© EUROMICRO
EUROMICRO Journal 5 (1979) 363-369

Implementation of Pascal on an 8080 Microcomputer

W. J. A. Pasman
Technische Hogeschool Twente, Dept. of Digital Technics, Postbus 217, AE 7500 Enschede, the Netherlands

With a portable Pascal compiler as a start, the possibility was created to develop programs in Pascal for an 8080 microcomputer. The compiler was installed on a DEC 10 and generated an intermediate code which was interpreted on the microcomputer. The possibility was created to give the programmer access to the file oriented monitor system and predefined 8080-machine-code routines. The interpreter and the necessary monitor routines needed about 7k of memory bits. The work was done at the Technische Hogeschool Twente within the Department of Digital Technics from October 1976 until March 1977. The next pages give a brief review of the most important decisions taken during the implementation.

1. INTRODUCTION

In August 1976 the Technische Hogeschool Twente (THT) decided to use Pascal as a high level language to run on 8080 microcomputers. The main reasons for this decision were:

1. A high level language was needed to decrease the programming effort in an application.
2. Pascal is well structured, well defined and self-documenting.
3. Pascal was already used at the THT on the DEC 10 and PDP 11 computers.

The intended applications were mainly text-formatting, documentation, etc., in which execution speed is not a critical factor. For microcomputers, however, the programs are quite large, so the basic goal of the implementer was to take care of an efficient memory usage.

At the ETH Zurich a portable compiler for Pascal was developed, which made it simple to implement Pascal on machines for which a compiler was not available. This compiler, called the P-compiler generated assembler statements for a hypothetica computer. These statements, called P-symbolic code, had to be converted to machine code by an assembler program. The influence of the architecture of the target computer was concentrated as much as possible in the assembler. The assembling could be seen as the last pass of the compilation process.

At the THT the P-compiler and assembler were available in a Pascal version and in a P-symbolic version. For microcomputers they were large. In order to keep compiling fast and with the limited amount of memory in mind, we installed the compiler and assembler on the DEC 10 system of the THT. See Fig. 1 and Fig. 2 for T-diagrams of the installation. An advantage of this strategy was also that editing and simulation of the programs could be done at the DEC 10. A disadvantage was that the programs for the microcomputer had to be transferred from the DEC 10 to the 8080 by papertape.

The names in the figures are explained as follows:

PASCAL: Program in a Pascal version;
PSYMB: Program in a P-symbolic version;
PMACH: Program in a P-machine version;
D10: Program in a DEC 10 language version.

The symbols in the figures are explained in Fig. 3.

For the code generation we had several possibilities:

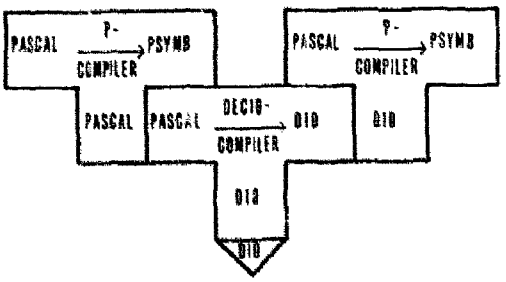


Fig. 1. The installation of the P-compiler.

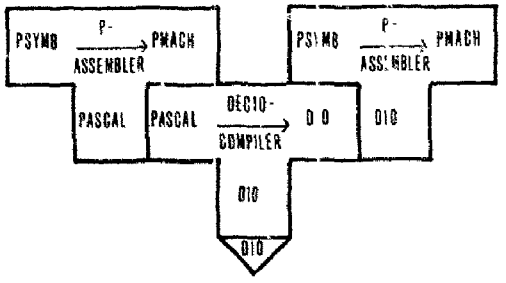
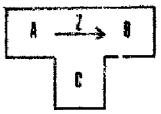


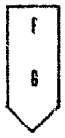
Fig.2. The installation of the P-assembler.



