A
NEW ABSTRACT MACHINE
FOR
S-ALGOL

Alan Dearle
Peter J.Bailey
Alfred L. Brown
Ronald Morrison

Department
of
Computational Science
University of St Andrews

CS/84/7

A New Abstract Machine for S-algol

Contents

Chapter

1 Introduction to the abstract machine

2 Heap Formats

3 The Stacks

4 Abstract Machine Code

5 Loading Code and Initialisation

6 Marking and Reachability

Appendices

I Compiler Output Format

II Code Generated by the Compiler

III Abstract Machine Operation Codes

1 Introduction to the abstract machine

The major difference between the new S machine and the old S machine is the instruction code format. The new machine uses a byte orientated system with most instructions having long and short forms for different lengths of operands.

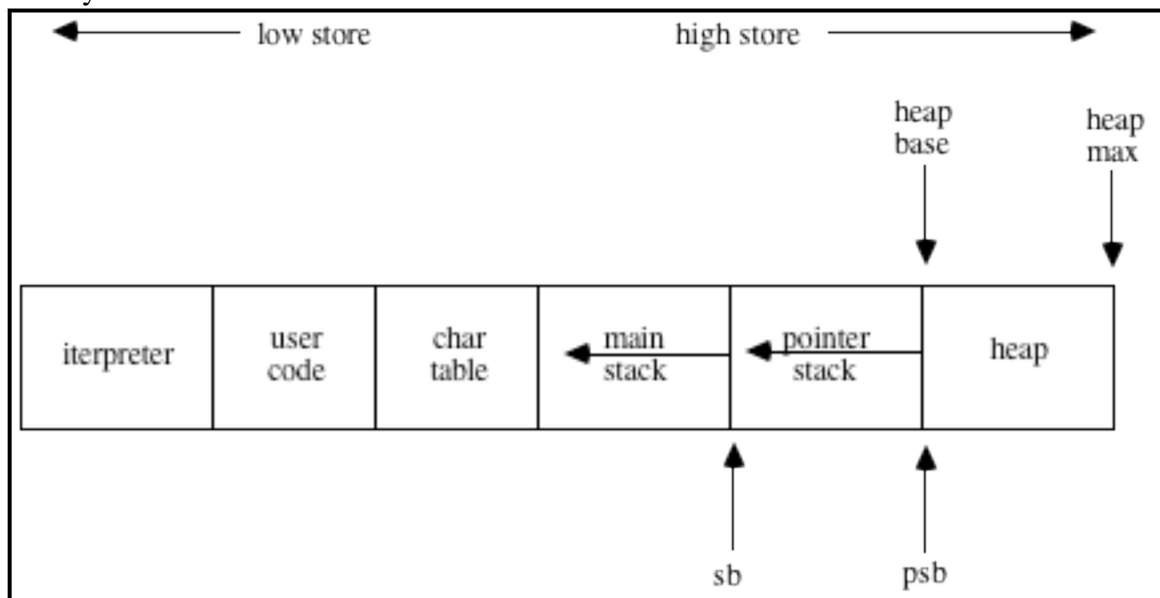
Abstract Machine Registers

The registers of the S machine are used to identify the main and pointer stacks and the code pointer. The registers are:

SF	stack front
PSF	pointer stack front
SP	main stack top
PSP	pointer stack top
SB	main stack base
PSB	pointer stack base
PC	code pointer

Memory Layout

The layout of store is like this :



2 The Heap

String, vectors, structures, and file descriptors reside in the heap. The empty string and the nil pointer are represented by a zero on the pointer stack so they don't reside on the heap. Instructions which use the strings and pointers must take account of this.

The heap looks like a contiguous store of 32 bit units, all heap items are made up from these units – thus every heap item consists of a whole number of words.

The heap then consists of the above items, a linked list of free areas, and left over units of one word (32 bits). These left overs are coalesced during garbage collection.

Every item has a leading tag word which describes what it is – these tag words are described below. Note that bit 31 of a word is the most significant, also note that the format is given in bits from the start of the heap item. The maximum size of a vector is limited by the indices only (in theory at least). The maximum size of a string is 64K chars. The maximum size of a file descriptor is 1M bytes. The maximum size of a structure is compiler dependent.

