The research reported in this paper was sponsored in part by the Advanced Research Projects Agency Information Processing Techniques Office and was monitored by the Electronic Systems Division, Air Force Systems Command under contract F1962867C0004, Information Processing Techniques, with the System Development Corporation

TECH MEMO *a working paper*

System Development Corporation / 2500 Colorado Avenue / Santa Monica, California 90406

Information International Inc. / 11161 Pico Boulevard / Los Angeles, California 90064

TM- 3417/400/00

AUTHOR Dane Crandell
D. Crandell

TECHNICAL Jeff Barnett
J. Barnett

RELEASE Clark Weissman
C. Weissman, S.D.C.
D. Anschultz, I.S.I.

for J. I. Schwartz

DATE 4/26/67 PAGE 1 OF 24 PAGES

(Page 2 is blank)

LISP 2 Assembly Program (LAP) Specification

ABSTRACT

This document specifies the functions performed by the LISP 2 Assembly Language (AL) and processor for the IBM System 360. Included are a description of LAP address field types, LAP pseudo-instructions, and the LAP assembly processor. Syntax equations for most LAP language forms are included in an appendix.

(Page 4 Blank)

TABLE OF CONTENTS

1. INTRODUCTION

The LISP 2 Assembly Program (LAP) generates a core image (a list of octals paired with relative locations) from a list of symbolic instructions and labels. LAP also allocates storage for variables on the pushdown stack, and insures that references to fluid and own variables are consistent among different compiled functions.

2. TOP OF THE LANGUAGE

A lap-definition has the following format:

lap definition = (LAP listing ref-list)

where

listing = ({FUNCTION|MACRO} (f-name value-type) par-list item$_*$)

item = S/360-inst|pseudo-inst|label

A lap-definition consists of two parts: a ref-list, whose format is as yet undefined and a listing. The ref-list supplies a complete list of global variables referenced freely from listing; it also supplies the necessary amount of declaration to do consistency checks of type, etc. The listing is the symbolic assembly language, AL, for either a function or macro. The name (f-name) and the type of datum produced (value-type) are specified. The list of items includes machine instructions, pseudo-instructions, and labels--the normal constituents of a symbolic assembly language.

3. LAP INSTRUCTION FIELDS

3.1 ENTRY

Syntax: (ENTRY identifier displacer)

This is used to reference an identifier defined by a previous ENTRY pseudo-instruction in this or a previous program. Provision is made for a displacement relative to this identifier by means of "displacer." This address usage will result in assembling a "load index register" instruction with the address of the entry identifier prior to assembling the given instruction.

3.2 QUOTE

Syntax: (QUOTE S-expression)

The normal case of a quote field causes the following: a quote cell is generated in quote structure space; a load register {1|12} via the quantized core map (adcons) is inserted in front of the current instruction; the address field concerned is assembled as ({1|12} displacement).

In all these adcon-demanding instructions, the assembler checks the current status of registers 1 and 12. If the required reference is in the same quantum, no load is necessary. The remembered contents of these registers is lost when a label or a function call is encountered.

3.3 UNIQUE

Syntax: (UNIQUE S-expression)

UNIQUE references have much in common with quotes. The outstanding difference is that the structure is shared and quote cells are not.

3.4 OWN VARIABLE

Syntax: own-variable

Once again, register 1 or 12 is used as a base register. The address reference created refers to the own binding cell for that own-variable. The own structure count is incremented. The own-variable must be dotted with the name of the section in which it was declared.

3.5 FLUID VARIABLE

Syntax: fluid-variable

The adcon required by the address reference to the fluid binding cell is very likely given by register 14. Register 14 always contains the base for the initial non-locative fluid structure space. Locative fluid references and those non-locatives that could not fit in the initial space will require an additional adcon in register {1|12}. The fluid-variable must be dotted with the name of the section in which it was declared.

3.6 LEXICAL VARIABLE

Syntax: lexical-variable

Lexical variables are dotted with LEX. and represent absolute or pointer stack references. The actual machine address produced will be a positive displacement modified by either base register 10 or 13. The absolute and pointer stacks occupy a single space and grow toward each other. Registers 10 and 13 represent pointers to their respective stacks but have negative offsets included so that references forward and backward from the stack boundaries can be accomplished with positive displacements. These offsets are $-\alpha$ for the pointer stack and $-\beta$ for the absolute stack. At exhaustion of stack space, a certain amount of "breathing space", δ, must still be present for the use of the garbage collector. Figure 1 is typical of the stack at exhaustion. In the figure, G is the function being called. Note also that registers 10 and 13 have the same value at this point.

