OVERC;
      SP,NP,JTAB,MP,SEG,BASE:-1..OVERS;
      I,J,K:0..CODESIZE;
      OP1,OP2:BYTE;
      BYTE1,BYTE2,BYTE3:BYTE;
      X:INTEGER;
      WORD1,WORD2,WORD3:WORD;
        EMULATING:BOOLEAN;


 FUNCTION BASELEV(L:BYTE):INTEGER;
   VAR B1:INTEGER;
   BEGIN B1:=MP;
     WHILE L.UB>0 DO
       BEGIN B1:= STACK[B1].VA; L.UB:=L.UB-1
       END;
     BASELEV:=B1
   END;(*BASELEV*)


 PROCEDURE ERRORM(STRING :MESSAGE);
   BEGIN WRITELN; WRITELN(STRING); GOTO 99
   END;(*ERRORM*)


 PROCEDURE GETB;
   BEGIN
     IF CODE[IPC].UB<128 THEN BEGIN
          WORD1.UW:= CODE[IPC].UB ;
          IPC:= IPC-1 END
     ELSE BEGIN
          WORD1.UW:= (CODE[IPC].UB MOD 128)*256 + CODE[IPC-1].UB;
          IPC:= IPC-2
           END
   END; (* GETB *)


 PROCEDURE BUILDMSCW;
   BEGIN
     BYTE1.UB:=CODE[IPC].UB;  IPC:= IPC-1;
     WORD1.UW:=STACK[SEG-BYTE1.UB].VA;
     WORD2.UW:=SP-(STACK[WORD1.UW-4].VW.UW+1) DIV 2
              -(STACK[WORD1.UW-3].VW.UW+1) DIV 2;
     IF WORD2.UW-6 <= NP THEN
       ERRORM('STACK OVERFLOW            ');
     (* MARKSTACK CONTROL WORD *)
     STACK[WORD2.UW-1].VA:=SP;
     STACK[WORD2.UW-2].VA:= IPC;
     STACK[WORD2.UW-3].VA:= SEG;
     STACK[WORD2.UW-4].VA:= JTAB;
     STACK[WORD2.UW-5].VA:= MP; (*DYN LINK *)
     (* SET UP NEW ENVIRONMENT *)
     SP:= WORD2.UW-6;
     JTAB:= WORD1.UW;
     IPC:= STACK[WORD1.UW-1].VA;
     SEG:= SEG;
 END; (* BUILDMSCW *)


 PROCEDURE INTERPRET;
   BEGIN WRITELN('START P-CODE INTERPRETER');
     IPC:=CODESIZE;  SP:=OVERS;
     NP:=-1;  MP:=OVERS;
     JTAB:=OVERS;
     SEG:=STKSIZE;  BASE:=OVERS;
     EMULATING:= TRUE;

     WHILE EMULATING DO
          BEGIN
          (* FETCH*)
          OP1.UB:=CODE[IPC].UB;
          IPC:=IPC-1;

          (* EXECUTE *)
          IF (OP1.UB<=127) AND (OP1.UB>=0) THEN
            (* CONSTANT ONE-WORD LOADS *)
            BEGIN  (* SLDC 0..127 *)
              SP:=SP-1;
              STACK[SP].VW.UW:= OP1.UB
            END
          ELSE
          CASE OP1.UB OF

            (* CONSTANT ONE-WORD LOADS *)
          199:BEGIN  (* LDCI W *)
                  SP:=SP-1;
                  STACK[SP].VW.UW:=CODE[IPC-1].UB*256+ CODE[IPC].UB;
                  IPC:=IPC-2
              END;

            (* LOCAL ONE-WORD LOADS AND STORES *)
          216,217,218,219,220,221,222,223,
          224 225 226 227 228 229 230 231·
          BEGIN  (* SLDL1..16 *)
         SP:=SP-1;
         BYTE1.UB:= OP1.UB -215;
         WORD1.UW:= STACK[MP+5].VA;
         STACK[SP].VW.UW:=STACK[WORD1.UW-BYTE1.UB].VW.UW
          END;
202:BEGIN  (* LDL B *)
         SP:=SP-1;
         GETB;
         STACK[SP].VW.UW:=STACK[STACK[MP+5].VA-WORD1.UW].VW.UW
     END;
190:BEGIN  (* LLA B *)
         SP:=SP-1;
         GETB;
         STACK[SP].VA:=STACK[STACK[MP+5].VA-WORD1.UW].VA
     END;
204:BEGIN  (* STL B *)
         GETB;
         STACK[STACK[MP+5].VA-WORD1.UW]:=STACK[SP];
         WRITELN(STACK[STACK[MP+5].VA -WORD1.UW].VW.UW);
         SP:=SP+1;
     END;

  (* GLOBAL ONE-WORD LOADS AND STORES *)
232.233.234.235.236.237.238.239.
240.241.242.243.244.245.246.247:
          BEGIN  (*SLDO1..16 *)
         SP:=SP-1;
         BYTE1.UB:= OP1.UB-231;
         WORD1.UW:=STACK[BASE+5].VA;
         STACK[SP].VW.UW:=STACK[WORD1.UW-BYTE1.UB].VW.UW
          END;
167:BEGIN  (*LDO B *)
         SP:=SP-1;
         GETB;
         STACK[SP].VW.UW:=STACK[STACK[BASE+5].VA-WORD1.UW].VW.UW
     END;
165:BEGIN  (* LAO B *)
         SP:=SP-1;
         GETB;
         STACK[SP].VA:= STACK[STACK[BASE+5].VA-WORD1.UW].VA
     END;
171:BEGIN  (* SRO B *)
         GETB;
         STACK[STACK[BASE+5].VA-WORD1.UW]:=STACK[SP];
         WRITELN(STACK[STACK[BASE+5].VA-WORD1.UW].VW.UW);
         SP:=SP+1
     END;

  (* INTERMEDIATE ONE-WORD LOADS AND STORES *)
182:BEGIN  (* LOD DB.B *)
         SP:=SP-1;
         BYTE1.UB:=CODE[IPC].UB;  IPC:=IPC-1;
         GETB;
         STACK[SP].VW.UW:= STACK[STACK[BASELEV(BYTE1)+5].VA -WORD1.UW].VW.UW
     END;
178:BEGIN  (* LDA DB.B *)
         SP:=SP-1;
         BYTE1.UB:=CODE[IPC].UB;  IPC:=IPC-1;
         GETB;
         STACK[SP].VA:= STACK[STACK[BASELEV(BYTE1)+5].VA -WORD1.UW].VA
     END;
184:BEGIN  (* STR DB.B *)
         BYTE1.UB:=CODE[IPC].UB;  IPC:=IPC-1;
         GETB;
         STACK[STACK[BASELEV(BYTE1)+5].VA -WORD1.UW]:=STACK[SP];
         WRITELN(STACK[STACK[BASELEV(BYTE1)+5].VA -WORD1.UW].VW.UW);
         SP:=SP+1
     END;

  (* INDIRECT ONE-WORD LOADS AND STORES *)
248.249.250.251.252.253.254.255:
          BEGIN  (* SINDO..7 *)
         BYTE1.UB:=OP1.UB-248;
         STACK[SP].VW.UW:= STACK[STACK[SP].VA-BYTE1.UB].VW.UW
          END;
154:BEGIN  (* STO *)
         WORD1.UW:=STACK[SP+1].VA;
         STACK[WORD1.UW]:=STACK[SP];
         SP:=SP+2
     END;

  (* TOP-OF-STACK ARITHMETIC *)

  (* LOGICAL *)
132:BEGIN  (* LAND *)
         SP:=SP+1;
         STACK[SP].VB:=STACK[SP].VB AND STACK[SP-1].VB
     END;
141:BEGIN  (* LOR *)
         SP:=SP+1;
         STACK[SP].VB:=STACK[SP].VB  OR STACK[SP-1].VB
     END;
147:  (* LNOT *)
         STACK[SP].VB:= NOT STACK[SP].VB;

  (* INTEGER *)
128:BEGIN  (* ABI *)
         IF STACK[SP].VI<=-32768 THEN
           ERRORM('VALUE UNDEFINED          ')
         ELSE STACK[SP].VI:= ABS(STACK[SP].VI)
    END;
130:BEGIN  (* ADI *)
         SP:=SP+1;
         STACK[SP].VI:= STACK[SP].VI + STACK[SP-1].VI
     END;
145:  (* NGI *)
         STACK[SP].VI:= -STACK[SP].VI;
149:BEGIN  (* SBI *)
         SP:=SP+1;
         STACK[SP].VI:= STACK[SP].VI - STACK[SP-1].VI
     END;
143:BEGIN  (* MPI *)
         SP:=SP+1;
         STACK[SP].VI:=STACK[SP].VI * STACK[SP-1].VI
     END;
152:  (* SQI *)
         STACK[SP].VI:= SQR(STACK[SP].VI);
134:BEGIN  (* DVI *)
         SP:=SP+1;
         STACK[SP].VI:= STACK[SP].VI DIV STACK[SP-1].VI
     END;
142:BEGIN  (* MODI *)
         SP:=SP+1;
         STACK[SP].VI:=STACK[SP].VI MOD STACK[SP-1].VI
     END;
  (* BOUNDARY CHECK *)
136:  (* CHK *)
         IF (STACK[SP+2].VI< STACK[SP+1].VI) OR
            (STACK[SP+2].VI> STACK[SP].VI) THEN
          ERRORM('VALUE OUT OF RANGE        ');
```