Headers

The markers in the first word look like this:

bits 0-19	key field or don't care
bit 20	mark bit for the garbage collector
bits 21-27	unused
bits 28-31	type (as above)

Bits 28-31 of the first word of a heap item identify the type of heap object. The following are possible:

Value	**Object**
0	free space
2	vector with pointers
4	file descriptor
6	vector of reals
7	single (used for garbage collection)
8	string
10	vector of integers or booleans
12	structure

Bits in header of free list link -

0-27	offset to next link , or 0 for end of list
28-31	type

String

word 1	bits 0-15	number of characters in the string
word 1	bits 16-31	reserved
word 2		the characters 1 per byte padded up to a 4 byte boundary.

Files

word 1	bits 0-15	The size of the file object which includes the buffer. The size is in bytes and is padded to a 4 byte boundary.
word 1	bits 32	The file descriptor.

Structures

word 1	bits 0-15	The trade mark
word 1	bits 21-27	Number of pointers including the class identifier
word 2		The fields with the pointer fields first.

Vectors

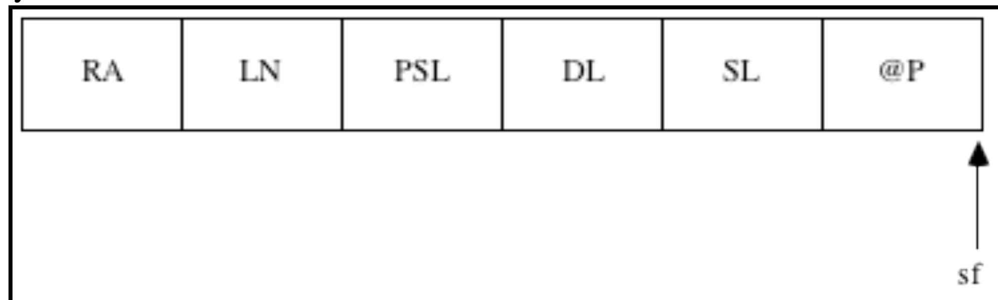
word 1		reserved
word 2		lower bound
word 3		upper bound
word 4		the elements

3 The Stacks

There are two stacks, the main stack and the pointer stack which both grow backwards. The compiler will calculate the maximum stack usage of each stack frame. When set up PSB and SB are set to point at the stack bases.

In order that the current stack frame may call others and return correctly we require to put a mark stack control word (MSCW) on the main stack.

The layout of a MSCW is:



The pointer stack link (PSL) points at the top of the pointer stack. The return address (RA) is the address from which the procedure was called. Similarly the line number(LN) contains the line number from which the procedure was called. Note that the static link(SL), the dynamic link(DL), and procedure address(@) are also part of the MSCW.

4 The Abstract Machine Code

The S-algol abstract machine code, S-code, is designed to fit exactly the needs of the S-algol language. Appendix II describes the code generated for each syntactic construct in the language and Appendix III the operation codes and format of each abstract machine instruction. Here the individual instructions are described. They fall naturally into groups. Typed instructions have an encoded name with the following convention.

ib	integer or boolean
r	real
s	string
p	pointer
pr	procedure
v	void

Jumps

All the jump addresses are relative to the location following the jump address. Only backward jumps have a short form. All jump addresses are word addresses.

fjump(L)	Unconditional jump forward to address L.
bjump(L)	Unconditional jump backwards to address L.
jumpf(L)	Jump forward to L if the top stack element is false. Remove the top element of the stack.
jumptt(L) remove	Jump forward to L if the top element is true. Otherwise the top stack element.

jumpff(L)	Jump forward to L if the top stack element is false. Otherwise remove the top stack element.
cjump.ib,r,s,p(L)	The type determines which stack to use. If the top two stack elements are equal, remove both and jump forward to L. Otherwise remove only the top stack element. Be careful on equality of strings.
bjumpt(L)	Jump backwards to L if the top stack element is true. Remove the top element of the stack.
fortest(L)	The control constant, increment and limit are the top three elements of the stack. If the increment is negative and the control constant is less than the limit or the increment is positive and the control constant is greater than the limit then remove all three from the stack and jump forward to M.
forstep.op(L)	Update the control constant by adding the increment. Then jump backwards to L.

