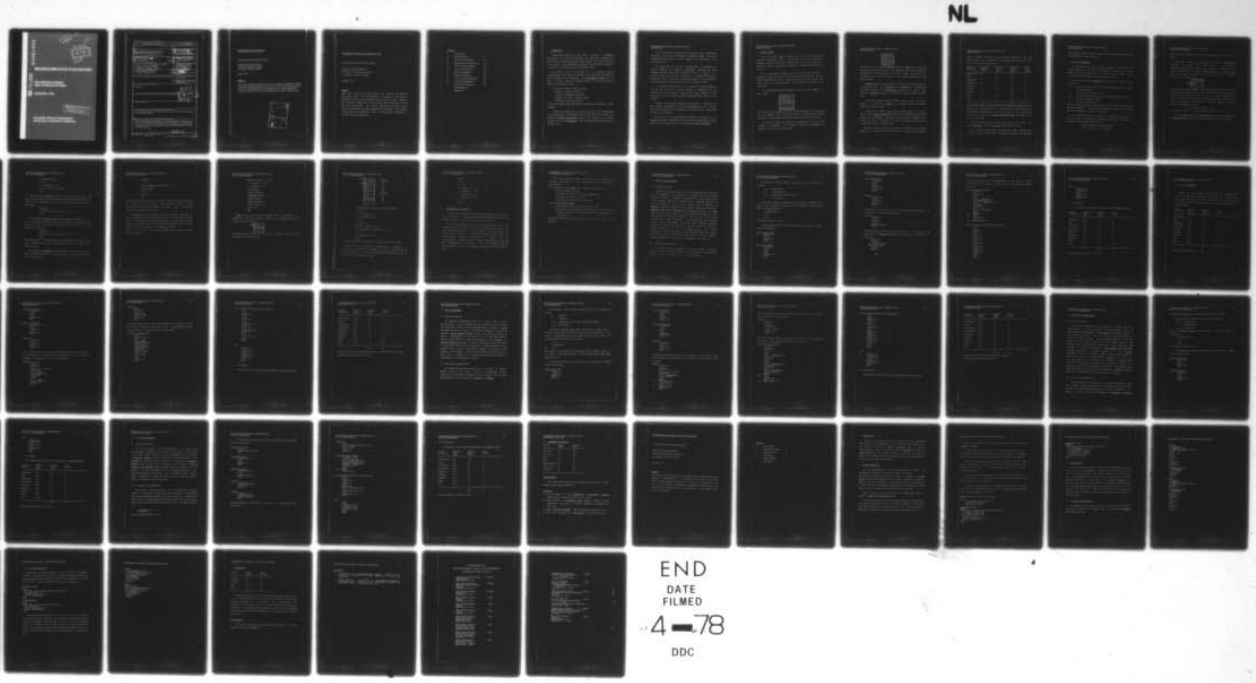
AD-A051 934 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT OF--ETC F/G 9/2
MICROCOMPUTER EVALUATION.(U)
JAN 78 P B HANSEN, C HAYDEN

UNCLASSIFIED N00014-77-C-0714
NL

| OF |
AD
A051 934



END
DATE
FILMED
4-78
DDC

MICROCOMPUTER EVALUATION

PER BRINCH HANSEN
AND CHARLES HAYDEN

JANUARY 1978

Computer Science Department
University of Southern California

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MICROCOMPUTER EVALUATION.	5. TYPE OF REPORT & PERIOD COVERED Technical Report.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Per Brinch Hansen [redacted] Charles Hayden	8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0714 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS New Computer Science Department University of Southern California Los Angeles, California 90007	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 049-415
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Arlington, Virginia 22217	12. REPORT DATE Jan [redacted] 78 / 13. NUMBER OF PAGES 43 + 10
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified / 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE Not applicable

16. DISTRIBUTION STATEMENT (of this Report)

Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

microcomputer evaluation, instruction sets, Concurrent Pascal interpreter, process switching time

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report evaluates the ability of 16-bit microcomputers to implement a subset of the programming language Concurrent Pascal. The first part of the report describes the most frequent virtual instructions and their implementation; the second part evaluates the process switching time of various microprocessors.

410617

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MICROCOMPUTER EVALUATION

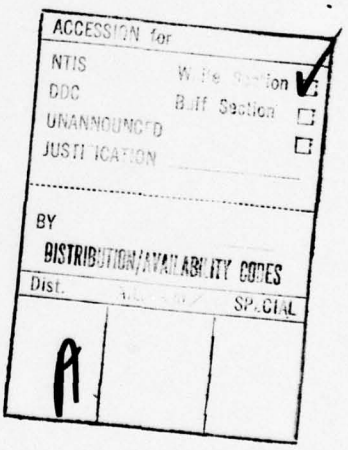
Per Brinch Hansen and Charles Hayden

Computer Science Department
University of Southern California
Los Angeles, California 90007

January 1978

Summary

This report evaluates the ability of 16-bit microcomputers to implement a subset of the programming language Concurrent Pascal. The first part of the report describes the most frequent virtual instructions and their implementation; the second part evaluates the process switching time of various microprocessors.



MICROCOMPUTER EVALUATION: INSTRUCTION SETS

Per Brinch Hansen and Charles Hayden

Computer Science Department
University of Southern California
Los Angeles, California 90007

December 1977

Summary

This report defines a precise method for comparing microcomputer instruction sets. Each microcomputer is judged by its ability to interpret the virtual code generated by the Concurrent Pascal compiler. The standard of comparison is the Concurrent Pascal interpreter for the PDP 11/45 minicomputer. This report identifies the most frequent virtual instructions and their implementation on the PDP 11/45 computer. It then evaluates the speed and size of interpreters implemented on various 16-bit microcomputers.

Contents

1. INTRODUCTION

A new research project at USC seeks to develop a programming methodology and machine architecture for concurrent programming on microprocessor networks with distributed storage. The starting point of this project is the programming language Concurrent Pascal implemented on the PDP 11/45 minicomputer [Brinch Hansen, 1977a].

The initial aim of the project is to evaluate existing microcomputers and select one of them for a uniprocessor system with a display terminal and a floppy disk. This microcomputer system will be able to execute concurrent programs written in a subset of Concurrent Pascal [Brinch Hansen, 1977b].

The Concurrent Pascal subset includes

processes, monitors, classes
routines, statements, expressions
enumerations, arrays, records, queues
terminal and disk input/output

It does not include reals and sets and cannot execute sequential Pascal programs.