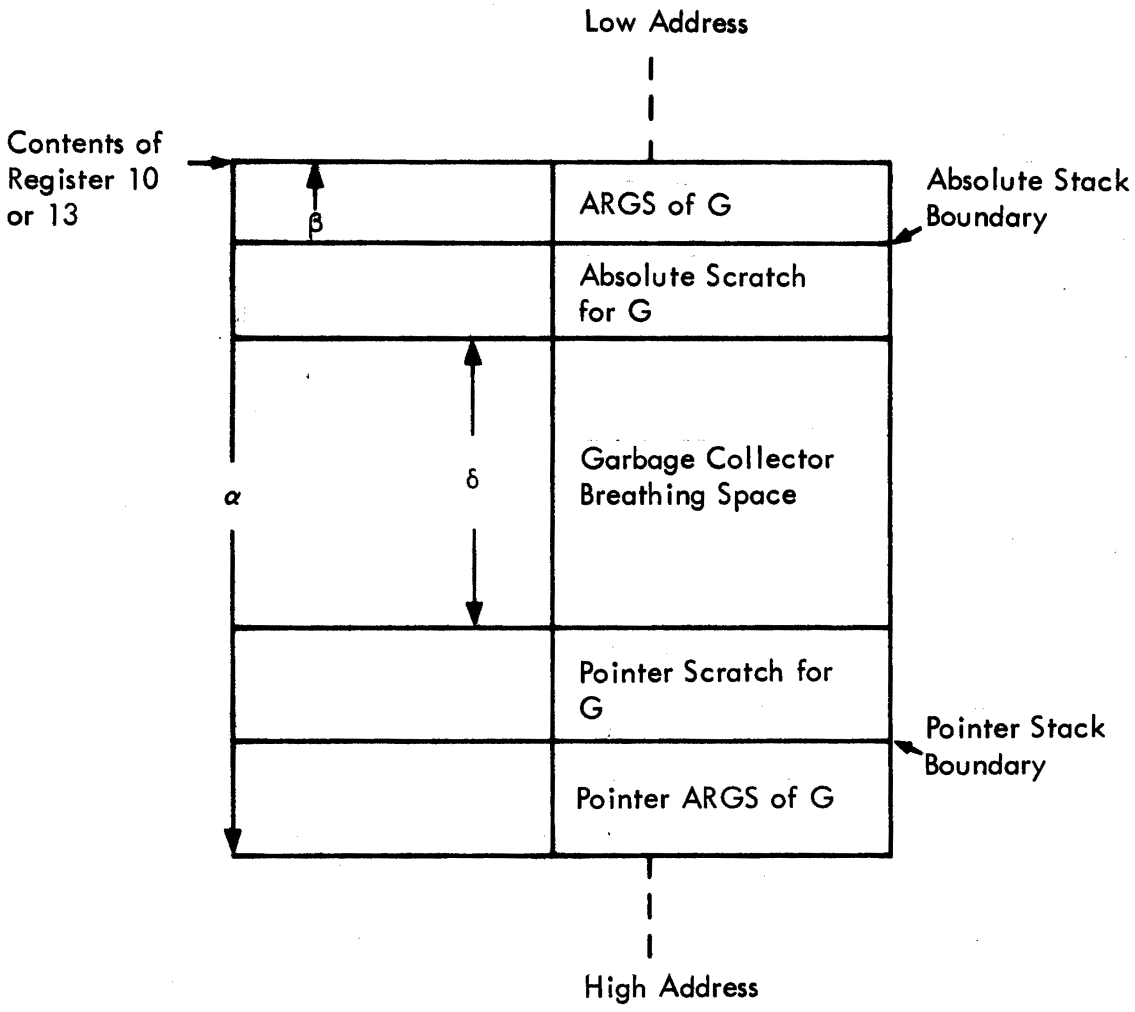


Figure 1. Assembly of Lexical Variable Address Fields

3.7 PUSH

Syntax: (PUSH {{A|P} [count]|AD})

This address causes the same action as does the PUSH pseudo-instruction; it also supplies the base register and displacement to the given instruction.

