P-CODE INTERMEDIATE ASSEMBLER LANGUAGE
(PAIL-4)

Erik J. Gilbert and David W. Wall

TECHNICAL NOTE NO. 148

March 1978

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

The authors wish to acknowledge crucial support for this work which has been received from the Department of the Navy via Office of Naval Research Order Numbers N00014-76-F-0023, N00014-77-F-0023, and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which is operated for the U.S. Department of Energy under Contract No. W-7405-Eng-48), from the Computations Group of the Stanford Linear Accelerator Center (supported by the U.S. Department of Energy under Contract No. EY-76-C-03-0515), and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation). The authors also wish to acknowledge the fellowship support of their graduate studies which was extended by the National Science Foundation during the academic year. This work has been performed under Contract No. LLL P09083403, Principal Investigator, Professor Gio Wiederhold.

P-CODE INTERMEDIATE ASSEMBLER LANGUAGE
(PAIL-4)

Erik J. Gilbert and David W. Wall

TECHNICAL NOTE NO. 148
March 1978

COMPUTER SYSTEMS LABORATORY
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

ABSTRACT

The syntax and semantics of P-Code, the intermediate language used in the current S-1 programming system is described.

INDEX TERMS: Intermediate language, P-Code, Semantics, S-1

O. Table of Contents

I. Introduction

A. Purpose

This document describes the intermediate code produced by the PASCAL compiler currently in use at SLAC. This intermediate code is called P-Code. It runs on a hypothetical machine called the Stack Computer (SC). The purpose behind compiling into this intermediate form is to make the PASCAL compiler more portable from one system to another; one need only rewrite the P-Code translator or interpreter to bring up the entire PASCAL compiler.

The purpose of this document is to describe the syntax and semantics of P-Code assembler language text as it is output by the PASCAL compiler, so that the PASCAL implementor may use this description to construct an interpreter or translator for a particular system. The existence of P-Code assembler language assumes the existence of an underlying machine (the Stack Computer). In order to most clearly define the P-Code assembler language it is necessary to make occasional references to the detailed structure of this underlying machine. However, it should be noted that the interface being defined by this document is that of the source level P-Code, NOT the underlying Stack Computer. Hence, for instance, an actual implementation of an interpreter for P-Code may vary significantly in detailed structure from the SC referred to herein.

Unfortunately, the definition of the interface between the PASCAL compiler and a P-Code interpreter or translator is not entirely clean and clear-cut. For instance, the code

for the PASCAL compiler contains a number of P-Code implementation-dependent parameters. Therefore, the reader is warned that this document is not an absolutely complete definition of the interface, since for picking out certain fine details it will undoubtedly be necessary to refer to the source code for the PASCAL compiler, the P-Code interpreter, etc.

B. Acknowledgement

The authors wish to acknowledge crucial support for this work which has been received from the Department of the Navy via Office of Naval Research Order Numbers N00014-76-F-0023, N00014-77-F-0023, and N00014-78-F-0023 to the University of California Lawrence Livermore Laboratory (which is operated for the U. S. Department of Energy under Contract No. W-7405-Eng-48), from the Computations Group of the Stanford Linear Accelerator Center (supported by the U. S. Department of Energy under Contract No. EY-76-C-03-0515), and from the Stanford Artificial Intelligence Laboratory (which receives support from the Defense Advanced Research Projects Agency and the National Science Foundation). The authors also wish to acknowledge the fellowship support of their graduate studies which was extended by the National Science Foundation during the academic year.

Portions of the second section of this document were taken with some modification from "The PASCAL (P) Compiler Implementation Notes" by Nori, Ammann, Jensen, and Nageli. That text has proven invaluable in the preparation of this document, and the reader seeking additional information (especially historical) should consult that text.

II. Architecture of the Stack Computer

A. Static environment

The Stack Computer consists of four registers and a memory. The registers are:

1) PC: the program counter;
2) SP: the stack pointer;
3) MP: the mark pointer;
4) NP: the new pointer.

The PC has the usual meaning. The meaning of SP, MP, and NP will become apparent when we describe the dynamic environment. The memory can be thought of as two linear arrays of storage units (words): one of these parts of memory is referred to as the code store, labelled CODE; the other part is referred to as the data store, labelled STORE. Their functions are obvious. Note that PC is always an index into CODE, and that SP, MP, and NP are always indices into STORE. Note also that the CODE array is read-only whereas the

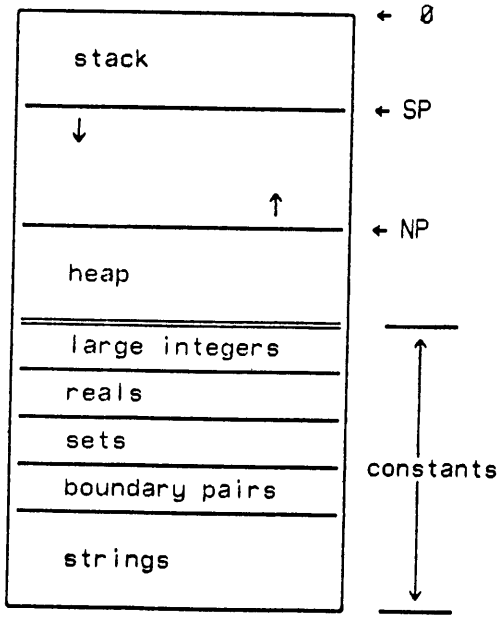
STORE array is read/write.

Each element of CODE is an instruction with four fields: the OP field, the T field, the P field, and the Q field. The actual lengths of these fields are implementation-dependent with the restrictions that the OP field should be at least 7 bits long, the T field should be at least 4 bits, the P field should be at least 4 bits, and the Q field should be at least large enough to hold any index into CODE or STORE. The OP field specifies the particular operation to be performed. The T field specifies the type(s) of one or more explicit or implicit operands. The P field (usually) specifies the lexical level of declaration of a variable being accessed. The meaning of the Q field is highly instruction-dependent, but it usually contains an offset or an item count of some kind.

Each element of STORE has two fields: the type field and the data field. The type field tells the datatype of the data field, e.g. INTEGER, REAL, BOOLEAN, SET, etc. The data field can have any value legal for the type specified.

B. Dynamic environment

At P-Code run time, STORE is subdivided into two parts: one part contains constants of various kinds, whereas the other caters to the varying demands of data store, as required by the execution of PASCAL programs. This is depicted below.



The stack grows from 0 upwards and consists of all directly addressable data according to the data declarations.

Storage overflow occurs if SP and NP meet. The heap grows downwards from the point where the constants begin; its growth is dictated by use of the standard procedure NEW.

The following points are worth noting regarding the dynamic use of elements of STORE: the compiler's use of the heap resembles a second stack and so a very simple heap

mechanism suffices. However, an implementor desiring more flexibility could implement a more complex free-storage handling mechanism. Though it should be clear from the picture, please note that SP points to the top of the stack and NP points to the top of the heap. Please also note that this usage of the word "heap" is quite different from the sorting data structure of the same name used by Knuth and others.

The stack has further internal structure; this structure allows a correspondence between the dynamic evaluation of a PASCAL program and its static text in that necessary links are maintained, dynamically, so that the accessible objects are those dictated by static program text (except for parameters - of course). To amplify, the stack consists of a sequence of "data segments," each of them "belonging" to an activation of a procedure or a function (except the first data segment, which starts at location 0, and which belongs to the outermost block, viz., the program block).

