A PASCAL ENVIRONMENT MACHINE (P-CODE)

by

Bent Brun Kristensen

Ole Lehrmann Madsen

Bent Bæk Jensen

DAIMI PB-28

April 1974

Institute of Mathematics University of Aarhus

DEPARTMENT OF COMPUTER SCIENCE

Ny Munkegade - 8000 Aarhus C - Denmark

Phone 06-12 83 55

A PASCAL ENVIRONMENT MACHINE (P-CODE)

by

Bent Bruun Kristensen
Ole Lehrmann Madsen
Bent Bæk Jensen

Abstract:

This paper describes the architecture and instruction set of a machine to support PASCAL. This so-called P-code machine is designed both to be emulated on microprogrammable computers, and to be an intermediate step in code generation for traditional computers. Furthermore, an interpreter on CDC 6400 and a microprogrammed version of P-code on a minicomputer system are described. A precise description of all P-code instructions and examples of PASCAL programs with generated P-code are shown in appendices.

CONTENTS

1. INTRODUCTION

In January 1973 it was decided to implement the language PASCAL [5] on a microprogrammable minicomputer system named RIKKE/MATHILDA [1, 2] presently being built at our department.

It then became natural to define a machine (P-code) to support PASCAL, and microprogram this machine on RIKKE/MATHILDA. A compiler which could generate code for this machine was implemented using a SLR(1) parser generator [11].

As well as being intended to be a total runtime environment for programs translated from PASCAL, the P-code machine is designed to satisfy the following requirements:

1. It should be possible to microprogram an efficient emulator on the RIKKE/MATHILDA system.

2. It should be possible to write code generators from P-code to other machines, P-code thereby becoming an intermediate language in the process of compiling PASCAL programs.

The first of these requirements has been the main goal of the project. The second is to ensure portability of the compiler in the same way as has been done with BCPL and O-code [8].

2. PREMISES

2.1 Easy code generation

In order to make the code generation easy and natural, the P-code machine became a stack machine. For the ease of storage allocation it is a block structured machine.

2.2 Correct code

P-code is designed using the philosophy that the compiler always generates correct code, and that no programmer will ever program directly in P-code.

2.3 Procedure oriented

The unit in P-code is a procedure, which consists of a code segment and, for each activation, a data segment. To each procedure is assigned a name, a block level number, and a record describing the procedure. The address of this record is referred to as the address of the procedure. The procedure descriptions are placed in a special data segment. Variables in a data segment are accessed via the block level number (BN) and an ordinal number (ON) within the data segment.

3. RESOURCES

The runtime environment consists of a runtime stack, an address stack, an evaluation stack, a display, a code memory and a backing store.

3.1 The runtime stack

In the runtime stack are found: a segment which describes all procedures, a global vector of constants, and one or more data segments for each activated procedure. RUSP points to the next free element of RUST.

3.2 The address stack

The address stack (ADST) is used to hold temporary values in the performance of address calculation and integer arithmetic on "small" integers. ADSP points to the next free element of ADST.

3.3 The evaluation stack

The evaluation stack (EVST) is used for evaluation of real and power set expressions. "Long" integers will also be operated here. Fields from packed records are packed and unpacked here. EVSP points to the next free element of EVST.

3.4 The display

The DISPLAY is a vector which contains elements which point to data segments in the RUST. DISPLAY[0] points to the segment describing the procedures. DISPLAY[1] points to the global vector of constants. DISPLAY[2].....DISPLAY[DISP-1] point to the data segments of the procedures which are visible from the currently active procedure, using the normal scope rules of PASCAL. As seen, the block numbering starts with 2. When entering and leaving a procedure, the DISPLAY is automatically updated.

3.5 The code memory

The code memory (CM) is the place where the code segment of a procedure resides for execution. A code segment is a unit consisting of purely re-entrant code. One can only leave the segment by using special enter and exit instructions. In particular, one cannot jump to locations in other segments. CM is byte oriented in the sense that one can address down to each byte in the segment. All jumping is relative to the instruction counter (IC), making the code completely reallocatable.

3.6 The backing store

The backing store (BS) is used to hold a P-code program, i.e., a compiled PASCAL program.It consists of blocks of bytes, and every block is directly accessible. It contains information to start execution of each P-code program. A copy of all code segments is kept here. At entry to a procedure its code segment is transferred to CM. A procedure may have a value segment in BS, used to initialize part of the data segment on entry to the procedure. BS also contains the procedure descriptions (data segment 0) and the global vector of constants (data segment 1). The first block of BS contains information about the number of procedures, and pointers in BS to data segment 0 and 1. At execution start, data segments 0 and 1 are transferred to the RUST, and execution starts with the procedure described at address 1 in data segment 0.

4. PROGRAM AND DATA FORMS

4.1 The procedure description

Data segment 0 is a vector of records. Each record describes a procedure. The following information is kept in each record:

PNAME	the name of the procedure.
BN	the block level of the procedure.
PRESENT	a boolean variable, which is true if the code segment of the procedure is present in CM.
ENTRY	if PRESENT, absolute address in CM of the code segment.
SEGLENGTH	number of BS blocks occupied by the procedure.
SEGADDR	the address of the code segment in BS.
DATALENGTH	length of the data segment in RUST.
VALUELENGTH	number of BS blocks occupied by the value segment.
VALUEADDR	address in BS of the value transfer.
VALUESTART	start address in the data segment of the value transfer.
PARAMETER DESCRIPTION	for each parameter there is a variable indicating whether it is a const, var, procedure or function parameter.