Symbol for a program Z which translates programs available in language B. The program Z is available in language C.

Symbol for a program D available in language D.

Symbol for a program D available in language D.



Symbol for a run on a machine with machine language G, which happens interpretive. The programs are written in a language F.

Fig. 3. Explanation of the symbols in Figs. 1 and 2.

1. to generate code directly for the 8080; every P-instruction would be substituted by 8080-instructions (or macros for the 8080-assembler)
2. to generate code directly for the 8080; every P-instruction would be substituted by a subroutine call. The 8080-code for each subroutine used, would be added to the program.
3. to generate code for a P-processor and interpret it on the 8080.

Possibility 1 would result in 10 times as much code compared to possibility 3. Possibility 2 would result in 2 times as much code compared to possibility 3. This is due to the fact that every P-operation would use 3 bytes for the call. In case of using an interpreter the opcode would use 1 byte (see also subsection 2.2.).

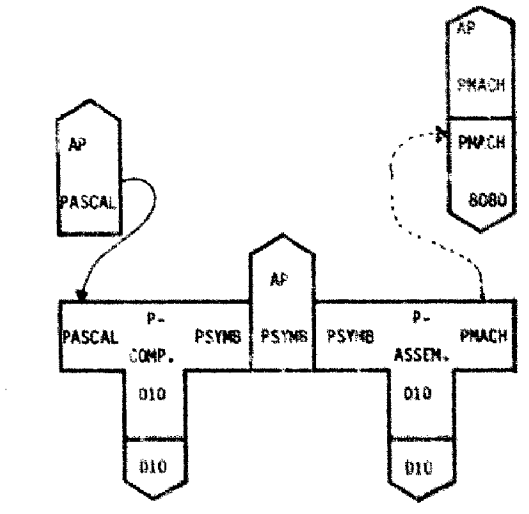


Fig. 4. The preparation of a program.

With the limited amount of memory in mind we decided to write an interpreter for P-machine code. See Fig. 4 for a survey of the actions necessary for the preparation of an application program written in Pascal.

2. IMPLEMENTATION

2.1. The Compiler

P-Pascal was defined as a subset of Standard Pascal (see for a definition of Standard Pascal ref. [1]). The features not implemented in P-Pascal were:

1. Formal procedures and functions.
2. "goto" labels outside a procedure or a function body.
3. Packing features.
4. The freedom to define files (P-Pascal knows only 2 predefined files for input and for output of type: "text").
5. The standard procedure DISPOSE (it was replaced by a combination of two new standard procedures MARK and RELEASE).
6. Variable string length (the compiler made all strings of a fixed length).

For details of these differences, see ref. [2].

Between Pascal on the DEC 10 and Standard Pascal there were also minor differences in the used character set, the indication of comment and in the symbols for set union. The P-compiler was adapted so that P-Pascal was a subset of DEC 10-Pascal.

To create the possibility of calling 8080-machine code routines in a Pascal program, P-Pascal was extended by a new standard procedure MONITOR. This procedure has 1 parameter: an integer indicating which routine should be called. 8080-machine code routines are available for:

1. Reset of the input-file pointers
2. IO from and to the system console
3. Linkage of the predefined Pascal-files with any other file in the 8080-system
4. Getting a value indicating the free space on the P-stack

In most cases the routines needed or produced information; e.g., file names, characters, status information. For passing these parameters the following construction is used:

DECLARATION:

```
PROCEDURE <ident.>(<formal par.section>);
BEGIN
   MONITOR(<unsigned integer>)
END;
```

CALL:

```
<procedure identifier>(<act.par.section>);
```

As an example, the declaration and call of the routine for input from the system console are given below:

DECLARATION:

```
PROCEDURE TTYIN(VAR CH:CHAR);
BEGIN
```

```
    MONITOR(1)
  END;
CALL:
  TTYIN(CHARIN);
```

This construction has the advantage that only a few changes were needed in the compiler and the introduction of only 1 new P-instruction was necessary (called MON). When the procedure:<identifier> is called, the actual parameters are pushed on the 8080-stack. The 8080-machine code routine can calculate with the aid of the P-stack pointer the addresses of the actual parameters.