Procedure entry and exit

The instruction sequence to call a procedure is

mst.load
evaluate the parameters
apply.op

The mst.load instruction, of which there are three forms depending on the scope of the procedure, loads the closure on to the top of the stack, fills in the dynamic link and the pointer stack link, and leaves space for the line number and the return address. After the parameter expressions have been evaluated on the stack, apply.op is used to call the procedure.

apply.op(m)	First calculate new SF = SP - m. Next calculate the new PSF = PSL and fill in the return address. Move the contents of the stack location pointed at by SF to PC. This will perform the branch. The first two words of the procedure are the maximum amount of stack space, on each stack, that the procedure may require. The apply instruction checks that the space is available. If there are no parameters the mst.load and apply instructions are combined into one instruction (load.apply). The code for the procedure itself ends with a return instruction and a retract instruction.
return.ib,r,s,p,pr,v	Return from a procedure. Move the result to the old stack top. Calculate the return address and reset the PC. Use the old frame to set up PSP, SP and LN. Remember to reset PSF and SF.

store.closure(n,m — Place the procedure closure on the stack. A closure is made up of the code vector address and the static link. If the address n is not the top of the stack, it is a forward declared procedure and n is the stack address. Add the value of PC to m to get the real address of the procedure. Then load m and SF on to the stack at the appropriate position.

mst.local(n),
mst.global(n) — Load the procedure closure at offset n in local or global frame on the stack, fill in the DL and PSL, and leave space for the rest of the MSCW.

mst.load(r,n) — As above but chain down static link r times then use n as an offset to find the MSCW.

mst.external(n) — This is for write real. The real number and the field width are on the main stack. Set up a call for the S-algol write real by placing the MSCW on the stack below these parameters remembering to duplicate the file descriptor on the pointer stack.

local.apply(n),
global.apply(n) — Load the MSCW from index n in the local or global frame and call the procedure. There are no parameters.

load.apply(r,n) — As above but chain down the static chain r times then use n as an offset to find the MSCW.

mst.st.func(n),
load.st.func(n),
apply.st.func(n) — These are the equivalent of the above but are used to find the standard functions which are addressed from sb but in the opposite direction on the stack from everything else.

forward.op — Leave space for the procedure closure on the stack.

retract.ib(n,m),r,s,p,v — Retract the main stack to n and the pointer stack to m. If the retract is not void move the value at the top of the appropriate stack to the new stack top.

Stack Load Instructions

To ease the problems of garbage collection the abstract machine uses two stacks. The main stack contains space for integers, reals and booleans while the pointer stack contains space for all the pointer objects. The pointer stack is used as the base for marking the heap. Objects on the pointer stack may be strings, vectors, pntrs, and files. Reals take two stack elements each on the main stack as do procedures (closures) on the pointer stack. A closure is made up of the code vector address and the static link. These instructions are used to load any data item that is in scope, on to the top of the stack. The data items may be in the local, global, standard or intermediate environments and a separate instruction exists for each form. Different instructions are also used for the separate stacks. The local and global forms of the instruction have a parameter which is the displacement of the item from the stack frame base. The intermediate form of the instruction requires the

address of the static environment as well as the displacement. Only one form of each type is described.

local(n),global(n)	Load on the main stack
load(r,n)	Load on the main stack the item found by chaining down static link r times and then taking an offset n.
dlocal(n),dglobal(n)	Load double length item main stack
dload(r,n)	Load double length item main stack in the same manner as load.
plocal(n),pglobal(n)	Load on the pointer stack
pload(r,n)	Load on the pointer stack in the same manner as load.

Stack Assignment

For each of the stack load instructions there is an equivalent stack assignment instruction. These instructions take the top element of the stack and assign it to the address indicated in the instruction. These instructions are: local.ass, plocal.ass, dlocal.ass, global.ass, pglobal.ass, dglobal.ass, load.ass, pload.ass, dload.ass.