```
        (* LOGICAL COMPARISONS *)
       195:BEGIN  (* EQUI *)
              SP:=SP+1;
              STACK[SP].VB:= STACK[SP].VI=STACK[SP-1].VI
          END;
       175: BEGIN  (*EQU*)
              OP2.UB:=CODE[IPC].UB;  IPC:=IPC-1;
              SP:=SP+1;
              CASE OP2.UB OF
              2: (* REAL *);
              4:  (* STRING *);
              6:  (* BOOL *)
                STACK[SP].VB:= STACK[SP].VB = STACK[SP-1].VB;
              8:  (* POWR *);
              10:  (* BYT *);
              12:  (* WORD *);
              END;
            END;
        203:BEGIN  (* NEQI *)
              SP:=SP+1;
              STACK[SP].VB:=STACK[SP].VI <> STACK[SP-1].VI
           END;
        193:BEGIN  (* NEQ *)
              OP2.UB:=CODE[IPC].UB;  IPC:=IPC-1;
              SP:=SP+1;
              CASE OP2.UB OF
              2:  (*REAL *);
              4:  (* STRING *);
              6:  (*BOOL *)
                 STACK[SP].VB:= STACK[SP].VB <> STACK[SP-1].VB;
              8:  (* POWR *);
              10:  (* BYT *);
              12:  (* WORD *);
              END;
           END;
         200:BEGIN  (* LEQI *)
              SP:=SP+1;
              STACK[SP].VB:= STACK[SP].VI <= STACK[SP-1].VI
           END;
         180:BEGIN  (* LEQ *)
              OP2.UB:=CODE[IPC].UB;  IPC:=IPC-1;
              SP:= SP+1;
              CASE OP2.UB OF
              2:  (* REAL *);
              4:  (* STRING *);
              6:  (* BOOL *)
                 STACK[SP].VB:= STACK[SP].VB <= STACK[SP-1].VB;
              8:  (* POWR *);
              10:  (* BYT *);
              12: ERRORM('<= COMPARISON FOR WORDS  ');
              END;
            END;
         201:BEGIN  (* LESI *)
              SP:=SP+1;
              STACK[SP].VB:= STACK[SP].VI < STACK[SP-1].VI
            END;
         181: BEGIN  (* LES *)
              OP2.UB:= CODE[IPC].UB;  IPC:= IPC-1;
              SP:=SP+1;
              CASE OP2.UB OF
              2:  (* REAL *);
              4:  (* STRING *);
              6:  (* BOOL *)
                 STACK[SP].VB:= STACK[SP].VB < STACK[SP-1].VB;
              8:  ERRORM('< COMPARISON FOR SET      ');
              10:  (* BYT *);
              12:  ERRORM('< COMPARISON FOR    DS   ');
              END;
            END;
          196:BEGIN  (* GEQI *)
              SP:=SP+1;
              STACK[SP].VB:= STACK[SP].VI >= STACK[SP-1].VI
            END;
          176: BEGIN  (* GEQ *)
              OP2.UB:= CODE[IPC].UB;  IPC:= IPC-1;
              SP:=SP+1;
              CASE OP2.UB OF
              2:  (* REAL *);
              4:  (* STRING *);
              6:  (* BOOL *)
                 STACK[SP].VB:= STACK[SP].VB >= STACK[SP-1].VB;
              8:  (* POWR *);
              10:  (* BYT*);
              12: ERRORM('>= COMPARISON FOR WORDS  ');
              END;
           END;
          197:BEGIN  (* GTRI *)
              SP:=SP+1;
              STACK[SP].VB:= STACK[SP].VI > STACK[SP-1].VI
            END;
          177: BEGIN  (* GTR *)
              OP2.UB:=CODE[IPC].UB;  IPC:=IPC-1;
              SP:= SP+1;
              CASE OP2.UB OF
              2:   (* REAL *);
              4:   (* STRING *);
              6:  (* BOOL *)
                 STACK[SP].VB:= STACK[SP].VB > STACK[SP-1].VB;
              8:  ERRORM('> COMPARISON FOR SETS    ');
              10:  (*BYT *);
              12:  ERRORM('> COMPARISON FOR WORDS   ');
              END;
            END;

          (* JUMPS *)
          185: BEGIN (* UJP SB *)
              BYTE1.UB:=CODE[IPC].UB;  IPC:=IPC-1;
              IF BYTE1.UB>128 THEN BEGIN (* JUMP TO JTAB *)
                IPC:= STACK[JTAB -4 -BYTE1.UB MOD 128].VA
                END
              ELSE IPC:=IPC - BYTE1.UB
            END;
          161: BEGIN  (* FJP SB *)
              BYTE1.UB:= CODE[IPC].UB;  IPC:=IPC-1;
              IF BYTE1.UB>128 THEN BEGIN (* JUMP TO JTAB *)
                IPC:= STACK[JTAB -5 -BYTE1.UB MOD 128].VA
                END
              ELSE BEGIN
                IF NOT STACK[SP].VB THEN IPC:=IPC- BYTE1.UB
                   END;
              SP:=SP+1
            END;
          211: BEGIN  (* EFJ SB *)
              BYTE1.UB:=CODE[IPC].UB;  IPC:=IPC-1;
              IF BYTE1.UB>128 THEN BEGIN
                IPC:= STACK[JTAB -4 -BYTE1.UB MOD 128].VA
                END
              ELSE BEGIN
                IF STACK[SP].VI<>STACK[SP-1].VI THEN
                  IPC:=IPC-BYTE1.UB
                   END;
            END;
        212: BEOIN  (* NFJ SB *)
                BYTE1.UB:=CODE[IPC].UB:  IPC:=IPC-1;
                IF BYTE1.UB>128 THEN BEGIN
                  IPC:= STACK[JTAD -4 -BYTE1.UB MOD 128].VA
                   END
                ELSE BEOIN
                  IF STACK[SP].VI= STACK[SP-1].VI THEN
                     IPC:=IPC-BYTE1.UB;
                     END;
            END;
        172:BEOIN (* XJP W1.W2.W3.<CASETABLE> *)
                WORD1.UW:= CODE[IPC-1].UB*256 + CODE[IPC].UB;
                IPC:=IPC-2;
                WORD2.UW:= CODE[IPC-1].UB*256 + CODE[IPC].UB;
                IPC:=IPC-2;
                IF (STACK[SP].VI<WORD1.UW) OR
                   (STACK[SP].VI>WORD2.UW) THEN
                    (* IPC ALREADY POINTS TO UJP INSTRUCTION *)
                ELSE BEOIN WORD3.UW:= STACK[SP].VI - WORD1.UW;
                           IPC:= CODE[WORD3.UW].UB - WORD3.UW
                     END;
            END;

          (* PROCEDURE AND FUNCTION CALLS AND RETURNS *)
        206: BEOIN (* CLP UB *)
                BUILDMSCW;
                STACK[WORD2.UW-6].VA:= MP; (*STAT LINK*)
                MP:= WORD2.UW-6;
            END;
        207:BEOIN (* COP UB *)
                BUILDMSCW;
                STACK[WORD2.UW-6].VA:=BASE; (* STAT LINK *)
                BASE:= WORD2.UW-6;
                MP:= WORD2.UW-6
            END;
        174: BEOIN (* CIP UB *)
                BUILDMSCW;
                BYTE2.UB:=STACK[JTAB].VW.UW MOD 256 ;
                IF BYTE2.UB>0 THEN BEOIN
                   WORD3.UW:=MP;
                   REPEAT
                      WORD3.UW:= STACK[WORD3.UW+1].VA;
                      BYTE3.UB:= STACK[STACK[WORD3.UW+2].VA].VW.UW MOD 256
                   UNTIL BYTE2.UB-BYTE3.UB = 1;
                   STACK[WORD2.UW-6].VA:= STACK[WORD3.UW].VA END
                ELSE
                  (* SIMILAR TO CBP UB *);
           END;
        194: BEOIN (* CBP UB *)
                (* NOT YET IMPLEMENTED *)
            END;
        173: BEOIN (* RNP DB *)
                BYTE1.UB:= CODE[IPC].UB;  IPC:= IPC-1;
                WORD1.UW:= STACK[MP + 5].VA;
                FOR I:=1 TO BYTE1.UB DO
                  STACK[WORD1.UW-I].VW.UW:=
                    STACK[MP + 6 + STACK[JTAB-4].VA - I].VW.UW;
                SP:= STACK[MP+5].VA -BYTE1.UB;
                IPC:=STACK[MP+4].VA;
                SEG:= STACK[MP+3].VA;
                JTAB:= STACK[MP+2].VA;
                MP:= STACK[MP+1].VA
            END;
        193: BEOIN (* RBP DB *)
                (* NOT YET IMPLEMENTED *)
           END;

        213: BEGIN (* BPT *)
                (* TEMPORARY HALT *)
                EMULATING:=FALSE;
                WRITE('END P-CODE INTERPRETER')
            END;

        END; (* CASE *)

        END; (* WHILE *)

     END;(* INTERPRET *)


BEOIN (* MAIN *)
        I:=CODESIZE;
        READLN(X);
        WHILE X<>256 DO
          BEGIN
           IF I<1 THEN
                BEGIN WRITE('PROGRAM TOO LONO');
                  OOTO 99
                END;
           CODE[I].UB:=X;
           I:=I-1;
           READLN(X)
          END;
        IF I<> CODESIZE THEN
          BEOIN
           FOR J:= CODESIZE DOWNTO I+1 DO
                 WRITELN(J,'   ', CODE[J].UB);
                   INTERPRET
                 END
        ELSE WRITE('NO CODE IN INPUT FILE');
99:WRITELN
END.(* PCODEINTERPRETER *)
```

APPENDIX B. P-857 Registers

P-857 REGISTER	FUNCTION
A0	P-857 program counter (also referred to as P)
A1, A2	double length arithmetic; multiply and divide registers
A3 to A5	general purpose registers
A7, A8	I/O registers
A9 to A14	general purpose registers
A15	P-857 Interrupt Stack Pointer