2.2. The Assembler

In Fig. 4 this program is called P-ASSEM. In the assembler the P-symbolic code is converted to P-machine code. See ref. [2] for a definition of the P-symbolic code. The compiler does not generate absolute addressesin jump and call instructions, but generates labels and refers to them in the program code. A task of the assembler is to calculate absolute addresses for the labels. The P-assembler also generates tables for strings and for values of boundary checks. Boundary checks are executed during run time in order to check whether the values of variables stay within the boundaries. The boundaries were implicit or explicit defined at the declaration of the variables in the Pascal program.

The assembler has to know the memory usage of the interpreter, e.g., the start address for the program code and the tables in the 8080-memory. In the assembler this kind of information is defined by constants. An option is built in to omit the instructions and tables generated for the boundary checks, for these had a very bad influence on the run time of a program.

2.3. The Interpreter

2.3.1. The Architecture of the P-Machine. The P-machine knows 2 types of memory and 6 registers. The 2 types of memory are called:

1. the program store
2. the data store

The program store contains the P-instructions. The processor has a register which contains a pointer to the next instruction to be executed. This register is called the program counter. Another register called the instruction register contains the instruction which is executed.

The data store contains the constants, addresses and variables. For the explanation of the use of the registers and the data store, it is necessary to explain something about the scope and lifetime of variables in Pascal. The scope of a variable is the part of the program where it can be used. In Pascal the scope is the procedure block to which the variable belongs. The lifetime of a variable is the time the procedure defining the variable is activated to the time a return is activated by the procedure. In Pascal procedure activation is a first in last out process, the use of a stack is an appropriate strategy. When a procedure is activated storage for its local variables is allocated on top of the stack and is disallocated at return from the procedure. When a procedure is activated we say that a new segment is created on top of the stack. The stack is also used for evaluation of expressions and for the calculation of the addresses of elements in arrays and records. The stack is called the P-stack and the register containing the pointer to the top of the stack is called the stack pointer.

The address of the first element of the last created segment is kept in a register of the P-machine, called the base register. Two addresses are needed for proper resumption of program execution after termination of the procedure. This is a return-address and the base-address of the segment which is after return on top of the stack. These addresses are stored in the first elements of a segment. The base addresses stored in the stack form a chain (called the dynamic chain), which reflects the history of procedure activation.

At compile time it is not known which segments will be on the stack, so the addresses of the variables must be calculated at run time. The compiler generates for every address calculation a displacement from the base address of the segment in which a variable is allocated. This displacement is not sufficient because the proper base address is also needed. The compiler does not know the history of procedure activation and so no use of the dynamic chain can be made. It knows only the statical hierarchical structure of the procedures. So another chain (called the static chain) is implemented; every segment has a pointer which refers to the segment of the block in which the procedure was declared. For making it possible to find the proper base address, the compiler generates a number which indicates the difference in static levels of the segments in which a variable is located and used. This number is called the static level difference.

An example of the use of the static and dynamic chain follows:

```
Example: B declared in A
         C declared in B
         A, B, C are procedures
         A calls B, B calls C,
         C calls B recursively
```

The segments will be linked, when C calls B for the first time, as depicted in Fig. 5.

In the data store another stack is used for allocating records, created by the Pascal standard procedure NEW. The records are pushed on this stack (called the heap) when NEW is called. A register in the P-machine contains a pointer to the top of the heap and is called the heap pointer (see Fig. 6).