4.2 The global constants

The data segment at level 1 is used to hold long constants, i.e., long integers, reals, powersets and string constants. Small integer constants are stored as arguments of the instructions.

4.3 Instruction format

All instructions consist of one or more bytes. The operation code always occupies one byte. The instructions of the P-code machine can be divided into three groups, depending on whether they have no argument, one argument or two arguments.

The first group consists mostly of stacktop operations, such as plus, minus, mult, etc.

The second group consists of jump instructions and a special version of the boolean, relational, and arithmetic dyadic operations, which operate on a stacktop element and the instruction argument. The argument occupies one or two bytes.

The third group is load/store instructions, where the first argument (one byte) is the BN, and the second argument (two bytes) is the ON. There also exist instructions with two one-byte arguments.

A precise description of all instructions is found in appendix A.

4.4 Calling sequence

To enter a procedure the following sequence of instructions must be performed:

MARK

{ evaluation of actual parameters

ENTER N

The first instruction reserves a word on RUST to contain return information. Next the actual parameters must be evaluated and placed on RUST, and at last the entering takes place. N is the address of the record in data segment 0, where the procedure is described. The mark will contain the following information:

DYN — a pointer to the data segment of the calling procedure.

STAT — a pointer to a data segment of the surrounding procedure in the static nesting in the PASCAL program. It points to the data segment which is accessible from the entered procedure.

RETURN — return address in the code segment of the calling procedure.

PDADDR — the address in data segment 0 of the calling procedure.

The ENTER instruction allocates a data segment immediately after the mark. This means that anything pushed on RUST after a MARK operation will be the first locations of the data segment (i.e. actual parameters).
Part of the data segment will be initialized with the value segment (if it exists) from BS.
If the code segment of the called procedure is not in CM, it will be transferred from BS.
Finally the DISPLAY is updated.

Four exit instructions exist.

EXIT — normal exit. The data segment is popped off RUST. Return is made to the procedure described in the mark on top of RUST. The mark is also popped.

EXITF — as EXIT, but location 1 in the data segment is pushed on ADST.

EXITFEV — as EXITF, but location 1 is pushed on EVST.

EXITL BN, ADDR — all data segments above DISPLAY[BN] are popped off RUST. Execution continues in address ADDR of the code segment corresponding to the data segment pointed at by DISPLAY[BN]. (1 < BN < DISP-2).

It is possible to pass a procedure as a parameter. For this purpose a word describing the procedure must be established. The instruction GPPW N (N is the address in data segment 0) generates such a word and pushes it on RUST.

The parameter word contains:

PADR	the address in data segment 0 of the procedure being passed as parameter.
STATIC	a pointer to the accessible data segment of the procedure surrounding the procedure passed as parameter. (Used to update the DISPLAY when activating the parameter.)

The following sequence of instructions will activate the procedure:

MARK

{ evaluation of actual parameters

ENTERP BN ON

The address couple (BN, ON) must be the address of a word generated by a GPPW instruction.

For further explanation the reader is referred to the examples in appendix B.

5. IMPLEMENTATION

5.1 General

The P-code machine has a structure which makes it relatively easy to write interpreters for it in high level languages. It is more difficult to say anything about microimplementations because the structures and microlanguages differ from machine to machine.

We have not yet tried to write a code generator from P-code to machine code of traditional machines. On the other hand, a P-code program is much like a reverse Polish form of the PASCAL program, hence standard techniques can be used [9, 10].

5.2 CDC 6400 implementation

An interpreter for the P-code machine has been written in PASCAL on CDC 6400 with the purpose to gain experience, and to experiment with modifications before the actual implementation on RIKKE/MATHILDA. Such an interpreter can be implemented very quickly and it is an easy way to get a PASCAL compiler. It will be slow, but fast enough for small student programs.

It was absolutely necessary for us to have the interpreter because it was the only way we could test the compiler.

5.3 The RIKKE/MATHILDA implementation

RIKKE is a 16 bit minicomputer with a 16 bit memory. MATHILDA is a 64 bit version of RIKKE, and is going to act as a fast functional unit for RIKKE. A 64 bit memory (wide store) controlled by RIKKE is connected to RIKKE/MATHILDA. For further details the reader is referred to [1, 2].

The structures of the P-code machine are realized as follows:

1. The RUST resides in the wide store.
2. The ADST is a register group of 16 registers in RIKKE.
3. The EVST is a register group of 16 or 256 registers in MATHILDA.
4. The DISPLAY is a register group of 16 registers in RIKKE.
5. CM is placed in the 16 bit memory of RIKKE.

6. A byte is 8 bit, (i.e. only 16 bit integers are implemented).
7. It is the least significant bits of a 64 bit word that are used in transfers between ADST and RUST or EVST.

The idea of this implementation is that RIKKE shall do the instruction decoding, address calculation, integer arithmetic, etc. Complicated operations are performed in MATHILDA. It is the idea to experiment with advanced floating point systems. Several proposals by Peter Kornerup and Bruce Shriver exist [3, 4].

The standard floating point instructions in the first version of P-code will be a subset of an instruction set implemented by Kaja Lando.

