A Pascal P-Code Interpreter for the Stanford Emmy

by

Donald Alpert

Technical Note No. 164

September 1979

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University

Stanford, California 94305

The work described herein was supported by the author's National Science Foundation Graduate Fellowship using facilities provided by the Department of Energy under contract EY-76-S-03-0326-PA 39.

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Sciences
Stanford University
Stanford, CA 94306

Technical Note No. 164

September 1979

A Pascal P-Code Interpreter for the Stanford Emmy

by

Donald Alpert

ABSTRACT

This report describes an interpreter for P-Code that runs on the Stanford Emmy. Programs written in Pascal may be compiled into P-Code. The P-Code is then assembled into a binary representation that is interpreted by a microprogram in the Emmy control store. File handling is performed with the aid of mini-UNIX running in a PDP11/05 attached to the Emmy.

KEYWORDS

Emmy Emulation Pascal P-Code

The work described herein was supported by the author's National Science Foundation Graduate Fellowship using facilities provided by the Department of Energy under contract EY-76-S-03-0326-PA 39.

Table of Contents

1 Introduction....1

2 Use of Pascal on the Emmy-PDP11/05 system....2

3 Description of P-Code....3
3.1 Machine architecture....3
3.2 Instruction set....4

4 P-Code implementation on Emmy....5
4.1 Emmy main memory....5
4.2 Interpretation....6
4.3 Data types....7
4.4 Procedure linkage....8
4.5 I/O implementation....9
4.5.1 Buffer variable....9
4.5.2 File i/o....10
4.5.3 Input....10
4.5.4 Output....11
4.5.5 Teletype i/o....12
4.6 Interrupts....12
4.7 Program termination....12

5 Bootstrap procedure....13

6 Performance....15

1.0 Introduction

P-Code is an assembly language for a hypothetical stack machine which is useful in transporting the programming language Pascal [7]. A relatively simple compiler, written in Pascal, translates Pascal source programs to P-Code. A host system may easily gain the capability of running this compiler and subsequently other Pascal programs in at least two ways.

(i) Write a program to translate P-Code programs into the host machine language instructions, most likely through macro expansions.
(ii) Write a program to interpret P-Code instructions without translation.

Either of these methods may include optimization on the P-Code to improve performance during interpretation by the particular host machine.

The second choice was selected for Emmy. A program in Emmy control store interprets a binary representation of the P-Code program in Emmy core. The use of Pascal on the combined Emmy-PDP11/05 system is described in section 2. A brief description of P-Code is given in section 3. The implementation of the interpreter is described in section 4.

The version of P-Code in the Emmy system comes from Sasan Hazeghi of the Computation Research Group at SLAC [2]. His version running at SLAC was the basis of a bootstrap procedure described in section 5.

I wish to thank Sasan Hazeghi, Jerry Huck, and Charlie Neuhauser for their patient assistance in answering a multitude of questions.

2.0 Use of Pascal on the Emmy-PDP11/05 system.

It is possible to compile and run Pascal programs under mini-UNIX on the PDP11/05 in the Emmy lab with almost complete transparency to the actual processing occurring in Emmy. The capabilities of the Emmy lab system are quite fully displayed. The Unix shell interprets commands to call programs written in C which initialize the Emmy control store interpreter and P-Code program. The P-Code program is executed by the Emmy while another C program provides the i/o interface to files and teletype through UNIX [3].

The use of Pascal on the lab system may change in the future but presently these are the salient features:

1. UNIX shell files pcompile(I) and prun(I) compile and execute Pascal programs and provide limited diagnostic support. See Appendix 4.

2. The P-Code compiler for Pascal provides only a large subset of Pascal. Consult references for differences [2] [7]. The most significant is that files are limited to type TEXT, sequential character files. Also, procedures and functions may not be passed as arguments.

3. The implementation limits files in the source program to 6 predefined names:
   INPUT, OUTPUT, PRD (input),
   PRR (output), QRD (input), QRR (output)
The significance of input or output is that a RESET or REWRITE is generated for the file automatically before execution.

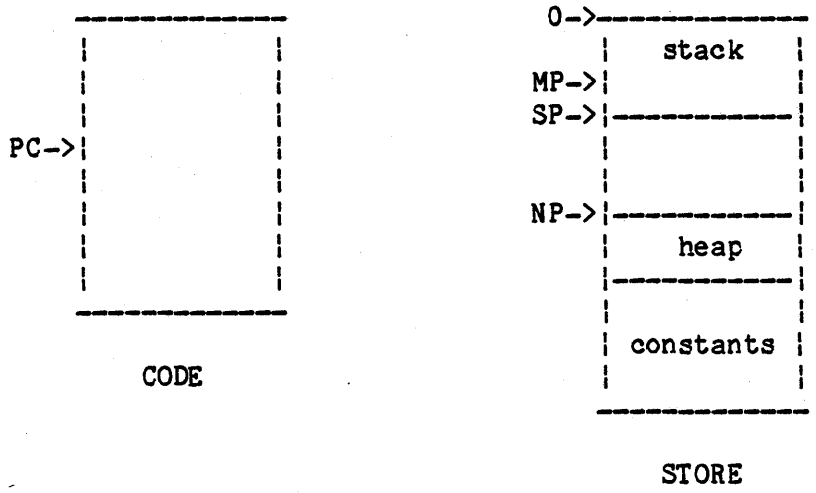
4. Real data types and operations are not supported by the implementation.