```
*******************************************************************
*** UCSD PASCAL P-CODE INTERPRETER              PHILIPS NAT.LAB.    ***
*** A.A. MANGASER                               15 APRIL 1980       ***
*******************************************************************


***   REGISTER ASSIGNMENTS   ***
*
*   A0          P857 PROGRAM COUNTER
*   A1.A2       DOUBLE LENGTH ARITHMETIC.MULTIPLY AND DIVIDE REGISTERS
*   A7.A8       LKM-INSTRUCTION REGISTERS FOR I/O
*   A15         P857 INTERRUPT STACK POINTER
*
#MACRO REGASG:
BASE      EQU       A3                                P-MACHINE:BASE OF GLOBAL DATA SEGMENT
IPC       EQU       A4                                P-MACHINE:INTERPRETER PROGRAM COUNTER
MP        EQU       A5                                P-MACHINE:BASE OF LOCAL DATA SEGMENT
SP        EQU       A6                                P-MACHINE:STACK POINTER
NP        EQU       A9                                P-MACHINE:HEAP POINTER
GOBACK    EQU       A13                               P-MACHINE:AUXILIARY REGISTER WHERE LABEL
                                                      OF NEXT ASSEMBLY LANGUAGE INSTRUCTION
                                                      IS SAVED FOR COMPLEX OPERATIONS
RETURN    EQU       A14                               P-MACHINE:AUXILIARY REGISTER WHERE LABEL
                                                      OF NEXT ASSEMBLY LANGUAGE INSTRUCTION
                                                      IS SAVED FOR COMPLEX OPERATIONS
#MEND:
*
***

***   CODE SEGMENT FORMAT   ***
*
*   SEG POINTS TO THE FIRST WORD OF THE SEGMENT DICTIONARY
*       BYTE1 CONTAINS THE SEGMENT NUMBER
*       BYTE0 CONTAINS THE NUMBER OF PROCEDURES
*       THE SUBSEQUENT WORDS ARE POINTERS TO THE PROCEDURE
*           DICTIONARIES OF THE SEGMENT'S PROCEDURES
*
*   JTAB POINTS TO THE FIRST WORD OF THE PROCEDURE DICTIONARY
*       BYTE1 CONTAINS THE PROCEDURE NUMBER
*       BYTE0 CONTAINS THE LEXICAL LEVEL
*       THE SUBSEQUENT WORDS CONTAIN INFORMATION ABOUT THE
*           PROCEDURE AND ITS JUMP TABLE
*
*   THE FOLLOWING ARE OFFSETS RELATIVE TO THE WORD POINTED TO
*       BY JTAB
*
#MACRO SEGFOR:
ENTRIC    EQU       -2                                ENTRY POINT OFFSET
EXITIC    EQU       -4                                EXIT POINT OFFSET
PARMSZ    EQU       -6                                # OF PARAMETERS TO BE COPIED AT ENTRY
DATASZ    EQU       -8                                # OF WORDS TO RESERVE IN THE STACK
#MEND:
*
*   THE JUMP TABLE CONTAIN POINTERS TO RELEVANT POINTS IN THE
*       PROCEDURE
*
***

***   MARK STACK CONTROL WORD FORMAT   ***
*
*   THE FOLLOWING ARE OFFSETS RELATIVE TO THE STATIC LINK WORD
*
#MACRO MSCWFORM:
MSSTAT    EQU       0                                 STAT LINK- POINTS TO PARENT'S STAT LINK
MSDYN   . EQU       2                                 DYNAMIC LINK- POINTS TO CALLER'S STAT LINK
MSJTAB    EQU       4                                 ABS MEM ADDRS OF CALLER JTAB
MSSEG     EQU       6                                 ABS MEM ADDRS OF SEGMENT TABLE OF CALLER
MSIPC     EQU       8                                 ABS MEM ADDRS OF NEXT OPCODE IN CALLER
MSSP      EQU       10                                MEM ADDRS OF CALLER'S SP
MSBASE    EQU       -2                                BASE REGISTER- ONLY IN BASE MSCW
#MEND:
*
***

***   INTERPRETER CONSTANTS   ***
*
#MACRO INTCONST:
NIL       EQU       /80                               VALUE OF NIL POINTER
MSDLTA    EQU       10
#MEND:
*
***

```

```
          IDENT     INITLZ
          EXTRN     INITSP.INIIPC.BACK.MEMTOP.INITNP.JTAB.SEG
#REGASG:
#INTCONST:

*** P-MACHINE INITIALIZATION ***
*
ENTECB    DATA      2.ENTMSG.26.0.0.0
ENTMSG    DATA      /0A0D                             CARRIAGE RETURN LINE FEED
          DATA      'START P-CODE INTERPRETER'
*
BEGIN     LDKL      SP.INITSP                         INITIALIZE REGISTERS AND POINTERS
          LDKL      IPC.INIIPC
          LDKL      NP.INITNP
          LDKL      BASE.NIL
          LDKL      MP.NIL
          LDKL      A10.NIL
          ST        A10.JTAB
          LDKL      A10.MEMTOP
          ST        A10.SEG
*
          LDK       A7./85                            PRINT HEADER
          LDKL      A8.ENTECB
          LKM
          DATA      1
*
          ABL       BACK                              START INTERPRETATION
*
*** END INITIALIZATION ***

          END       BEGIN
```

```
          IDENT PRINT1
          ENTRY     PRINT.CFSTAK
#REGASG:

*** CONVERT AND PRINT ***
*
*   THIS IS A PROCEDURE WHICH CONVERTS A HEXADECIMAL INTO ITS
```

```
•      CORRESPONDING ASCII CHARACTERS AND AFTERWARDS PRINTS IT.
•   IT USES A2 TO PASS THE HEXADECIMAL NUMBER.
•   IT STORES THE CONVERTED NUMBER INTO ASCII+2 &ASCII+4.
•
NUMECB     DATA      2.ASCII.6.0.0.0
ASCII      DATA      /0A0D                     CARRIAGE RETURN AND LINE FEED
           RES       2                         THE CONVERTED NUMBER
HEXTBL     DATA      '0123456789ABCDEF'
HEXA       RES       1                         THE NUMBER TO BE CONVERTED
CFSTAK     RES       2                         PROCEDURE CALL STACK FOR A0 & PSW
•
PRINT      ST        A2.HEXA                   BEGIN CONVERSION
           ANK       A2./0F                    LS NIBBLE
           LC        A1.HEXTBL.A2
           SC        A1.ASCII+5
           LC        A2.HEXA+1
           ANK       A2./F0                    NEXT NIBBLE
           SRL       A2.4
           LC        A1.HEXTBL.A2
           SC        A1.ASCII+4
           LC        A2.HEXA
           ANK       A2./0F                    NEXT NIBBLE
           LC        A1.HEXTBL.A2
           SC        A1.ASCII+3
           LC        A2.HEXA
           ANK       A2./F0                    MS NIBBLE
           SRL       A2.4
           LC        A1.HEXTBL.A2
           SC        A1.ASCII+2                END OF CONVERSION
•
           LDK       A7./85                    PREPARE TO PRINT
           LDKL      A8.NUMECB
           LKM
           DATA      1
•
           RTN       RETURN
•
••• END OF CONVERT AND PRINT •••

           END
```

```
           IDENT MODL01
           ENTRY     XFRTBL
           EXTRN     ABI.ABR.ADI.ADR
           EXTRN     AND.DIF.DVI.DVR
           EXTRN     CHK.FLO.FLT.INN
           EXTRN     INT.IOR.MOD.MPI
         • EXTRN     MPR.NGI.NGR.NOT
           EXTRN     SRS.SBI.SBR.SGS
           EXTRN     SQI.SQR.STO.IXS
           EXTRN     UNI.S2P.CSP.LDCN
           EXTRN     ADJ.FJP.INC.IND
           EXTRN     IXA.LAO.LCA.LDO
           EXTRN     MOV.MVB.SAS.SRO
           EXTRN     XJP.RNP.CIP.COMPAR
           EXTRN     LDA.LDC
           EXTRN     LOD
           EXTRN     STR.UJP.LDP.STP
           EXTRN     LDM.STM.LDB.STB
           EXTRN     IXP.RBP.CBP.EQUI
           EXTRN     GEQI.GTRI.LLA.LDCI
           EXTRN     LEQI.LESI.LDL.NEQI
           EXTRN     STL.CXP.CLP.CGP
           EXTRN     S1P.IXB.BYT.EFJ
           EXTRN     NFJ.BPT.XIT.NOP
           EXTRN     SLDLS
           EXTRN     SLDOS
           EXTRN     SINDS