Most dynamically allocated storage entities (particularly program variables) are accessed using this internal structure of the stack. Each such entity is statically associated with a particular block of program text, which is in turn statically nested inside other blocks. Thus, with each such entity may be associated a "static level number," i.e. the nesting level of the block in which it is declared. Since the program can at one time access variables declared in at most one block having a particular static level number, the specification at instruction execution time of a level number together with a data segment displacement uniquely identifies each such dynamic entity.

Thus, the STORE addressing mechanism of P-Code instructions is defined in terms of this data segment structure. A piece of the dynamic stack can be addressed directly in terms of its absolute address, but it is often addressed by a pair of numbers (P,Q) as follows: P is the static level number of the entity being accessed, and Q is the displacement into the data segment (which is dynamically defined by P) of the actual data of the entity. Static level numbers of storage begin at 1 for data entities in the outermost static level, and increase by 1 for each nested procedure. This corresponds to the indexing of the procedure nesting levels used by the ENT instruction; the main program has level 1 and the level increases by 1 for each nesting. This has the effect that variables declared in a given procedure have a static level number equal to the nesting level number of that procedure. From all this it follows that the Stack Computer stack implementation must be defined such that all dynamic entities may be accessed in this way.

Since any given target machine may use a somewhat different addressing structure from that used by the hypothetical Stack Computer, one should be careful to ensure that P-Code addresses actually point to the relevant data object, and not to a hypothetical base address, e.g. virtual zero-origin addressing of arrays. This may mean that a P-Code program will have apparently unnecessary address adjustments (e.g. INCs and DECs) which a good translator will then optimize out.

The outermost data segment (containing entities with static level 1) contains a few specially distinguished elements, used for communication with the outside world. The addresses of these elements are called "file addresses," since they are used by the emitted P-Code to identify the different files used. Each element is a storage unit of type character, and is used as the PASCAL-defined associated buffer variable for the corresponding file. Since the elements are in the outermost data segment, the static level number of any address pair used to reference them is 1. The displacements are assigned by the PASCAL compiler starting from the value of the PASCAL compiler CONST parameter "FIRSTFILBUF." At present, six files are predefined, so user defined files are assigned starting at FIRSTFILBUF+6. The predefined files are as follows:

"file address" displacement	file name
FIRSTFILBUF	INPUT
FIRSTFILBUF+1	OUTPUT
FIRSTFILBUF+2	PRD
FIRSTFILBUF+3	PRR
FIRSTFILBUF+4	QRD
FIRSTFILBUF+5	QRR

A data segment consists of the following sequence of information: a "mark-stack" part; a "parameter" section if there are any parameters to the procedure or function to which the data segment belongs; a "local data" section if there are any local variables declared within the procedure or function to which the data segment belongs; and finally, any temporary elements which may be required in the program evaluation process. The register MP always points to the mark-stack part of the most recently allocated data segment in the stack.

The mark-stack part consists of several consecutive fields containing information necessary to maintain the dynamic environment and to allow the old dynamic environment to be restored upon return from this one. This information may vary from implementation to implementation, but is likely to contain such items as the return address, static and dynamic links, a function return value, etc. An initial mark-stack part is set up by executing an MST instruction. An implementor of a P-Code translator or interpreter must decide on a precise format for the mark-stack part and implement the MST instruction accordingly.

The parameter section consists of two parts, both of which may be empty. The first part consists of elements which are either: (a) pointers (indices into STORE) in case the corresponding parameters are of type "call-by-reference" or of type "call-by-value" but the size of the parameter is larger than the size of a scalar or set; or (b) the parameter is "call-by-value" and the value itself is passed as it requires less than or equal to the amount

of space occupied by a scalar or set. The second part pertains only to call-by-value parameters whose size is greater than the amount of space occupied by a scalar or set. In such a case, for each of such parameters, space is allocated as required by their respective sizes.

In order to effect a procedure/function call, a mark-stack instruction (MST) is executed with a parameter which allows the links to be filled. Then follows a series of expression evaluations to fill in the first part of the parameter section. After this a call-user-procedure instruction (CUP) or a call-standard-procedure instruction (CSP) is executed with appropriate parameters. The reserving of space for the second part of the parameter section as well as the local data is done by the ENT instruction, the first to be executed in the procedure body. The copying of large call-by-value parameters into the second part of the parameter section is done by instructions immediately after the ENT instruction.

The MST, CUP, CSP, ENT, and RET instructions must be implemented so as to keep the addressing structure of the stack consistent with the assumptions made by the P-Code emitted by the PASCAL compiler. Specifically, the parameters to a called procedure or function must, after execution of the ENT which comes first in the called code, be located at the proper displacements into the current data segment. These displacements are assigned starting from the value of the PASCAL compiler CONST parameter "LCAFTMST," the number of storage units which are expected to be added to the stack by the MST instruction. Also, for a function (which returns a value), the compiler emits code to store the value at displacement "FNCRSLT" (another CONST parameter) prior to the RET instruction. The implementor must ensure that upon return the stack is restored to its original state before the call, plus the function value pushed onto the top of the stack.

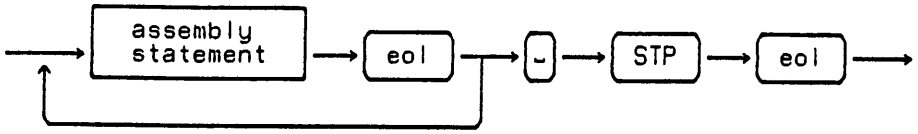
"Lif had picked up a brick from the heap and put it in place on the stack and smiled in embarrassment."

- Ursula K. LeGuin, "Things"