3.0 Description of P-Code

This section contains a brief description of the P-Code stack machine (P-machine). For a more complete description consult [1] and [7] from which most of this information was taken. Appendix 3 contains an example of P-Code assembly language for which the Pascal source may be found in Appendix 2.

3.1 Machine architecture

The hypothetical P-machine has two regions of storage: CODE and STORE. CODE is read only and contains the P-code insructions. The current instruction is pointed to by the register PC. STORE is read/write memory and contains constants, stack, and dynamic storage allocation (called heap). The top of the stack is pointed to by SP, the bottom of the current data segment by MP, and the top of the dynamic allocation by NP. See Figure 3.1. (Note that the P-machine as implemented on Emmy differs in several respects. See Figure 4.1.)



P-machine

Figure 3.1

Each time a procedure is called, a new data segment is created and pointed to by MP. This data segment contains "mark stack" information to allow return including return value, return address, dynamic link, static link, etc. The exact information is implementation dependent. Also in the data segment is space for parameters, local variables, and expression evaluation.

Address references into STORE in the P-Code instructions may be expressed as a pair (P,Q) where P designates the static nesting level and Q the displacement into the most recently activated data segment at that level. It is necessary to use the static links or a display to determine the real address in the stack at run time.

3.2 Instruction set

P-Code instructions contain four fields, although not all are always used. The OP field must be at least 7 bits long to contain the opcode. The T field must be at least 4 bits to contain the operand type. (The type field is an addition to the original P-Code by Sasan Hazeghi [2].) The P field is at least 4 bits; usually it contains a static nesting level for describing operand location. The Q field contains enough bits to address all of STORE or CODE. It usually contains an absolute address, a data segment displacement, a jump destination address, or a short constant.

A typical P-Code assembly instruction looks like

```
OPC [T,][P,][Q]
```

OPC is a three letter opcode mnemonic
T is a letter from {A,B,C,I,R,S, etc.} for type address,boolean, character,integer,real,set, etc.
P is usually an integer specifying a static level for addressing
Q is an integer or label

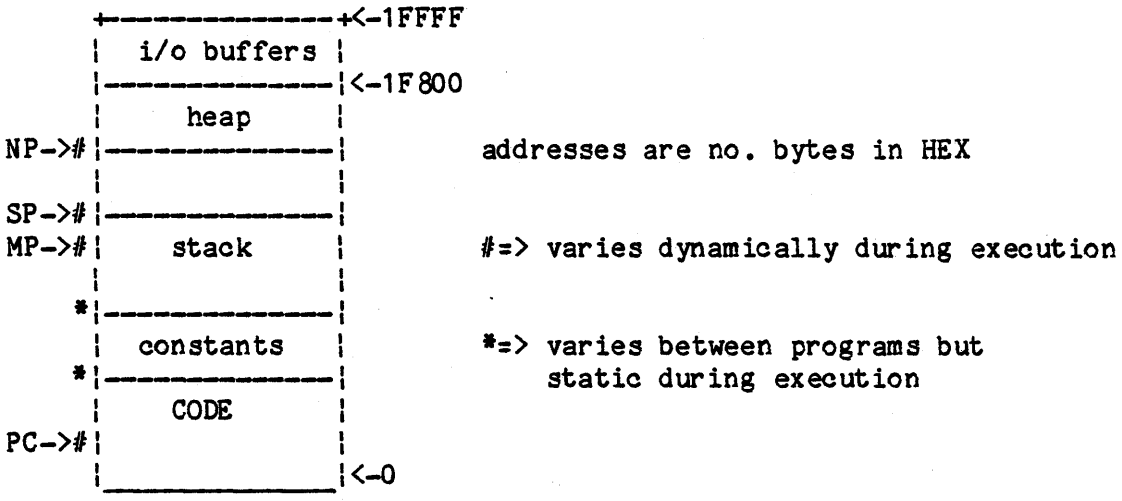
For more information about the particular instructions see Appendix 1 or references [1] or [7].

4.0 P-Code implementation on Emmy

This description of the P-Code implementation on Emmy can be used as an overview or as a guide to more detailed study of the interpreter. It assumes familiarity with the Emmy [5].

4.1 Emmy main memory

The P-machine CODE and STORE are stored in the Emmy main memory (Figure 4.1). CODE is stored as consecutive 32 bit words from the beginning of memory. On top of CODE is storage for constants. On top of the constants is the bottom of the stack. Since the base address of the constants and the stack depends on the P-Code program, their location is passed to the interpreter by the assembler when the program is loaded. Also passed to the interpreter is the entry point to begin execution from CODE. The heap grows downward from a fixed point near the top of main memory. At the end of main memory is an area reserved to buffer i/o transfers from UNIX.