The P-stack starts at one end of the data store and the heap at the other. To make it possible to test if both stacks do not overwrite each other, a register in the P-machine is reserved for keeping the value for the maximum growth of a segment, while this segment is on top of the stack. This register is called the extreme stack

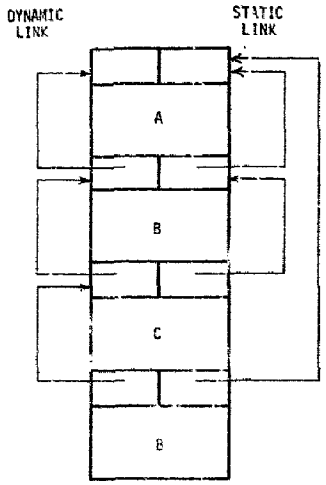


Fig. 5. The dynamic and static chain.

pointer. The maximum growth is calculated by the compiler and at activation of a procedure the value is added to the value of the new base address and stored in the extreme stack pointer. Then the value is compared with the heap pointer and if not enough space is left for the new segment the execution is stopped. When the heap grows the value of the heap pointer is compared with the extreme stack pointer. Execution is stopped if not enough space for the new record is left on the stack. At procedure activation the old value of the extreme stack pointer is saved in the new segment and restored at return from the procedure.

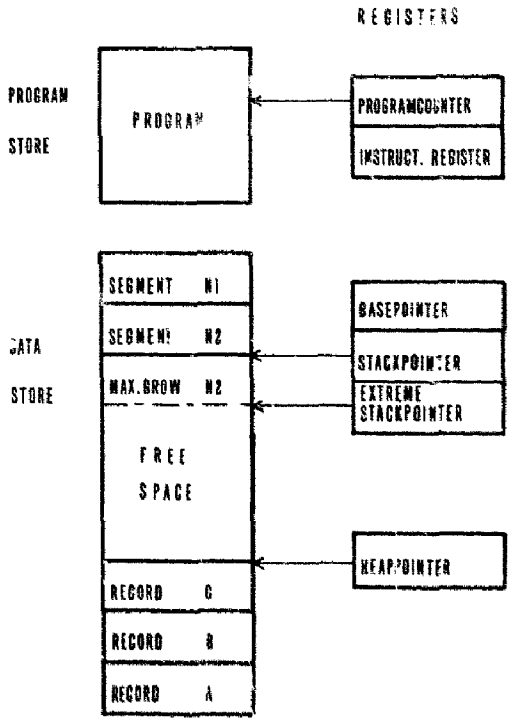


Fig. 6. Use of the store and the registers.

The IO of the P-machine is directly derived from the Pascal IO. It knows only files of type: TEXT and has the standard routines like GET, PUT, READ, WRITE, READLN and WRITELN. For indication of en-of-file conditions it has for every input file 2 flags.

2.3.2. The Data Types. The P-machine distinguishes the following data types: reals, integers, addresses, booleans, characters and sets. The decisions taken on the formats for each of the data types are specified below.

INTEGER: In most cases the integers are used for counting and indexing. Therefore, a large range is not necessary and 16 bits to represent an integer is acceptable. They are represented in two-complement notation; see Fig. 7.

REAL: For the applications we had in mind, we did not need reals, so we did not implement them.

BOOLEAN: In the 8080 no instructions for bit processing are present, so we represent every boolean in 1 byte; see Fig. 8.

CHARACTER: They are represented in one byte by the value in the ASCII table, with the high order bit of the byte zero; see Fig. 9.

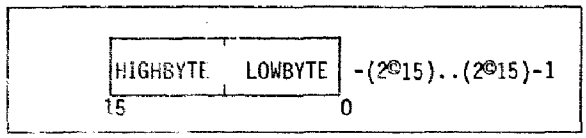


Fig. 7. INTEGER Data Type.

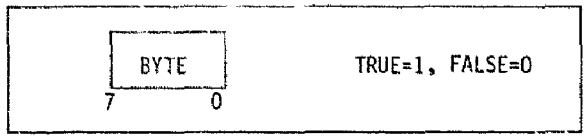


Fig. 8. BOOLEAN Data Type.