•••   TRANSFER TABLE   •••
•  THIS TABLE CONTAINS THE LABELS OF THE CORRESPONDING ASSEMBLY
•  LANGUAGE PORTIONS WHICH CARRY OUT THE DESIRED P-CODE INSTRUCTIONS.
•
•          DATA      SLDC0....     .SLDC127         00/000
XFRTBL     DATA      ABI.ABR.ADI.ADR                HEX=80/DEC=128
           DATA      AND.DIF.DVI.DVR                84/132
           DATA      CHK.FLO.FLT.INN                88/136
           DATA      INT.IOR.MOD.MPI                8C/140
           DATA      MPR.NGI.NGR.NOT                90/144
           DATA      SRS.SBI.SBR.SGS                94/148
           DATA      SQI.SQR.STO.IXS                98/152
           DATA      UNI.S2P.CSP.LDCN               9C/156
           DATA      ADJ.FJP.INC.IND                A0/160
           DATA      IXA.LAO.LCA.LDO                A4/164
           DATA      MOV.MVB.SAS.SRO                A8/168
           DATA      XJP.RNP.CIP.COMPAR             AC/172
           DATA      COMPAR.COMPAR.LDA.LDC          B0/176
           DATA      COMPAR.COMPAR.LOD.COMPAR       B4/180
           DATA      STR.UJP.LDP.STP                B8/184
           DATA      LDM.STM.LDB.STB                BC/188
           DATA      IXP.RBP.CBP.EQUI               C0/192
           DATA      GEQI.GTRI.LLA.LDCI             C4/196
           DATA      LEQI.LESI.LDL.NEQI             C8/200
           DATA      STL.CXP.CLP.CGP                CC/204
           DATA      S1P.IXB.BYT.EFJ                D0/208
           DATA      NFJ.BPT.XIT.NOP                D4/212
           DATA      SLDLS.SLDLS.SLDLS.SLDLS        D8/216
           DATA      SLDLS.SLDLS.SLDLS.SLDLS        DC/220
           DATA      SLDLS.SLDLS.SLDLS.SLDLS        E0/224
           DATA      SLDLS.SLDLS.SLDLS.SLOLS        E4/228
           DATA      SLDOS.SLDOS.SLDOS.SLDOS        E8/232
           DATA      SLDOS.SLDOS.SLDOS.SLDOS        EC/236
           DATA      SLDOS.SLDOS.SLDOS.SLDOS        F0/240
           DATA      SLDOS.SLDOS.SLDOS.SLDOS        F4/244
           DATA      SINDS.SINDS.SINDS.SINDS        F8/248
           DATA      SINDS.SINDS.SINDS.SINDS        FC/252
•
••• END OF TRANSFER TABLE •••

           END
```

```
           IDENT MODL02
           ENTRY     JTAB.SEG.STKBAS.LASTMP.OLDSEG
$INTCONST:

••• OTHER P-MACHINE POINTERS •••
•                                          POINTER TO FIRST WORD OF PROCEDURE DICTIONARY
JTAB       DATA      NIL                   POINTER TO FIRST WORD OF SEGMENT DICTIONARY
SEG        DATA      NIL
•
•••

•••   SYSCOM AREA   •••
•                                          PERMANENT BASE REGISTER
STKBAS     DATA      NIL                   PERMANENT MP REGISTER
LASTMP     DATA      NIL                   PERMANENT SEG POINTER
OLDSEG     DATA      NIL
```

```
*
***

          END
```

```
          IDENT MODL03
          ENTRY     ABI.ADI.DVI.MOD.MPI.SQI.NGI.SBI.CHK
          EXTRN     OVRFLW.INVNDX
8REGASG:

***   INTEGER ARITHMETIC   ***
*
ABI       EQU       *                             INTEGER ABSOLUTE VALUE
          LDR*      A10.SP
          RF(NN)    PLUS                          IF POSITIVE THEN BRANCH
          C2R       SP                            ELSE TAKE ABSOLUTE VALUE
          ABL(N)    OVRFLW                        IF EQUAL TO -32768 THEN OVERFLOW
PLUS      LDR       P.GOBACK                      POINT PC TO NEXT INSTRUCTION
*
ADI       EQU       *                             INTEGER ADDITION
          LDR*      A10.SP
          ADK       SP.2
          ADRS      A10.SP
          LDR       P.GOBACK
*
DVI       EQU       *                             INTEGER DIVISION
          LDR*      A10.SP
          ADK       SP.2
          XRR       A1.A1                         CLEAR A1
          LDR*      A2.SP                         A2 CONTAINS DIVIDEND
          RF(Z)     ZERO1                         IF ZERO THEN BRANCH
          RF(P)     PLUS1                         ELSE IF POSITIVE THEN BRANCH
          ORKL      A1./FFFF                      ELSE EXTEND SIGN BIT
PLUS1     DVR       A10                           NOTE:DIVISOR IS IN A10
ZERO1     STR       A2.SP                         INTEGER QOUTIENT IN A2
          LDR       P.GOBACK
*
MOD       EQU       *                             REMAINDER OF INTEGER DIVISION
          LDR*      A10.SP
          ADK       SP.2
          XRR       A1.A1
          LDR*      A2.SP
          RF(Z)     ZERO2
          RF(P)     PLUS2
          ORKL      A1./FFFF
PLUS2     DVR       A10
          ANKL      A1./7FFF                      CLEAR SIGN BIT OF REMAINDER
ZERO2     STR       A1.SP                         REMAINDER IN A1
          LDR       P.GOBACK
*
NGI       EQU       *                             INTEGER NEGATION
          C2R       SP
          LDR       P.GOBACK
*
SBI       EQU       *                             INTEGER SUBTRACTION
          LDR*      A10.SP
          ADK       SP.2
          LDR*      A11.SP
          SUR       A11.A10
          STR       A11.SP
          LDR       P.GOBACK
*
SQI       EQU       *                             SQUARE INTEGER
          LDR*      A10.SP
          SUK       SP.2                          ADJUST SP TO POINT OTO TOS+1
          STR       A10.SP                        COPY INTEGER TO BE SQUARED INTO STACK
          RF        MPI                           BRANCH TO INTEGER MULTIPLY: TRICKY!
*
CHK       EQU       *                             CHECK INDEX AGAINST RANGE
          LDR*      A10.SP                        'POP' MAXIMUM INDEX
          ADK       SP.2
          LDR*      A11.SP                        'POP' MINIMUM INDEX
          ADK       SP.2
          CUR*      A10.SP                        IF ((SP)) > MAX INDEX THEN
          ABL(L)    INVNDX                        BRANCH TO INVALID INDEX ROUTINE
          CUR*      A11.SP                        IF ((SP)) < MIN INDEX THEN
          ABL(G)    INVNDX                        BRANCH TO INVALID INDEX ROUTINE
          LDR       P.GOBACK
*
MPI       EQU       *                             INTEGER MULTIPLY
          LDR*      A10.SP                        'POP' MULTIPLIER
          ADK       SP.2
          LDR*      A2.SP                         'POP' MULTIPLICAND
          MUR       A10                           A1.A2 CONTAIN THE PRODUCT
          RF(N)     MINUS3                        IF PRODUCT IS NEGATIVE THEN BRANCH
          CWK       A1./0000
          ABL(NE)   OVRFLW                        POSITIVE OVERFLOW
          STR       A2.SP                         'PUSH' PRODUCT INTO STACK
MORE3     LDR       P.GOBACK                      GO ON W/ INTERPRETATION
MINUS3    CWK       A1./FFFF
          ABL(NE)   OVRFLW                        NEGATIVE OVERFLOW
          ORKL      A2./8000                      RESTORE SIGN
          RB        MORE3
*
*** END INTEGER ARITHMETIC ***

          END
```

```
          IDENT MODL04
          ENTRY     EQUI.NEQI.LEQI.LESI.GEQI.GTRI
8REGASG:

***   INTEGER COMPARE   ***
*
EQUI      EQU       *                             INTEGER EQUAL COMPARE
          LDR*      A10.SP                        'POP' TOP-OF-STACK (TOS)
          ADK       SP.2
          LDR*      A11.SP                        'POP' TOS-1
          CWR       A11.A10
          RF(E)     PSHTRU
PSHFLS    CMR       SP
          LDR       P.GOBACK                      GO ON W/ INTERPRETATION
PSHTRU    LDKL      A10.1
          STR       A10.SP
          LDR       P.GOBACK                      GO ON W/ INTERPRETATION
*
NEQI      EQU       *                             INTEGER NOT EQUAL COMPARE
          LDR*      A10.SP
          ADK       SP.2
          LDR*      A11.SP
          CWR       A11.A10
          RB(NE)    PSHTRU
          RB        PSHFLS
*
LEQI      EQU       *                             INTEGER LESS OR EQUAL COMPARE
          LDR*      A10.SP
          ADK       SP.2
          LDR*      A11.SP
          CWR       A11.A10
          RB(NG)    PSHTRU
```

```
            RB        PSHFLS
•
LESI        EQU       •                             INTEGER LESS THAN COMPARE
            LDR•      A10.SP
            ADK       SP.2
            LDR•      A11.SP
            CWR       A11.A10
            RB(L)     PSHTRU
            RB        PSHFLS
•
GEQI        EQU       •                             INTEGER GREATER OR EQUAL COMP
            LDR•      A10.SP
            ADK       SP.2
            LDR•      A11.SP
            CWR       A11.A10
            RB(NL)    PSHTRU
            RB        PSHFLS
•
GTRI        EQU       •                             INTEGER GREATER THAN COMPARE
            LDR•      A10.SP
            ADK       SP.2
            LDR•      A11.SP
            CWR       A11.A10
            RB(G)     PSHTRU
            RB        PSHFLS
•
••• END INTEGER COMPARE •••

            END
```

```
            IDENT MODL05
            ENTRY     AND.IOR.NOT.COMPAR
            EXTRN     XFRTBL.NOTIMP
$REGASC:

•••   BOOLEAN ARITHMETIC   •••
•
AND         EQU       •                             LOGICAL AND
            LDR•      A10.SP
            ADK       SP.2
            ANRS      A10.SP
            LDR       P.GOBACK                      CONTINUE INTERPRETATION
•
IOR         EQU       •                             LOGICAL OR
            LDR•      A10.SP
            ADK       SP.2
            ORRS      A10.SP
            LDR       P.GOBACK
•
NOT         EQU       •                             LOGICAL NOT
            C1RS      SP
            LDR       P.GOBACK
•
••• END BOOLEAN ARITHMETIC •••

•••   COMPLEX COMPARE   •••
•
•   THE FOLLOWING IS A TABLE OF INDICES
•
CMPTBL      DATA      0
            DATA      REALCM
            DATA      STRGCM
            DATA      BOOLCM
            DATA      POWRCM
            DATA      BYTECM
            DATA      WORDCM
•
•
COMPAR      EQU       •                             COMPARE COMPLEX THINGS
            LDR•      A11.IPC                       GET COMPARISON TYPE
            ADK       IPC.1
            LD        P.CMPTBL.A11                  TRANSFER TO PROPER CODE
•
•
BOOLCM      EQU       •                             COMPARE BOOLEANS
            LDKL      A11./0001
            ADK       SP.2                          POINT SP TO TOS-1
            ANRS      A11.SP                        GET ONLY LS BIT
            SUK       SP.2                          POINT SP BACK TO TOS
            ANRS      A11.SP
            LD        P.XFRTBL+40.A10               BRANCH TO INTEGER COMP
•
•
•  THE FOLLOWING ARE NOT YET IMPLEMENTED
•
REALCM      EQU       •                             COMPARE REALS
            ABL       NOTIMP
•
STRGCM      EQU       •                             COMPARE STRINGS
            ABL       NOTIMP
•
POWRCM      EQU       •                             COMPARE SETS
            ABL       NOTIMP
•
BYTECM      EQU       •                             COMPARE BYTES
            ABL       NOTIMP
•
WORDCM      EQU       •                             COMPARE WORDS
            ABL       NOTIMP
•
••• END COMPLEX COMPARISONS •••

            END
```

```
            IDENT MODL06
            ENTRY     BACK.SLDC.LDCI.LDCN.SLDLS.SLDOS.SINDS
            EXTRN     XFRTBL
$REGASC:
$INTCONST:

•••   FETCH   •••
•
•  ENTER HERE TO FETCH NEXT P-MACHINE INSTRUCTION
•    SLDC IS INCORPORATED WITH THE FETCH
•
SLDC        EQU       •                             SHORT LOAD CONSTANT
            SUK       SP.2
            STR       A10.SP                        'PUSH' CONSTANT INTO STACK
BACK        XRR       A10.A10                       CLEAR A10
            LCR       A10.IPC                       FETCH P-CODE BYTE
            ADK       IPC.1
            LDKL      A11./0080
            TM        A10.A11                       TEST BYTE'S SIGN BIT
            RB(D)     SLDC                          IF NOT SET THEN BRANCH TO SLDC
            ADR       A10.A10                       DOUBLE FOR WORD INDEXING
            SUKL      A10.256                       CALCULATE EFFECTIVE INDEX: 2•P-CODE - 2•128
            LD        P.XFRTBL.A10                  TRANSFER CONTROL TO PROPER SECTION
•
••• END OF FETCH •••
```

```

*** CONSTANT ONE-WORD LOADS ***
*
LDCI     EQU      *                LOAD CONSTANT WORD OR LONG INTEGER CONSTANT
         XRR      A10.A10
         LCR      A10.IPC          GET LS BYTE
         ADK      IPC.1
         XRR      A11.A11
         LCR      A11.IPC          GET MS BYTE
         ADK      IPC.1
         LDR      A1.A11           PREPARE FOR SHIFTING
         SLA      A1.8             PUT MS BYTE IN PROPER PLACE
         LDR      A11.A1           PUT BACK INTO A11
         ORR      A10.A11          CONCATENATE MS &LS BYTES
         SUK      SP.2
         STR      A10.SP           'PUSH' WORD
         LDR      P.GOBACK         CONTINUE INTERPRETATION
*
LDCN     EQU      *                LOAD CONSTANT NIL POINTER
         LDKL     A10.NIL
         SUK      SP.2
         STR      A10.SP
         LDR      P.GOBACK
*
*** END CONSTANT ONE-WORD LOADS ***

*** MOST COMMON P-CODES ***
*
SLDLS    EQU      *                SHORT LOAD LOCAL VAR
         SUKL     A10.174          (2*P-CODE - 256) - (2*215 - 256)
         ADKL     A10.MSDLTA       OFFSET DUE TO MSCW
         ADR      A10.MP           A10 NOW POINTS TO VAR
         LDR*     A10.A10          PUT VAR INTO A10
         SUK      SP.2
         STR      A10.SP           'PUSH' VAR
         LDR      P.GOBACK
*
SLDOS    EQU      *    .           SHORT LOAD GLOBAL VAR
         SUKL     A10.206          (2*P-CODE - 256) - (2*231 - 256)
         ADKL     A10.MSDLTA
         ADR      A10.BASE
         LDR*     A10.A10
         SUK      SP.2
         STR      A10.SP
         LDR      P.GOBACK
*
SINDS    EQU      *                SHORT INDEX AND LOAD WORD
         SUKL     A10.240          (2*P-CODE - 256) - (2*248 - 256)
         LDR*     A11.SP           GET ADDRESS TO BE INDEXED
         ADR      A10.A11          A10 POINTS TO VAR
         LDR*     A10.A10
         STR      A10.SP
         LDR      P.GOBACK
*
*** END MOST COMMON P-CODES ***

         END
```

```
         IDENT MODL07
         ENTRY    EFJ.NFJ.FJP.UJP.XJP
         EXTRN    JTAB
$REGASG;

***   JUMPS   ***
*
EFJ      EQU      *                EQUAL FALSE JUMP
         LDR*     A10.SP
         ADK      SP.2
         LDR*     A11.SP
         ADK      SP.2
         CWR      A10.A11
         RF(E)    NOJUMP
         RF       UJP              NOT EQUAL THEN JUMP
*
NFJ      EQU      *                NOT EQUAL FALSE JUMP
         LDR*     A10.SP
         ADK      SP.2
         LDR*     A11.SP
         ADK      SP.2
         CWR      A10.A11
         RF(NE)   NOJUMP
         RF       UJP              EQUAL THEN JUMP
*
FJP      EQU      *                JUMP IF TOS IS FALSE
         LDR*     A10.SP
         ADK      SP.2
         LDKL     A11.1
         TM       A10.A11          TEST IF TRUE
         RF(O)    UJP              FALSE THEN JUMP
NOJUMP   ADK      IPC.1            SKIP OVER JUMP OFFSET
         LDR      P.GOBACK         CONTINUE INTERPRETATION
*
UJP      EQU      *                UNCONDITIONAL JUMP
         XRR      A10.A10
         LCR      A10.IPC          GET JUMP OFFSET
         ADK      IPC.1
         LDKL     A11./0080
         TM       A10.A11          TEST BYTE SIGN BIT
         RF(4)    LONGJP           IF NEG THEN LONGJP
         ADR      IPC.A10          ELSE ADD OFFSET TO IPC
         LDR      P.GOBACK         CONTINUE INTERPRETATION
LONGJP   LD       IPC.JTAB         GET PROC. DICT.P POINTER
         ORKL     A10./FF00        EXTEND SIGN OF OFFSET
         ADR      IPC.A10          ACTUALLY A SUBTRACTION
         SUR*     IPC.IPC          IPC NOW POINTS TO NEXT CODE
         LDR      P.GOBACK         CONTINUE INTERPRETATION
*
XJP      EQU      *                CASE OR INDEX JUMP
         ADK      IPC.1
         ANKL     IPC./FFFE        ADJUST TO WORD BOUND.
         LDR*     A10.SP           GET INDEX
         ADK      SP.2
         LDR*     A11.IPC          GET MIN CASE INDEX
         ADK      IPC.2            NOTE WORD LENGTH
         CWR      A10.A11
         RF(L)    MINERR           INDEX<MIN GOTO MINERR
         LDR*     A12.IPC          GET MAX CASE INDEX
         ADK      IPC.2
         CWR      A10.A12
         RF(G)    MAXERR           INDEX>MAX GOTO MAXERR
         ADK      IPC.2            SKIP OVER ELSE JUMP WORD
         SUR      A10.A11          ADJUST INDEX TO 0..N
         ADR      A10.A10          DOUBLE INDEX FOR WORD
         ADR      IPC.A10          IPC POINTS AT PROPER JTAB INDX
         SUR*     IPC.IPC          IPC POINTS TO PROPER CODE
         LDR      P.GOBACK         CONTINUE INTERPRETATION
MINERR   ADK      IPC.2            POINT IPC AT ELSE JUMP LOC
MAXERR   LDR      P.GOBACK         IPC ALREADY POINTS AT ELSE LOC
*
***   END OF JUMPS   ***

         END
```

```
           IDENT MODL08
           ENTRY     GETBIG.LLA.LDL.STL.LAO.LDO.SRO
           EXTRN     CFSTAK.PRINT
$REGASC;
$INTCONST;