A concurrent program will be compiled on a PDP 11/45 minicomputer and transferred to a microcomputer via a floppy disk. The compiled code consists of virtual instructions that will be executed by a machine program called the interpreter. Such an interpreter already exists in the PDP 11/45.

INTRODUCTION

Our initial goal is to compare the instruction sets, input/output facilities, and prices of existing microcomputer systems. This report concentrates on the first of these aspects: the efficiency of the instruction sets.

Our primary goal is to use a microprocessor to implement an abstract language for concurrent programming. Although the final language chosen may differ from Concurrent Pascal in details it is likely to have many things in common with it (among them processes, procedures, arrays, records, and enumeration types). The code generated should therefore be quite similar to that of Concurrent Pascal.

It is therefore reasonable to use the Concurrent Pascal interpreter for the PDP 11/45 as a precise standard of comparison for the instruction sets. This report identifies the most frequent virtual instructions and the machine code that interprets them on the PDP 11/45 computer.

The report also defines the machine code needed to interpret the same virtual instructions on various microcomputers. This is used to estimate the absolute size and speeds of Concurrent Pascal interpreters for these microcomputers.

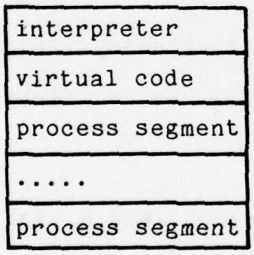
Since our aim is to develop <u>new</u> microcomputer technology that will <u>simplify</u> language implementation for non-trivial applications we will ignore 8-bit microprocessors and concentrate on <u>16-bit processors</u>.

2. VIRTUAL MACHINE

Since the virtual machine spends most of its time executing sequential statements within a process we will only describe the sequential aspects of the virtual machine here. The virtual machine for Concurrent Pascal is described in more detail elsewhere [Brinch Hansen, 1977a].

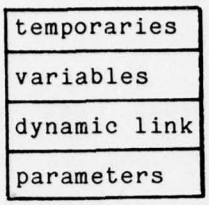
In a microcomputer network each processor should perform a reasonably simple task that does not require too much storage. We will therefore assume that a virtual instruction potentially can address the entire store of a single processor. Each processor can access its own store only.

In a uniprocessor system the store is divided into segments of fixed length



The lengths of the virtual code and the process segments are determined by the compiler. This is possible because a Concurrent Pascal program consists of a fixed number of processes without recursive procedures.

Each process segment is organized as a stack that grows from high towards low addresses. The storage of procedure parameters and variables in the stack is shown below.



The store consists of 16-bit words divided into 2 bytes each. Words are addressed by even byte indices. An enumeration type (or an address) is represented by a single word in the stack. Arrays and records are represented by one or more words. A text string consists of an even number of bytes.

A dynamic link of 5 words defines the state of the process prior to a procedure call. The parameters and variables are addressed by positive and negative even displacements relative to the dynamic link. Temporary addresses computed in the stack are absolute within the whole store.

A concurrent program can only access variables that are either local to a procedure or global to a class, monitor, or process. Procedures cannot be nested.

A concurrent program is compiled into a single segment of virtual code. Each virtual instruction consists of an operation code followed by zero or more arguments. The operation codes and their arguments each occupy one word. The virtual code refers to program labels and stack variables by relative addresses only.

The Solo operating system for the PDP 11/45 computer is a typical example of a non-trivial Concurrent Pascal program [Brinch Hansen,

1977a]. A dynamic analysis of the virtual code executed by the Solo system shows that 9 kinds of virtual instructions account for 71 per cent of all instructions executed.

virtual instruction	dynamic frequency (%)	execution time (usec)	weighted time (usec)	code piece (words)
pushvariable	26	9	2.3	4
index	7	19	1.3	12
pushaddress	13	8	1.0	3
pushconstant	9	6	0.5	2
equalword	5	10	0.5	6
copyword	5	7	0.4	2
copybyte	4	7	0.3	2
enter	1	22	0.2	10
exit	1	15	0.2	8
	71		6.7	49

The execution time of a particular virtual instruction weighted by its frequency of execution defines the contribution of that instruction to the average instruction time. Since 71 per cent of all instructions contribute 6.7 usec the average instruction time on the PDP 11/45 is 6.7/0.71 = 9.5 usec.

The total length of the selected code pieces is 49 words. The whole interpreter is about 1000 words long on the PDP 11/45.

We will choose these virtual instructions as being representative of the 50 virtual instructions or so needed to implement the Concurrent

Pascal subset. This will be our basis for evaluating the speed and size of interpreters on other machines.

3. PDP 11/45 INTERPRETER

In implementing the Concurrent Pascal subset, we will ignore the address mapping of the PDP 11/45 computer and view it as a 16-bit machine that can address a store of 64 K bytes (or 32 K words). Words are addressed by even (byte) addresses [Digital, 1975].

The machine has 8 word registers that are used as follows by the interpreter

w, x, y	scratch registers used during the interpretation of a single virtual instruction.
s	the address of the stack top.
b, g	the base address of the current local and global variables.
q	the virtual program counter.
p	the real program counter.

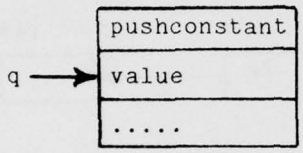
The virtual program counter q points to the next virtual instruction (or one of its arguments). The real program counter p points to the machine code that interprets the virtual instruction.

The interpreter code will be defined in Pascal-like notation so that each line corresponds to one machine instruction. The store and the registers are declared as follows:

```
var byte: array [integer] of integer;
w, x, y, b, g, q, s, p: integer;
```

Most of the time, however, we will refer to the store as an array of words.

Among other things the store contains the virtual instructions. Consider, for example, the virtual instruction that pushes an enumeration constant on the stack. This instruction consists of two words representing the operation pushconstant and a constant value. When this instruction is executed the virtual instruction counter q points at the constant value:

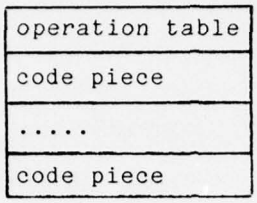


Since the stack grows towards low addresses the interpreter first decrements the stack top s by one word (2 bytes) and copies the constant value from the virtual code to the stack. It then increments the virtual program counter q by one word (2 bytes) to make it point at the next operation code. All this is done by a single machine instruction which can be defined as follows

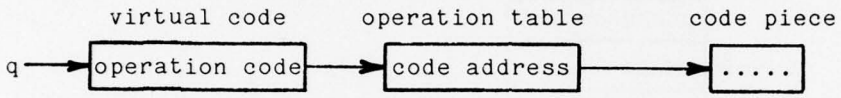
```
s :- 2; word(s) := word(q); q :+ 2;
```

(An assignment such as s :- 2 is a short-hand for s := s - 2)

The interpreter contains a code piece for each virtual instruction. An operation table defines the starting addresses of the code pieces.