Relational Operations

The relational operations act on the data types int, real and string. The top two elements of the stack are compared and removed. The boolean result true or false is left on the main stack. Care should again be taken in the comparison of strings which means element by element comparison. Equality is defined on all the data objects in the language. There is a separate form of the instruction for each type.

ge.i,r,s	Greater than or equal to
gt.i,r,s	Greater than
le.i,r,s	Less than or equal
lt.i,r,s	Less than
eq.ib,r,s,p	Equal to
neq.ib,r,s,p	Not equal to

Arithmetic Operators

These instructions operate on the data types real and integer. The top two elements of the stack are replaced by the result except for negate and float1 which use only the top element and float2 which uses the second top element.

plus,fplus	Add
times,ftimes	Multiply
minus,fminus	Subtract
fdivide	Divide real
div	Divide int leaving quotient
rem	Divide int leaving remainder
neg,fneg	Negate
float1	Coerce the int to a real on top of the stack
float2	Coerce the int to a real second top stack element

Vector and Structure Creation Instructions

These instructions take information off the stack and create heap objects. These objects are then initialised and the pointer to them left on the top of the pointer stack.

make.vec.ib,r,s,p(m,n)	m points to the position of the lower bound on the main stack. The difference between SP and n or SP and m depending on which stack is in use, gives the number of vector elements. The instruction creates a vector and fills in the elements. The stack pointers are then reduced to m and n with the pointer to the vector being placed on the pointer stack
iliffe.op.ib,r,s,p(n)	n pairs of bounds are on the main stack. However, the top of one of the stacks will contain the initial value. The instruction creates an iliffe vector of the shape indicated by the bound pairs and the value of the initial expression is copied into the elements of the last dimension. The expression value and the bound pairs are removed from the stack and the pointer to the vector is placed on the pointer stack
form.structure(m,n)	The expressions which initialise the structure fields have been evaluated on the appropriates stacks. This includes the trade mark which is an integer. m is the total size of the object on the heap and n is the number of pointers. After moving the fields from the stacks the pointer to the structure is placed on the pointer stack

Vector and Structure Accessing Instructions

These instructions are generated by the compiler to index a vector or a structure. The index of the vector must be checked against the bounds before the indexing is done. Similarly the structure class of a structure must be checked.

subv.ib,r,s,p	The vector index is on the top of the main stack and the vector pointer on the pointer stack. These are used to check that the index is legal and then to find the required value. They are removed from the stack and replaced by the value.
subs.ib,r,s,p	The class identifier and the structure pointer are on the top of the pointer stack. The field address is on the main stack. The class identifier is checked against the structure class identifier and if it is the same the field address is added to the pointer to yield the absolute field address. The class identifier, field address and the structure pointer are replaced on the stack by the result.
subvass.ib,r,s,p	This assigns a value to a vector element. The value is on the top of the stack and the address is calculated as in subv.
subsass.ib,r,s,p	This assigns a value to a structure field. The value is on the top of the stack and the address is calculated as in subs.
lwb	Remove the pointer to the vector from the pointer stack and place its lower bound on the main stack
upb	Remove the pointer to the vector from the pointer stack and place its upper bound on the main stack
is.op	The class identifier is found by dereferencing the structure pointed at by the top element of the pointer stack and is compared with top element of the main stack. Remove both and place the boolean result of the comparison on the main stack
isnt.op	This is the same as is.op except it has the opposite test
load.trademark(m,n)	This loads the trade mark on to the main stack. The trade mark is an index into the structure table. m is an index into the structure table of the procedures compilation unit. n is the relative address of a word containing the start position of the structure table in the real structure table.

Load Literal Instructions

These are used to load the value of a literal on to the stack. The literal usually follows the instruction in the code stream and so the CP register has to be updated accordingly.

LL.nil.pntr	Load the pntr value nil
LL.nil.string	Load the empty string on to the pointer stack
LL.file	Load the nullfile on to the pointer stack
LL.bool(n)	Load the boolean value n (true or false) on to the main stack
LL.string(n)	Load the string address on to the pointer stack n is the number of the string in the string vector of the current procedure.
LL.lint	Load a long integer, 32 bits, on to the main stack
LL.sint	Load a short integer, 8 bits, on to the main stack
LL.char(n)	Load the character n as a string of length 1.