•••   GETBIG   •••
•
GETBIG     EQU       •                             GET PARAM W/C IS EITHER 1-BYTE OR 2-BYTES
           XRR       A10.A10
           LCR       A10.IPC                       GET LS BYTE
           ADK       IPC.1
           LDKL      A12./0080
           TM        A10.A12                       TEST LS BYTE'S SIGN BIT
           RF(0)     NOTBIG                        IF POSITIVE THEN BRANCH
           ANKL      A10./FF7F                     ELSE CLEAR SIGN BIT
           LDR       A1.A10                        PREPARE FOR SHIFTING
           SLA       A1.8                          CONVERT AS MS BYTE
           LDR       A10.A1                        PUT BACK IN A10
           XRR       A12.A12
           LDR•      A12.IPC                       GET ACTUAL LS BYTE
           ADK       IPC.1
           ORR       A10.A12                       CONCATENATE MS &LS BYTES
NOTBIG     LDR       P.RETURN                      GO ON WITH PREVIOUS INSTRUCTION
•
••• END  GETBIG •••

•••   LOCAL STORE AND LOADS   •••
•
LLA        EQU       •                             LOAD LOCAL ADDRESS
           LDKL      RETURN.ENDLLA
           RB        GETBIG                        GET DISPLACEMENT
ENDLLA     ADR       A10.A10                       DOUBLE FOR WORD INDEXING
           ADKL      A10.MSDLTA                    OFFSET CORRECTION
           ADR       A10.MP                        A10 CONTAINS ADDRESS
           SUK       SP.2
           STR       A10.SP                        'PUSH' ADDRESS
           LDR       P.GOBACK                      CONTINUE INTERPRETATION
•
LDL        EQU       •                             LOAD LOCAL WORD
           LDKL      RETURN.ENDLDL
           RB        GETBIG                        GET DISPLACEMENT
ENDLDL     ADR       A10.A10
           ADKL      A10.MSDLTA
           ADR       A10.MP                        A10 CONTAINS ADDRESS
           LDR•      A10.A10                       A10 NOW CONTAINS THE WORD
           SUK       SP.2
           STR       A10.SP
           LDR       P.GOBACK
•
STL        EQU       •                             STORE LOCAL WORD
           LDKL      RETURN.ENDSTL
           RB        GETBIG
ENDSTL     ADR       A10.A10
           ADKL      A10.MSDLTA
           ADR       A10.MP                        A10 CONTAINS ADDRESS
           LDR•      A11.SP                        'POP' WORD
           ADK       SP.2
           STR       A11.A10                       STORE WORD
           LDR       A2.A11                        COPY A11 AS PARAMETER FOR CALL
           LDKL      RETURN.CFSTAK
           CF        RETURN.PRINT                  PRINT VALUE AS HEXADEC
           LDR       P.GOBACK
•
••• END  LOCAL STORE AND LOADS •••

•••   GLOBAL STORE AND LOADS   •••
•
LAO        EQU       •                             LOAD GLOBAL ADDRESS
           LDKL      RETURN.ENDLAO
           RB .      GETBIG
ENDLAO     ADR       A10.A10
           ADKL      A10.MSDLTA
           ADR       A10.BASE                      CHANGE: BASE INSTEAD OF MP
           SUK       SP.2
           STR       A10.SP
           LDR       P.GOBACK
•
LDO        EQU       •                             LOAD GLOBAL WORD
           LDKL      RETURN.ENDLDO
           RB        GETBIG
ENDLDO     ADR       A10.A10
           ADKL      A10.MSDLTA
           ADR       A10.BASE                      NOTE CHANGE!
           LDR•      A10.A10
           SUK       SP.2
           STR       A10.SP
           LDR       P.GOBACK
•
SRO        EQU       •                             STORE GLOBAL WORD
           LDKL      RETURN.ENDSRO
           RB        GETBIG
ENDSRO     ADR       A10.A10
           ADKL      A10.MSDLTA
           ADR       A10.BASE                      NOTE ONLY ALTERATION!
           LDR•      A11.SP
           ADK       SP.2
           STR       A11.A10
           LDR       A2.A11
           LDKL      RETURN.CFSTAK
           CF        RETURN.PRINT
           LDR       P.GOBACK
•
••• END GLOBAL STORE AND LOADS •••

           END
```

```
           IDENT MODL09
           ENTRY     STO.LDA.LOD.STR
           EXTRN     GETBIG.CFSTAK.PRINT
$REGASC;
$INTCONST;

•
STO        EQU       •                             STORE INDIRECT
           LDR•      A10.SP                        'POP' WORD
           ADK       SP.2
           LDR•      A11.SP                        'POP' ADDRESS
           ADK       SP.2
           STR       A10.A11                       STORE WORD INTO ADDRESS
           LDR       A2.A10                        COPY A10 AS PARAMETER FOR CALL
           LDKL      RETURN.CFSTAK
           CF        RETURN.PRINT
           LDR       P.GOBACK
•

•••   INTERMEDIATE STORE AND LOADS   •••
•
LDA        EQU       •                             LOAD INTERMEDIATE ADDRESS
           XRR       A10.A10
           LCR       A10.IPC                       GET DELTA LEX. LEVELS
           ADK       IPC.1
```

```
LOOP     LDR      A11.MP                                POINT A11 AT STAT LINKS
         LDR*     A11.A11                               LINK DOWN UNTIL
         SUKL     A10.1                                 DELTA LEX LEVELS = 0
         RB(NZ)   LOOP
         LDKL     RETURN.ENDLDA
         ABL      GETBIG                                GET DISPLACEMENT
ENDLDA   ADR      A10.A10                               DOUBLE FOR WORD INDEXING
         ADKL     A10.MSDLTA                            OFFSET CORRECTION
         ADR      A10.A11                               A10 CONTAINS ADDRESS
         SUK      SP.2
         STR      A10.SP                                'PUSH' ADDRESS
         LDR      P.GOBACK
*
LOD      EQU      *                                     LOAD INTEREMEDIATE WORD
         XRR      A10.A10
         LCR      A10.IPC
         ADK      IPC.1
LOOP1    LDR      A11.MP
         LDR*     A11.A11
         SUKL     A10.1
         RB(NZ)   LOOP1
         LDKL     RETURN.ENDLOD
         ABL      GETBIG
ENDLOD   ADR      A10.A10
         ADKL     A10.MSDLTA
         ADR      A10.A11
         LDR*     A10.A10
         SUK      SP.2
         STR      A10.SP
         LDR      P.GOBACK
*
STR      EQU      *                                     STORE INTERMEDIATE WORD
         XRR      A10.A10
         LCR      A10.IPC
         ADK      IPC.1
LOOP2    LDR      A11.MP
         LDR*     A11.A11
         SUKL     A10.1
         RB(NZ)   LOOP2
         LDKL     RETURN.ENDSTR
         ABL      GETBIG
ENDSTR   ADR      A10.A10
         ADKL     A10.MSDLTA
         ADR      A10.A11
         LDR*     A11.SP
         ADK      SP.2
         STR      A11.A10
         LDR      A2.A11
         LDKL     RETURN.CFSTAK
         CF       RETURN.PRINT
         LDR      P.GOBACK
*
*** END INTERMEDIATE STORE AND LOADS ***

         END
```

```
         IDENT MODL10
         ENTRY    RBP.RNP
         EXTRN    STKBAS.SEG.JTAB.LASTMP.NOTIMP
$REGASG:
$MSCWFORM:
$INTCONST:

***   PROCEDURE RETURNS   ***
*
RBP      EQU      *                                     RETURN FROM BASE LEVEL PROCEDURE
         LDR      A10.MP
         ADKL     A10.MSBASE                            A10 HAS BASE FROM MSCW
         LDR      BASE.A10                              PUT IN BASE REGISTER
         ST       BASE.STKBAS                           SAVE IN PERM SYSCOM WORD
RNP      EQU      *                                     RETURN FROM NORMAL PROCEDURE
         LD*      A10.SEG                               GET SEGMENT #
         ANKL     A10./00FF
         LDR      A11.MP
         ADKL     A11.MSSEG                             A11 POINTS MSCW'S MSSEG
         LDR*     A11.A11                               A11 NOW POINTS TO OLD SEGMENT
         LDR*     A11.A11                               GET OLD SEG #
         ANKL     A11./00FF
         CWR      A10.A11                               SAME SEGMENT?
         RF(E)    SAMSEG                                IF YES THEN GOTO SAMSEG
         ABL      NOTIMP                                ELSE GOTO NOT IMPLEMENTED
SAMSEG   LDR      A10.MP
         ADKL     A10.MSSP                              A10 HAS SP FROM MSCW
         XRR      A11.A11
         LCR      A11.IPC                               GET # OF WORDS TO RETURN
         ADK      IPC.1
         CWK      A11.0
         RF(E)    RETCOD                                IF EQUAL THEN BRANCH TO RETURN CODE
         ADK      MP.MSDLTA
         ADK      MP.2                                  THIS POINTS MP ABOVE FUNCTION VALUE
         ADR      MP.A11                                DO THIS TWICE FOR WORD INDEXING
         ADR      MP.A11                                MP POINTS TO WHERE WORDS ARE TO BE IN
LOOP3    SUK      MP.2
         LDR*     A12.MP                                A12 HAS WORD TO BE RETURNED
         SUKL     A10.2                                 A10 IS OLD SP
         STR      A12.A10                               'PUSH' RETURN WORD INTO STACK
         SUKL     A11.1
         RB(NZ)   LOOP3                                 DO THIS FOR TOTAL # OF WORDS
         LD       MP.LASTMP                             RESTORE OLD MP VALUE
RETCOD   LDR      A11.MP                                RESTORE STATE FROM MSCW
         ADKL     A11.2                                 SKIP OVER STAT LINK
         LDR*     MP.A11                                DYNAMIC LINK
         ADKL     A11.2
         LDR*     A12.A11                               JTAB
         ST       A12.JTAB                              STORE IN PERM REGISTER WORD
         ADKL     A11.2
         LDR*     A12.A11                               SEG
         ST       A12.SEG                               STORE IN PERM REGISTER WORD
         ADKL     A11.2
         LDR*     IPC.A11                               IPC
         ADKL     A11.2
         ST       MP.LASTMP                             STORE IN PERM SYSCOM WORD
         LDR      SP.A10                                PREVIOUS STATE RESTORED
         LDR      P.GOBACK                              CONTINUE INTERPRETATION
*
*** END PROCEDURE RETURNS ***