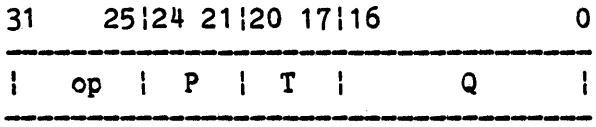


Emmy main store

Figure 4.1

4.2 Interpretation

The interpreter and a portion of the P-machine reside in Emmy control store. This occupies approximately 2K words. The P-machine SP and PC are stored in two of the Emmy's registers while MP, NP, and the display are in control store. The basic interpretation cycle fetches the next instruction word from main memory using the word address in PC, extracts the opcode, and jumps to the appropriate routine to execute the instruction. Each instruction is stored in the format shown in Figure 4.2.



P-Code instruction

Figure 4.2

There is a debug flag in the Emmy state register that allows halt before instruction execution at selected locations in the P-Code program, after a selected number of P-Code instructions have been executed, or on a selected opcode. This checking is normally disabled to speed execution. Another debugging aid allows Pascal programs compiled and assembled with the debug option to halt at a source code line number.

Most P-Code instructions are entirely straightforward in their meaning and execution. Some which require detailed explanation are included in the remainder of this section.

4.3 Data types

The interpreter supports the following simple data types.

type	size in bytes
char	1
boolean	4
integer	4
address	4
set	8

Real data type is not presently supported. Address is used for pointers or machine representation invisible to the Pascal programmer. Sets of 64 or fewer elements are allowed.

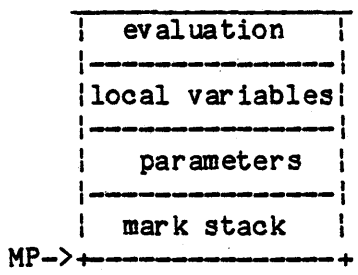
All addressing in STORE is done in bytes. Data are stored without alignment consideration. Data types of more than one byte are stored with more significant bytes at lower addresses. The high order bits of the address are used by the memory controller to select the length and justification of the data [6]. SP always points to the first free byte on top of the stack. When access is performed on the top of the stack the data type is known and its address is given by (SP)-length. When data are pushed on the stack they are stored beginning at the byte pointed to by SP and SP is incremented by the length.

4.4 Procedure linkage

A procedure or function call contains six phases:

MST reserve area on the stack for return linkage
push parameters if any
CUP transfer control to procedure
ENT reserve area on stack for local variable storage
procedure computation
RET return to calling program, possibly with a value

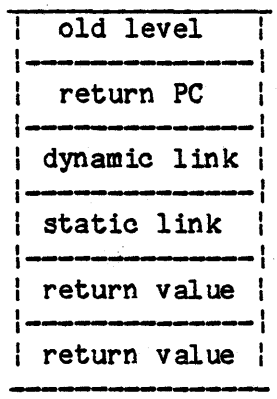
Each procedure activation has a data segment (Figure 4.4.1).



Procedure data segment

Figure 4.4.1

The Mark Stack area has six entries of four bytes each.



Mark Stack area

Figure 4.4.2

When MST is executed the current MP is stored as the dynamic link. MP is set to the present value of SP and SP is incremented to reserve the Mark Stack area. When CUP is executed PC is stored in the return PC word of the Mark Stack area and the address of procedure entry is placed in PC. When ENT is executed SP is incremented to allow for local variable storage and SP is checked against NP for stack overflow.

The level of the calling procedure is stored in the Mark Stack area and the new level is saved in a particular control store location. The pointer to the data segment of the last procedure invoked at the new level is found in the display and stored as the static link. The current value of MP replaces it in the display.

When the procedure executes RET all of the linkage in the Mark Stack area is restored. If a function executes RET the type of the returned value determines how many bytes to leave on top of the stack.

4.5 I/O implementation

The i/o handling represents one of the largest portions of the effort in developing the interpreter. Because i/o is so machine dependent, P-Code buries the details in relatively high level standard procedures. (See Appendix 1.2) This implementation is described in some detail for the interest of future designers on Emmy. In the following description it is assumed that the reader is familiar with Vaccess, a program running on the PDP11 which allows Emmy to interface to the UNIX file system [3].

4.5.1 Buffer variable

All files used in Pascal P-Code are of type TEXT. At present only six predefined files may be used in Pascal programs. The six files are INPUT,OUTPUT,PRD,PRR,QRD,QRR. INPUT,PRD,QRD are input files, the others are output files. The significance of input or output definition is that an automatic RESET or REWRITE is performed before program execution. Otherwise the definition is arbitrary. Pascal programs have a window for each file through which they may read or write the next character in sequence. This window is called a buffer variable [3, p.55]. Pascal i/o operations are translated into the following P-Code framework

```
LDA buffer variable
CSP SIO          start i/o
i/o operations
CSP EIO          end i/o
```

The buffer variable address on the stack is used by the interpreter to identify the file on which to perform i/o.

4.5.2 File i/o

This section concerns the implementation of file i/o in the interpreter. The slight differences for teletype i/o are explained in a later section.

Each file has a buffer of 256 bytes near the top of Emmy main memory used for transfers to UNIX on the PDP11 through Vaccess. These buffers are used by the interpreter but are invisible to the Pascal program. One complication associated with this buffer is that the significance of the high and low bytes of a PDP11 word are reversed from Emmy's byte addressing. When addressing into an i/o buffer using a counter the LSB of the counter is complemented.