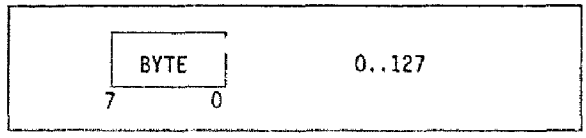


Fig. 9. CHARACTER Data Type.

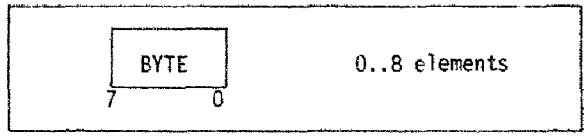


Fig. 10. SET Data Type.

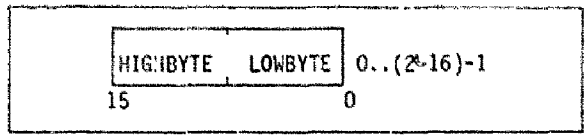


Fig. 11. ADDRESS Data Type.

SET: A set is represented by its characteristic function. This is a bit string in which the value of every bit specifies the presence or absence of an element. Set operations are only useful if they are fast; in most cases they correspond with logical operations. In the 8080 logical operations are performed on 1-byte operands, so we decide to use 1 byte. Applications did not show the need for another choice; see Fig. 10.

ADDRESS: Because we used the memory of the 8080 to represent the program- and data-store of the P-machine, we took for the P-address range the same range as for an address in the 8080; see Fig. 11.

2.3.3. The Instruction Format. Every instruction consists of an operation code and 0, 1 or 2 operands. There wer less than 256 different P-instructions, so we chose to represent the operation code in 1 byte. The operands could be of type: address, integer, level difference, boolean, set or character. For the use of the level difference, see subsection 2.3.1. For the level difference we took 1 byte, this had as result that nesting of procedures could not exceed 256 levels. For further information about the representation of the operands in the program code, see section 3.

2.3.4. The Register Allocation. The P-machine knows 6 registers:

PC: the program counter points to the next instruction to be executed.
IR: the instruction register, contains the instruction to be executed.
SP: the stack pointer, points to the top of the segment stack.
NP: the heap pointer, points to the top of the heap.
MP: the base register, points to the base of the last activated segment on the stack.

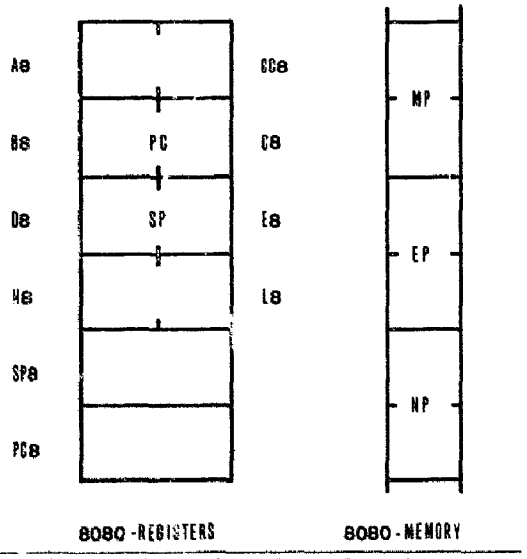


Fig. 12. The allocation of the P-registers.

EP: the extreme stack pointer, points to the location which will be used when the stack grows to its maximum size.

The instruction register is not explicitly allocated in the 8080, because the instruction operands are immediately used after they are fetched.

The program counter and the stack pointer are referred to most frequently and are located in 2 general purpose register pairs of the 8080. The program counter is located in the BC-register pair and the stack pointer is located in the DE-register pair. The heap pointer, the base register and the extreme stack pointer are located at fixed places in the memory of the 8080. The HL-register pair is used in the interpreter as temporary storage to prevent too many memory references and because of its special function in the 8080. See Fig. 12.