The operation codes are used as indices in the operation table to locate the corresponding code pieces. When one code piece has been executed the next piece is located as follows:



The virtual program counter q points to the next operation code which in turn is the absolute address of an entry in the operation table. The selected entry in the operation table contains the address of the corresponding code piece that will perform the operation.

So the next code piece is located by a doubly-indirect jump instruction

```
p := word(word(q)); q :+ 2;
```

Each code piece ends with such a jump to the next code piece. This single instruction is the only overhead of code interpretation on the PDP 11/45 computer. This efficient form of code interpretation is called threaded code [Bell, 1973].

The purpose of the operation table is to enable the compiler to generate fixed operation codes that are independent of any modifications to the code pieces. The only requirement is that the table must start

at a fixed absolute address.

The general notation used to define a code piece is illustrated below:

```
pushconstant(value):
s :- 2; word(s) := word(q); q :+ 2;
next;
```

It defines the name and arguments of the virtual instruction and the code that interprets it. Each line corresponds to one machine instruction. The instruction next is the indirect jump defined earlier.

The rest of the selected code pieces are defined below.

The virtual instruction pushaddress adds the displacement of a variable to the local base address b and pushes the resulting absolute address on the stack:

```
pushaddress(displ):
s :- 2; word(s) := b;
word(s) :+ word(q); q :+ 2;
next;
```

When this code piece is executed the virtual instruction counter q points to its argument (the displacement). This is a general rule because the jump to the next code piece increments q as soon as it has used the operation code as a table index.

The instruction pushvariable computes the absolute address of an enumeration variable and pushes the value of the variable on the stack:

```
pushvariable(displ):
w := b;
w :+ word(q); q :+ 2;
s :- 2; word(s) := word(w);
next;
```

The instruction copyword assigns the value stored in the top of the stack to the enumeration variable whose address is stored below the top of the stack. The address and the value of the variable are popped from the stack:

```
copyword:
word(word(s + 2)) := word(s); s :+ 4;
next;
```

The instruction copybyte assigns the rightmost byte stored in the top of the stack to the byte variable whose address is stored below the top of the stack. The address and the value of the variable are then popped from the stack:

```
copybyte:
byte(word(s + 2)) := byte(s); s :+ 4;
next;
```

This instruction is included because the processing of text strings is a fairly frequent case that is awkward to handle on some 16-bit microprocessors.

The instruction equalword compares two words in the top of the stack and replaces them with a boolean value (1 or 0) defining whether or not they are equal:

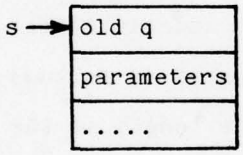
```
equalword:
w := 0;
word(s) compare word(s + 2); s :+ 2;
if equal then
  w :+ 1;
word(s) := w;
next
```

The comparison instruction sets a condition register which can be tested only (but not stored directly). The testing is done by a skip instruction followed by an increment instruction. This is described here by an *if* statement of two lines.

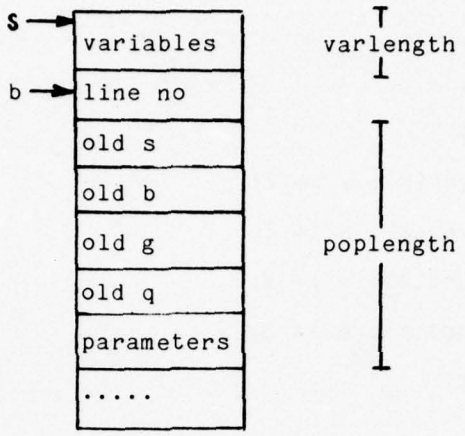
The *index* instruction assumes that the stack already contains the absolute address of an array variable and the value of an index into the array. The instruction checks that the index i is within a fixed range ($min \leq i \leq max$) and replaces the array address @A and the index i by the address of the array element A(i) computed as follows:
$@A(i) = @A + (i - min) * length$ (where length is the number of bytes per array element):

```
index(min, max-min, length):
x := word(s); s :+ 2;
x :- word(q); q :+ 2;
if less then
  goto rangeerror;
x compare word(q); q :+ 2;
if greater then
  goto rangeerror;
x :* word(q); q :+ 2;
word(s) :+ x;
next;
```

Enter is the first instruction executed within a procedure. It assumes that the stack already contains the parameters and return address (old q) of the call



The instruction allocates stack space for the dynamic link and the variables of the procedure call




```
enter(notused, poplength, lineno, varlength):
q :+ 2;
s :- 2; word(s) := g;
s :- 2; word(s) := b;
s :- 2;
word(s) := s;
word(s) :+ word(q); q :+ 2;
s :- 2; word(s) := word(q); q :+ 2;
b := s;
s :- word(q); q :+ 2;
next;
```

The unused argument is irrelevant in the Concurrent Pascal subset.

Exit is the last instruction executed within a procedure. It removes the local variables, the dynamic link, and the parameters of the procedure call from the stack and reestablishes the previous values of the registers from the dynamic link:

```
exit:
s := b;
s :+ 2;
w := word(s); s :+ 2;
b := word(s); s :+ 2;
g := word(s): s :+ 2;
q := word(s); s :+ 2;
s := w;
next;
```

4. IMPLEMENTATION CONSTRAINTS

Since Concurrent Pascal originally was implemented on the PDP 11/45 the virtual code is somewhat biased by the general features of that machine (although it does not reflect any of its details).

In evaluating other machines it therefore makes sense to give the implementor some freedom to modify the virtual code slightly to use these machines efficiently. There are, however, very good reasons for limiting this freedom. Since the long-term goal of this project is to invent simpler computer architectures an obsession with the peculiarities of existing machines would only distract us from that goal. Furthermore, it is essential to use the existing Concurrent Pascal compiler witn as few changes as possible to limit the initial effort.