         END
```

```
         IDENT MODL11
         ENTRY    CLP.CGP.CBP.CIP
         EXTRN    NOTIMP.STKOVF.SEG.OLDSEG.JTAB.LASTMP.STKBAS.BACK
$REGASG:
$SEGFOR:
$MSCWFORM:
$INTCONST:

TEMPMP   RES      1                                     TEMPORARY STORE FOR COMPARISON

***   PROCEDURE CALLS   ***
```

```
*
CLP       EQU       *                          CALL LOCAL PROCEDURE
          LD        A10.SEG
          ST        A10.OLDSEG                 USUALLY NO SEGMENT CHANGE
          LDR       A10.SP                     A10 HAS VALUE OF SP
XCLP      EQU       *                          ENTRY FOR EXTRN CALLS. A10 & OLDSEG DIFFERENT
          XRR       A11.A11
          LCR       A11.IPC                    GET PROCEDURE NUMBER
          ADK       IPC.1
          ADR       A11.A11                    DOUBLE FOR WORD INDEXING
          NGR       A11.A11                    ENSURE IT IS NEGATIVE
          AD        A11.SEG                    A11 POINTS AT SEGTABLE ENTRY FOR PROC
          SUR*      A11.A11                    A11 NOW POINTS TO PROC DICTIONARY
          LDR       A7.A11                     HAVE COPY OF JTAB
          ADKL      A11.1                      ADJUST TO POINT TO CORRECT BYTE
          LDKL      A12.0
          CCR       A12.A11                    CHECK IF PROC NO IS 0
          ABL(E)    NOTIMP
          SUKL      A11.1
          ADKL      A11.DATASZ                 POINT A11 TO DATA SIZE WORD
          SUR*      SP.A11                     RESERVE SPACES FOR DATA
          ST        NP.TEMPNP                  CHECK FOR ENOUGH SPACE IN STACK
          ECR       A1.SP                      EXCHANGED CHAR POS. OF SP IN A1
          CC        A1.TEMPNP                  COMPARE MS BYTES
          ABL(L)    STKOVF
          RF(E)     MORECC
          RF        NOCARE
MORECC    CC        SP.TEMPNP+1                COMPARE LS BYTES
          ABL(NG)   STKOVF
NOCARE    SUK       SP.2                       SPACE FOR SAVING SP
          SUK       SP.2                       BUILD MSCW
          STR       IPC.SP                     SAVE MSIPC
          SUK       SP.2
          LD        A12.OLDSEG                 GET PREVIOUS SEG
          STR       A12.SP                     SAVE MSSEG
          SUK       SP.2
          LD        A12.JTAB                   GET JTAB
          STR       A12.SP                     SAVE MSJTAB
          SUK       SP.2
          STR       MP.SP                      SAVE MSDYN
          SUK       SP.2
          STR       MP.SP                      SAVE MSSTAT
          LDR       A11.A7                     A11 IS ALSO JTAB NOW
          ADKL      A11.PARMSZ                 POINT A11 TO PARAM SIZE WORD
          LDR*      A1.A11                     A1 HAS # OF PARAMETERS
          RF(0)     NOPARM                     IF # OF PARAM = 0 THEN BRANCH
          SRA       A1.1                       HALVE # OF BYTES FOR # OF WORDS
          LDR       A2.SP                      SET UP A2 TO PARAM COPY PLACE
          ADK       A2.2
          ADK       A2.MSDLTA                  A2 NOW POINTS ABOVE MSCW
LOOP4     LDR*      A12.A10                    GET PARAMETER
          ADKL      A10.2
          STR       A12.A2                     COPY PARAM TO ITS NEW PLACE
          ADK       A2.2
          SUK       A1.1
          RB(NZ)    LOOP4                      UNTIL # OF PARAM = 0
NOPARM    LDR       MP.SP                      POINT MP AT STAT LINK
          ST        MP.LASTMP                  SAVE IN PERM. SYSCOM WORD
          LDR       A12.MP
          ADKL      A12.MSSP                   A12 POINTS TO MSCW'S MSSP
          STR       A10.A12                    SAVE OLD SP VALUE
          ST        A7.JTAB                    NEW JTAB POINTER
          LDR       IPC.A7                     POINT IPC TO FIRST BYTE OF CODE
          ADKL      IPC.ENTRIC
          SUR*      IPC.IPC
          LDR       P.GOBACK                   GO ON W/ INTERPRETATION
*
CGP       EQU       *                          CALL GLOBAL PROCEDURE
          LDKL      GOBACK.ENDCGP              SET UP FOR RETURN
          RB        CLP                        AND CALL LOCAL PROCEDURE
ENDCGP    STR       BASE.MP                    CHANGE STATIC LINK TO BASE
          LDKL      GOBACK.BACK                SET UP TO CONTINUE INTERPRETATION
          LDR       P.GOBACK                   GO ON W/ INTERPRETATION
*
CBP       EQU       *                          CALL BASE PROCEDURE
          LDKL      GOBACK.ENDCBP
          RB        CLP
ENDCBP    SUK       SP.2                       ADD ON EXTRA MSCW WORD
          STR       BASE.SP                    SAVE MSBASE
          LDR*      A12.BASE
          STR       A12.MP                     POINT STAT LINK AT OUTER BLOCK
          LDR       BASE.MP                    SET BASE REG TO THIS NEW PROC
          ST        BASE.STKBAS                STORE IN PERM SYSCOM WORD
          LDKL      GOBACK.BACK
          LDR       P.GOBACK                   CONTINUE INTERPRETATION
*
CIP       EQU       *                          CALL INTERMEDIATE PROCEDURE
          LDKL      GOBACK.ENDCIP
          RB        CLP
ENDCIP    XRR       A1.A1
          LCR       A1.A7                      GET LEX LEV OF CALLED PROC
          LDKL      A12./0080
          TM        A1.A12                     IF < 0 THEN BASE PROC CALL
          RB(4)     ENDCBP
          CCK       A1./0000                   IF = 0 THEN STILL BASE PROC CALL
          RB(0)     ENDCBP
          LDR       A10.MP
LOOP5     LDR       A12.A10
          ADKL      A12.MSJTAB                 GET MSCW'S JTAB
          LDR*      A11.A12                    A11 IS NOW JTAB
          CCR       A1.A11                     COMPARE LEX LEVS
          RF(L)     FOUND                      IF LESS THEN BRANCH
          ADKL      A10.MSDYN                  ELSE LINK DOWN
          LDR*      A10.A10                    TO CALLER OF CURRENT PROC
          RB        LOOP5
FOUND     LDR*      A12.A10                    GET DESIRED MP
          STR       A12.MP
          LDKL      GOBACK.BACK
          LDR       P.GOBACK                   CONTINUE INTERPRETATION
*
*** END PROCEDURE CALLS ***

          END
```

```
          IDENT MODL12
          ENTRY     ABR.ADR.DVR.FLO.FLT.MPR.NGR.SBR.SQR
          ENTRY     DIF.INN.INT.SRS.SGS.UNI.ADJ
          ENTRY     NOP.BPT.CXP.CSP
          ENTRY     LDM.STM.LDC
          ENTRY     BYT.LDB.STB.MVB.IXB
          ENTRY     LCA.SAS.S1P.S2P.IXS
          ENTRY     MOV.IND.INC.IXA.IXP.LDP.STP
          ENTRY     NOTIMP.STKOVF.OVRFLW.INVNDX.XIT

$REGASG:

***   NOT YET IMPLEMENTED P-CODES   ***
*

***   P-CODES FOR REALS   ***
```

```
*
ABR       EQU       *                                REAL ABSOLUTE VALUE
          RF        NOTIMP
*
ADR       EQU       *                                ADD REALS
          RF        NOTIMP
*
DVR       EQU       *                                DIVIDE REALS
          RF        NOTIMP
*
FLO       EQU       *                                FLOAT TOS-1
          RF        NOTIMP
*
FLT       EQU       *                                FLOAT TOS
          RF        NOTIMP
*
MPR       EQU       *                                MULTIPLY REALS
          RF        NOTIMP
*
NGR       EQU       *                                NEGATE REAL
          RF        NOTIMP
*
SBR       EQU       *                                SUBTRACT REALS
          RF        NOTIMP
*
SQR       EQU       *                                SQUARE REAL
          RF        NOTIMP
*
*** END P-CODES FOR REALS ***

***    P-CODES FOR SETS    ***
*
DIF       EQU       *                                SET DIFFERENCE
          RF        NOTIMP
*
INN       EQU       *                                SET MEMBERSHIP
          RF        NOTIMP
*
INT       EQU       *                                SET INTERSECTION
          RF        NOTIMP
*
SRS       EQU       *                                SUBRANGE SET
          RF        NOTIMP
*
SGS       EQU       *                                SINGLETON SET
          RF        NOTIMP
*
UNI       EQU       *                                SET UNION
          RF        NOTIMP
*
ADJ       EQU       *                                ADJUST SET
          RF        NOTIMP
*
*** END P-CODES FOR SETS ***