Each file also has an entry in a table of file descriptors held in control store with the following information

1. buffer address in main memory
2. device number assigned by Vaccess
3. count of characters into buffer
4. count of characters out from buffer
5. flags

The reason for separate counts in and out is related to the peculiarities of the teletype. There are five flag bits used to specify whether the file is a teletype, whether an i/o request to UNIX has been requested but not completed, whether the file is read or write, and for read files whether end of line or end of file have been reached.

When start i/o (CSP SIO) is executed by the interpreter the buffer variable address is used to calculate an internal pointer to the correct entry in the table of file descriptors. This pointer is used by subsequent i/o operations to speed access to the correct file. CSP EIO signals the end of the i/o sequence. It causes the buffer variable address to be popped from the stack. A sequence of CSP SIO, i/o operations, CSP EIO will not be nested within another such sequence in code produced by the compiler.

Input and output share all aspects of the file handling described thus far. Further explanation is described in the next two sections, separately for input and output.

4.5.3 Input

All i/o routines in the interpreter performing input call one routine, GETCH, each time they require a character. GETCH uses the descriptor table pointer to find the buffer address of the file. If end of file has not been reached then the count of characters out from the buffer is used as an index to read the next character. The character is tested to properly set the end of line flag. The count of characters out from the buffer is incremented and compared to the count in. If more characters remain in the buffer then control returns. Otherwise a new buffer of 256 characters is requested from Vaccess and processing waits for the transfer to complete. Double

buffering and i/o overlapped with computation are not used, to minimize the record keeping. When the transfer has completed the count into the buffer is set to the length of the transfer reported by Vaccess and control returns to the calling routine. The length of the transfer may be less than 256 if end of file has been reached in UNIX. When the next buffer request is made a length of zero will be returned; the interpreter recognizes this as end of file and sets the flag.

4.5.4 Output

All i/o routines in the interpreter performing output call one routine, PUTCH, each time they output a character. PUTCH uses the table pointer to find the buffer address of the file. The count into the buffer is used to place the character in the next locoation. The count is incremented and if it has not yet reached 256 then control returns. Otherwise the buffer is full and a request is made to Vaccess to transfer the entire buffer while processing waits. Double buffering and i/o overlapped with computation are not used, to minimize the record keeping. The count is set to zero and control returns to the calling routine. At program termination partially filled buffers are emptied.

4.5.5 Teletype i/o

When the file on which i/o is being performed is a teletype then the implementation differs slightly from that for files. On output the characters are not buffered but are sent directly to UNIX. This is better for interactive programs.

On input there is a complication caused by Vaccess. The KEYBOARD is different from other virtual devices in that it will interrupt the interpreter whenever a character has been entered from the keyboard. These unsolicited characters may be received even when no program input requests are pending. A special pointer is reserved to locate the KEYBOARD entry in the table of file descriptors maintained by the interpreter. The interrupt for the input character may be received during a potentially critical section when another character is being removed from the buffer. A circular buffer with separate counters in and out avoids the critical use of shared variables.

4.6 Interrupts

The only Emmy interrupts used by the interpreter are the two millisecond console timer and mailbox notices from Vaccess in the PDP11. Any other interrupts cause the interpreter to return an error condition and to halt.

4.7 Program termination

The program may terminate in three expected ways: normal completion, a Pascal program EXIT is executed with a return code, or the interpreter detects an error. Some of the errors detected by the interpreter are variable out of bounds, arithmetic overflow, and read past end of file. Before termination all output buffers are emptied and Vaccess is called with a return code to indicate the cause of termination.

5.0 Bootstrap procedure

The first step in the bootstrap procedure was to write the P-Code interpreter in Emmy assembly languange. At SLAC a version of the Pascal to P-Code compiler, written in Pascal, was modified to be compatible with the interpreter. Some of the modifications were to the size of data types, location of i/o buffer variables, and position of function return value in the stack. The Pascal version of the compiler is independent of the character set representation. In transporting to another machine, however, character set dependencies arose. In particular, when a character variable was used as the switch in a case statement, the jump table created at SLAC assumed the EBCDIC representation. The compiler was modified to produce P-Code to run on a machine using the Pascal character set of 64 members. The interpreter also translated from ASCII characters used by UNIX to the Pascal character set. It is unfortunate that (for reasons not explained here) the compiler was not modified to produce P-Code for an ASCII character set directly.

A Pascal program was also written to assemble the P-Code into a hex representation to be loaded into the Emmy core for interpretation. A simple test program was compiled and assembled at SLAC and brought to the Emmy lab for debug of the interpreter. This step was followed several times as bugs were located in the assembler as well as the interpreter.

When the initial debug was complete, the compiler in Pascal and P-Code along with the assembler in Pascal, P-Code, and hex were brought to the Emmy lab from SLAC. The assembler and interpreter were debugged together, with patches to the assembler made in the hex image. Most of these changes resulted from the different i/o environment between SLAC and the Emmy lab. For example, at SLAC all of the input were treated as card images padded with blanks to 80 columns but in the UNIX files the trailing blanks were not present. At this point patches in the interpreted programs could be made more easily in the P-Code version and then translated into hex by the assembler. In this way a P-Code version of the compiler was debugged. Now patches could be made in the original Pascal programs and consistent, working versions of the assembler and compiler were available and easily modifiable.