With these conflicting goals in mind we have established the following constraints on the implementation of interpreters on microcomputers:

The following will not change from one machine to another

The word length (16 bits)

The format and size of the virtual instructions

The format and size of variables

The structure and direction of the stack

The following may be changed:

The operation codes (as long as they are fixed)

The encoding of byte addresses in the virtual code and the stack

The table, registers, and machine code used to implement the interpreter.

The changes must only require modifications to the last pass of the compiler.

5. GA 16/110 MICROCOMPUTER

5.1 Machine Properties

The General Automation 16/110 and 16/220 computers are single board LSI microcomputers. The machines are similar and will be treated together. There are 7 registers: 3 accumulators, 3 index registers, and a base register. There are two separate sets of registers, but only one set is accessible at a time, so this feature will be ignored. In absolute addressing mode the first 32 words of memory can be accessed. Indirect addressing and indexing may be used with this mode. In based mode an offset in the range 0 to 31 may be added to the contents of the base register to determine the address. This too may be used with indirect and indexed addressing. Some instructions allow program counter relative addressing, and others allow immediate operands. Addresses are usually word addresses. They are either 15 or 16 bits in length, depending on a bit in the machine state. Two machine instructions use byte addresses, in which the least significant address bit is used to select a byte and the rest is used as a word address. The byte addressing within a word is the opposite of the PDP 11s.

5.2 Interpreter Implementation

All displacements and increments will be in words. Addresses stored in the stack will generally be word addresses, except that the address of a character within a string is a byte address.

The Concurrent Pascal machine registers will be allocated as follows:

x, y	accumulators
s, w	index registers
b, g	low memory (locations 0 to 31)
q	base register

The transfer operation next uses the technique of treaded code. Each virtual instruction contains an absolute address in the operation table. The next operation is

```
x:=word(word(q));
q:+1;
p:=x;
```

This sequence takes 8.2 usec.

The following describes the code pieces for the nine most frequent virtual instructions.

```
pushconstant(value):
        x:=word(q);
        q:+1;
        s:-1;
        word(s):=x;
        next;

pushaddress(displ):
        x:=word(q);
        q:+1;
        y:=b;
        x:+y;
        s:-1;
        word(s):=x;
        next;
```

```
pushvariable(displ):
        x:=word(q);
        q:+1;
        w:=b;
        w:+x;
        x:=word(w);
        s:-1;
        word(s):=x;
        next;


copyword:
        x:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        word(w):=x;
        next;
```

The exclusive or instruction in the following code piece changes a PDP 11 byte index to a GA 16/110 byte index.

```
copybyte:
        x:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        w: xor 1;
        byte(w):=right(x);
        next;
```

The if statement in the following code piece is a conditional jump instruction. The compare instruction sets indicators for the test.

```
equalword:
        w:=0;
        x:=word(s);
        s:+1;
        x compare word(s);
        if equal then
          w:+1;
        word(s):=w;
        next;
```

In the next code piece, the length parameter is the length in words except for character arrays. In this case it is zero and a byte address is computed.

```
index(min,max-min,length):
        y:=word(s);
        s:+1;
        x:=word(q);
        q:+1;
        y:-x;
        if x < 0 then goto err;
        y compare word(q);
        if greater then goto err;
        q:+1;
        w:=word(s);
        x:=word(q);
        q:+1;
        if x = 0 then goto A;
        y:*x;
        goto B;
  A:    w:*2;
  B:    y:+w;
        word(s):=y;
        next;
  err:  goto rangeerror;


enter(notused,poplength,lineno,varlength):
        q:+1;
        s:-1;
        x:=g;
        word(s):=x;
        s:-1;
        x:=b;
        word(s):=x;
        s:-1;
        x:=word(q);
        q:+1;
        x:+s;
        word(s):=x;
        s:-1;
        x:=word(q);
        q:+1;
        word(s):=x;
        b:=s;
        x:=word(q);
        q:+1;
        s:-x;
        next;
```

```
exit:
        w:=b;
        s:=word(w+1);
        x:=word(w+2);
        b:=x;
        x:=word(w+3);
        g:=x;
        q:=word(w+4);
        next;
```

5.3 Performance

The execution times and space requirements are summarized below.

virtual instruction	execution time (usec)	weighted time (usec)	size (words)
pushvariable	26.2	6.8	10
index	48.2	3.4	22
pushaddress	23.6	3.1	9
pushconstant	18.9	1.7	7
equalword	26.6	1.3	10
copyword	21.5	1.1	8
copybyte	24.6	1.0	9
enter	60.7	.6	23
exit	27.2	.3	10
		19.3	108

Average instruction time: 27.2 usec.

6. LSI 11 MICROCOMPUTER

The LSI 11 is a microcomputer version of the PDP 11 minicomputer. It uses the same basic instructions as the PDP 11/45. The differences do not affect the code pieces of the interpreter. Thus only the speeds of the machines differ. The execution times and space requirements are summarized below.

virtual instruction	execution time (usec)	weighted time (usec)	size (words)
pushvariable	24.2	6.3	4
index	77.4	5.4	12
pushaddress	22.0	2.9	3
pushconstant	13.6	1.2	2
equalword	29.8	1.5	6
copyword	13.0	.6	2
copybyte	13.0	.5	2
enter	54.6	.5	10
exit	39.9	.4	8
		19.3	49

Average instruction time: 27.2 usec.

7. NOVA MICROCOMPUTER

7.1 Machine Properties

The NOVA computer family is a range of 16 bit processors, including a single chip microprocessor. All execute the same basic instruction set. The machine has 4 registers, 2 of which may be used as index registers. Arithmetic can only be done between register pairs. The stack operations PUSH and POP are included in the instruction set, but the stack grows in the opposite direction from the Concurrent Pascal stack and is therefore is not used for our purposes. The addressing modes of interest are direct and based. In the direct mode the instruction can directly access a word with memory address 0-255. In based mode an offset in the range -128 to 127 is added to the contents of one of the index registers to determine the memory address. Indirect addressing may be specified in addition to each of these modes. Auto-increment locations, memory words 16-23, are incremented automatically after serving as indirect addresses. All addresses are 15 bit word addresses.

7.2 Interpreter Implementation

All displacements and increments will be in words. A variable address stored in the stack is usually an absolute word address. The only exception is the address of a character within a string. This is a byte address which will be interpreted by one of the instructions pushbyte or copybyte.