6. EVALUATION

The work of designing a stack machine for supporting PASCAL has been quite valuable for the authors because the instruction set was not a manifest set at the beginning of the compiler writing. Every instruction, which was not obvious, was discussed in detail before any decision of including it in P-code was taken. Several instructions, found in earlier versions, have been removed from this (so far) final version, some new instructions have been created and some are not yet used. But for the ease of developing the compiler further and for the sake of completeness, the instructions were included in P-code. Only a few of the P-code instructions have been difficult to implement such as the enter and exit instructions. They are quite complicated.

The P-code machine does not support all the changes to PASCAL mentioned in [7]. No facility to support the dynamic allocation of the class variable exists. However, it is intended to extend P-code to cover this new facility when the new PASCAL compiler becomes better known.

Because of the P-code machine and a parser generator, the work of writing a PASCAL compiler has been concentrated to semantic analysis, which is the interesting part of the compiler writing.

The authors want to thank their advisor Peter Kornerup who has had great influence on this project. They also want to thank Robert F. Rosin who read the manuscript and gave valuable comments and critisism.

Appendix A

The following pages contain a precise description of all P-code instructions. The description consists of (1) the symbolic name of the instruction, (2) the argument(s) (in brackets the number of bytes occupied by each), (3) a PASCAL description and (4) an English description.

In the PASCAL description the following abbreviations are used:

IC	instruction counter of the P-code machine
α MARK	address of the current active mark
ACP	address in data segment 0 of the current active procedure

The term "procedure N" means the procedure described at address N in data segment 0.

SYMBOLIC NAME	ARGUMENTS	FUNCTION	COMMENTS

EXIT
EXITF
EXITFEV
		IF EXITF THEN
		BEGIN
		ADST[ADSP]:=RUST[DISPLAY[DISP-1]+1];
		ADSP:=ADSP+1;
		END ELSE
		IF EXITFEV THEN
		BEGIN
		EVST[EVSP]:=RUST[DISPLAY[DISP-1]+1];
		EVSP:=EVSP+1;
		END;
		RUSP:= α MARK;
		α MARK:=RUST[RUSP].DYN;
		N:=RUST[α MARK].PDADDR;
		DISP:=RUST[DISPLAY[0]+N].BN+1;
		
		P:= α MARK;
		Q:=DISP-1;
		REPEAT
		DISPLAY[Q]:=P;
		Q:=Q-1;
		P:=RUST[P].STAT;
		UNTIL DISPLAY[Q]=P;
		
		IF ¬(RUST[DISPLAY[0]+N].PRESENT) THEN FETCH (N);
		IC:=RUST[DISPLAY[0]+N].ENTRY+RUST[RUSP].RE-
		TURN;
		ACP:=N;
			These instructions will return the control to the calling procedure. The data segment is popped off the RUST and the DISPLAY is restored. EXITF also pushes location 1 of the data segment on ADST. EXITFV pushes location 1 on EVST.

EXITL	BN(1) ADDR(2)	Q:=DISPLAY[BN+1]; WHILE Q ≠ DISPLAY[BN] DO Q:=RUST[Q].DYN; ACP:=RUST[Q].PDADDR; IF ¬ (RUST[DISPLAY[0]+ACP].PRESENT) THEN FETCH (ACP); RUSP:=Q; α MARK := DISPLAY[BN]; DISP:=BN+1; IC:=RUST[DISPLAY[0]+ACP].ENTRY+ADDR;	All data segments above DISPLAY[BN] are popped off RUST. Execution continues in address ADDR of the code segment corresponding to the data segment pointed at by DISPLAY[BN]. (1 < BN < DISP-2);
COPALI	BN(1) ON(2)	N:=RUST[DISPLAY[BN]+ON].PADR; I:=ADST[ADSP-1]; ADSP:=ADSP-1; IF {parameter no. "I" of procedure "N" is a "CONST" parameter} THEN RUST[RUSP]:=RUST[ADST[ADSP-1]] ELSE RUST[RUSP]:=ADST[ADSP-1]; ADSP:=ADSP-1; RUSP:=RUSP+1;	This instruction is used when evaluating variable parameters for a formal procedure. It is not possible to see, whether such a parameter is a "VAR" on "CONST" parameter. See "COPALA".
COPALA	N(1 or 2)	I:=ADST[ADSP-1]; ADSP:=ADSP-1; IF {parameter no. "I" of procedure "N" is a "CONST" parameter} THEN RUST[RUSP]:=RUST[ADST[ADSP-1]] ELSE RUST[RUSP]:=ADST[ADSP-1]; ADSP:=ADSP-1; RUSP:=RUSP+1;	This instruction makes it possible to avoid "FORWARD"declarations of procedures. In a call of a procedure it is not indicated whether a variable parameter is a "VAR" or a "CONST" parameter (shall the address or the contents of the variable be passed as the parameter). The instruction checks this on run-time. Of course this instruction can be avoided in a multipass compiler.