3.8 POP

Syntax: (POP {A|P} [count])

This address also acts the same as its pseudo-instruction counterpart, and supplies the base register and displacement to the given instruction.

3.9 LABEL

Syntax: (LABEL label)

Normally, label reference fields for the S/360 simply assemble as a positive displacement. Label references may occur outside the code section in which the label is defined. In such cases, label references will refer to a link block consisting of a "load register 11 and branch". This unfortunately complicates the assembler, but should not occur often.

3.10 LITERAL

Syntax: literal

Literal reference fields cause an address reference to a quantity in the literal pool. Parts of this literal pool may be in fixed program space. Stores into literals are prohibited. Normally, a literal will be generated in the constants region of the current BPI.

3.11 REGISTER DISPLACER

Syntax: (reg displacer)

This address form is a means of explicitly stating the register and displacement to be assembled into the given instruction.

3.12 REGISTER

Syntax: reg

A reg denotes the register to use as a base or index register in a 360 instruction.

4. LAP PSEUDO-INSTRUCTIONS

4.1 ENTRY

Syntax: (ENTRY identifier displacer)

The ENTRY pseudo-instruction is used to generate an adcon in the entry pool that may be referred to in subsequent assemblies by referring to the identifier. This pseudo-instruction is intended for system-making purposes, and will cause the program in which it occurs to be assembled in fixed program space.

4.2 BEGIN

Syntax: (BEGIN)

This pseudo-instruction causes fluid binding of the input parameters that are fluid.

4.3 ARGUMENT

Syntax: (ARG)

The ARG pseudo-instruction is used to mark the beginning of the preparation of the arguments for the next FUNCTION call. It causes the assembler to note the status of the stacks so that this status can be restored after the call.

4.4 CALL

Syntax: (CALL function-name map)

This causes the generation of a calling sequence for the function given by function-name. The entity "map" reflects the status of the general registers. The right bit of map will be 0 if register 0 is absolute and 1 if register 0 is a pointer quantity. Successive bits to the left in map describe successively lower-numbered registers. The map will be put in the calling sequence along with the amounts to increment the absolute and pointer stack during the call. After the call is generated, LAP returns the stacks to the status at the corresponding ARG pseudo-instruction. This has the effect of doing a POP pseudo-operation for each stack argument of this call.

4.5 PUSH

Syntax: (PUSH {A|AD|P} [count])

PUSH causes LAP to reserve a space on either the absolute or pointer stack as given by A or P. AD is used to indicate double-length absolute pushes. If count is not specified, then one such entry is made, otherwise count specifies the number of entries.

4.6 POP

Syntax: (POP {A|P} [count])

The POP pseudo-instruction is used to de-allocate stack space previously allocated by a PUSH. A or P indicates whether the absolute or pointer stack is affected. The optional count field can be used to indicate the number of stack entries to remove. An empty count field implies a count of 1. Stack entries are removed on a last-in, first-out basis.

4.7 BLOCK

Syntax: (BLOCK $item_*$ declare $item_*$)

LAP is a block-structured assembler. This affects both label referencing and variable bindings. Labels defined within the block are invisible outside; labels defined outside are visible inside, however. If several labels are defined with the same name at different block levels, the innermost visible label will satisfy the reference. (If they are defined at the same block level, there is an error.) The items encountered before the declaration are a series of computations ending with PUSHes. The declaration gives variable names to the entities left on the stacks. Any variables declared fluid are fluid-bound at the declaration. After execution of the last item in the block, fluid-variables bound in the block are restored and the lexical variables on the stack are POPped automatically. An end pseudo-op is usually the last item in a block.

4.8 RETURN

Syntax: (RETURN)

RETURN generates the code to reload the safe-over-function-call registers that were not arguments, and the system registers. Control is returned to the calling program.

4.9 END

Syntax: (END)

END signals the end of a block and revokes all variable bindings and label definitions of the block. The fluid variables are restored. If END does not occur in the context of a block, its function is to restore the fluid variables, and mark the end of the function.