String Operations

These are used to perform the string operations in S-algol.

concat.op	Remove the two strings from the top of the pointer stack and replace them with a new string which is the concatenation of them
substr.op	A new string is created from the one at the top of the pointer stack and replaces it. It is formed by using the length at the top of the main stack and the starting position at the second top. After checking that these are legal they are removed

Input and Output

read.op(n) The stream descriptor is on the top of the pointer stack. This is removed and the value read is placed on the appropriate stack. n indicates which read function to use. They are:

read	Read a character and form it into a string
reads	Read a string
readi	Read an integer
readr	Read a real
readb	Read a boolean
read.name	Read a S-algol identifier and form it into a string
peek	Same as read but do not advance the input stream
read.byte	Read an 8 bit byte and return it as an integer
read.a.line	Read from the current input position to the first newline character. Return a string of the characters excluding the newline
eof	Test for end of file on the input stream.
write.op(n)	he field width is on the top of the main stack and the item to be written out either under it or on the pointer stack. The stream descriptor is under all this on the pointer stack. The field width and the item are removed from the stack. In the case of out.byte the file descriptor is also removed from the stack. n indicates which function. They are:

write.int	Write an integer
write.bool	Write a boolean
write.string	Write a string
out.byte	Write an 8 bit byte.

If the field width is not specified then i.w and r.w come into use for int and real. s.w spaces are always written after integers or reals for character streams.

Miscellaneous

rev.ms,rev.ps	Swap the top two elements of the stack
erase.ib,r,s,p	Remove an element from the stack
finish.op	Stop the program execution
abort.op	Stop the program execution
not.op	Perform a not on the boolean at the top of the stack
no.op	Do nothing - used for padding in some implementations
newline(n)	Set the line indicator to n.

5 Loading and initialisation Code

Initialisation

The initialisation sequence for the S-algol interpreter may be split into five sections. These are listed below.

- i reate single character strings
- ii nitialise the stacks
- iii initialise the heap
- iv read in the user code
- v read in the real I/O code (if appropriate)

Note the order in which the interpreter executes these sections is up to the implementor. The order given about reflects that of the interpreter written at St. Andrews. Each section will now be explained in more detail.

Single character strings

Purely for efficiency considerations - reducing heap traffic; speeding up single string comparisons, it has been found advantageous to provide the run time system with a set of single character strings within the heap. This means that S-algol I/O routines such as **read** or **peek** need not go through the storage allocation process. Instead the address of one of these single character strings is calculated (normally one or two machine instructions). This makes execution faster and also helps reduce heap fragmentation. Notice that substr.op and concat.op should make use of this facility.

Initialisation of the stacks

There are two stacks, the pointer stack and the main stack. Space must be found for both of these. Both stacks grow down store. Space must be found for the standard functions. These are indexed from the stack base (SB) in the opposite direction from everything else on the main stack. This alows new standard functions to be added to the machine without

the need to recompile old programs. Space must be made for the first stack frame and then the standard identifiers must be placed on the stack. The standard identifiers are i.w, s.w, r.w, epsilon, pi,maxint and maxreal. The last thing to do is set up the stack base and stack fronts for the two stacks. Be careful with the main stack. The stack base points between the last standard function and the mark stack control word which is on the bottom of the stack.

Heap initialisation

The heap provides the interpreter with dynamic storage for the S-algol data structures. As this space may be re-used (garbage collection) a free space list is kept. This consists of a linked list with a size field at each node indicating the number of bytes available. By keeping the free space list in the heap itself it is possible to use the link address to indicate the starting address of the free space area. To initialise the heap it is simply neccessary to obtain a section of contiguous store (operating system dependent) and create a free space list of one link, the size field of which is the total heap size. To facilitate re-creation of the free space list after garbage collection it has been found useful to install a dummy link at the start of the free space list with a size field of zero. Having initialised the heap it is now possible to use the interpreter's storage allocation package to aid the remainder of the initialisation sequence.