The Concurrent Pascal registers will be allocated as follows:

x, y	registers
w, z	index registers
s, b, g	low memory (below 255)
q	auto-increment memory (16-23)

The transfer operation next uses a modified form of threaded code. Each code piece finishes by jumping to the next virtual instruction, which jumps to the next code piece. An operation table in low memory contains the absolute address of all code pieces. So the next operation is

```
p:=q; q:+1;
```

and the virtual instruction is

```
p:=word(addr)
```

where addr is a fixed address in the operation table. The whole sequence takes about 9 usec.

The following describes the code pieces for the nine most frequent virtual instructions.

```
pushconstant(value):
        x:=word(q); q:+1;
        s:-1;
        word(s):=x;
        next;
```

```
pushaddress(displ):
        x:=word(q); q:+1;
        y:=b;
        y:+x;
        s:-1;
        word(s):=y;
        next;


pushvariable(displ):
        x:=word(q); q:+1;
        w:=b;
        w:+x;
        x:=word(w);
        s:-1;
        word(s):=x;
        next;


copyword:
        x:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        word(w):=x;
        next;
```

The c register used in the following code piece is a one bit register. The if statements are conditional skip instructions. An octal constant #177400 is a byte mask and is stored in low memory.

```
copybyte:
        x:=word(s);
        s:+1;
        y:=word(s);
        s:+1;
        w:=y div 2; c:= y mod 2;
        y:=word(w);
        if c = 1 then
          swap bytes in y;
        z:=#177400;
        y: and z;
        y:+x;
        if c = 1 then
          swap bytes in y;
        word(w):=y;
        next;
```

```
equalword:
        w:=0;
        x:=word(s);
        s:+1;
        y:=word(s);
        if x = y then
          w:+1;
        word(s):=w;
        next;
```

In the following code piece the length argument is the length in words. For character strings the length will be zero. A byte address is formed by concatenating a low order bit to the word address.

```
index(min,max-min,length):
        y:=word(s);
        s:+1;
        x:=word(q); q:+1;
        y:-x;
        if y < 0 then
          goto rangeerror;
        w:=word(q); q:+1;
        if y > w then
          goto rangeerror;
        w:=word(s);
        x:=word(q); q:+1;
        if x = 0 then
          w:*2;
        if x > 1 then
          y:*x;
        y:+w;
        word(s):=y;
        next;
```

```
enter(notused,poplength,lineno,varlength):
        q:+1;
        s:-1;
        x:=g;
        word(s):=x;
        s:-1;
        x:=b;
        word(s):=x;
        s:-1;
        x:=s;
        y:=word(q); q:+1;
        x:+y;
        word(s):=x;
        s:-1;
        x:=word(q); q:+1;
        word(s):=x;
        x:=s;
        b:=x;
        y:=word(q); q:+1;
        x:-y;
        s:=x;
        next;


exit:
        w:=b;
        x:=word(w+1);
        s:=x;
        x:=word(w+2);
        b:=x;
        x:=word(w+3);
        g:=x;
        x:=word(w+4);
        q:=x;
        next;
```

7.3 Performance

The execution times and space requirements are summarized below.

virtual instruction	execution time (usec)	weighted time (usec)	size (words)
pushvariable	30.4	7.9	7
index*	57.4	4.0	18
pushaddress	27.5	3.6	6
pushconstant	22.2	2.0	4
equalword	31.9	1.6	8
copyword	27.5	1.4	6
copybyte	50.1	2.0	15
enter	79.1	.8	21
exit	34.8	.3	10
		23.6	95

*Time given is for character string access. Others require 96.3 usec.

Average instruction time: 33.2 usec.

8. PACE MICROCOMPUTER

8.1 Machine Properties

The National Semiconductor PACE is a single chip 16 bit microprocessor. It has 4 accumulators, two of which may also be used as index registers. There are five addressing modes of interest. Direct addressing allows any word in low memory (addresses 0 to 255) to be accessed. Direct indexed addressing forms the effective address by adding a displacement in the range -128 to 127 to an index register. Indirect addressing can be used with these modes to give indirect and indirect indexed addressing. If indirect addressing is used, the source or destination register must be accumulator 0. Some instructions allow program counter relative instructions (jumps), and others provide immediate 8 bit operands. The PACE includes stack instructions but the stack holds a maximum of 10 elements and thus does not satisfy our requirements. All addressing is by word.

8.2 Interpreter Implementation

All displacements and increments will be by words. A variable address stored in the stack is usually a word address. The only exception is the address of a character within a string. This is a byte address which will be interpreted by pushbyte and copybyte.

The Concurrent Pascal machine registers will be allocated as follows:

x	register 0
y	register 1
w, s	registers 2 and 3, which are index registers
b, g, q	low memory

The transfer operation next jumps from the end of a code piece to the next virtual instruction, which jumps to the next code piece. The q register is incremented as the first operation of each code piece. So the next operation consists of the steps:

```
p:=q;
p:=word(addr);
```

and

```
q:+1;
```

Here addr is a fixed address in the operation table, which is stored in low memory. The whole sequence, which will be written simply next, takes 10.5 usec.

The following describes the code pieces for the nine most frequent virtual instructions.

```
pushconstant(value):
        s:-1;
        x:=word(q);
        q:+1;
        word(s):=x;
        next;
```

```
pushaddress(displ):
        x:=word(q);
        q:+1;
        x:+b;
        s:-1;
        word(s):=x;
        next;


pushvariable(displ):
        x:=word(q);
        q:+1;
        x:+b;
        s:-1;
        w:=x;
        x:=word(w);
        word(s):=x;
        next;


copyword:
        x:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        word(w):=x;
        next;
```

The following code piece uses a one bit register c, which can be tested by a conditional branch instruction. The octal constant #177600 is used as a mask.

```
copybyte:
        y:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        w: div 2; c:= w mod 2;
        x:=word(w);
        if c = 1 then goto A;
        x: and #177600;
        y:+x;
        goto B;
  A:    swap bytes in x;
        x: and #177600
        y:+x;
        swap bytes in y;
  B:    word(w):=y;
        next;
```

The following code piece uses exclusive or (xor) to test for equality since subtraction is not available.

```
equalword:
        w:=1;
        y:=word(s);
        s:+1;
        x:=word(s);
        x: xor y;
        if x <> 0 then
          w:=0;
        word(s):=w;
        next;
```