The compiler and assembler were then modified to produce code that assumed an ASCII representation for characters. This required the new programs to be compiled and assembled with the old versions and then processed through themselves. The interpreter was modified to avoid translation between ASCII and Pascal sets. The compiler and assembler were modified once more to handle the character set operations more efficiently with use of CHR and ORD functions instead of explicitly using the ASCII character codes.

Many of the tedious details of the bootstrap procedure have not been described. Although several levels of program were involved: Pascal, P-Code, hex, and Emmy interpreter, the process was nearly as straightforward as explained because the debugging proceeded orderly up the hierarchy. By the time the compiler was being debugged it was infrequent that problems had to be located at the level of single stepping the interpreter. It was important to debug each lower level fully before proceding to the next level.

6.0 Performance

It is difficult to make a meaningful measure of the performance of the P-Code implementation and even more difficult to make comparison with other implementations. Some specific examples and measurements are provided below. It should be noted that no attempt has been made to improve the P-Code representation or to tune the interpreter. By way of comparison the Emmy can emulate the PDP11 at 50 KIPS and the IBM 360 at 60 KIPS (highly optimized).

The P-Code compiler was run on Emmy to compile itself with no other users on the PDP11. The following measurements were made.

no. source lines	5339
no. source lines that generate code	3032
no. P-Code instructions in compiler	15938
time to compile	458 sec
percentage of compilation time in i/o wait	7.0 %
average P-Code execution rate	58 KIPS

A benchmark program performs the quicksort algorithm on 20,000 pseudorandom numbers (see Appendix 2). The inner sorting loop (source lines 53 to 94) contains no i/o operations. The P-Code for this loop executes in 91 seconds at an average rate of 71 KIPS. The same loop with debugging enabled executes in 107 seconds at 73 KIPS. (Debugging includes CHK for bounds checking and LOC to test for breakpoint.) The interpreter was used in a slower mode to capture the execution counts for the P-Code operations in the loop. The results for this loop are presented in Table 6.1. The execution times for the operations were measured by forcing each operation into a one instruction loop. This involved modification of the interpreter for PC and SP adjustment. These artifacts were approximately subtracted out.

The IBM 370/168 at SLAC executes the same loop translated from P-Code into its machine code with extensive optimization in 1.5 seconds at a rate of 2 MIPS. Typically the translation maps one P-Code instruction into one 370 RX instruction [2].

There are two ways in which the performance can be easily improved. First, new operations may be generated at assembly time and the interpreter may be tuned to be more efficient but less structured. For example, this process may be applied to the introduction of three separate LOD operations for types of 1, 4, or 8 bytes. This should reduce the execution time of LOD to about 12 us and decrease the execution time for the quiksort loop by 6.2%. Secondly, peephole optimization can be used at assembly time to reduce the number of operations performed. For example, in the quiksort loop all of the DEC operations are used to create zero offset indexing into the arrays that were declared to have lower index 1. These DEC operations can all be eliminated by adjusting the base address used in accessing the arrays.

Quicksort loop execution statistics

OP	EXEC TIME (us)	COUNT	TIME (sec)	COUNT %	TIME %
ADI	11.7	184450	2.16	2.8	2.4
DEC	12.8	662589	8.48	10.2	9.2
EQU	18.6	957	0.18	0.0	0.2
FJP	10.0	499867	5.00	7.7	5.4
GEQ	18.6	203575	3.79	3.1	4.1
GRT	18.6	6327	0.12	0.1	0.1
INC	12.8	19999	0.26	0.3	0.3
IND	14.8	518680	7.68	8.0	8.4
IXA	19.1	662589	12.7	10.2	13.8
LDA	10.0	662589	6.63	10.2	7.2
LDC	10.8	472961	5.11	7.3	5.6
LEQ	18.6	190103	3.54	2.9	3.9
LES	18.6	20000	0.37	0.3	0.4
LOD	15.6	1578046	24.6	24.2	26.8
NEQ	18.6	23642	0.44	0.4	0.5
SBI	11.7	239573	2.80	3.7	3.0
STO	14.0	143909	2.01	2.2	2.2
STR	14.8	394695	5.84	6.1	6.4
UJP	7.1	27344	0.19	0.4	0.2
TOTAL	----	6511895	91.2	100	100

Notes: 1. All operations with type field had type integer.

2. IXA execution time depends on the storage size in the Q field. Each time IXA was executed in the loop the storage size was 4.

3. The compare operations (EQU, GEQ, etc.) and FJP take varying time to execute depending on the truth value of the result or the test. This difference is 2% or less of the execution time and was ignored.