***    P-CODES FOR MULTIPLE WORD LOADS & STORES    ***
*
LDM       EQU       *                                LOAD MULTIPLE WORDS
          RF        NOTIMP
*
STM       EQU       *                                STORE MULTIPLE WORDS
          RF        NOTIMP
*
LDC       EQU       *                                LOAD MULTIPLE WORD CONSTANT
          RF        NOTIMP
*
*** END P-CODES FOR MULTIPLE WORD LOADS & STORES ***

***    P-CODES FOR BYTE ARRAYS    ***
*
BYT       EQU       *                                BYTE CONVERSION
          RF        NOTIMP
*
LDB       EQU       *                                LOAD BYTE
          RF        NOTIMP
*
STB       EQU       *                                STORE BYTE
          RF        NOTIMP
*
MVB       EQU       *                                MOVE BYTE
          RF        NOTIMP
*
IXB       EQU       *                                INDEX BYTE ARRAY
          RF        NOTIMP
*
*** END P-CODES FOR BYTE ARRAYS ***

***    P-CODES FOR STRINGS    ***
*
LCA       EQU       *                                LOAD CONSTANT STRING ADDRESS
          RF        NOTIMP
*
SAS       EQU       *                                STRING ASSIGN
          RF        NOTIMP
*
S1P       EQU       *                                STRING TO PACKED CONV. ON TOS
          RF        NOTIMP
*
S2P       EQU       *                                STRING TO PACKED CONV. ON TOS-1
          RF        NOTIMP
*
IXS       EQU       *                                INDEX STRING ARRAY
          RF        NOTIMP
*
*** END P-CODES FOR STRINGS ***

***    P-CODES FOR RECORD & ARRAY INDEXING & ASSIGNMENT    ***
*
MOV       EQU       *                                MOVE WORDS
          RF        NOTIMP
*
IND       EQU       *                                STATIC INDEX AND LOAD WORD
          RF        NOTIMP
*
INC       EQU       *                                INCREMENT FIELD POINTER
          RF        NOTIMP
*
IXA       EQU       *                                INDEX ARRAY
          RF        NOTIMP
*
IXP       EQU       *                                INDEX PACKED ARRAY
          RF        NOTIMP
*
LDP       EQU       *                                LOAD A PACKED FIELD
          RF        NOTIMP
*
STP       EQU       *                                STORE INTO A PACKED FIELD
          RF        NOTIMP
*
*** END P-CODES FOR RECORD & ARRAY INDEXING & ASSIGNMENT ***

***    MISCELLANEOUS P-CODES    ***
```

```
*
BPT       EQU      *                            BREAKPOINT
          RF       NOTIMP
*
CXP       EQU      *                            CALL EXTERNAL PROCEDURE
          RF       NOTIMP
*
CSP       EQU      *                            CALL STANDARD PROCEDURE
          RF       NOTIMP
*
*** END MISCELLANEOUS P-CODES ***

*
*** END NOT YET IMPLEMENTED P-CODES ***

***   CONTINGENCY ROUTINES   ***
*
ECB1      DATA     2.MESSG1.28.0.0.0
MESSG1    DATA     /0A0D                        CARRIAGE RETURN LINE FEED
          DATA     'P-CODE NOT YET IMPLEMENTED'
ECB2      DATA     2.MESSG2.26.0.0.0
MESSG2    DATA     /0A0D
          DATA     'P-MACHINE STACK OVERFLOW'
ECB3      DATA     2.MESSG3.18.0.0.0
MESSG3    DATA     /0A0D
          DATA     'INTEGER OVERFLOW'
ECB4      DATA     2.MESSG4.20.0.0.0
MESSG4    DATA     /0A0D
          DATA     'INDEX OUT OF RANGE'
*
NOTIMP    EQU      *                            NOT YET IMPLEMENTED P-CODE
          LDK      A7./85                       PREPARE FOR PRINTING
          LDKL     A8.ECB1
          LKM
          DATA     1
          RF       XIT
*
STKOVF    EQU      *                            STACK OVERFLOW
          LDK      A7./85
          LDKL     A8.ECB2
          LKM
          DATA     1
          RF       XIT
*
OVRFLW    EQU      *                            REGISTER OVERFLOW
          LDK      A7./85
          LDKL     A8.ECB3
          LKM
          DATA     1
          RF       XIT
*
INVNDX    EQU      *                            INVALID INDEX
          LDK      A7./85
          LDKL     A8.ECB4
          LKM
          DATA     1
          RF       XIT
*
*** END CONTINGENCY ROUTINES ***

***   EXIT & NO OPERATION   ***
*
XITECB    DATA     2.XITMSG.24.0.0.0
XITMSG    DATA     /0A0D
          DATA     'END P-CODE INTERPRETER'
*
NOP       EQU      *                            NO OPERATION
          LDR      P.GOBACK                     CONTINUE INTERPRETATION
*
XIT       EQU      *                            EXIT FROM INTERPRETER TO DOM
          LDK      A7./85
          LDKL     A8.XITECB
          LKM
          DATA     1
          LKM
          DATA     3
*** END EXIT ***

          END
```

APPENDIX D. Implemented P-code Instructions

MNEMONIC	OP-CODE	PARAMETERS	FULL NAME AND OPERATION

I. ONE-WORD FETCHING, STORING AND INDEXING

I.A CONSTANT ONE-WORD LOADS

MNEMONIC	OP-CODE	PARAMETERS	FULL NAME AND OPERATION
SLDC0 .. SLDC127 (Pascal version) SLDCS (P-857 version)	0..127		Short load word constant.
LDCN	159		Load constant nil. (Not implemented in Pascal version.)
LDCI	199	W	Load constant word.

I.B LOCAL ONE-WORD LOADS AND STORE

MNEMONIC	OP-CODE	PARAMETERS	FULL NAME AND OPERATION
SLDL1 .. SLDL16 (Pascal) SLDLS (P-857)	216 .. 231		Short load local word.

MNEMONIC	OP-CODE	PARAMETERS	FULL NAME AND OPERATION
LDL	202	B	Load local word.
LLA	198	B	Load local address.
STL	204	B	Store local word.

I.C GLOBAL ONE-WORD LOADS AND STORE

SLDO1 .. SLDO16 (Pascal) SLDOS (P-857)	232 .. 247		Short load global word.
LDO	167	B	Load global word.
LAO	165	B	Load global address.
SRO	171	B	Store global word.

I.D INTERMEDIATE ONE-WORD LOADS AND STORE

LOD	182	DB,B	Load intermediate word.
LDA	178	DB,B	Load intermediate address.
STR	184	DB,B	Store intermediate word.

I.E INDIRECT ONE-WORD LOADS AND STORE

SINDO .. SIND7 (Pascal) SINDS (P-857)	248 .. 255		Short index and load word.
STO	154		Store indirect.

II. TOP-OF-STACK ARITHMETIC AND COMPARISONS

II.A LOGICAL (BOOLEAN)

II.A.1 LOGICAL ARITHMETIC

LAND (Pascal) AND (P-857)	132		Logical and
LOR (Pascal) IOR (P-857)	141		Logical or
LNOT (Pascal) NOT (P-857)	147		Logical not

II.A.2 LOGICAL COMPARISONS

EQU (Pascal) COMPAR (P-857)	175 6		Boolean = comparison.
NEQ (Pascal) COMPAR (P-857)	183 6		Boolean <> comparison.
LEQ (Pascal) COMPAR (P-857)	180 6		Boolean <= comparison.
LES (Pascal) COMPAR (P-857)	181 6		Boolean < comparison.
GEQ (Pascal)	176 6		Boolean >= comparison.

MNEMONIC	OP-CODE	PARAMETERS	FULL NAME AND OPERATION
COMPAR (P-857)			
GTR (Pascal) COMPAR (P-857)	177 6		Boolean > comparison.

II.8 INTEGER

II.8.1 INTEGER ARITHMETIC

ABI	128		Integer absolute value.
ADI	130		Integer addition.
NGI	145		Integer negation.
SBI	149		Integer subtraction.
MPI	143		Integer multiplication.
SQI	152		Integer square.
DVI	134		Integer division.
MODI (Pascal) MOD (P-857)	142		Integer module.
CHK	136		Check against subrange bounds.

II.8.2 INTEGER COMPARISONS

EQUI	195		Integer = comparison.
NEQI	203		Integer <> comparison.
LEQI	200		Integer <= comparison.
LESI	201		Integer < comparison.
GEQI	196		Integer >= comparison.
GTRI	197		Integer > comparison.

III. JUMPS

UJP	185	SB	Unconditional jump.
FJP	161	SB	Jump on false tos.
EFJ	211	SB	Equal false jump.
NFJ	212	SB	Not equal false jump.
XJP	172	W_1,W_2,W_3, <casetable>	Case or index jump.

IV. PROCEDURE/FUNCTION CALLS AND RETURNS

CLP	206	UB	Call local procedure.
CGP	207	UB	Call global procedure.
CIP	174	UB	Call intermediate procedure.
CBP	194	UB	Call base procedure. (Not implemented in Pascal version.)
RNP	173	DB	Return from normal procedure.
RBP	193	DB	Return from base procedure. (Not implemented in Pascal version.)

V. MISCELLANEOUS INSTRUCTIONS

BPT	213		Breakpoint. (Not implemented in P-857 version.)
XIT	214		Exit. (Not implemented in Pascal version.)
NOP	215		No operation. (Not implemented in Pascal version.)