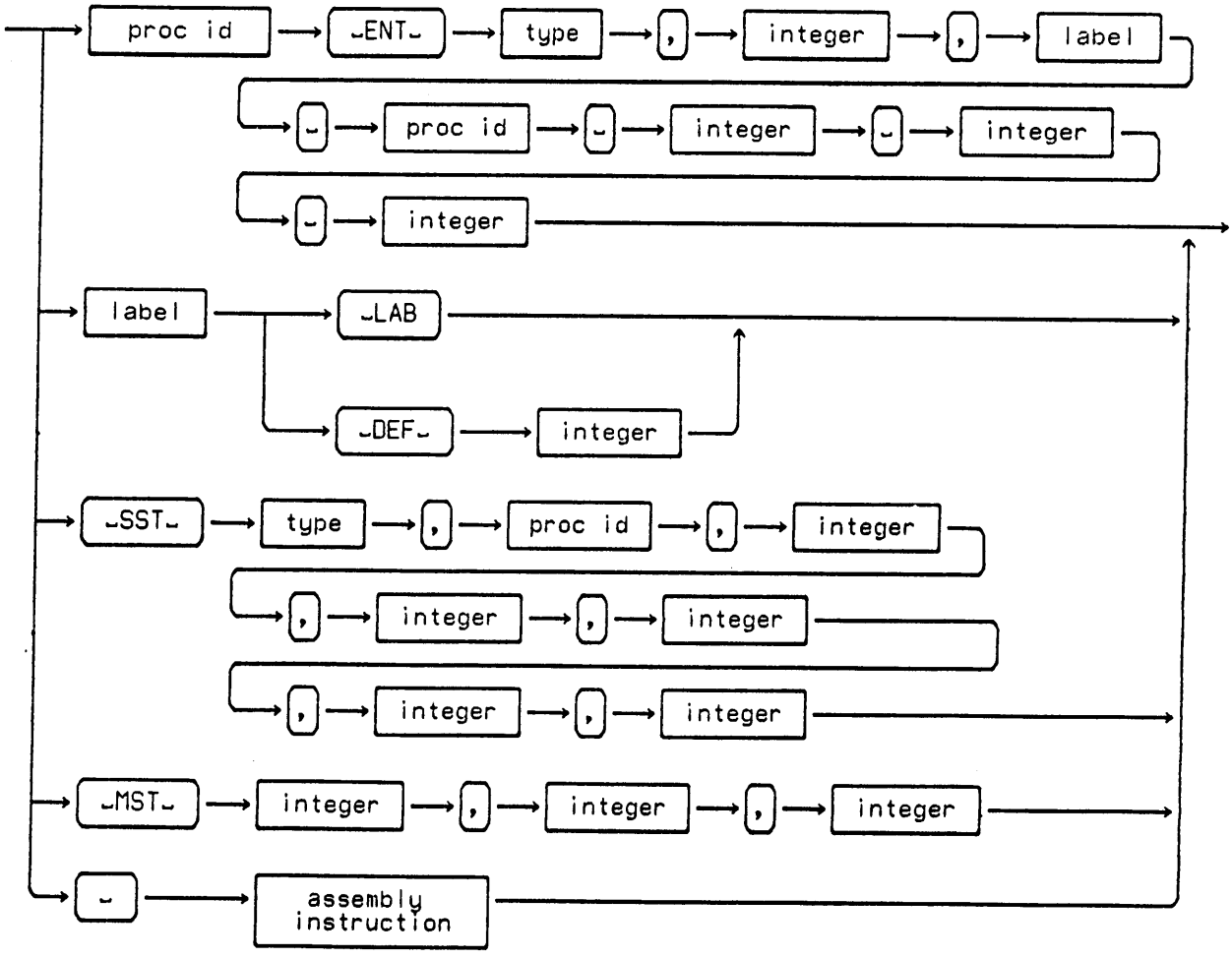
III. Detailed Language Description

A. Syntax diagrams

Assembly program:

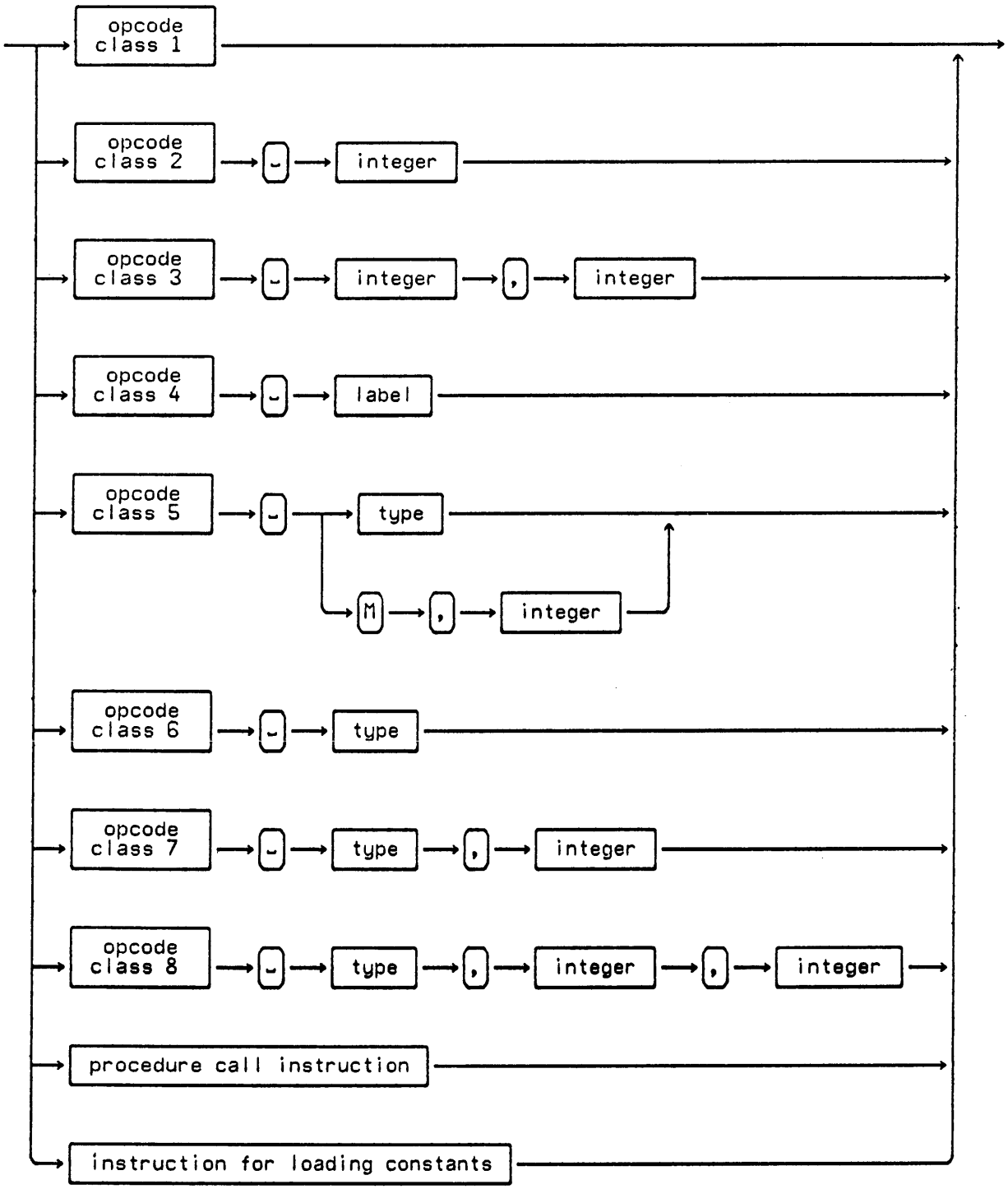


Assembly statement:

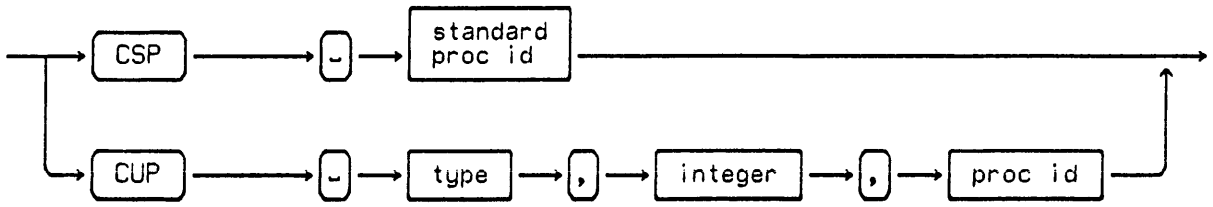


Label:

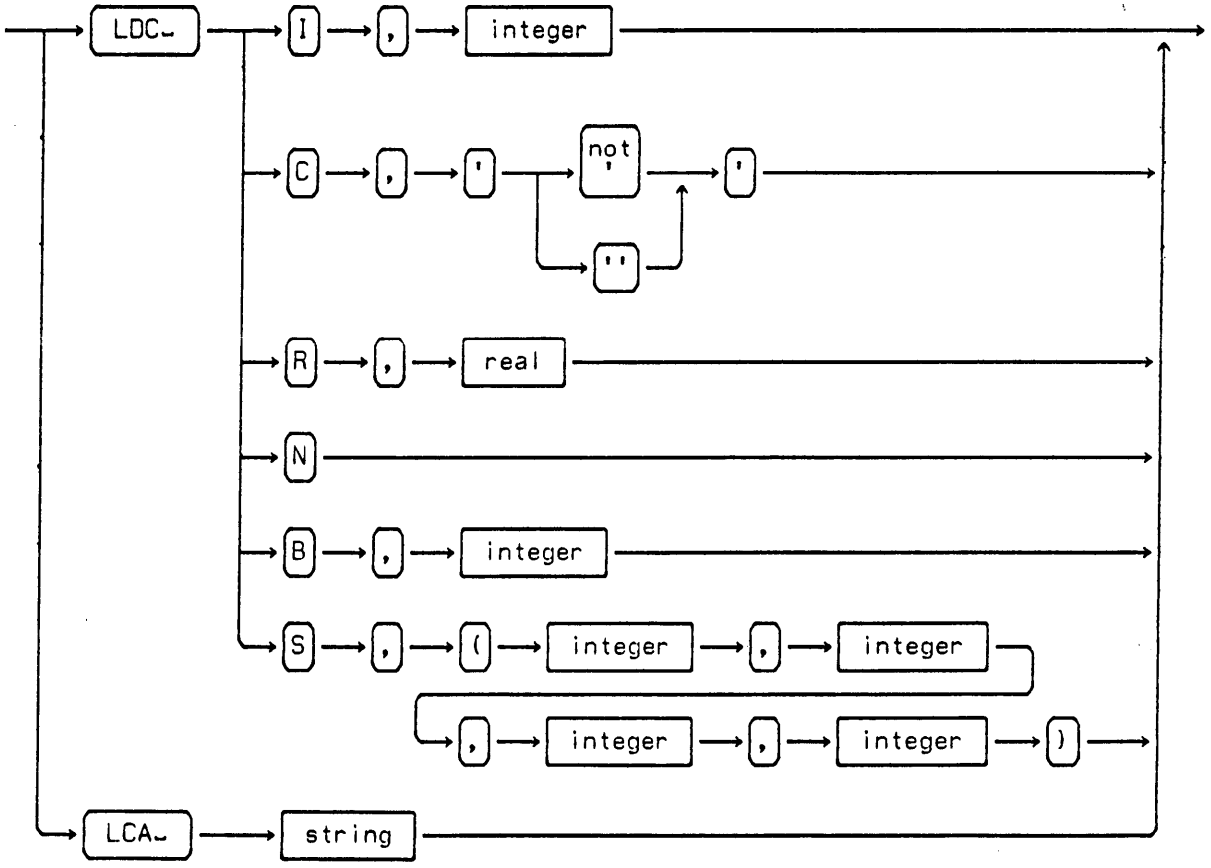
Assembly instruction:



Procedure call instruction:



Instruction for loading constants:




```
opcode class 1 = { ABI, ABR, ADI, ADR, AND, CHR, DIF, DVI, DVR, EOF,
                   FLO, FLT, INN, INT, IOR, MOD, MPI, MPR, NGI, NGR,
                   NOT, ODD, ORD, PRE, RST, SAV, SBI, SBR, SGS, SQI,
                   SQR, SUC, TOF, TON, TRC, UNI }

opcode class 2 = { BGN, IXA, LAO, LOC, MOV, NEW }
```

```
opcode class 3 = { LDA }

opcode class 4 = { FJP, UJP, XJP }

opcode class 5 = { EQU, GEQ, GRT, LEQ, LES, NEQ }

opcode class 6 = { PAR, RET, STO }

opcode class 7 = { DEC, INC, IND, LDO, SRO }

opcode class 8 = { CHK, LOD, STR }

type = { A, B, C, D, H, I, J, M, N, P, Q, R, S, X }

standard proc id = { ATN, CLK, COS, EIO, ELN, EOF, EXP, GET, LOG, NEW,
                     PUT, RDC, RDI, RDR, RDS, RES, REW, RLN, RST, SAV,
                     SIN, SIO, SQT, WLN, WRB, WRC, WRI, WRR, WRS, XIT }

proc id = identifier


integer, string, real, and identifier as defined in PASCAL syntax.
```

B. Instruction and standard procedure summary

Alphabetic List of Instructions:

The stack contents are described in terms of the type of the value on the stack. Please note that this "type" is neither the same as the "types" used in PASCAL source code, nor the same as whatever concept of "type" the eventual target machine may implement. The P-Code emitted by the PASCAL compiler is "aware" of precisely the following set of types: {int,char,real,bool,set,adr}, where "int" means an integer (which may be a quarter-word, half-word, single-word, or double-word), and "real" may be a single-word or a double-word.

Another comment is needed on the handling of multiple size representations for types integer and real. In PASCAL, arithmetic on subrange types may legally yield a result outside the subrange. Thus, the P-Code implementation must be such that arithmetic on quarter-word or half-word integers may yield as large a result as a single-word integer. This does not necessarily imply that such arithmetic will always yield a single-word

result, but rather that it might yield up to a single-word result in order to avoid overflow. The general principle here is that the Stack Computer may make implicit conversions between subrange types and different sizes of types.

Notes:

... = the remainder of the stack, untouched by this instruction.
CTI = compile time instruction, hence no use of stack.
ICRBSA = int,char,real,bool,set, or adr, depending on type param.
ICBA = int,char,bool, or adr, depending on type param.
ICB = int,char, or bool, depending on type param.

Mnem.	Param.	Stack before			Stack after		
		top-2	top-1	top	top-2	top-1	top
ABI			...	int		...	int
ABR			...	real		...	real
ADI		...	int	int		...	int
ADR		...	real	real		...	real
AND		...	bool	bool		...	bool
BGN	value			*CTI*			*CTI*
CHK	t,lb,ub		...	int		...	int
CHR			...	int		...	char
CSP	stdprocid	(see individual standard procedure descriptions)					
CUP	t,nprmfl,procid	(see detailed description of CUP instruction)					
DEC	t,decamt		...	ICBA		...	ICBA
DEF	label DEF value			*CTI*			*CTI*
DIF		...	set	set		...	set
DVI		...	int	int		...	int
DVR		...	real	real		...	real
ENT	t,lev,dslen,etc.	(see detailed description of ENT instruction)					
EOF		(not generated, generates CSP EOF instead)					
EQU	t v M,strlen	...	ICRBSA	ICRBSA		...	bool
FJP	label		...	bool			...
FLO		...	int	any	...	real	any
FLT			...	int		...	real
GEQ	t v M,strlen	...	ICRBSA	ICRBSA		...	bool
GRT	t v M,strlen	...	ICRBSA	ICRBSA		...	bool
INC	t,incamt		...	ICBA		...	ICBA
IND	t,index		...	adr		...	ICRBSA