4. TOTAL measurements may disagree slightly with column sums.

Table 6.1

Appendix 1.1

P-Code assembly mnemonics

MNEMONIC	OPERATION
ABI	pop integer on top of stack and push its absolute value
ABR	pop real on top of stack and push its absolute value
ADI	pop two integers on top of stack and push sum
ADR	pop two reals on top of stack and push sum
AND	pop two booleans on top of stack and push their logical AND
CHK	perform bounds check on top of stack, error if out of bounds
CHR	pop integer on top of stack and push its character equivalent
CSP	call standard procedure
CUP	call user procedure
DEC	decrement top of stack by Q field
DIF	perform set difference of TOP and NTOP and push result
DVI	integer divide NTOP by TOP and push quotient
DVR	real divide NTOP by TOP and push quotient
ENT	enter user procedure
EOF	not used, see CSP EOF
EQU	pop TOP,NTOP, compare according to type and push NTOP = TOP
FJP	pop boolean TOP and if it is FALSE jump to address in Q
FLO	convert integer NTOP to real NTOP
FLT	pop integer TOP, convert it to real and push
GEQ	pop TOP,NTOP, compare according to type and push NTOP >= TOP
GRT	pop TOP,NTOP, compare according to type and push NTOP > TOP
INC	increment top of stack by Q field
IND	pop address TOP, add index Q, and push from that address
INN	pop TOP,NTOP and push boolean result NTOP IN TOP
INT	pop two sets on top of stack and push their set intersection
IOR	pop two booleans on top of stack and push their inclusive OR
IXA	pop TOP,NTOP and push address NTOP + TOP*Q
LAO	push address of global, in static level 1, with displacement Q
LCA	push address of string
LCI	generated by assembler to load constants too long for Q field
LDA	push address (P,Q)
LDC	load the constant immediately specified in Q
LEQ	pop TOP,NTOP, compare according to type and push NTOP <= TOP
LES	pop TOP,NTOP, compare according to type and push NTOP < TOP
LOC	specifies source code line number
LOD	push (P,Q)@
MOD	integer divide NTOP by TOP and push remainder
MOV	pop TOP,NTOP and move Q bytes starting at address TOP to NTOP
MPI	pop two integers on top of stack and push their product
MPR	pop two reals on top of stack and push their product
MST	reserve and initialize Mark Stack area prior to procedure call
NEQ	pop TOP,NTOP, compare according to type and push NTOP <> TOP
NEW	pop address TOP, allocate Q bytes on heap, store NP in TOP
NGI	pop integer on top of stack and push its additive inverse
NGR	pop real on top of stack and push its additive inverse
NOT	pop boolean on top of stack and push its logical NOT
ODD	pop integer on top of stack and push boolean ODD(TOP)

MNEMONIC	OPERATION
ORD	pop TOP and push its ordinal integer value
RET	return from procedure
RST	deallocate heap: pop address TOP and assign it to NP
SAV	pop address TOP and store NP at that address
SBI	pop integers TOP,NTOP and push NTOP - TOP
SBR	pop reals TOP,NTOP and push NTOP - TOP
SGS	pop integer on top of stack and push its singleton set
SQI	pop integer on top of stack and push its square
SQR	pop real on top of stack and push its square
SRO	pop TOP to store as global in (1,Q)
STO	pop TOP and NTOP and store TOP in address NTOP
STP	normal program termination
STR	pop TOP and store in (P,Q)
TRC	pop real TOP, truncate it to an integer and push
UJP	unconditional jump to Q
UNI	pop two sets on top of stack and push their set union
XJP	pop integer TOP and if in range use it as index into jump table

Notes:
1. TOP and NTOP refer to the items on top of the stack and next to the top of the stack before the instruction is executed.

2. (P,Q) represents the address specified by the static level, displacement pair.
(P,Q)@ represents the data item in address (P,Q)

Appendix 1.2

P-Code standard procedures

MNEMONIC	OPERATION
ATN	pop real TOP and push its arctangent
CLK	pop integer TOP and select clock value to push
COS	pop real TOP and push its cosine
EIO	end of i/o sequence, pop file buffer address TOP
ELN	pop file buffer address TOP, push boolean result for end of line test and push address for EIO to pop
EOF	pop file buffer address TOP, push boolean result for end of file test and push address for EIO to pop
EXP	pop real TOP and push its exponential
GET	use file buffer address TOP to get next character from the file
LOG	pop real TOP and push its logarithm
PUT	use file buffer address TOP to put next character to the file
RDB	pop adress TOP, use file buffer address NTOP to read a boolean value from the file and store the value in TOP
RDC	pop adress TOP, use file buffer address NTOP to read a character from the file and store the value in TOP
RDI	pop adress TOP, use file buffer address NTOP to read an integer from the file and store the value in TOP
RDR	pop adress TOP, use file buffer address NTOP to read a real from the file and store the value in TOP
RDS	read a string with file, destination, and length specified in stack
RES	use file buffer address TOP to perform RESET on file
REW	use file buffer address TOP to perform REWRITE on file
RLN	use file buffer address TOP to perform READLN on file
SIN	pop real TOP and push its sine
SIO	start i/o sequence for file with buffer address TOP
SQT	pop real TOP and push its square root
WLN	use file buffer address TOP to perform WRITELN on file
WRB	write boolean value to file with format specified in stack
WRC	write character to file with format specified in stack
WRI	write integer to file with format specified in stack
WRR	write real to file with format specified in stack
WRS	write string to file with format specified in stack
XIT	terminate program with return code TOP

Note: 1. TOP and NTOP refer to the items on top of the stack and next to the top of the stack before the instruction is executed.