SYMBOLIC NAME	ARGUMENTS	FUNCTION	COMMENTS
GPPW	N(1 or 2)	B:=RUST[DISPLAY[0]+N].BN; RUST[RUSP].STATIC:=DISPLAY[B-1]; RUST[RUSP].PADR:=N; RUSP:=RUSP+1;	A parameter word is generated for the procedure described at Address "N" in datasegment 0. The instruction is used when the procedure is passed as a parameter. The parameter word contains the descriptor-address (N), and the first element to be used in the static chain, when N actually is called.
MARK		ADST[ADSP]:=RUSP; ADSP:=ADSP+1; RUST[RUSP].DYN:= α MARK; RUSP:=RUSP+1;	Reserves a word on RUST to contain return-information, save old value of RUSP on ADST.
ENTER	N(1 or 2)	DISP:=RUST[DISPLAY[0]+N].BN; S:=DISPLAY[DISP-1]; ADSP:=ADSP-1; α MARK:= ADST[ADSP]; RUST[α MARK].PDADDR:=ACP; RUST[α MARK].STAT:=S; RUST[α MARK].RETURN:= IC+1-RUST[DISPLAY[0]+ACP].ENTRY; DISPLAY[DISP]:=α MARK; DISP:=DISP+1; IF ¬ (RUST[DISPLAY[0]+N].PRESENT) THEN FETCH(N); INITBLOCK; {VALUE LENGTH, VALUE START, DATALENGTH and VALUEADDR in the description record are used to transfer a block of data from BS to initialize part of the procedure's data segment.} ACP:=N; IC:=RUST[DISPLAY[0]+N].ENTRY;	The instruction is used to enter the procedure described at address "N" in data segment 0. The display is updated. If the procedure is not in memory, it is fetched from BS. Part of the data segment of the procedure is possibly initialized.
ENTERP	BN(1) ON(2)	N:=RUST[DISPLAY[BN]+ON].PADR; S:=RUST[DISPLAY[BN]+ON].STATIC; DISP:=RUST[DISPLAY[0]+N].BN; ADSP:=ADSP-1; α MARK:=ADST[ADSP]; RUST[α MARK].PDADDR:=ACP; RUST[α MARK].STAT:=S; RUST[α MARK].RETURN:= IC+1-RUST[DISPLAY[0]+ACP].ENTRY; DISP:=DISP+1; P:= α MARK; Q:=DISP-1; REPEAT DISPLAY[Q]:=P; Q:=Q-1; P:=RUST[P].STAT; UNTIL DISPLAY[Q]=P; IF ¬(RUST[DISPLAY[0]+N].PRESENT) THEN FETCH(N); INITBLOCK; {see ENTER} ACP:=N; IC:=RUST[DISPLAY[0]+N].ENTRY;	The instruction is used to enter a procedure, which is a formal parameter. "BN", "ON" is the address of a word containing a description of the procedure, which is actually to be entered. The instruction GPPW must be used to create such a word. ENTERP does essentially the same as ENTER, except that updating of the DISPLAY is more complicated.

SYMBOLIC NAME	ARGUMENTS	FUNCTION	COMMENTS
FLOATC	C(1)	EVST[EVSP]:=FLOAT(C); EVSP:=EVSP+1;	The constant C is floated to real which is placed on EVST
SETBIT		EVST[EVSP-1]:=EVST[EVSP-1] ∨ [ADST[ADSP-1]] ADSP:=ADSP-1;	Bit no. "ADST[ADSP-1]" is set to one in the top element of EVST. The ∨ is meant as set-union.
TESTBIT		ADST[ADSP-1]:=ADST[ADSP-1] IN EVST[EVSP-1]; EVSP:=EVSP-1 ;	If bit no. "ADST[ADSP-1]" is one in the top element of EVST, a true value is put on ADST, else a false value.
TESTBITC	C(1)	ADST[ADSP]:=C IN EVST[EVSP-1]; ADSP:=ADSP+1; EVSP:=EVSP-1;	If bit no. C is one in the top element of EVST, a true value is put on ADST else a false value.
DOUBLE		ADST[ADSP]:=ADST[ADSP-1]; ADSP:=ADSP+1;	The top element of ADST is duplicated.
DELETE		ADSP:=ADSP-1;	Top element of ADST is deleted
SWITCH		I:=ADST[ADSP-1]; ADST[ADSP-1]:=ADST[ADSP-2]; ADST[ADSP-2]:=I;	The two top elements of ADST are interchanged.
SWITCHEV		S:=EVST[EVSP-1]; EVST[EVSP-1]:=EVST[EVSP-2]; EVST[EVSP-2]:=S;	The two top elements of EVST are exchanged.
NOOP		;	No operation.
MSKIN	I(1) J(1)	$EVST[EVSP-2]_{<W-I,\ W-I-J+1>} := EVST[EVSP-1]_{<J-1,\ 0>}$ EVSP:=EVSP-1;	EVST is assumed to contain W+1 bits per word. EVST[I] $EVST[I]_{<m,n>}$ means bit n,n+1,...,m of EVST[I]. The instruction is used to pack a field into a packed record.
MSKOUT	I(1) J(1)	$EVST[EVSP-1] := EVST[EVSP-1]_{<W-I,\ W-I-abs(J)+1>}$ IF J<0 THEN perform sign extension.	Is used to pack a field out of a packed record. If J is negative then the field is treated as a signed number.
INCL		EVST[EVSP-2]:=EVST[EVSP-2] ≤ EVST[EVSP-1]; EVSP:=EVSP-1;	The powerset operation ⊆ (set inclusion) is performed on the two top elements of EVST.
EXCL		EVST[EVSP-2]:=EVST[EVSP-2] ≥ EVST[EVSP-1]; EVSP:=EVSP-1;	The powerset operation ⊇ (set inclusion) is performed on the two top elements of EVST.
NON		ADST[ADSP-1]:= ¬ ADST[ADSP-1];	The boolean value of the top element of ADST is negated.
ODD		IF ODD(ADST[ADSP-1]) THEN ADST[ADSP-1]:=TRUE ELSE ADST[ADSP-1]:=FALSE	If the top element of ADST is odd, it is replaced by a TRUE value, else by a FALSE value.
LN	N(1 or 2)	ADST[ADSP]:=N; ADSP:=ADSP+1;	Push the constant "N" on ADST.