Note: The register names with an "8" at the end are 8080-registers.

We did not use the 8080 stack pointer as location for the P-stack pointer for the following reasons:

1. The 8080 stack pointer was used by the interpreter to call subroutines and to save data. So each time the interpreter uses the stack the stack pointer would have to be restored.
2. The 8080 push- and pop-instructions operate on 2-byte operands, each time a 1-byte operand is used the stack pointer should have to be up-dated.
3. Moving the 8080 stack pointer to a register pair cannot be achieved by 1 instruction.

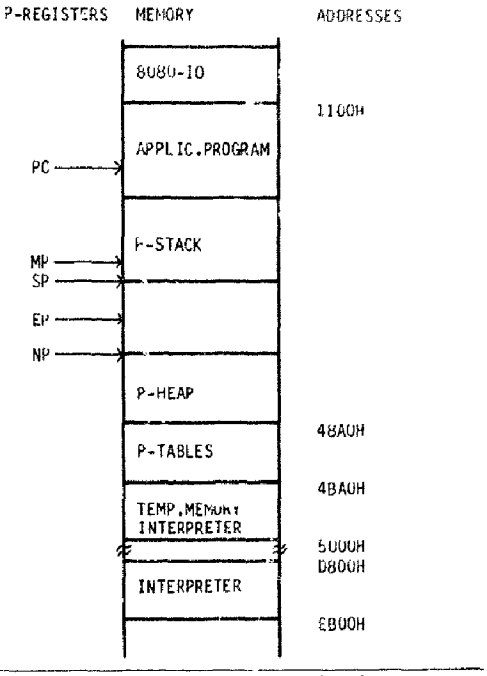


Fig. 13. The use of the memory by the interpreter.

2.3.5. The Lay-Out of the Memory. Note 1: See for more information about the space requirements section 4. Note 2: The P-stack starts always after the application program. See Fig. 13.

3. OPTIMALIZATION OF THE P-MACHINE CODE

In the first version of the interpreter, the representation of the program code was directly derived from the P-symbolic code. Every integer and address used 2 bytes, every other constant 1 byte.

We did some measurements to the frequency of the instructions and the values of the operands in the program code after completion of the first version. They are based on a program for text-formatting and a program for documentation of hardware. The measurements were "static" because the main objective of the optimalization was to reduce the space for the program code.

Most frequently were present:

1. Constant operands with the value of 0 or 1.
2. Level differences with the value of 0 or 1.
3. Constant integers with a value smaller than 256.

In the first version we use 80 different opcodes. There was also space left for 176 new opcodes. We decided to introduce new opcodes for instructions with the following operands:

1. Integers with the value 0.
2. Integers with the value 1.
3. 1-byte constants with the value 0.
4. 1-byte constants with the value 1.
5. Integers with a value smaller than 256.
6. Level differences with the value 0.
7. Level differences with the value 1.

In case 1, 2, 3, 4, 6 and 7 the operand field could be omitted. In case 5 the space for the representation of the integer was reduced by 1 byte. The encoding of the instructions was done in the P-assemble program. There were 59 new opcodes introduced. This way the representation of the P-instructions was optimized for the application.

4. SPACE REQUIREMENTS

Table 1 lists the space in bytes needed by the interpreter program (version 2). The values in brackets are for a minimum system. Such a system has very simple IO and no messages to the console.

Note 1. These are routines to adapt the Pascal IO to the 8080 system IO.

Note 2: The chosen string length was 16. 32 strings could be loaded in the table. In the boundary table 64 bourdary pairs could be loaded.

Table 2 shows the values for the space requirements of some application programs written in Pascal.

The average instruction length was reduced by the code optimalization from 2.8 to 1.8 bytes per P-instruction.