4.10 DECLARE

Syntax: (DECLARE bound-declaration$_*$)

When this pseudo-instruction is encountered, the list of bound-declarations is reversed; each bound declaration is taken in order and paired with a pushdown constituent. Fluid binding is done for the fluid variables declared. See Section 4.7.

4.11 SYMBOL

Syntax: (SYMBOL (identifier . integer)$_*$)

This pseudo-instruction can be used to define register names or other symbolic equivalents. This definition must occur earlier in the code than the use of the given symbol. This works as EQU does in most other assemblers.

4.12 ORG

Syntax: (ORG [integer])

This pseudo-instruction causes the program to be assembled in fixed space. The integer indicates the origin of the program in fixed space. If the integer is not specified, some previously specified value is taken as the default value.

4.13 CASEGO

Syntax: (CASEGO reg label$_{*+1}$)

The CASEGO pseudo-instruction performs an operation similar to the CASE operation in IL. An integer value, n, is assumed to be in the general register specified by reg. Computer instructions are generated to transfer control to the instruction with the nth label of the given label list. If n is less than 1 or greater than the given number of labels, control is transferred to the last of the given labels.

4.14 ALIGN

Syntax: (ALIGN a [b])

This pseudo-instruction causes alignment of the instruction counter. The parameter "a" specifies the desired alignment as follows:

a	alignment
4	full word
8	double word

Alignment is effected by adding half-word NOP instructions to the instruction stream. Half-word alignment of instructions should be maintained by the user at all times. The optional parameter b is used to specify the number of bytes of contiguous coding that will follow this pseudo-instruction. "Contiguous" means that the coding cannot be put on two different pages, e.g., an in-line calling sequence.

5. LISP ASSEMBLY PROCESSOR

LAP 2 for the 360 is influenced by two major constraints: the block structuring of its input language (AL), and the paging orientation of the 360's addressing scheme. This most significantly affects the label processing, and to a lesser degree, the method of handling literals.

The LAP assembler has two versions that essentially differ only in what they do with the assembled code. LAP either places code to be run in the memory of the machine on which it is operating (normal running mode), or it outputs a form of the code onto secondary storage to be subsequently loaded onto another system (core image generation mode).

LAP's initial output is basically a list of items that are to be stored, paired with the relative location in which they are to be stored. The post-processor which plants this list in core or outputs it onto secondary storage will not be considered further in this document. All other areas are treated as though a planting in core (normal running mode) was planned for the initial output of LAP.

5.1 CALLING FUNCTION LAP

LAP is a function which is called with one argument. This argument is the definition of a function (or a macro) in assembly language.

LAP's argument, (LAP listing ref-list) consists of the ref-list (or declaration) list, which is a list of the names of all variables used "free" in this function,* together with pertinent declaration information, and the listing--a list of the assembly language (AL) items output by the compiler, which makes up the main body of the function definition. The ref-list is processed first.

The ref-list is processed by applying the primitive MAKEFREE to each of the variables in turn. MAKEFREE creates a variable structure (with the appropriate class of structure and value type) for each of the variables for which a variable structure does not then exist, and checks for consistency those variable structures that already exist. In all cases, it returns as its value the location of that structure.

---

*"Function" is being used to indicate function, macro, etc.

The listing ({FUNCTION|MACRO} (f-name value-type) par-list item$_*$) is then processed.

The following list is passed, as an argument, to MAKEFREE:

- f-name, i.e., the name of the function, or macro, etc.
- usage, i.e., the type of structure being defined: FUNCTION or MACRO
- value-type, i.e., the type of value the function will return
- par-list

MAKEFREE creates the function descriptor and returns as its value the location of that descriptor.

The par-list only gives information to LAP. Variable names on the par-list are associated with preceding locations on the stacks, starting with current locations of the stack pointers and going back to successively earlier locations in the stacks.

In the proposed LAP/360 implementation, there are actually two stacks, one for absolute quantities; a second for pointer quantities and recording the contents of the registers prior to a function call.

This, then, leaves only the items unprocessed. The items of the listing make up the main body of the function definition, and their processing mechanism makes up the main body of LAP--a function called RAP (Recursive Assembly Processor).

5.2 RAP OPERATION

RAP operates on its input, the free variable Listing, which is a list of items. It outputs additions to the free variable Corim, which is a list of assembled items, each paired with the location number of the site at which it is to be planted.