SYMBOLIC NAME	ARGUMENTS	FUNCTION	COMMENTS
EQ		ADST[ADSP-2]:=ADST[ADSP-2] = ADST[ADSP-1]; ADSP:=ADSP-1;	"EQUAL" on ADST
NE		ADST[ADSP-2]:=ADST[ADSP-2] ≠ ADST[ADSP-1]; ADSP:=ADSP-1;	"NOT EQUAL" on ADST
LTC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] < C;	The relational operation "LESS THAN" is performed between the top element of ADST and the argument C.
GTC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] > C;	
LEC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] ≤ C;	
GEC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] ≥ C;	
EQC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] = C;	
NEC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] ≠ C;	
LTEV		ADST[ADSP]:=EVST[EVSP-2] < EVST[EVSP-1]; EVSP:=EVSP-2; ADSP:=ADSP+1;	The relational operation "LESS THAN" is performed between the two top elements of EVST(real values).
GTEV		ADST[ADSP]:=EVST[EVSP-2] > EVST[EVSP-1]; EVSP:=EVSP-2; ADSP:=ADSP+1;	"GREATER THAN" on EVST
LEEV		ADST[ADSP]:=EVST[EVSP-2] ≤ EVST[EVSP-1]; EVSP:=EVSP-2; ADSP:=ADSP+1;	"LESS THAN or EQUAL" on EVST
GEEV		ADST[ADSP]:=EVST[EVSP-2] ≥ EVST[EVSP-1]; EVSP:=EVSP-2; ADSP:=ADSP+1;	"GREATER THAN or EQUAL" on EVST
EQEV		ADST[ADSP]:=EVST[EVSP-2] = EVST[EVSP-1]; EVSP:=EVSP-2; ADSP:=ADSP+1;	"EQUAL" on EVST
NEEV		ADST[ADSP]:=EVST[EVSP-2] ≠ EVST[EVSP-1]; EVSP:=EVSP-2; ADSP:=ADSP+1;	"NOT EQUAL" on EVST
EQV	N(1 or 2)	BOO:=FALSE; J:=ADST[ADSP-1]; K:=ADST[ADSP-2]; ADSP:=ADSP-1; FOR I:=0 TO N-1 DO IF RUST[J+I] ≠ RUST[K+I] THEN GOTO 1; BOO:=TRUE; 1: ADST[ADSP-1]:=BOO;	The vectors of length "N" starting at addresses "J" and "K" in RUST are compared. If they are equal a TRUE value is pushed on ADST, else a FALSE value is pushed
NEV	N(1 or 2)	BOO:=TRUE; J:=ADST[ADSP-1]; K:=ADST[ADSP-2]; ADSP:=ADSP-1; FOR I:=0 TO N-1 DO IF RUST[J+I] ≠ RUST[K+I] THEN GOTO 1; BOO:=FALSE; 1: ADST[ADSP-1]:=BOO;	The vectors of length "N" starting at addresses "J" and "K" in RUST are compared. If they are not equal, a TRUE value is pushed on ADST, else a FALSE value is pushed.
ANDEV		EVST[EVSP-2]:=EVST[EVSP-2] ∧ EVST[EVSP-1]; EVSP:=EVSP-1;	Intersection (logical and) between the two top elements of EVST.
OREV		EVST[EVSP-2]:=EVST[EVSP-2] ∨ EVST[EVSP-1]; EVSP:=EVSP-1;	Union (logical or) between the two top elements of EVST