INN		...	int	set	...	bool
INT		...	set	set	...	set
IOR		...	bool	bool	...	bool
IXA	xmult	...	adr	ICB	...	adr
LAB	label LAB			*CTI*		*CTI*
LAO	levoneadr			...	...	adr
LCA	string			...	...	adr
LDA	lev,reladr			...	...	adr
LDC	I,ivalue			...	...	int
	v C,'char'			...	...	char
	v R,rvalue			...	...	real
	v N			...	...	adr(nil)
	v B,bvalue			...	...	bool
	v S,(i1,i2,i3,i4)			...	...	set
LDO	t,levoneadr			...	...	ICRBSA
LEQ	t v M,strlen	...	ICRBSA	ICRBSA	...	bool
LES	t v M,strlen	...	ICRBSA	ICRBSA	...	bool
LOC	codeloc			*CTI*		*CTI*
LOD	t,lev,reladr			...	...	ICRBSA
MOD		...	int	int	...	int
MOV	nunits	...	adr	adr		...
MPI		...	int	int	...	int
MPR		...	real	real	...	real
MST	lev,fpsiz,rpsiz (see detailed description of MST instruction)					
NEQ	t v M,strlen	...	ICRBSA	ICRBSA	...	bool
NEW	nunits		...	adr		...
NGI			...	int	...	int
NGR			...	real	...	real
NOT			...	bool	...	bool
ODD			...	int	...	bool
ORD			...	ICB	...	int
PAR	t		...	ICRBSA	...	ICRBSA
PRE		(not generated, generates DEC 1 instead)				
RET	t	(see detailed description of RET instruction)				
RST			...	adr		...
SAV			...	adr		...
SBI		...	int	int	...	int
SBR		...	real	real	...	real
SGS			...	ICB	...	set
SQI			...	int	...	int
SQR			...	real	...	real
SRO	t,levoneadr		...	ICRBSA		...

SST	t,procid,lev,i2,i3,i4,i5			*CTI*			*CTI*
STO	t	...	adr	ICRBSA			...
STP				*CTI*			*CTI*
STR	t,lev,reladr		...	ICRBSA			...
SUC	(not generated, generates INC 1 instead)						
TOF				...			...
TON				...			...
TRC			...	real		...	int
UJP	label			...			...
UNI		...	set	set		...	set
XJP	label		...	int			...

Alphabetic List of Standard Procedures/Functions:

Notes:

... = the remainder of the stack, untouched by this instruction.

Mnem.	Action	Stack before				Stack after		
		top-3	top-2	top-1	top	top-2	top-1	top
ATN	arctan function			...	real		...	real
CLK	clock function			...	int		...	int
COS	cosine function			...	real		...	real
EIO	end I/O group			...	adr			...
ELN	EOLN test function			...	adr	...	bool	undef
EOF	EOF test function			...	adr	...	bool	undef
EXP	exponential function			...	real		...	real
GET	get next textfile char			...	adr		...	adr
LOG	natural log function			...	real		...	real
NEW	(not generated, generates NEW instruction instead)							
PAK	(not generated, intended for PACK procedure - unimplemented)							
PUT	put next textfile char			...	adr		...	adr
RDB	read bool from file		...	adr	adr		...	adr
RDC	read char from file		...	adr	adr		...	adr
RDI	read int from file		...	adr	adr		...	adr
RDR	read real from file		...	adr	adr		...	adr

RDS	read string from file	...	adr	adr	int	...	adr
RES	RESET file			...	adr	...	adr
REW	REWRITE file			...	adr	...	adr
RLN	READLN file			...	adr	...	adr
RST	(not generated, generates RST instruction instead)						
SAV	(not generated, generates SAV instruction instead)						
SIN	sine function			...	real	...	real
SIO	start I/O group			...	adr	...	adr
SQT	square root function			...	real	...	real
TRP	TRAP to external adr		...	int	adr		...
WLN	WRITELN file			...	adr	...	adr
WRB	write bool to file	...	adr	bool	int	...	adr
WRC	write char to file	...	adr	char	int	...	adr
WRI	write int to file	...	adr	int	int	...	adr
WRO	(not generated, intended function unknown)						
WRR	write real to file	...	adr	real	int	...	adr
WRS	write string to file	adr	adr	int	int	...	adr
XIT	stop program w/rtn code			...	int		...

C. Detailed instruction descriptions

In the descriptions below, the notation "<int>" means any integer type, namely "Q,H,I,D" indicating quarter-, half-, single-, and double-word integers. The notation "<real>" correspondingly means "R,X" indicating single- and double-word reals.

ABI Evaluates the absolute value of the integer on top of the stack, pops that integer, and pushes the absolute value.

ABR Evaluates the absolute value of the real on top of the stack, pops that real, and pushes the absolute value.

ADI Evaluates the sum of the top two integers on the stack, pops those two integers, and pushes the sum.

ADR Evaluates the sum of the top two reals on the stack, pops those two reals, and pushes the sum.

AND Evaluates the logical AND of the top two booleans on the stack, pops those two booleans, and pushes the logical AND.

BGN (Compile-time instruction) Specifies translator options. If the instruction parameter is 1, translates the P-Code into a 370/assembler source text; otherwise translates it into an object module suitable for loading.

CHK The first parameter in the instruction is a type from the set {A,C,<int>,J,S,P}, indicating that the top item on the stack is an address, character, integer, index (also an integer), an ordinal number for an element of a set, or parameter to a procedure call. The second and third parameters are the lower and upper bounds which are legal for the item on top of the stack; in the case of sets these bounds are determined by the maximum number of elements that can appear in a set. In the case of addresses, the lower bound may be either 0 or -1; if -1 it means that the nil address is also allowed. The top item is tested against these bounds. If it is not between them, an error condition is raised; otherwise nothing happens.

CHR Converts the ordinal integer on top of the stack into a character, pops that integer, and pushes the character equivalent.

CSP Calls the standard procedure specified in the instruction, saving the return address. The exact behavior of the stack depends on which procedure is called; see the descriptions of the standard procedures.

CUP Calls a specified user procedure. The first parameter in the instruction is a type from the set {P,A,B,C,<int>,<real>}, indicating that the procedure is untyped (i.e. of type "procedure") or of type address, boolean, character, integer, or real. The second parameter contains some coded information: it is twice the number of parameters for the procedure called plus a bit which is 1 if and only if a floating point save area is required. The third parameter is the name of the user procedure being called.

DEC The first parameter in the instruction is a type from the set {A,B,C,<int>}, indicating that the top item on the stack is to be treated as an address, boolean, character, integer, or index (also an integer). The second parameter is an integer decrement amount. This instruction examines the top item on the stack and replaces it by the item of the same type whose ordinal number is less than the ordinal number of the original top item by exactly the decrement amount. Loosely, this instruction decrements the top of the stack by an amount given as the second parameter in the instruction.

DEF (Compile-time instruction) Defines the meaning of the label on this instruction to be the value of the integer parameter.

DIF Evaluates the set difference given by subtracting the set on top of the stack from the set which is second on the stack, pops those two sets, and pushes the set

difference.

DVI Evaluates the quotient without remainder (DIV) given by dividing the integer which is second on the stack by the integer on top of the stack, pops those two integers, and pushes the quotient.

DVR Evaluates the floating-point quotient given by dividing the real which is second on the stack by the real which is on top of the stack, pops those two reals, and pushes the quotient.