User code

An S-algol code file consists of a code vector optionally followed by a structure table and then by padding characters followed by 24 bytes of information. The number of padding characters ensures that the total size in bytes of the code file is a multiple of 128.

The 24 bytes of information at the end of the code file consist of the following: the first four bytes indicate the total code size in bytes not including the padding bytes or information bytes; the second four bytes are an offset from the beginning of the file indicating the start of the main program segment; the next word (4 bytes) indicate the size of the code vector; the next two bytes are the external flag used by the binder; next follows two bytes containing the line count, this is used by the flow summary option; the next two bytes are the version number of the compiler which created it; next follow four bytes holding the overall size of the block; finally come two byes containing the block number which is used by the binder.

Having read in the code, checked the version number, it is only required to remember the starting address for the user code file initialisation to be complete.

Real I/O code

The reading in of the real I/O code is similar to that for the user code as is its initialisation. The real I/O code is written in S-algol and is loaded into the interpreter in a specific place so that the closures for the I/O functions may be found when called.

Standard functions

It is important that the interpreter provides the same environment for the user program as was assumed by the compiler at compile time. This is the implementor's problem. The ordering of these names is provided to the compiler by a standard declaration file.

The standard functions fall into two groups, those written in S-algol and those `hard wired' into the interpreter. The real I/O functions such as readr and writer are written in S-algol along with eformat,fformat and gformat. All the others are coded in the interpreter.

All the standard function closures are indexed from the bottom of the main stack. They are indexed from the stack base in the opposite direction from everything else on the stack. The closures of the functions written in S-algol must be loaded on the stack in the appropriate position. These can then be called in the normal fashion when the program starts to execute. Standard functions which are wired into the interpreter can be distinguished by having their static link set to zero. This can be trapped and the appropriate routine in the interpreter called.

6 Marking for garbage collection

Once the free space list is unable to satisfy a given request, it is neccessary to initiate a garbage collection sequence which re-creates the free space list. In order to distinguish garbage from wanted items all those items which are wanted need to be marked as such. This is where the marker comes in. The marker is a routine which recursively marks any heap items which are reachable from any pointer on the pointer stack. There are basically four distinct types of heap item, these being:

- i strings,files
- ii structures
- iii vectors of pointers
- iv vectors of non-pointers

There now follows a discussion of the intricacies of marking these types of heap item

Strings and Files

Strings and files contain no pointers to other heap objects, so the marker need go no further.

Structures

An instance of a structure contains a count of the number of pointer fields within its body. Any pointer fields occur at the beginning of the structure. This rearrangement of the structure fields is handled automatically by the compiler and is transparent to the user.

When marking a structure therefore, it is simply a case of iterating through the number of pointer fields.

Vectors of pointers

This category consists of vectors of pointers,vectors of strings and vectors of files. The number of pointer elements is calculated from the vector bounds held in words two and three of the vector header. The elements follow the bounds.

Vectors of non-pointers

This category consists of vectors of integers, booleans and reals. As with strings all that needs to be done with these is to mark them.

Appendix I
The Compiler Output Format

At the end of each program the compiler will pad the output code to a 128 byte boundary less the twenty four bytes of information. That is the compiler outputs the code vector followed by some padding characters. This is followed by the following twenty four bytes of information:

code size and structure table size in bytes (4 bytes)
start address in the code (4 bytes)
structure table address in the code (4 bytes)
realio flags (2 bytes)
line count (2 bytes)
version number (2 bytes)
overall size of whole block in bytes (4 bytes)
block number (2 bytes)

Appendix II
S-code Generated by the (8 bit) S-algol Compiler

A summary of the S-code generated by the S-algol compiler for each syntactic construct is given here. In the description E, in the source code represents an expression and E, in the code represents the S-code for that expression. Sometimes the expressions are of type void. A description of the instructions themselves is given in Appendix I.