By the time RAP operates on Listing, f-name, usage, value-type, and par-list have been processed and all that remains is the main body of the function definition, items.

Therefore,

$$\text{listing} = (\text{item}_*)$$

and

$$\text{item} = \text{S/360-inst} \mid \text{pseudo-inst} \mid \text{label}$$

An item may be an S/360-inst (an assembly language S/360 instruction which is assembled and output), a pseudo-instruction (which operates on LAP's internal lists and variables, and may produce an assembled output), or a label (which specifies a point in the code structure, causes LAP to record this point so that it may be referenced by other code, and causes no assembled output).

5.2.1 Label Processing

A label is an identifier or an integer. Encountering a label produces no assembled code output on Corim, and serves only to define the location of a point in code for reference.

When a label is encountered, all of the code preceding it has been assembled and assigned a location, so that the location of the label is defined as the next location at which code will be planted.

Whenever a label is encountered, it is added to a list of labels and paired with its location (actually the location at which the code immediately following it is to be planted).

$$\text{Labels list} = ((\text{label} \ . \ \text{ilc})_*)$$

The ilc (instruction location counter) is the location at which the code will be planted.

Then the list of label references is searched (i.e., the locations of instructions in Corim that make reference to labels are searched).

$$\text{Lblref list} = ((\text{label} \ \ell_*)_*)$$

ℓ is the current location of the code element on the Corim list. The location includes the page on which the reference occurs.

If Lblref is found to contain references to this label, the code (whose address field had been left empty) is fixed up to refer to this label location, and the references are pruned from the Lblref.

The process is actually not quite this simple, due to the page-oriented addressing (one can address locations only from 0 to 1023 words forward of a base register). At run time there is always one register loaded with a value which is the address of either the beginning of the currently executing binary program (BP) or a point n x 1024 words down from this point. Thus the binary program register always points to the beginning of the current "page" of the BP (that is the beginning of the 1024-word segment of the binary program that is currently being executed).

This makes it possible to address any label on the current page directly using the binary program register (BPR) as a base and displacing up to 1024 words from this register.

To address (branch to) points off the current page, as might have to be done in large multi-page binary programs, it is necessary to load a base register with a value which is within 1024 words of the desired address. This is done by adding (or subtracting) a multiple of 1024 from the BPR, and then branching to the location specified by the BPR displaced by a quantity less than 1023 words.

In practice, then, each time a label is encountered, it is added to the Labels list, paired with its location--page number (relative to the beginning of the BP)--and its location (displacement down from the beginning of the page).

Labels = ((label (pg . ilc))$_*$)

Then Lblref is searched.

If the just-encountered label is found in this list, then all references to the label (the list of locations of references to the label is dotted with the label name in Lblref) are fixed up to point to the label. The label, and its associated structure, is then pruned from Lblref.

5.2.2 S/360 Instruction Processing

A second class of items that may occur in the listing consists of S/360 instructions. These instructions make up the main body of the BP definition. The syntactic structure of an S/360 instruction is:

```
S/360-inst = (op|
              op reg|
              op reg reg|
              op reg addr|
              op reg addr reg|
              op length addr length addr|
              op (immed) addr)
```

These correspond to the RR, SI, RS, RX, and SS instruction types of the S/360 machine instructions.

The first element of an S/360 instruction is always the op, which is a list consisting of an op symbol or number followed by zero or more op-modifier symbols or numbers.

The RAP program contains a standard list of op and op-modifier symbols (on the Symbtabel list), and this list may be expanded by any user by adding symbol definitions to Symblist (which is appended to Symbtabel) using the SYMBOL pseudo-instruction (discussed in the section on pseudo-instructions).

The op, then, is

$$op = (\{op\text{-}symbol \mid number\}\{op\text{-}modifier\text{-}symbol \mid number\}_*)$$

(See the Appendix for a set of standard op and op-modifier symbols.)

The op is processed by exclusive-OR'ing each of its subelements together. Numeric values are used as is, and symbolic values are looked up (first in Symbtabel, and then, if not found there, in Symblist). The numeric value defined in the list is used.

The syntactic structure of the address field is then analyzed and a constant, which indicates instruction class, is EOR'ed into the opcode.

Finally,the address fields themselves are assembled into the instruction. The complete instruction is then added to Corim and the next item on the listing is processed.