ENT This is the first instruction executed by any program or procedure. Its exact effect depends on the implementation of the Stack Computer, but its general purpose is to record the information necessary to restore the static environment of the calling routine upon return, and to set up the new static environment for the freshly-invoked procedure. This may include recording static pointers, allocating data areas, updating displays, and so forth. The ENT must be preceded by a proc id which uniquely identifies the procedure or program and which is the label used by CUPs to refer to their destination. The ENT is followed by seven parameters (see syntax diagram). The first parameter is a type from the set {P,A,B,C,<int>,<real>}, indicating that the entered procedure is untyped (i.e. of type "procedure") or of type address, boolean, character, integer, or real. The second parameter is the static level number of the procedure (starting at 1 for the main program). The third parameter is a label whose value is the length of the data area, including any save areas allocated by the MST instruction, for this procedure. The fourth parameter is a prefix of the original PASCAL procedure name. The fifth parameter is a one if a general purpose register save area is required, otherwise zero. The sixth parameter is a one if a floating point register save area is required (assuming this is a real valued function), otherwise zero. The seventh parameter is one to indicate that debugging is in effect if the QRD file is being processed, otherwise zero.

EOF This instruction is never generated, but its function is to check for end-of-file for the input file whose address is on top of the stack. For a detailed description see the standard procedure EOF.

EQU The first parameter in the instruction is a type from the set {A,<int>,C,<real>,B,S,M}, meaning respectively that the top two elements on the stack are addresses, integers, characters, reals, booleans, sets, or the addresses of multiple-unit arrays or records. If the first parameter is M, there will be a second parameter - an integer - which tells how many storage units the structures occupy. This instruction compares the two addresses, integers, characters, reals, booleans, sets, or entire multiple-unit structures for equality, pops the top two elements from the stack, and pushes the boolean TRUE if the items were equal and FALSE

otherwise.

FJP Jumps to the address given in the instruction if the boolean on top of the stack is false; whether true or false, the boolean is popped. The stack must be empty after this instruction is executed.

FLO Converts the integer which is second on the stack to a real value, and replaces the integer by the real, leaving undisturbed the value on top of the stack.

FLT Converts the integer on top of the stack to a real value, and replaces the integer by the real.

GEQ The parameters in this instruction are the same as for the EQU instruction. The items are compared and popped from the stack. If the item which is second on the stack is greater than or equal to the item on top of the stack, the boolean TRUE is pushed; otherwise the boolean FALSE is pushed. Note that in the case of sets, >= refers to the superset operation, and that in the case of multiple unit structures, the structures compared are addressed by the top two elements of the stack but are not themselves on the stack.

GRT The parameters in this instruction are the same as in the EQU instruction, except that the type may not be S (set). The items are compared and popped from the stack. If the item which is second on the stack is greater than the item on top of the stack, the boolean value TRUE is pushed on the stack; otherwise FALSE is pushed. Note that in the case of multiple-unit structures, the structures compared are addressed by the top two elements on the stack, but are not themselves on the stack.

INC The first parameter in the instruction is a type from the set {A,B,C,<int>}, indicating that the top item on the stack is to be treated as an address, boolean, character, integer, or index (also an integer). The second parameter is an integer increment amount. This instruction examines the top item on the stack and replaces it by the item of the same type whose ordinal number is greater than the ordinal number of the original top item by exactly the decrement amount. Loosely, this instruction increments the top of the stack by an amount given as the second parameter in the instruction.

IND The first parameter in the instruction is a type from the set {A,B,C,<int>,<real>,S}; the second parameter is an integer index. The top of the stack contains an address. This instruction gets the item whose address is the sum of the address on the stack plus the index in the instruction. The type of this item is given by the type in the instruction. The address is popped from the stack, and the new item is pushed onto the stack.

INN Checks to see if the ordinal integer which is second from the top of the stack is in the set on top of the stack. Pops the integer and the set, and pushes the boolean TRUE if the integer is a member of the set and FALSE otherwise.

INT Evaluates the set intersection of the set on the top of the stack and the set which is second on the stack, pops those two sets, and pushes the intersection.

IOR Evaluates the logical inclusive OR of the two booleans on top of the stack, pops those two booleans, and pushes the inclusive OR.

IXA The integer parameter in this instruction is a number of storage units required by an instance of a given data type. The index on top of the stack is multiplied by the storage size in the instruction and added to the address which is second on the stack, to give a new address. The base address and index are popped, and the new address is pushed onto the stack.

LAB (Compile-time instruction) Defines the meaning of the label on this instruction to be the current value of the location counter. The stack must be empty when this instruction is seen.

LAO Pushes an address onto the stack. The instruction has as a parameter an integer offset into the data area of static level number 1. The address pushed is the address of the location at that offset in the global (i.e. static level 1) data area.

LCA Pushes the address of the string given in the instruction onto the stack. The string itself will be in the constant area.

LDA The first parameter in the instruction is a static level number; the second parameter is an offset in the most recently activated data area for that static level. Calculates the absolute address of the location thus specified and pushes that address onto the stack. **Warning:** Because the addressing structure of the target machine may be slightly different from that of the Stack Computer, a P-Code program should never LDA an address in the first parameter area unless that address is intended for use outside the local scope, i.e. as a reference parameter to a called procedure.

LDC Pushes a constant onto the stack. The first parameter in the instruction is a type from the set {I,C,R,N,B,S}, indicating that the constant to be loaded is integer, character, real, the nil pointer, boolean, or set. If the type is I, C, or R, the second parameter is a constant of that type, just as expressed in PASCAL. If the type is N, there is no second parameter; the nil pointer is simply pushed. If the type is B, the second parameter is the integer 0 or 1, representing the booleans FALSE and TRUE.

If the type is S, the second parameter is a list of four integers, separated by commas and surrounded by parentheses, whose low order 16 bits can be concatenated to produce the 64-bit representation of the set to be loaded.

LDO Pushes the value of a location in the data area of the outermost static level. The first parameter in the instruction is a type from the set {A,B,C,<int>,<real>,S}, indicating that the item to be loaded is an address, boolean, character, integer, real, or set. The second parameter is the offset of the desired location in the data area of the outermost static level.

LEQ The parameters in this instruction are the same as for the EQU instruction. The items are compared and popped from the stack. If the item which is second on the stack is less than or equal to the item on top of the stack, the boolean TRUE is pushed; otherwise the boolean FALSE is pushed. Note that in the case of sets, <= refers to the subset operation, and that in the case of multiple unit structures, the structures compared are addressed by the top two elements of the stack but are not themselves on the stack.

LES The parameters in this instruction are the same as in the EQU instruction, except that the type may not be S (set). The items are compared and popped from the stack. If the item which is second on the stack is less than the item on top of the stack, the boolean value TRUE is pushed on the stack; otherwise FALSE is pushed. Note that in the case of multiple-unit structures, the structures compared are addressed by the top two elements on the stack, but are not themselves on the stack.

LOC (Compile-time instruction) This instruction appears at regular intervals in the P-Code text to allow identification of code locations. The single integer parameter is the value of the location counter at the time the instruction is encountered. It could theoretically be used actually to set the value of the location counter, but implementations so far do not do this.