Appendix 2

Quicksort program

```
 1 | (*$L-*)
 2 | PROGRAM QUICKSORT( OUTPUT ) ;
 3 |
 4 | (* PARTITION-XCHANGE SORT, AFTER B. SEDGEWICK,  S. HAZEGHI.
 5 |   WITH MODIFICATIONS TO OUTPUT FORMAT BY DONALD ALPERT
 6 |    M ::= SIZE OF THE PARTITIONS TO BE BUBBLE-SORTED
 7 |    N ::= NUMBER OF ELEMENTS TO BE SORTED ;
 8 |    N1 ::= N+1 ;
 9 |    STACK_SIZE ::= MAX # OF UNSORTED PARTITIONS ( >= 2*(LOG2(N)-3) )
10 | *)
11 |
12 |     LABEL 101,111 ;
13 |
14 |     CONST   M = 9 ;  N = 20000;  N1 = 20001 ;STACK_SIZE = 25;
15 |
16 |     VAR L,R,P,I,J,V,T,TIM : INTEGER ;
17 |         STACK : ARRAY [1..STACK_SIZE] OF INTEGER ;
18 |         (* STACK_SIZE  2*(LG(N)-3) *)
19 |         A : ARRAY [1..N1] OF INTEGER ;
20 |
21 |     PROCEDURE PRINTDATA ;
22 |
23 |         (* TO PRINT THE RAW AND SORTED DATA, 10 NUMBERS PER LINE *)
24 |
25 |         BEGIN
26 |         FOR I := 1 TO N1 DO
27 |             BEGIN
28 |             IF (I MOD 10) = 1 THEN  WRITELN() ;
29 |             WRITE('  ',A[I]:11 ) ;
30 |             END;
31 |         WRITELN() ;
32 |         END ;
33 |
34 |
35 |     BEGIN  (* QUIKSORT *)
36 |
37 |     (* I- GENERATE RANDOM DATA FOR SORTING *)
38 |
39 |     A[1] := 0; L := 2; R := N1; P := 0;
40 |     FOR I := 1 TO N DO
41 |         BEGIN  A[I+1] := A[I]*314159269+453806245;
42 |         IF A[I+1] < 0 THEN
43 |             BEGIN  A[I+1] := A[I+1]+2147483647 ;
44 |             A[I+1] := A[I+1]+1
45 |             END
46 |         END ;
47 |
48 |     PRINTDATA ;
49 |     TIM := CLOCK(1) ;
```

```
50 |
51 |     (* II- PARTITION THE INPUT DATA *)
52 |
53 |     REPEAT  I := L-1; J := R; V := A[R];
54 |         REPEAT
55 |             REPEAT I := I+1 UNTIL (A[I] >= V) ;
56 |             A[J] := A[I];
57 |             REPEAT J := J-1 UNTIL (A[J] <= V) ;
58 |             IF I >= J THEN GOTO 101 ;
59 |             A[I] := A[J]
60 |         UNTIL  FALSE ;
61 |     101:IF I <> J THEN  J := J+1;
62 |         A[J] := V;
63 |         IF J-L > R-J THEN
64 |             BEGIN
65 |             IF M >= J-L THEN
66 |                 BEGIN  IF P = 0 THEN GOTO 111 ;
67 |                 R := STACK[P+1];  L := STACK[P];  P := P-2;
68 |                 END
69 |             ELSE IF R-J > M THEN
70 |                     BEGIN  P := P+2 ;  STACK[P] := L ;
71 |                     STACK[P+1] := J-1 ;  L := J+1
72 |                     END
73 |                 ELSE  R := J-1
74 |             END
75 |         ELSE IF M >= R-J THEN
76 |                     BEGIN  IF P = 0 THEN GOTO 111 ;
77 |                     R := STACK[P+1];  L := STACK[P];  P := P-2;
78 |                     END
79 |                 ELSE IF J-L > M  THEN
80 |                         BEGIN  P := P+2 ;  STACK[P] := J+1 ;
81 |                         STACK[P+1] := R ;  R := J-1
82 |                         END
83 |                     ELSE  L := J+1
84 |     UNTIL  FALSE ;
85 |
86 |     (* III- EXCHANGE SORT EACH PARTITION *)
87 |
88 | 111:FOR I := 2 TO N1 DO
89 |         IF A[I] < A[I-1] THEN
90 |             BEGIN
91 |             V := A[I] ;  J := I-1 ;
92 |             REPEAT  A[J+1] := A[J];  J := J-1  UNTIL (A[J] <= V) ;
93 |             A[J+1] := V
94 |             END ;
95 |
96 |     TIM := (CLOCK(1) - TIM) DIV 10 ;
97 |     WRITELN() ;
98 |   WRITELN('  SORTING TIME =', TIM DIV 100:4,'.',TIM MOD 100:2,' SECONDS');
99 |     PRINTDATA ;
100 |
101 |     END (* QUIKSORT *).
102 | (**)
```