5.2.3 Address Field Processing

Address-field processing is handled by calling a function that determines the address type, creates any necessary side effects of that address, and returns as its value the assembled address field.

There are two functions of this type: REG and ADR. The first handles the assembling of register type address fields. The syntax of the field that is passed to reg as an argument is:

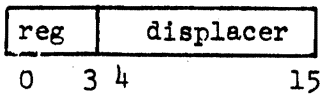
$$reg = integer \mid symbol$$

where integer is a number with a value from 0 to 15 that directly represents one of the 16 general registers of the S/360, and symbol is a previously defined symbol in Symblst whose value is an integer between 0 and 15. REG has no side effects and merely returns as its value the integer, or the integer value of the symbol.

The second function, ADR, is a bit more complex. ADR processes address fields of the "address, displacer" type. Its argument, addr, may be in one of several forms, defined by the equations in the appendix.

The first form of addr, (reg displacer), expressly indicates the base register that is to be used and the amount of displacement from this base register. If either is in symbolic form, it is replaced by its numeric value, found in Symblst. The two values, reg (= 0 to 15) and displacer (= 0 to 4095 bytes or 1024 words), are EOR'ed together (after reg has been shifted 12 bits left) to form the address, and this value is returned as the value of ADR.

addr =



The second type of addr is the literal,({2|4|8} integer). This is a reference to a literal value whose length is 2, 4, or 8 bytes in length (half, full, or double-word size). Literals (except those found in fixed program space) are placed at the end of the page of binary program that references them, so that they may be directly referenced using the BPR as the base register. Literals are shared on their page, so that if several instructions on the same page refer to literals whose values are the same, then only one literal is created.

The location (at the end of the page) where the literal will be placed is not known at the time it is encountered. Therefore the addr field of the instruction is left blank. The literal is added to the Litthspg list if it is not on the list, and ℓ (the location of this instruction in Corim) is associated with it. The instruction's addr field can be fixed up to point at the literal, when the literal is assigned a location on the page. The function returns the value of BPR with no displacer.

Lexical-variables are variables whose values are stored on one of the two pushdown stacks. Each of the stacks has an associated base register (the ASP and the PSP) whose location remains fixed during execution of a function. All lexical-variable addresses are displaced from these two registers. The variable base register and displacement are determined by reference to the lexical variable list. The function returns as its value ASP or PSP and a displacer.

Own-variable values are stored in own-variable space. Own-variable space has no reserved base register, so a base register must be loaded with a value within 1024 words of the referenced own variable. One of the scratch registers (registers 1 and 12) is used for this purpose.

The actual procedure is as follows. First, a function MAKEFREE is called, with (own-variable-name . section-name) as its argument; it returns with the location of that own variable. This location is separated into two parts: n x 1024 words + displacer, with displacer < 1024 words. A test is made to see if either of the LAP scratch registers contains the value n x 1024; if so, this register is used as the base register. If not, an instruction is created which loads one of the registers with the number n x 1024 words.

The number n x 1024 comes from a table in fixed program space that contains adcons (multiples of 1024 words). That is, the instruction created loads a LAP scratch register with the contents of a location specified by the FPS register displaced by an amount sufficient to reference the constant n x 1024 words. The instruction thus created is added to Corim and assigned a location just prior to that of the instruction currently being assembled. The function returns as its value the register number of the LAP scratch register loaded as a base register, and the displacer portion of the value returned by MAKEFREE.

Fluid variables are handled in a similar fashion, except that fluid-variable space does have a reserved base register which always contains the location of the beginning of fluid-variable space. This means that those fluid variables found within 1024 words of this base can be directly addressed, without first inserting a "load base register" instruction. The function returns the number of the fluid-base register, or a LAP scratch register, as appropriate, and a displacer.

Identifiers are handled similarly, except that a function MAKEID is called instead of MAKEFREE. Normally, the identifier will already exist, having been created at the time it was first encountered when read in, and MAKEID will just return its location. Identifier space has no reserved base register. A "load LAP scratch register" instruction usually must be inserted. The function returns the appropriate LAP scratch register and displacer as its value.

Quotes and uniques are created at assembly time. The function MAKEQUOTE or UNIQUOTE is called, with the S-expression as its argument. Quote space has no reserved base register. The base register and value returned are as above.