LOD Pushes the value of a location in an arbitrary static level. The first parameter in the instruction is a type from the set {A,B,C,<int>,<real>,S}, indicating that the location to be loaded contains an address, boolean, character, integer, real, or set. The second parameter is a static level number; the third parameter is an offset in the data area for that static level.

MOD Evaluates the remainder after integer division produced by dividing the integer which is second on the stack by the integer on top of the stack, pops those two integers, and pushes the remainder.

MOV The parameter in the instruction is a number of storage units. Copies a block of that

many values starting at the address on top of the stack to a block of that many storage units starting at the address which is second on the stack. The stack must be empty after this instruction is executed.

MPI Multiplies the two integers on top of the stack, pops those integers, and pushes the integer product onto the stack.

MPR Multiplies the two reals on top of the stack, pops those reals, and pushes the real product.

MST This instruction is executed preparatory to loading the parameters on the stack for a user procedure call. The MST instruction is followed by code to evaluate and stack the parameters, if any, which is in turned followed by a CUP instruction. Its exact effect depends on the implementation of the Stack Computer, but its general purpose is to record the information necessary to restore the dynamic environment of the calling routine upon return. This may include recording dynamic pointers, allocating save areas, and so forth. The first parameter tells the level of the called procedure. The second parameter is the size in storage units of the first part of the parameter section for the callee. The third parameter is the regparm area size for the callee, and should be the same value as is specified in the SST instruction for the callee. See the SST instruction for details.

NEQ The parameters in this instruction are the same as for the EQU instruction. The items are compared and popped from the stack. If the item which is second on the stack is not equal to the item on top of the stack, the boolean TRUE is pushed; otherwise the boolean FALSE is pushed. Note that in the case of multiple unit structures, the structures compared are addressed by the top two elements of the stack but are not themselves on the stack.

NEW The parameter in the instruction is a number of storage units. Allocates that many new units on the heap, stores NP, which contains the new address of the top of the heap, into the location whose address is on the top of the stack, and pops that address from the stack. The stack must be empty after executing this statement.

NGI Calculates the negative (additive inverse) of the integer on top of the stack, pops that integer, and pushes the integer negative.

NGR Calculates the negative (additive inverse) of the real on top of the stack, pops that real, and pushes the real negative.

NOT Calculates the logical NOT of the boolean value on top of the stack, pops that boolean, and pushes the logical NOT.

ODD Determines whether the integer on top of the stack is odd or even, pops that integer, and pushes the boolean value TRUE if it was odd and FALSE if it was even.

ORD Determines the ordinal value of the character, boolean, or integer value on top of the stack, pops the stack, and pushes this ordinal integer.

PAR Notes that the value on top of the stack is about to be passed as a parameter to a procedure. The only argument to PAR is the type of the formal parameter to which it is being passed.

PRE This instruction is never generated, but its function is to replace the scalar value on top of the stack by its immediate predecessor. The compiler generates a DEC 1 instruction instead.

RET This instruction returns control to the procedure which called the current one. Its exact effect will depend on the implementation of the Stack Computer, but is likely to include resetting of static and dynamic pointers and displays, deallocation of data and save areas, and of course the physical jump back to the calling point. It has a single parameter which is a type from the set {P,A,B,C,<int>,<real>,S}, indicating that the procedure is untyped (i.e. of type "procedure") or is returning a function value of type address, boolean, character, integer, real, or set.

RST Deallocates all locations on the heap which appear on top of the location whose address is on top of the stack by setting NP (the "new" pointer) to that address, and pops that address from the stack. The stack must be empty after executing this statement.

SAV Stores NP, which contains the address of the top element in the heap, at the location whose address is on top of the stack, and then pops that address from the stack. The stack must be empty after executing this statement.

SBI Evaluates the integer given by subtracting the integer on top of the stack from the integer which is second on the stack, pops those two integers, and pushes the integer difference onto the stack.

SBR Evaluates the real given by subtracting the real on top of the stack from the real which is second on the stack, pops those two reals, and pushes the real difference.

SGS Creates a singleton set containing the ordinal integer on the top of the stack, pops that ordinal integer, and pushes the singleton set.

SQI Calculates the square of the integer on top of the stack, pops that integer, and pushes the integer square onto the stack.

SQR Calculates the square of the real on top of the stack, pops that real, and pushes the real square.

SRO Pops a value from the stack into a location in the outermost static level. The first parameter in the instruction is a type from the set {A,B,C,<int>,<real>,S}, indicating that the value to be stored is an address, boolean, character, integer, real, or set. The second parameter is an offset into the data area of the outermost static level. The stack must be empty after this instruction is executed.

SST (Compile-time instruction) Specifies certain attributes of the procedure whose label is the second parameter of the instruction. The first parameter is the type of the procedure, selected from {P,A,B,C,<int>,<real>,S}. The third parameter is the static level of that procedure. The fourth parameter is the size in storage units of the first part of the parameter section. The fifth parameter is the size in storage units of the second part of the parameter section. The sixth parameter is the size in storage units of the locally declared variable storage. The seventh parameter is the regparm area size. This is the size of the initial portion of the first parameter section which is to be kept in registers. It should be at least 0, no more than the constant value MAXPAREG*WORDUNITS given in the P-Code translator, and no more than the size of the first parameter section. In addition, it should stop on a parameter boundary - i.e. the regparm area should not contain half of a double-word parameter. In practice it is desirable to make this value as large as possible subject to these constraints. Note - in no other context should the compiler worry about the fact that registers are used for parameters (except in the MST instruction, which gives this quantity in different contexts).

STO Stores the value on top of the stack into the address which is second on the stack, and pops both from the stack. This instruction has as a parameter a type from the set {A,B,C,<int>,<real>,S}, indicating that the value to be stored is an address, boolean, character, integer, real, or set. The stack must be empty after this instruction is executed.

STP (Compile-time instruction) Signals the end of the input program.

STR Pops the value on top of the stack into a location in an arbitrary static level. The first parameter in this instruction is a type from the set {A,B,C,<int>,<real>,S}, indicating that the value to be stored is an address, boolean, character, integer, real, or set. The second parameter is a static level number; the third parameter is an offset into the data area of that static level. The stack must be empty after this instruction

is executed.

SUC This instruction is never generated, but its effect is to replace the scalar value on top of the stack by its immediate successor. The compiler generates an INC 1 instruction instead.

TOF In the interpreter, turns off the execution trace. In the P-Code translator, turns off one of several kinds of traces based upon a keyword argument. These keyword arguments are not an official part of P-Code, but rather an addition to support the translator; they are described in "The SOPAIPILLA Maintenance Manual."

TON In the interpreter, turns on the execution trace. In the P-Code translator, provides one of several kinds of traces based upon keyword arguments. These keyword arguments are not an official part of P-Code, but rather an addition to support the translator; they are described in "The SOPAIPILLA Maintenance Manual."

TRC Calculates the value found by truncating the real number on top of the stack to an integer value, pops that real, and pushes the integer truncation.

UJP Jumps unconditionally to the label given in the instruction. The stack must be empty after this instruction is executed.

UNI Evaluates the set union of the two sets on top of the stack, pops those two sets, and pushes the union set.