SYMBOLIC NAME	ARGUMENTS	FUNCTION	COMMENTS
MULT		ADST[ADSP-2]:= ADST[ADSP-2]*ADST[ADSP-1]; ADSP:=ADSP-1;	Integer multiplication on two top elements of ADST
DIV		ADST[ADSP-2]:=ADST[ADSP-2]DIV ADST[ADSP-1]; ADSP:=ADSP-1;	Integer division on two top elements of ADST
REM		ADST[ADSP-2]:=ADST[ADSP-2]MOD ADST[ADSP-1]; ADSP:=ADSP-1;	Remainder by integer division of two top elements of ADST
FLOAT		EVST[EVSP]:=FLOAT(ADST[ADSP-1]); EVSP:=EVSP+1; ADSP:=ADSP-1;	Top element on ADST is floated to real and moved to EVST
FIX		ADST[ADSP]:=TRUNC (EVST[EVSP-1]); ADSP:=ADSP+1; EVSP:=EVSP-1;	Top element on EVST is truncated to integer and moved to ADST
NEG		ADST[ADSP-1]:=-ADST[ADSP-1];	Sign change on top element of ADST
ABS		ADST[ADSP-1]:=ABS(ADST[ADSP-1]);	The top element of ADST is changed to its absolute value
PLUSC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1]+C;	The constant C is added to the top element of ADST
MINUSC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1]-C;	The constant C is subtracted from the top element of ADST
DIVC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] DIV C;	The top element of ADST is integer divided by the constant C
REMC	C(1)	ADST[ADSP-1]:=ADST[ADSP-1] MOD C;	The remainder from integer division of the top element of ADST by the constant C is placed on ADST
PLUSEV		EVST[EVSP-2]:=EVST[EVSP-2]+EVST[EVSP-1]; EVSP:=EVSP-1;	Floating point addition of the two top elements of EVST
MINUSEV		EVST[EVSP-2]:=EVST[EVSP-2]-EVST[EVSP-1]; EVSP:=EVSP-1;	Floating point subtraction of the two top elements of EVST
MULTEV		EVST[EVSP-2]:=EVST[EVSP-2]*EVST[EVSP-1]; EVSP:=EVSP-1;	Floating point multiplication of the two top elements of EVST
DIVEV		EVST[EVSP-2]:=EVST[EVSP-2]/EVST[EVSP-1]; EVSP:=EVSP-1;	Floating point division of the two top elements of EVST
NEGEV		EVST[EVSP-1]:= -EVST[EVSP-1];	Sign change on top element of EVST
ABSEV		EVST[EVSP-1]:=ABS(EVST[EVSP-1]);	The top element of EVST is changed to its absolute value
LT		ADST[ADSP-2]:=ADST[ADSP-2] < ADST[ADSP-1]; ADSP:=ADSP-1;	The relational operation "LESS THAN" is performed on the top elements of ADST
GT		ADST[ADSP-2]:=ADST[ADSP-2] > ADST[ADSP-1]; ADSP:=ADSP-1;	"GREATER THAN" on ADST
LE		ADST[ADSP-2]:=ADST[ADSP-2] ≤ ADST[ADSP-1]; ADSP:=ADSP-1;	"LESS THAN OR EQUAL" on ADST
GE		ADST[ADSP-2]:=ADST[ADSP-2] ≥ ADST[ADSP-1]; ADSP:=ADSP-1;	"GREATER THAN OR EQUAL" on ADST

SYMBOLIC NAME	ARGUMENTS	FUNCTION	COMMENTS
SADI	BN(1) ON(2)	RUST [DISPLAY[BN]+ON]:=ADST[ADSP-1]; ADSP:=ADSP-1;	Indexed store in RUST from ADST
LADI	BN(1) ON(2)	ADST[ADSP]:=RUST[DISPLAY[BN]+ON]; ADSP:=ADSP+1;	Indexed load from RUST to ADST
LAD		ADST[ADSP-1]:=RUST[ADST[ADSP-1]];	Absolute load from RUST to ADST
SAD		RUST[ADST[ADSP-2]]:=ADST[ADSP-1]; ADSP:=ADSP-2;	Absolute store in RUST from ADST
SEVI	BN(1) ON(2)	RUST[DISPLAY[BN]+ON]:=EVST[EVSP-1]; EVSP:=EVSP-1;	Indexed store in RUST from EVST
LEVI	BN(1) ON(2)	EVST[EVST]:=RUST[DISPLAY[BN]+ON]; EVSP:=EVSP+1;	Indexed load from RUST to EVST
SEV		RUST[ADST[ADSP-1]]:=EVST[EVSP-1]; ADSP:=ADSP-1; EVSP:=EVSP-1;	Absolute store in RUST from EVST
LEV		EVST[EVSP]:=RUST[ADST[ADSP-1]]; EVSP:=EVSP+1; ADSP:=ADSP-1;	Absolute load from RUST to EVST
LA		ADST[ADSP]:=EVST[EVSP-1]; ADSP:=ADSP+1; EVSP:=EVSP-1;	Top element of EVST is moved to ADST
SA		EVST[EVSP]:=ADST[ADSP-1]; EVSP:=EVSP+1; ADSP:=ADSP-1;	Top element of ADST is moved to EVST
LADR	BN(1) ON(2)	ADST[ADSP]:=DISPLAY[BN]+ON; ADSP:=ADSP+1;	The absolute address of BN, ON in RUST is pushed on ADST
SADV	N(1 or 2)	D:=ADST[ADSP-1]; S:=ADST[ADSP-2]; ADSP:=ADSP-2; FOR I:=0 TO N-1 DO RUST[D+I]:=RUST[S+I];	N elements in RUST are copied to another place in RUST
MVADRU		RUST[RUSP]:=ADST[ADSP-1]; RUSP:=RUSP+1; ADSP:=ADSP-1;	The top element of ADST is moved to RUST
MVEVRU		RUST[RUSP]:=EVST[EVSP-1]; RUSP:=RUSP+1; EVSP:=EVSP-1;	The top element of EVST is moved to RUST
JMP		IC:=IC+ADST[ADSP-1]; ADSP:=ADSP-1;	Jump to a computed address
JMPU	K(1 or 2)	IC:=IC+K;	Unconditional jump
JMPF	K(1 or 2)	IF ADST[ADSP-1]=FALSE THEN IC:=IC+K; ADSP:=ADSP-1;	Jump false
JMPT	K(1 or 2)	IF ADST[ADSP-1]= TRUE THEN IC:=IC+K; ADSP:=ADSP-1;	Jump true
JMPFN	K(1 or 2)	IF ADST[ADSP-1]=FALSE THEN IC:=IC+K ELSE ADSP:=ADSP-1;	Jump false and keep a false on ADST
JMPTN	K(1 or 2)	IF ADST[ADSP-1]= TRUE THEN IC:=IC+K ELSE ADSP:=ADSP-1;	Jump true and keep a true on ADST
PLUS		ADST[ADSP-2]:=ADST[ADSP-2]+ADST[ADSP-1]; ADSP:=ADSP-1;	Integer addition on two top elements of ADST
MINUS		ADST[ADSP-2]:=ADST[ADSP-2]-ADST[ADSP-1]; ADSP:=ADSP-1;	Integer subtraction on two top elements of ADST

Appendix B

Following are two examples of PASCAL programs run on the compiler based on P-code. After the body of each procedure, the P-code which has been generated for that procedure is listed.