In the following code piece a comment notes the lack of a multiplication instruction. Multiplication must be done by software.

```
index(min,max-min,length):
        y:=word(s);
        s:+1;
        x:=word(q);
        q:+1;
        x:=-x;
        x:+y;
        if x < 0 then goto err;
        y:=x;
        x:=word(q);
        q:+1;
        x:=-x;
        x:+y;
        if x > 0 then goto err;
        w:=word(s);
        x:=word(q);
        q:+1;
        if x = 0 then goto A;
        " multiply y:*x  "
        goto B;
  A:    w:*2;
  B:    w:+y;
        word(s):=w;
        next;
  err:  goto rangeerror;
```

```
enter(notused,poplength,lineno,varlength):
        q:+1;
        s:-1;
        x:=g;
        word(s):=x;
        s:-1;
        x:=b;
        word(s):=x;
        s:-1;
        x:=word(q);
        q:+1;
        x:+s;
        word(s):=x;
        s:-1;
        x:=word(q);
        q:+1;
        word(s):=x;
        b:=s;
        w:=word(q);
        q:+1;
        x:=-x;
        s:+x;
        next;


exit:
        w:=b;
        s:=word(w+1);
        x:=word(w+2);
        b:=x;
        x:=word(w+3);
        g:=x;
        x:=word(w+4);
        q:=x;
        next;
```

8.3 Performance

The execution times and space requirements are summarized below.

virtual instruction	execution time (usec)	weighted time (usec)	size (words)
pushvariable	33.6	8.7	9
index*	80.5	5.6	24
pushaddress	28.0	3.6	7
pushconstant	25.2	2.3	6
equalword	34.3	1.7	10
copyword	25.9	1.3	7
copybyte	51.1	2.0	17
enter	81.9	.8	23
exit	32.9	.3	10
		26.3	113

* Time and space for multiplication not included.

Average instruction time: 37.0 usec.

9. CA LSI 4/10 MICROCOMPUTER

9.1 Machine Properties

The Computer Automation LSI 4/10 computer is a single board LSI microprocessor. It has 2 accumulators and 2 index registers. There are four relevant addressing modes. Short absolute addressing accesses a word in memory locations 0 to 63. To this may be added the contents of one or both index registers. Long absolute addressing allows any address to be selected, modified by one or both index registers. Long addressing mode requires two word instructions. Some instructions are available in long or short form only. Short indirect addressing finds the memory address in low memory (locations 0 to 63). Relative addressing is used for program counter relative jump instructions. Addresses are either (1) fifteen bit word addresses; (2) sixteen bit word addresses; or (3) sixteen bit byte addresses. Two bits in the machine state determine the addressing currently in effect. Byte addressing within a word is opposite of the PDP 11. This machine has a stack, but there are no instructions for pushing or popping individual items, so it is unsuitable for our purposes and is not used.

9.2 Interpreter Implementation

All displacements and increments will be words. The machine state will normally specify word addressing. A variable stored in the stack is normally a word address except when it refers to a character in a string, in which case it is a byte address. The pushbyte and copybyte

instructions temporarily change the state to byte addressing.

Concurrent Pascal registers are allocated as follows:

x, y accumulators

s, w index registers

b, g, q low memory

The transfer operator next uses the technique of threaded code. The operation is

```
w:=word(word(q));
q:+1;
p:=w;
```

This sequence takes 19.7 usec.

The following describes the code pieces for the nine most frequent virtual instructions.

```
pushconstant(value):
        x:=word(q);
        q:+1;
        s:-1;
        word(s):=x;
        next;

pushaddress(displ):
        x:=word(q);
        q:+1;
        x:+b;
        s:-1;
        word(s):=x;
        next;
```

```
pushvariable(displ):
        w:=word(q);
        q:+1;
        w:+b;
        x:=word(w);
        s:-1;
        word(s):=x;
        next;


copyword:
        x:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        word(w):=x;
        next;
```

The exclusive or instruction in the following code piece is used to change a LSI 4/10 byte index into a PDP 11 byte index.

```
copybyte:
        x:=word(s);
        s:+1;
        w:=word(s);
        s:+1;
        w: xor 1;
        change to byte mode;
        byte(w):=right(x);
        change to word mode;
        next;
```

The if statement in the following code piece is a conditional jump instruction.

```
equalword:
        y:=0;
        x:=word(s);
        s:+1;
        x:-word(s);
        if x = 0 then
          y:+1;
        word(s):=y;
        next;
```

```
index(min,max-min,length):
        y:=word(s);
        s:+1;
        y:-word(q);
        q:+1;
        if y < 0 then goto err;
        x:=word(q);
        q:+1;
        x:-y;
        if x > 0 then goto err;
        x:=word(s);
        w:=word(q);
        if w <> 0 then goto A;
        x:*2;
        y:+x;
        goto B;
  A:    y:*word(q)+x;
  B:    q:+1;
        word(s):=y;
        next;
  err:  goto rangeerror;

enter(notused,poplength,lineno,varlength):
        q:+1;
        s:-1;
        x:=g;
        word(s):=x;
        s:-1;
        x:=b;
        word(s):=x;
        s:-1;
        x:=word(q);
        q:+1;
        x:+s;
        word(s):=x;
        s:-1;
        x:=word(q);
        q:+1;
        word(s):=x;
        b:=s;
        s:-word(q);
        q:+1;
        next;
```

```
exit:
        w:=b;
        s:=word(w+1);
        x:=word(w+2);
        b:=x;
        x:=word(w+3);
        g:=x;
        x:=word(w+4);
        q:=x;
        next;
```

9.3 Performance

The execution times and space requirements are summarized below.

virtual instruction	execution time (usec)	weighted time (usec)	size (words)
pushvariable	46.1	12.0	9
index	106.9	7.5	22
pushaddress	42.4	5.5	8
pushconstant	38.6	3.5	7
equalword	46.0	2.3	10
copyword	39.6	2.0	8
copybyte	57.8	2.3	11
enter	116.2	1.2	22
exit	51.9	.5	11
		36.8	108

Average instruction time: 51.8 usec.

10. TI 9900 MICROCOMPUTER

10.1 Machine Properties

The Texas Instruments 9900 microcomputer is a single chip microprocessor compatible with a family of minicomputers. The machine has 16 general purpose registers, which are implemented in main memory. A workspace pointer contains the address of the register block. The register mode specifies that the operand is in a register. The register indirect mode specifies that the address is in a register. Register indirect auto-increment mode specifies that the address in in a register, and that the register is to be incremented after it is used. Some instructions use other modes, such as program counter relative jumps. Addresses are byte addresses, and each instruction specifies whether it operates on byte or word data.