XJP This instruction performs a jump to a location indexed by the ordinal integer on top of the stack. It is used to implement CASE statements. It has as a parameter a label which is first of four lexicographically consecutive labels (e.g. L28, L29, L30, and L31). These four labels have, respectively, the following values:

- The lowest legal value for the index on the stack,
- The highest legal value for the index on the stack,
- The code address of the branch table, and
- The default address to which it should jump if the index is not in the permissible range.

The branch table is a table of UJP (unconditional jump) instructions. This instruction first checks to see whether or not the index is in range; if not it jumps to the default address. If it is in range, it subtracts the lowest legal value from the ordinal integer and jumps to that displacement from the start of the branch table, which is in turn a jump to the appropriate piece of code. In either case, the index is popped from the stack. The stack must be empty after this instruction is executed.

D. Detailed standard procedure descriptions

ATN Evaluates the arctangent of the real on top of the stack, pops that real, and pushes the result.

CLK Uses the integer argument on top of the stack to select among types of clock functions. At present, the only possible argument value is 1. When the argument is 1, the number of milliseconds of run time spent in the PASCAL program thus far is computed. The argument is popped and replaced by pushing the result.

COS Evaluates the cosine of the real on top of the stack, pops that real, and pushes the result.

EIO Signals the end of a group of I/O related instructions (which was started by a CSP SIO instruction). EIO may be used to signal any events which are convenient to place at the end of such a group of I/O instructions, but in particular its function includes popping a single value off the top of the stack. This value is usually the "file address," which is pushed on the stack prior to the SIO and remains there throughout the I/O group.

ELN Tests the EOLN condition for the file whose address is on top of the stack. The address is popped, the boolean result is pushed, and then an undefined value is pushed (so that the following CSP EIO instruction will have something to pop without popping the result).

EOF Tests the EOF condition for the file whose address is on top of the stack. The address is popped, the boolean result is pushed, and then an undefined value is pushed (so that the following CSP EIO instruction will have something to pop without popping the result).

EXP Evaluates the exponential of the real on top of the stack, pops that real, and pushes the result.

GET Performs a GET on the file whose address is on top of the stack, appropriately filling the associated buffer with the gotten value. Leaves the file address on top of the stack. The stack must contain only the file address after this call is completed.

LOG Evaluates the natural logarithm of the real on top of the stack, pops that real, and pushes the result.

NEW This "standard procedure" appears in the implemented P-Code interpreter, but calls to it are never explicitly generated. Instead, the PASCAL compiler generates a NEW instruction (which the PASCAL interpreter translates into a CSP NEW). The stack must be empty after this call is completed. (See also the description of the NEW instruction.)

PAK This procedure is defined (only) in the PASCAL compiler, but calls to it are not yet generated. Apparently it is intended for use by the PASCAL procedure PACK, which is not yet implemented in the P-Code PASCAL compiler.

PUT Performs a PUT on the file whose address is on top of the stack. The associated buffer is then considered to have an undefined value. Leaves the file address on top of the stack. The stack must contain only the file address after this call is completed.

RDB Reads a boolean value (represented externally by 0 for FALSE and 1 for TRUE) from the file whose address is second from the top of the stack into the boolean variable whose address is on top of the stack; note the automatic updating of the buffer associated with the file. The address on top of the stack is popped off, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

RDC Reads a single character from the file whose address is second from the top of the stack into the variable whose address is on top of the stack; note the automatic updating of the buffer associated with the file. The address on top of the stack is popped off, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

RDI Reads an integer from the file whose address is second from the top of the stack into the integer variable whose address is on top of the stack; note the automatic updating of the buffer associated with the file. The address on top of the stack is popped off, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

RDR Reads a real from the file whose address is second from the top of the stack into the real variable whose address is on top of the stack; note the automatic updating of the buffer associated with the file. The address on top of the stack is popped off, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

RDS Reads a string from the file whose address is third from the top of the stack into the area whose address is second from the top of the stack; note the automatic updating

of the buffer associated with the file. The length of the string is given by the integer on top of the stack. The integer and the address on top of the stack are popped off, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

RES Performs the RESET operation for the file whose address is on top of the stack, updating the associated buffer accordingly. Leaves the file address on top of the stack. The stack must contain only the file address after this call is completed.

REW Performs the REWRITE operation for the file whose address is on top of the stack, updating the associated buffer accordingly. Leaves the file address on top of the stack. The stack must contain only the file address after this call is completed.

RLN Performs the READLN operation for the file whose address is on top of the stack, updating the associated buffer accordingly. Leaves the file address on top of the stack. The stack must contain only the file address after this call is completed.

RST This "standard procedure" appears in the implemented P-Code interpreter, but calls to it are never explicitly generated. Instead, the PASCAL compiler generates a RST instruction (which the PASCAL interpreter translates into a CSP RST). The stack must be empty after this call is completed. (See also the description of the RST instruction.)

SAV This "standard procedure" appears in the implemented P-Code interpreter, but calls to it are never explicitly generated. Instead, the PASCAL compiler generates a SAV instruction which the PASCAL interpreter translates into a CSP SAV). The stack must be empty after this call is completed. (See also the description of the SAV instruction.)

SIN Evaluates the sine of the real on top of the stack, pops that real, and pushes the result.

SIO Signals the start of a group of I/O related instructions (which will be ended by a CSP EIO instruction). SIO may be used to signal any events which are convenient to place at the start of such a group of I/O instructions, but it is not required to perform any operation at all. When SIO is executed, the "file address" for the file to which the I/O group applies will be on top of the stack, and it should remain there after the SIO.

SQT Evaluates the square root of the real on top of the stack, pops that real, and pushes the result.

TRP Traps to a user defined external routine, passing on the parameters given as an integer second from the top of the stack, and an address (of some variable) on the top of the stack. The two arguments are popped off the stack. The stack must be empty after this call is completed.

WLN Performs the WRITELN operation for the file whose address is on top of the stack, updating the associated buffer accordingly. Leaves the file address on top of the stack. The stack must contain only the file address after this call is completed.

WRB Writes a boolean value (represented externally by 0 for FALSE and 1 for TRUE) to the file whose address is third from the top of the stack from the boolean value which is second from the top of the stack. The width (in characters) of the field to be written is in the integer on top of the stack. The boolean value and the integer are popped, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

WRC Writes a single character to the file whose address is third from the top of the stack from the character value which is second from the top of the stack. The width (in characters) of the field to be written is in the integer on top of the stack. The character value and the integer are popped, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

WRI Writes an integer to the file whose address is third from the top of the stack from the integer which is second from the top of the stack. The width (in characters) of the field to be written is in the integer on top of the stack. Both integers are popped, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

WRO This procedure is defined (only) in the PASCAL compiler, but no calls to it are generated. It is apparently intended for some I/O write purpose, but its exact intended function is unknown.

WRR Writes a real to the file whose address is third from the top of the stack from the real which is second from the top of the stack. The width (in characters) of the field to be written is in the integer on top of the stack. The real and the integer are popped, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

WRS Writes a string to the file whose address is fourth from the top of the stack from the area whose address is third from the top of the stack. The length of the string is given by the integer on top, whereas the actual width (in characters) of the field to be written is in the integer second from the top of the stack. The string area address

and the two integers are popped, leaving the file address on top of the stack. The stack must contain only the file address after this call is completed.

XIT Terminates program execution, with a final "return code" given by the integer on the top of the stack. The stack must be empty after this call is completed, at which time it vanishes in a cloud of greasy black smoke.