A few instructions concerning input/output not mentioned in the P-code description appear in the listing.

INCH ININT OUTCH OUTINT
Preliminary I/O instructions from paper tape reader to ADST and from ADST to printer. INCH reads a char, OUTCH writes a char, ININT reads an integer, and OUTINT writes an integer.

The number listed before each instruction is the byte address inside the code segment.

```
PROCEDURE P(B : BOOLEAN ; PROCEDURE Q) ;
VAR  X: INTEGER;

    PROCEDURE R ;
    BEGIN ↱BODY OF R↓
       X:=X+1;
    END;

         0 : LADI              3        3
         4 : PLUSC             1
         6 : SADI              3        3
        10 : EXIT
 ↱R↓

 BEGIN ↱BODY OF P↓
    X:=0;
    IF B THEN Q ELSE Q(TRUE,R);
    WRITE(≡ ≡,X,EOL);
 END;

         0 : LN                0
         2 : SADI              3        3
         6 : LADI              3        1
        10 : JMPF             14
        14 : MARK
        15 : NOOP
        16 : ENTERP            3        2
        20 : JMPU             14
        24 : MARK
        25 : LN                1
        27 : MVADRU
        28 : GPPW              9
        30 : ENTERP            3        2
        34 : LN               45
        36 : OUTCH
        37 : NOOP
        38 : LADI              3        3
        42 : OUTINT
        43 : LN                0
        45 : OUTCH
        46 : EXIT
↱P↓

BEGIN ↱MAIN PROGRAM↓
   P(FALSE,P);
END.

         0 : MARK
         1 : LN                0
         3 : MVADRU
         4 : GPPW              5
         6 : ENTER             5
         8 : STOP


***** BOBS P-CODE SIMULATION *****

         0
         1
```