10.2 Interpreter Implementation

The Concurrent Pascal registers are all allocated in general registers. Addressing is identical to that on the PDP 11. The transfer operator next implements threaded code, as on the PDP 11. The operation table may be located anywhere in memory, and the virtual instructions contain absolute address of entries in this table. The next operation is:

```
x:=word(q); q:+2;
p:=word(x);
```

This takes approximately 17 usec.

The following describes the code pieces for the nine most frequent virtual instructions.

```
pushconstant(value):
        s:-2;
        word(s):=word(q); q:+2;
        next;


pushaddress(displ):
        x:=word(q); q:+2;
        s:-2;
        x:+b;
        word(s):=x;
        next;


pushvariable(displ):
        x:=word(q); q:+2;
        x:+b;
        s:-2;
        word(s):=word(x);
        next;


copyword:
        x:=word(s); s:+2;
        y:=word(s); s:+2;
        word(y):=x;
        next;


copybyte:
        x:=word(s); s:+2;
        y:=word(s); s:+2;
        byte(y):=right(x);
        next;
```

The <u>if</u> statement in the following code piece is a conditional skip instruction.

```
equalword:
        w:=0;
        word(s) compare word(s+2); s:+2;
        if x = y then
          w:+1;
        word(s):=w;
        next;


index(min,max-min,length):
        x:=word(s); s:+2;
        x:-word(q); q:+2;
        if x < 0 then goto err;
        x compare word(q); q:+2;
        if greater then goto err
        x:*word(q); q:+2;
        word(s):+x;
        next;
  err:  goto
          rangeerror;


enter(notused,poplength,lineno,varlength):
        q:+2;
        s:-2;
        word(s):=g;
        s:-2;
        word(s):=b;
        s:-2;
        word(s):=s;
        word(s):+word(q); q:+2;
        s:-2;
        word(s):=word(q); q:+2;
        b:=s;
        s:-word(q); q:+2;
        next;


exit:
        s:=b;
        s:+2;
        w:=word(s); s:+2;
        b:=word(s); s:+2;
        g:=word(s); s:+2;
        q:=word(s); s:+2;
        s:=w;
        next;
```

10.3 Performance

The execution times and space requirements are summarized below.

virtual instruction	execution time (usec)	weighted time (usec)	size (words)
pushvariable	52.7	13.7	6
index	92.7	6.5	11
pushaddress	50.7	6.6	6
pushconstant	36.0	3.2	4
equalword	53.3	2.7	7
copyword	49.3	2.5	5
copybyte	49.3	2.0	5
enter	117.3	1.2	14
exit	82.7	.8	9
		39.2	67

Average instruction time: 55.2 usec.

11. PERFORMANCE COMPARISON

Machine	Speed (usec)	Relative Size
PDP 11/45	10	100
GA 16/110	27	220
LSI 11	27	100
NOVA	33	194
PACE (no mult)	37	231
CA LSI 4/10	52	220
TI 9900	55	137

Acknowledgment

This research has been partially supported by the Office of Naval Research under contract NR049-415.

References

(1) Brinch Hansen, P., The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, NJ, July 1977a.

(2) Brinch Hansen, P., A Concurrent Pascal Subset. Computer Science Department, University of Southern California, Los Angeles, CA, Nov. 1977b.

(3) PDP 11 Processor Handbook. Digital Equipment Corporation, 1975.

(4) Bell, J.R., Threaded Code. Comm. ACM 16, 6 (June 1973), 370-72.

MICROCOMPUTER EVALUATION: PROCESSOR MULTIPLEXING

Per Brinch Hansen and Charles Hayden

Computer Science Department
University of Southern California
Los Angeles, California 90007

January 1978

Summary

This is one of several reports which compare 16 bit microcomputers with respect to implementation of a Concurrent Pascal subset. This report evaluates the process switching time of various microprocessors, that is, the time required to preempt one process and resume another. This influences the response time of a microcomputer to real-time events (such as interrupts).

Contents

1. INTRODUCTION

This is one of several reports which compare 16 bit microcomputers with respect to implementation of a Concurrent Pascal subset [1,2]. This report evaluates the process switching time of various microprocessors, that is the time required to preempt one process and resume another one. The process switching time and the scheduling policy of processes determine the response time of a microcomputer to real-time events (such as interrupts).

2. PROCESS SCHEDULING

In the implementation of the Concurrent Pascal subset, the scheduling of processes is simplified as much as possible:
The processor executes one process at a time. In general a running process continues its execution until it either terminates or waits for some condition to be satisfied. A process can wait until a monitor gate is open, or until a continue operation has been performed on a monitor queue or until an input/output operation has been completed.

When a process is not running it waits in a ready queue which is served in first-come, first-served order.

A process waiting for one of the conditions mentioned earlier will be resumed periodically (just as any other process). If the process finds that its condition is satisfied then it will continue execution; otherwise it reenters the ready queue to wait for another turn.

After initialization the only operations on the ready queue are preempt and resume.

Preempt saves the register values of the running process on top of its stack and enters the stack address in the ready queue. If the ready queue is full the concurrent program terminates with an error message.

Resume restores the stack address of a process from the ready queue, restores the register values from the stack, and continues the execution of the process. If the ready queue is empty the concurrent program terminates.

Since process switching takes place only after the completion of a virtual instruction it is necessary only to save and restore the values of the registers q, g, b, and s.

In Pascal the ready queue can be represented by an array of words and three variables defining the indices of the head and tail elements as well as the length of the queue:

```
const proclim = capacity of ready queue;

var
     list: array[1..proclim] of word;
     head, tail: 1..proclim;
     length: 0..proclim;

procedure preempt;
begin
  if length = proclim then terminate(proclimit);
  s:= pred(s); word[s]:=q;
  s:= pred(s); word[s]:=g;
  s:= pred(s); word[s]:=b;
  list[tail]:= s;
  tail:= tail mod proclim + 1;
  length:= succ(length)
end;
```

```
procedure resume;
begin
  if length = 0 then terminate(terminated);
  s:= list[head];
  head:= head mod proclim + 1;
  b:= word[s]; s:= succ(s);
  g:= word[s]; s:= succ(s);
  q:= word[s]; s:= succ(s);
  length:= pred(length)
end;
```

3. IMPLEMENTATION

On three of the four machines evaluated the implementation is quite close to the Pascal procedures. The registers are stored on the stack just as in a procedure call. The TI 9900 processor, on the other hand, has a fast processor switching mechanism. On this machine the general registers are stored in main memory. The workspace pointer is a register that defines the base address of the current register block. On this machine the variables head, tail, and length, as well as needed constants are stored in a separate register block that is used only during process switching. The previous workspace pointer is stored in the ready queue.

