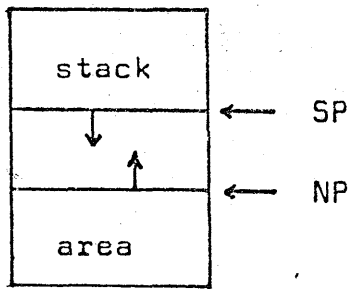
Eidg. Technische Hochschule Zürich
Fachgruppe Computer-Wissenschaften

Clausiusstrasse 55
CH-8006 Zürich

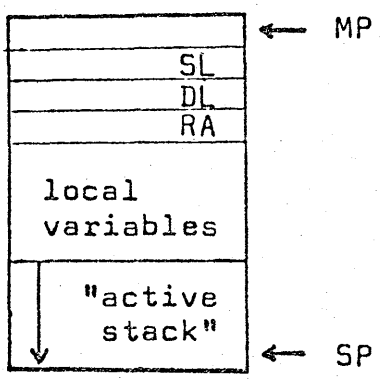
The 10. January letter of Mr. Ammann promised a description of the simple stack computer and its assembly language, as well as a Pascal-written interpreter. The following note is directed to this point.

The computer consists of a program, tables of constants, the store (for variables), and registers. The registers are the program counter (PC), the instruction register (OP, P, Q), and three address registers (SP, MP, NP). The store is divided into a stack, growing from one end, and an area of dynamically allocated variables growing from the other end. SP points to the top of the stack, NP points to the top (or bottom) of the area.



store

The stack consists of a series of data segments. Each segment is a block of information headed by a four-word unit called a mark. The first word is used to store the result, for the case where the segment belongs to a function. The instruction "mark stack" (MST) reserves these words by incrementing the stack pointer while recording the static and dynamic links. A "call user procedure" (CUP) immediately follows. This instruction sets MP , the base address, to the beginning of the block and records the return address. MP therefore is always the base address of the youngest data



segment. "Enter" (ENT) then increments the stack pointer, thereby reserving stack space representing the local variables and initializing them to the value "undefined". Every procedure or function, including the main program, is entered in this manner. "Return" (RET) resets the pointers thus "popping" the segment.

To provide dynamic allocation for variables generated by the standard procedure new, the interpreter allocates storage at the location designated by NP and yields NP as the address of the new variable. In the case where the new variable is a record specifying a tag field, new (p,t), an assignment is made to the tag field. The standard procedure reset then allows a release of this area.

The compiler generated instructions have no label field. Instead, a simple counter is used which is incremented by one with each new generation. The jump instructions then use this count as the referenced address.

An instruction, beginning in collumn one and ending with an eol, is a three-letter mnemonic followed by one or two parameters. The first parameter of the compare instructions (EQU, NEQ, GEQ, GRT, LEQ, LES), load constant (LDC), and return (RET) is a "type key" and appears in position four. Compare instructions use the code:

A	for	address comparison	
I		integer	
R		real	
B		Boolean	
S		set	
M		multiple	(e.g. an array)

A multiple comparison is followed by a second parameter which indicates the number of elements to be compared. The key of LDC signals the type of the constant appearing as the next parameter:

I	implies	integer	
R		real	
C		character	
(		set	
N		the nil pointer	
B		Boolean	(0,1)

Where the constant is of type set, the immediately following character is either an I or a C to distinguish an integer from a character set. The set is then terminated by a')'.

RET is followed by a F or P to differenciate between a function or procedure return.

Following CSP (call standard procedure) is a three-letter mnemonic, beginning in position five and referencing a Pascal standard procedure. A LDA (load constant address) is followed by a string constant. In position five is an apostrophe marking the beginning of a string which is then terminated by a second apostrophe. The other instructions have either integer parameters which can be written in free format, or no parameters.

The assembler reads the symbolic code and translates it into an internal representation. (The packing of two instructions into one component of the array CODE is due to the large wordsize of our CDC computer, and is of no further importance.) It was necessary to generate a "load constant indirect" for those values that do not fit into the Q-field.

For bootstrapping purposes, it is necessary to translate the given assembler-interpreter into any available language. One then has a means, admittedly slow, of executing a program of Pascal symbolic code. In particular, one can then execute the symbolic code of the new Pascal compiler which, in turn, is capable of compiling a Pascal program, namely that of a new Pascal compiler modified to generate the target computer's machine-code.

We include a listing of the Assembler-Interpreter. Input to this program was the hand generated symbolic code for the following program:

```
var a,b: integer;
begin a := 0;  b := 1;
      write(a+b);
      repeat a := sqr(b) + b;
             b := a div b
      until a > 20;
      write(a,b,a+b,eol)
end.
```

All test output is shown. We hasten to add that this is an early result and offer no claim of its total correctness. However, we do invite your comments and questions.

Yours sincerely,

Kathleen Jensen

Kathleen Jensen, Assistant

Symbolic Instructions of PASCAL-CODE KJ 22.1.73

Each instruction is packed into a 30-bit field. The op-code occupies a 6-bit field, parameter P a 4-bit field, and parameter Q a 20-bit (address) field.

Alphabetic List of Instructions:

code	mnemonic	parameters	description
40	ABI		absolute value of integer
41	ABR		absolute value of real number
28	ADI		integer addition
29	ADR		real addition
43	AND		Boolean "and"
26	CHK	Q	check against upper and lower bounds
15	CSP	Q	call standard procedure
12	CUP	P Q	call user procedure
57	DEC	Q	decrement address
45	DIF		set difference
53	DVI		integer division
54	DVR		real division
13	ENT	Q	enter block
27	EOF		test on end of file
17	EQU	P (Q)	compare on equal
24	FJP	Q	false jump
34	FLO		float next to the top
33	FLT		float top of the stack
19	GEQ	P (Q)	greater or equal
20	GRT	P (Q)	greater than
10	INC	Q	increment address
9	IND	Q	indexed fetch
48	INN		test set membership (in)
46	INT		set intersection
44	IOR		Boolean "inclusive or"
16	IXA	Q	compute indexed address

code	mnemonic	parameters	description
5	LAO	Q	load base-level address
56	LCA	Q	load address of constant
4	LDA	P Q	load address
7	LDC	P Q	load constant
1	LDO	Q	load contents of base-level address
21	LEQ	P (Q)	less than or equal
22	LES	P (Q)	less than
0	LOD	P Q	load contents of address
49	MOD		modulus
55	MOV	Q	move
51	MPI		integer multiplication
52	MPR		real multiplication
11	MST	P	mark stack
18	NEQ	P (Q)	not equal
36	NGI		integer sign inversion
37	NGR		real sign inversion
42	NOT		Boolean "not"
50	ODD		test on odd
14	RET	P	return from block
30	SBI		integer subtraction
31	SBR		real subtraction
32	SGS		generate singleton set
38	SQI		square integer
39	SQR		square real
3	SRO	Q	store
6	STO		store at base-level address
58	STP		stop
2	STR	P Q	store at address
35	TRC		truncation
23	UJP	Q	unconditional jump
47	UNI		set union
25	XJP	Q	indexed jump
8		P Q	load constant indirect, an assembler-generated instruction