Table 1
Space for the Interpreter

Function	Space		Kind of memory
Loading P-program and error handling	619	(100)	ROM
IO-routines (note 1)	510	(30)	ROM
Character strings for messages	316	(0)	ROM
Fetch and decode of P-opcode	560	(560)	ROM
Execution P-instruct.	2405	(2405)	ROM
IO-bufters	1024	(4)	RAM
Tables for strings and bound.pairs (note 2)	768	(768)	RAM
Temporary store (incl. 8080 stack)	96	(96)	RAM
Total	4410	(3095)	ROM
	1888	(868)	RAM

Note 1: Not counted were the lines which existed only of comment or declarations.

Note 2: Version 1 of the interpreter was before the code optimalization, version 2 afterwards.

5. CONCLUSION

Other strategies have been used to make it possible to develop Pascal programs for microcomputers (see refs. [5,6]). Most of them lead to implementation of the compiler on the microcomputer. These strategies have two disadvantages:

1. Compilation takes a long time compared to the time needed for compilation on e.g. a DEC 10.
2. A lot of memory is needed for the compilation.

The strategy we followed resulted in a short compilation time and a small amount of memory, but had also disadvantages. We had to transfer the P-machine code from the DEC 10 to the micro by papertape. Sometimes this took a few hours and nullified the advantage for the short compilation time. If no satisfactory solution will be found for this problem, implementation of the compiler on the micro will be an alternative for us.

Table 2
Space requirements of 2 application programs

PROGRAM FUNCTION	PASCALCODE [LINES] (NOTE1)	P-SYMB.CODE [INSTR.]	P-MACH.CODE [BYTES]	
			VERS.1 (NOTE2)	VERS.2
TEXTFORMATTING	750	3300	9200	6000
DOCUMENTATION OF HARDWARE	1145	5300	15100	9500

Another problem we met, made implementation of the compiler on the micro more attractive. When we installed the P-compiler on the DEC 10, the DEC 10 compiler did not accept P-Pascal and we had to adapt the P-compiler. Afterwards a new compiler on the DEC 10 was installed and we had to adapt the P-compiler again. Problems arose from the fact that the results from the standard procedures DISPOSE, MARK and RELEASE was defined the same in both compilers.

Some decisions would have been different if we had more experience in writing application programs in Pascal. The allocation of the strings in the string table caused an inefficient use of memory space. A better solution would be storing the strings in the program code after the load-instruction (strings are always referred to by load-address-of-string-instructions).

Also the IO via predefined files could be made more flexible by extending the number of files and reserving space for IO-buffers depending on the number of files used. Now we reserve space for always 4 files.

When writing the interpreter it appeared that the architecture of the 8080 was not very well suited to represent the P-machine. Disadvantages were the absence in the 8080 of:

1. 2 more register pairs for the base register and the heap pointer.
2. A base register for jumps via tables.
3. An auto-increment and auto-decrement mode for addressing via register pairs.
4. Instructions for moving words from memory to the BC- and DE-registers and vice versa.
5. Instructions for subtracting words.

More suited would be an 8-bit machine like the Z80 or 16-bit machines like the TMS 9900, Intel 8086 or the Z8000.

Most of our application programs were for interactive use, the speed of the programs appeared to be sufficient.

REFERENCES

[1] Kathleen Jensen, Niklaus Wirth, Pascal Users Manual and Report (Springer Verlag, Berlin, 1974).
[2] Niklaus Wirth, Algorithms + Data Structures = Programs (Prentice Hall, Englewood Cliffs, 1976).
[3] Nori, Amann, Jensen, Naegeli, Jacobi, The Pascal <P> Compiler: Implementation Notes, ETH Zurich (1976).
[4] W.J.A. Pasman, The P4-Interpreter, M.Sc. Report Nr. 1201.2056, Technische Hogeschool Twente (1977).
[5] Kin-Man Chung, Herbert Suen, A "Tiny" Pascal Compiler, Byte (September, October, November 1978).
[6] Kenneth Bowles, Microcomputer Problem Solving using Pascal (Springer Verlag, Berlin, 1977).