3.1. GA 16/110 Implementation

The variables length, head, and tail are allocated in low memory so they may be accessed in absolute mode. Several two-word immediate instructions are used.

```
preempt:
  x:= length;
  x compare proclim;
  if equal then goto proclimit;
  s:-1;
  word(s):=q;
  x:=g;
  s:-1;
  word(s):=x;
  x:=b;
  s:-1;
  word(s):=x;
  w:=tail;
  word(w):=s;
  w:+1;
  w compare lasttail;
  if equal then
    w:-proclim;
  tail:=w;
  length:+1;
  return;

resume:
  x:=length;
  x compare 0;
  if equal then goto terminated;
  w:=head;
  s:=word(w);
  w:+1;
  w compare lasthead;
  if equal then
    w:-proclim;
  head:=w;
  length:-1;
  x:=word(s);
  b:=x;
  s:+1;
  x:=word(s);
  g:=x;
  s:+1;
  q:=word(s);
  s:+1;
  return;
```

3.2. LSI 11 Implementation

The variables length, head, and tail are stored in memory. The constants proclim and lasttail are immediate operands.

```
preempt:
  length compare proclim;
  if equal then goto proclimit;
  s:-2; word(s):=q;
  s:-2; word(s):=g;
  s:-2; word(s):=b;
  y:=tail;
  word(y):=s; y:+2;
  y compare lasttail;
  if equal then
    y:-2*proclim;
  tail:=y;
  length:+1;
  return;

resume:
  test length;
  if zero then goto terminated;
  x:=head;
  s:-word(x); x:+2;
  x compare lasthead;
  if equal then
    x:-2*proclim;
  head:=x;
  b:=word(s); s:+2;
  g:=word(s); s:+2;
  q:=word(s); s:+2;
  length:-1;
  return;
```

3.3. NOVA Implementation

The variables length, head, and tail, as well as constants proclim and lasttail are stored in low memory.

```
preempt:
  x:=length;
  y:=proclim;
  if x = y then
    goto proclimit;
  s:-1;
  word(s):=q;
  s:-1;
  word(s):=g;
  s:-1;
  word(s):=b;
  x:=s;
  w:=tail;
  word(w):=x;
  w:+1;
  x:=lasttail;
  if x = w then
    w:-y;
  tail:=w;
  length:+1;
  return;

resume:
  x:=length;
  if x = 0 then
    goto terminated;
  w:=head;
  x:=word(w);
  s:=x;
  w:+1;
  y:=proclim;
  x:=lasthead;
  if x = w then
    w:-y;
  head:=w;
  x:=word(s);
  s:+1;
  b:=x;
  x:=word(s);
  s:+1;
  g:=x;
  x:=word(s);
  s:+1;
  q:=x;
  return;
```

3.4. TI 9900 Implementation

The preempt and resume programs for the TI 9900 are somewhat different. The work space pointer WP is saved and restored from the the ready queue and the other registers are saved and restored as a side effect of this. Thus the corresponding Pascal procedures are:

```
procedure preempt;

begin
  if length = proclim then terminate(proclimit);
  list[tail]:=WP;
  tail:=tail mod proclim + 1;
  length:= succ(length)
end;

procedureresume;

begin
  if length = 0 then terminate(terminated);
  WP:=list[head];
  head:= head mod proclim + 1;
  length:= pred(length)
end;
```

The variables length, head, and tail, as well as the constant proclim, are stored in registers. The WP register contains the previous value of the workspace pointer (before the call), and the workspace pointer is restored from the same register by a return instruction. Using this scheme each process requires a dedicated register block of 16 words. This could be allocated out of its stack space at initialization time.

```
preempt:
  length compare proclim;
  if equal then goto proclimit;
  word(tail):=WP; tail:+2;
  tail compare lasttail;
  if equal then
    tail:-proclim;
  length:+2;
  return;

resume:
  test length;
  if zero then goto terminated;
  WP:= word(head); head:+2;
  head compare lasthead;
  if equal then
    head:-proclim;
  length:-2;
  return;
```

4. PERFORMANCE

Machine	Speed (usec)	Space (words)
TI 9900	80	16
GA 16/110	111	44
NOVA	120	42
LSI 11	143	39

Process switching time for all machines takes about as much time as 2 to 5 virtual instructions. The TI 9900 clearly benefits from the fast switching mechanism in its instruction set. Unfortunately the flexibility that allowed this was partly responsible for its poor performance in interpreting Concurrent Pascal code. The other machines are similar to each other in performance and in the nature of the length of the code.

Acknowledgment

This research has been partially supported by the Office of Naval Research under contract NR049-415.

References

(1) Brinch Hansen, P., A Concurrent Pascal subset. Computer Science Department, University of Southern California, Los Angeles (In preparation).

(2) Brinch Hansen, P., and Hayden, C., Microcomputer evaluation: instruction sets. Computer Science Department, University of Southern California, Los Angeles, Dec. 1977.

DISTRIBUTION LIST

FOR THE TECHNICAL, ANNUAL, AND FINAL REPORTS

FOR CONTRACT N00014-77-C-0714

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, VA 22217	2 copies
Office of Naval Research Code 715LD Arlington, VA 22217	6 copies
Office of Naval Research Code 200 Arlington, VA 22217	1 copy
Office of Naval Research Code 455 Arlington, VA 22217	1 copy
Office of Naval Research Code 458 Arlington, VA 22217	1 copy
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA 91106	1 copy
Office of Naval Research New York Area Office 715 Broadway - 5th Floor New York, NY 10003	1 copy

Naval Research Laboratory — 6 copies
Technical Information Division
Code 2627
Washington, D. C. 20375

Dr. A. L. Slafkosky — 1 copy
Scientific Advisor
Commandant of the Marine Corps
(Code RD-1)
Washington, D. C. 20380

Naval Ocean Systems Center — 1 copy
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Research & Development
Center
Computation and Mathematics Department
Bethesda, MD 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350

Mr. Kin B. Thompson — 1 copy
NAVDAC 33
Washington Navy Yard
Washington, D.C. 20374