Appendix 3

P-Code program for PRINTDATA procedure in Appendix 2

```
PRINT001 ENT P,2,L1  PRINTDATA      1 1 1   1
 LDC I,1
 STR I,1,260
 LDC I,20001
 STR I,2,24
 LOD I,1,260
 LOD I,2,24
 LEQ I
 FJP L3
L2 LAB
 LOC    28
 LOD I,1,260
 LDC I,10
 MOD
 LDC I,1
 EQU I
 FJP L4
 LDA 1,193
 CSP SIO
 CSP WLN
 CSP EIO
L4 LAB
 LOC    29
 LDA 1,193
 CSP SIO
 LCA ' '
 LDC I,2
 LDC I,2
 CSP WRS
 LDA 1,376
 LOD I,1,260
 CHK I,1,20001
 DEC I,1
 IXA I,4
 IND I,0
 LDC I,11
 CSP WRI
 CSP EIO
 LOC    30
 LOD I,1,260
 LOD I,2,24
 NEQ I
 FJP L3
 LOD I,1,260
 INC I,1
 STR I,1,260
 UJP L2
L3 LAB
 LOC    31
 LDA 1,193
```

```
 CSP SIO
 CSP WLN
 CSP EIO
 LOC    32
 RET P
L1 DEF        28
```

Appendix 4

UNIX shell file descriptions

PRUN(I) 7/19/79 PRUN(I)

NAME

prun - interpret Pascal P-Code program on emmy

SYNOPSIS

prun file input output [prd [prr [qrd [qrr]]]]

DESCRIPTION

Prun is a shell command which is used to run Pascal programs on emmy. The Pascal program file.pas should previously have been compiled and assembled into the P-Code program file.em with pcompile(I). The P-Code program is interpreted on emmy with the named files assigned to Pascal source INPUT,OUTPUT,PRD,PRR,QRD,QRR. INPUT,PRD,QRD are input files, the others are output files. The significance of input or output definition is that an automatic RESET or REWRITE is performed before program execution. Otherwise the definition is arbitrary. INPUT or OUTPUT can be directed to the terminal by the assignment "tty". Unassigned files must be represented with "-" except that trailing unused files may be omitted.

E.g. "prun foo tty - - foo.out " will interpret the P-Code program foo.em with INPUT assigned to the terminal keyboard and PRR assigned to the file foo.out.

FILES

file.em	assembled version of source program in em-load format
file.init	temporary file used for initialization before running emmy
file.log	on exceptional termination of P-Code program this file contains some diagnostic information
file.msg	file used to record standard output from some programs called by pcompile for use in case of errors in processing

SEE ALSO

pcompile(I)

BUGS

Under certain error conditions, e.g. emmy not powered on, the program will hang. Type one or more rubouts and look at file.msg to try to determine the cause. The emmy is not a protected resource on the unix system. If more than one user tries to run on emmy they will be in conflict.

NAME

pcompile - compile Pascal program

SYNOPSIS

pcompile file {-p | -a | input output [prd [prr [qrd [qrr]]]]}

DESCRIPTION

Pcompile is a shell command which allows pascal programs to be compiled into P-Code and interpreted on emmy. The Pascal source program should be in file.pas. During compilation a symbol table and referencing statistics are collected in file.stat. A program listing with error diagnostics is included in file.lst. The compiled P-Code program is in file.pcode. If no compilation errors are found the assembler removes file.pcode and creates file.em which can be interpreted on emmy.

-p suspends processing after compilation and leaves file.pcode.

-a suspends processing after assembly.

If no assembly errors are found the user program is executed with a call to prun(I) and the named files.

FILES

file.em	assembled version of source program in em-load format
file.err	if errors occur during assembly they will be reported here
file.init	temporary file used for initialization before running emmy
file.log	on exceptional termination of P-Code program this file contains some diagnostic information
file.lst	listing of source program including compiler diagnostics
file.msg	file used to record standard output from some programs called by pcompile for use in case of errors in processing
file.pas	pascal source program
file.pcode	compiled version of source in P-Code
file.stat	symbol table for compiled program with referencing statistics

SEE ALSO

prun(I)

References

[1] Gilbert, Erik J. and Wall, David W., "P-Code Intermediate Assembler Language (PAIL-4)", Technical Note No. 148, Computer Systems Laboratory, Stanford University, Stanford CA 94305

[2] Hazeghi, Sasan, personal conversations and unpublished work, SLAC Computation Research Group, Stanford CA 94305

[3] Huck, Jerry, "A Virtual Input/Output System for the Stanford Emmy — V-Access", Technical Note No. 144, Computer Systems Laboratory, Stanford University, Stanford CA 94305

[4] Jensen, Kathleen and Wirth, Niklaus, "PASCAL User Manual and Report", Second Edition, New York, Springer-Verlag, 1974

[5] Neuhauser, C., "Emmy System Processor — Principles of Operation", Technical Note No. 114, Computer Systems Laboratory, Stanford University, Stanford CA 94305

[6] Neuhauser, C., "Emmy System Peripherals — Principles of Operation", Technical Note No. 77, Computer Systems Laboratory, Stanford University, Stanford CA 94305

[7] Nori, K. V. et al, "The PASCAL <P> Compiler: Implementation Notes", Eidgenossische Technische Hochschule Zurich, Instituts fur Informatik, July, 1976