The two address fields, (PUSH ...) and (POP ...), allow the compiler to create instructions that add values to, and remove values from, the two pushdown stacks.

First some of the features of stack management will be described. At the time any function is called, the pushdown stack pointer (the ASP and PSP registers) are set to point at the first location beyond the last argument for the function being called. This is the leading edge of the stack at that time. (For the S/360 implementation, the registers actually contain a value which is lower than this by some constant offset, so that it is possible to refer to locations both before and beyond the pointer. The S/360 allows only positive displacements from a base register.) During the execution of a function, the stack pointers do not move, but the compiler may add values to the stack (PUSH ...), and then remove (POP ...) these values. The assembler keeps track of the current contents of the stack (Aslist and Pslist), the current leading edges (the virtual stack pointers Deltaasp and Deltapsp, which give the distance from the stack pointer to the leading edge of the stack), and the locations of the leading edges of the stacks just prior to the addition of the first argument of a function (Argspoint list). (This last list stores the condition that exists just before and just after the function call, as all arguments of a function are assumed to be gone from the stack just after the call to that function has returned.)

In instructions using the PUSH and POP address fields, A indicates the leading edge of the absolute stack is being referenced, and P indicates the leading edge of the pointer stack. Count indicates the number of items being pushed onto the stack in that instruction (this is used when the push is done with a "store multiple" instruction). If absent, the count is one.

The (PUSH ...) address field returns as its value the appropriate stack register (ASP or PSP), and the virtual pointer (Deltaasp or Deltapsp) as its displacer. That is, it returns the location of the leading edge of the stack. The virtual stack pointer for the referenced stack is then updated. In a simple push, it is moved one word beyond its previous location in an "absolute stack, double word push" (AD). It is advanced two words, and it is advanced "count" words when this item is present, indicating that this field is in a "store multiple" instruction. This is accomplished by updating the value of Deltaasp or Deltapsp. The appropriate stack list (Aslist or Pslist) is also updated.

The (POP ...) addr field creates a reference to a location on one of the push-down stacks and causes the item POPped to be removed so that it cannot be referenced again. The location referenced is the last value pushed onto the referenced stack (A or P). The function (ADR) returns the appropriate stack register displaced by an amount that is one word before the value of the appropriate virtual stack pointer. (The virtual pointers point just beyond the end of the stack, that is, they point to the word just beyond the last value on the stack. The meanings of "before" and "beyond" depend on which stack is being discussed, since the absolute stack grows downward, toward higher-numbered core, and the pointer stack grows upward, toward lower-numbered core. For the absolute stack, "beyond" is down and "before" is up; for the pointer stack, this is reversed.)

Thus instructions exist in the calling sequence that update ASP and PSP registers at run time. The virtual stack pointers (as represented by Deltaasp and Deltapsp) and the lists associating names with stack locations exist only at assembly time. They are used by the assembler to determine the location assignment of values put on the stack.

The (LABEL ...) addr field refers to the labeled points in code (see the discussion of label item types). Whenever a (LABEL ...) addr field is encountered, Labels is searched. If the label is found and its associated page number is the current page number, the addr function simply returns BPR and the value of the ilc (instruction location counter, relative to the BPR) associated with the label in Labels as the displacer. If the label does not exist in Labels or if the labels page is not the current page, then the function returns as its value the BPR and an empty displacer field, to be filled in later, and the label is added to Lblref, so that the addr may later be fixed up, in the following manner.

After the pair of instructions have been added to Corim, the address fields of all the instructions that refer to this label and need fixup [found in Corim, at the locations pointed to by the ℓ's of the list (ℓ_*)] are fixed up to point to the ilc of the first instruction of the pair added. Then the list (ℓ_*) is removed from Lblref and is replaced by the pair (pg . ℓ) where ℓ is the location in Corim of the first instruction of the two added instructions and pg is the current page number. This, then, is sufficient information to enable the add and branch instructions to be fixed up, when the label is encountered.