Source — **S-code**

~E — E not.op

+E — E

-E — E neg.op

unary.function(E) — E unary.function.op

write E1:E1',.....En:En' — s.o E1 E1' write.op.......
En En' write.op erase.op

Write operates for reals, ints, bools and strings.

output E0,E1:E1'....En:En' — E0 E1 E1' write.op..........
En En' write.op erase.op

out.byte E0,E1,E2 — E0 E1 E2 write.op

read — s.i read.op

read(E) — E read.op

similarly for peek, read.name, reads, readi, readb, eof and read.byte.

E1(E2) := E3 — E1 E2 E3 subvass or subsass

E1 **or** E2 — E1 jumptt(l) E2 l:

E1 **and** E2 — E1 jumpff(l) E2 l:

E1 <binary.op> E2 — E1 E2 binary.op

(E) — E

E1(E2|E3) — E1 E2 E3 substr.op

E1(E2) — E1 E2 subs or subv

E(E1,.....En) — mst.load E E1....En apply.op

@E **of** T[E1,....En] — E E1.......En make.vector

E(E1,.....En) — E E1.......En formvec.op

E ? — E finish.op

abort — abort.op

vector E1::E1'..
,..En::En' **of** E — E1 E1'...En En' E iliffe.op

if E1 **do** E2 — E1 jumpf(l) E2 l:

if E1 **then** E2 **else** E3 E1 jumpf(l) E2 jump(m) l: E3 m:
repeat E1 **while** E2 l: E1 E2 bjumpt(l)
repeat E1 **while** E2 **do** E3 l: E1 E2 jumpf(m) E3 jump(l) m:
while E1 **do** E2 l: E1 jumpf(m) E2 jump(l) m:
for I=E1 **to** E2 **by** E3 **do** E4 E1 E2 E3 l: fortest.op(m) E4
Forstep.op(l) m:
let I = E E
let I := E E
procedure ; E E return
structure load.trademark
<literal> ll.literal dependent on type
<identifier> load.stack
<identifier> := E E load.stack.assign

A load.stack instruction may be one ofload, local, global, pload, plocal, pglobal, dload, dlocal or dglobal. There is an equivalent instruction for each assignment.

The unary functions are upb, lwb, float.

The binary operations are eq.op, neq.op, lt.op, le.op, gt.op, ge.op, plus.op, times.op, minus.op, div.op, rem.op, divide.op, is.op, isnt.op and concat.op.

case E0 **of** E0
E11,E12,...E1n : E10 E11 cjump(l1)
E12 cjump(l1) ...
E1n cjump(l1)
jump(M1)
l1 : E10 jump(xit)
E21,E22,...E2n : E20 M1:E21........

default : Ek+1 0 Mk:Ek+1 0
xit:

Appendix III
S-algol (8 bit) Abstract Machine Operation Codes

Note

The operation codes are held as 8 bit quantities. If the instruction requires an operand then the operand will follow in the code. Therefore most of these instructions have a long and a short form. If there are two forms the operation codes usually differ by 128. The numbers in brackets following a command name denotes the number of bytes which the operand(s) take up. Typed instructions have an encoded name with the following convention.

ib	integer or boolean
r	real
s	string
p	pointer
v	void

Jump Instructions

The code address in the instruction is relative to the code pointer. That is the address of the instruction following the jump in the code stream.

0 no.op	128 fjump(2)
1 bjump(1)	129 bjump(2)
2	130 jumpf(2)
3	131 jumpff(2)
4	132 jumptt(2)
5	133 for.test(2)
6 for.step(1)	134 for.step(2)
7 cjump.ib(2)	135 cjump.r(2)
8 cjump.s(2)	136 cjump.p(2)
9	137
10 bjumpt(1)	138 bjumpt(2)
11	139

Stack accessing instructions

The address is the stack offset from SB or PSB. The instructions starting with the letter p refer to the pointer stack.