Example 2

```
 ↱ RED,WHITE AND BLUE IS READ INTO ARRAY A. THE ARRAY IS THEN
     SORTED IN THE ORDER RED,WHITE AND BLUE ↓
CONST
    N=20;
TYPE
    COLOUR=(RED,WHITE,BLUE);
VAR
    A: ARRAY[1..N] OF COLOUR;
    CH : CHAR;
    V : COLOUR;
    I,R,W,B :INTEGER;

PROCEDURE SWAP(I,J:INTEGER);
VAR
    C : COLOUR;
BEGIN ↱ BODY OF PROCEDURE SWAP ↓
    C:=A[J];
    A[J]:=A[I];
    A[I]:=C;
END;
```

```
 0 : LADI          3     2
 4 : LADR          2     0
 8 : PLUS
 9 : LAD
10 : SADI          3     3
14 : LADI          3     2
18 : LADR          2     0
22 : PLUS
23 : NOOP
24 : LADI          3     1
28 : LADR          2     0
32 : PLUS
33 : LAD
34 : SAD
35 : NOOP
36 : LADI          3     1
40 : LADR          2     0
44 : PLUS
45 : NOOP
46 : LADI          3     3
50 : SAD
51 : EXIT
```

```
FUNCTION COL(I: INTEGER): COLOUR;
BEGIN ↱ BODY OF FUNCTION COL ↓
   COL:=A[I];
END;
```

```
 0 : LADI                    3          2
 4 : LADR                    2          0
 8 : PLUS
 9 : LAD
10 : SADI                    3          1
14 : EXITF
```

```
PROCEDURE RESULT;
BEGIN ↱BODY OF PROCEDURE RESULT ↓
   FOR I:=1 TO N DO
    CASE A[I] OF
     RED  : WRITE(≡ RED≡) ;
     WHITE: WRITE(≡ WHITE≡) ;
     BLUE : WRITE(≡ BLUE≡);
    END;
   WRITE(EOL);
END;
```

```
  0 : LN          1
  2 : DOUBLE
  3 : LEC        20
  5 : JMPF      107
  8 : SADI        2        23
 12 : LADI        2        23
 16 : LADR        2         0
 20 : PLUS
 21 : LAD
 22 : MULTC       4
 24 : LN         63
 28 : PLUS
 29 : JMP
 30 : JMPU       74
 34 : LN         45
 36 : OUTCH
 37 : LN         18
 39 : OUTCH
 40 : LN          5
 42 : OUTCH
 43 : LN          4
 45 : OUTCH
 46 : JMPU       58
 50 : LN         45
 52 : OUTCH
 53 : LN         23
 55 : OUTCH
 56 : LN          8
 58 : OUTCH
 59 : LN          9
 61 : OUTCH
 62 : LN         20
 64 : OUTCH
 65 : LN          5
 67 : OUTCH
 68 : JMPU       36
 72 : LN         45
 74 : OUTCH
 75 : LN          2
 77 : OUTCH
 78 : LN         12
 80 : OUTCH
 81 : LN         21
 83 : OUTCH
 84 : LN          5
 86 : OUTCH
 87 : NOOP
 88 : JMPU       16
 92 : JMPU      -58
 96 : JMPU      -46
100 : JMPU      -28
104 : LADI        2        23
108 : PLUSC       1
110 : JMPU     -108
112 : DELETE
113 : LN          0
115 : OUTCH
116 : EXIT
```

```
BEGIN ↱ MAIN PROGRAM ↓
   I:=0;
   WHILE I < N DO
   BEGIN READ(CH); I:=I+1;
      IF CH = ΞRΞ THEN A[I]:=RED

      ELSE
       IF CH = ΞWΞ THEN A[I]:=WHITE
       ELSE
        IF CH = ΞBΞ THEN A[I]:=BLUE
        ELSE I:=I-1;
   END;

   R:=1; W:=N; B:=N;
   WHILE W≥R DO
   BEGIN
      V:=COL(W);
      IF V=RED THEN
       BEGIN
          SWAP(R,W); R:=R+1;
       END
      ELSE
       BEGIN
          IF V=BLUE THEN
           BEGIN
              SWAP(W,B); B:=B-1;
           END;
          W:=W-1;
       END;
   END;

   RESULT;

END.
```

```
  0 : LN        0
  2 : SADI      2      23
  6 : LADI      2      23
 10 : LTC      20
 12 : JMPF    110
 16 : INCH
 17 : NOOP
 18 : SADI      2      21
 22 : LADI      2      23
 26 : PLUSC     1
 28 : SADI      2      23
 32 : LADI      2      21
 36 : EQC      18
 38 : JMPF     20
 42 : LADI      2      23
 46 : LADR      2       0
 50 : PLUS
 51 : LN        0
 53 : SAD
 54 : JMPU     66
 58 : LADI      2      21
 62 : EQC      23
 64 : JMPF     20
 68 : LADI      2      23
 72 : LADR      2       0
 76 : PLUS
 77 : LN        1
 79 : SAD
 80 : JMPU     40
 84 : LADI      2      21
 88 : EQC       2
 90 : JMPF     20
 94 : LADI      2      23
 98 : LADR      2       0
102 : PLUS
103 : LN        2
105 : SAD
106 : JMPU     14
110 : LADI      2      23
114 : PLUSC    -1
116 : SADI      2      23
120 : JMPU   -114
122 : LN        1
124 : SADI      2      24
128 : LN       20
130 : SADI      2      25
134 : LN       20
136 : SADI      2      26
140 : LADI      2      25
144 : LADI      2      24
148 : GE
149 : JMPF    107
152 : MARK
153 : LN        0
155 : MVADRU
156 : LADI      2      25
160 : MVADRU
161 : ENTER     9
163 : NOOP
164 : SADI      2      22
168 : LADI      2      22
172 : EQC       0
174 : JMPF     34
178 : MARK
179 : NOOP
```

```
180 : LADI        2     24
184 : MVADRU
185 : NOOP
186 : LADI        2     25
190 : MVADRU
191 : ENTER       5
193 : NOOP
194 : LADI        2     24
198 : PLUSC       1
200 : SADI        2     24
204 : JMPU       50
208 : LADI        2     22
212 : EQC         2
214 : JMPF       30
218 : MARK
219 : NOOP
220 : LADI        2     25
224 : MVADRU
225 : NOOP
226 : LADI        2     26
230 : MVADRU
231 : ENTER       5
233 : NOOP
234 : LADI        2     26
238 : PLUSC      -1
240 : SADI        2     26
244 : LADI        2     25
248 : PLUSC      -1
250 : SADI        2     25
254 : JMPU     -114
256 : MARK
257 : ENTER      13
259 : STOP
```

```
***** BOBS P-CODE SIMULATION *****

RED RED RED RED RED WHITE WHITE WHITE WHITE WHITE WHITE WHITE WHITE WHITE


BLUE BLUE BLUE BLUE BLUE BLUE
```

REFERENCES

[1] "A Description of the MATHILDA System", Bruce Shriver, DAIMI PB-13, April 1973.

[2] "RIKKE-1 reference manual", Jørgen Staunstrup, DAIMI MD (to appear).

[3] "A proposal for a unified number representation giving normalized, unnormalized and interval arithmetic"

"A unified numeric data type in PASCAL" , Peter Kornerup, DAIMI (unpublished papers).

[4] "A small group of research projects in machine design for scientific computation", Bruce Shriver, DAIMI PB-14, June 1973.

[5] "The programming language PASCAL", N. Wirth, ACTA INFORMATICA 1, 35-63 (1973).

[6] "The design of a PASCAL compiler", N. Wirth, SOFTWARE-PRACTICE and EXPERIENCE, vol. 1, 309-333 (1971).

[7] "Planned changes to the programming language PASCAL", N. Wirth, June 1972.

[8] "The portability of the BCPL compiler", M. Richards, SOFTWARE - PRACTICE and EXPERIENCE, vol. 1, 135-146 (1971).

[9] "Compiler construction for digital computers", David Gries, John Wiley & Sons, Inc., 1971.

[10] "A compiler generator", W.M. McKeeman, J.J. Horning, D.B. Wortman, Prentice Hall, Englewood Cliffs, 1970.

[11] "A short description of a translator writing system (BOBS-system)", Bent Bruun Kristensen, Ole Lehrmann Madsen, Bent Bæk Jensen, Søren Henrik Eriksen, DAIMI PB-11, February 1973.