12 local(1)	140 local(2)
13 plocal(1)	141 plocal(2)
14 dlocal(1)	142 dlocal(2)
15 mst.local(1)	143 mst.local(2)

16 global(1)   144 global(2)
17 pglobal(1)   145 pglobal(2)
18 dglobal(1)   146 dglobal(2)
19 mst.global(1)   147 mst.global(2)
20   148
21   149
22   150
23   151
24 load(1,1)   152 load(1,2)
25 pload(1,1)   153 pload(1,2)
26 dload(1,1)   154 dload(1,2)
27 mst.load(1,1)   155 mst.load(1,2)
28 local.ass(1)   156 local.ass(2)
29 plocal.ass(1)   157 plocal.ass(2)
30 dlocal.ass(1)   158 dlocal.ass(2)
31   59
32 global.ass(1)   160 global.ass(2)
33 pglobal.ass(1)   161 pglobal.ass(2)
34 dglobal.ass(1)   162 dglobal.ass(2)
35   163
36   164
37   165
38   166
39   167
40 load.ass(1,1)   168 load.ass(1,2)
41 pload.ass(1,1)   169 pload.ass(1,2)
42 dload.ass(1,1)   170 dload.ass(1,2)
43   171

The load instructions have as a first operand the reverse lexical level and as the second operand the stack address.

Procedure and block entry and exit

44 apply.op(1)   172 apply.op(2)
45 store.closure(2,4)   173
46   174
47 return.ib   175 return.r
48 return.s   176 return.p

49	177 return.v
50	178

Miscallaneous instructions

51 global.apply(1)	179 global.apply(2)
52 local.apply(1)	180 local.apply(2)
53 retract.ib(1,1)	181 retract.ib(2,2)
53 retract.r(1,1)	82 retract.r(2,2)
55 retract.s(1,1)	183 retract.s(2,2)
56 retract.p(1,1)	184 retract.p(2,2)
57	185
58 retract.v(1,1)	186 retract.v(2,2)
59 load.standard.fn(1)	187 mst.standard.fn(1)
60 apply.standard.fn(1)	188
61 mst.external(1)	189
62 forward	190
63 load.apply(1,1)	191 load.apply(1,2)

Structure and Vector instructions

64 orm.structure(2)	192
65 is.op	193 isnt.op
66 subs.ib	194 subs.r
67 subs.s	195 subs.p
68	196
69 subsass.ib	197 subsass.r
70 subsass.s	198 subsass.p
71	199
72 makev.ib(2)	200 makev.r(2)
73 makev.s(2)	201 makev.p(2)
74	202
75 iliffe.ib(2)	203 iliffe.r(2)
76 iliffe.s(2)	204 iliffe.p(2)
77	205
78 subv.ib	206 subv.r
79 subv.s	207 subv.p
80	208
81 subvass.ib	209 subvass.r
82 subvass.s	210 subvass.p

83	211
84 upb.op	212 lwb.op
85 concat.op	213 substr.op
86 load.tm(1,4)	214 load.tm(2,4)

Load literals

90 ll.int(1)	218 ll.int(4)
91 ll.bool(1)	219 ll.real(8)
92 ll.string(1)	220
93 ll.char(1)	221 ll.file
94 ll.nil.string	222 ll.nil.pntr
95	223

Comparison instructions

96 eq.ib	224 eq.r
97 eq.s	225 eq.p
98	226
99 neq.ib	227 neq.r
100 neq.s	228 neq.p
101	229
102 lt.i	230 lt.r
103 lt.s	231
104 le.i	232 le.r
105 le.s	233
106 gt.i	234 gt.r
107 gt.s	235
108 ge.i	236 ge.r
109 ge.s	237

Arithmetic Instructions

110 plus	238 times
111 minus	239 div
112 rem	240 neg
113 fplus	241 ftimes
114 fminus	242 fdivide
115 not.op	243 fneg
116 float1	244 float2

Stack manipulation, termination and I/O instructions

120 erase.ib	248 erase.r
121 erase.s	249 erase.p
122	250
123 rev.ms	251 rev.ps
124 newline(1)	252 newline(2)
125 finish.op	253 abort.op

* read.op uses the second 8 bits for the following functions

0 readi	1 reads
2 readb	3 read.byte
4 peek	5 read
6 eof	7 read.name
8 read.a.line	9 readr

** write.op uses the second 8 bits for the following functions

0 write.i	1 write.s
2 write.b	3 out.byte
4 write.r	