One further operation takes place when a label reference field is encountered. Since the add and branch instructions take two words for each "off-the-page" label reference (two per label referenced, not two per reference), and since it must be assumed that all currently undefined label references are off-the-page references until the label is encountered, the amount of space on the page is reduced by two words each time a label reference new to the current page is added to Lblref. Note also that PAGEROOM is increased by two each time a label referenced on the page is encountered on this page. Pageroom is reduced by the instruction size each time an instruction is added to Corim, and that Pageroom is reduced by the literal's size each time a literal is encountered (literals are stored in the BP at the end of the page).

If the label does not already exist on Lblref, it is added and paired with the empty list of lists, ((())). If the first item of the first list (of the lists paired with the label in Lblref) is a number (the page number of a non-current page), then a new list is inserted, (nil). If the label was found in Labels, but on a previous page, then the first item of the first list is set to (pg . ilc), the pg and ilc of the label found in Labels. Finally, the location ℓ (in the Corim) of the instruction in which this addr field exists is always inserted, as the second item in the first list following the label in Lblref. This makes it possible to go back and fix up the addr field of the instruction in Corim, when the label is encountered or at the end of the page.

At the end of the page, all label references on the current page that have not been fixed up (that is, all that are still in Lblref in the form: (nil . (ℓ_*)) or ((pg . ilc) . (ℓ_*))) are assumed to refer to labels that are not on the current page, and so a pair of instructions is inserted at the bottom of the page. The instructions are of the form: an add or subtract of an adcon (multiple or 1024 stored in a table of adcons in fixed program space) from BPR, then a branch to the BPR displaced by some quantity. At the time of their creation, the adcon locator and displacer fields are left blank, as the location of the label that is being referenced is not known.

APPENDIX

Lisp 2 Assembly Language Syntax

TOP-LEVEL EQUATIONS

Lap-definition = (LAP listing ref-list)
ref-list = {undefined}
listing = ({FUNCTION|MACRO} (f-name value-type) par-list item*)
variable = (variable-name . {section-name|LEX.})
f-name = variable
value-type = {undefined}
par-list = {undefined}
item = S/360-inst|pseudo-inst|label
variable-name = identifier
section-name = identifier
label = identifier|integer

SYSTEM/360 INSTRUCTION EQUATIONS

S/360-inst = rr|rx|rs|ss|si
rr = (op)|(op reg)|(op reg reg)
rx = (op reg addr)|(op reg addr reg)
rs = rx
ss = (op length addr length addr)
si = (op (immed) addr)

APPENDIX (Cont.)

```
Op      = {AND|OR|XOR|CPL}  |
          ({APP|SUB|MULT|CPA} {H|F|SF|DF|DEC})  |  (Overbar indicates
          {ADDL|SUBL|DIV} {F|SF|DF}  |                default type)
          B{[UC NEVER XH XIE AL CT]  |
            [GR LS EQ NG NL NE]  |
            [{OF|POS|NEG|ZE}*]  |
            [{ONES|MIXED|ZEROES}*] }
          SHIFT [SING|DOUB] [LOG|ARI] [RT|LFT]
          LD    [POS|NEG|TEST|COMP] [H|F|SF|DF|MULT]  |
          STO   [H|F|SF|DF|MULT|DEC]  |
          MV    [NUM|ZON|OFS]  |
          HALVE {SF|DF}  |
          PK  |
          UNPK  |
          TRANS  |
          TRANST  |
          EDIT  |
          EDITM

reg     = integer symbol

addr    = (reg displacer)  |
            literal  |
            lexical-variable  |
            own-variable  |
            fluid-variable  |
            (UNIQUE S-expression)  |
            (QUOTE S-expression)  |
            (PUSH {{A|P} [COUNT]|AD})  |
            (POP {A|P})  |
            (LABEL label)
            (ENTRY identifier displacer)
```

APPENDIX (Cont.)

```
length          = integer|symbol

immed           = character|integer

symbol          = identifier

displacer       = {integer|symbol}*   (Note: EOR'ed together to produce displacer)

literal         = ({2|4|8} integer)   (Note: Integer has size of 2|4|8 bytes)

lexical-variable = (lex-var-name . LEX.)

own-variable    = (own-var-name . section-name)

fluid-variable  = (fluid-var-name . section-name)

lex-name        = identifier          (Used to name a lex-var on the pushdown
                                       stack)

count           = integer             (Range 1-16, the number of items pushed
                                       in a multiple PUSH)

word-count      = integer             (Used to reference specific word in a
                                       multiple PUSH item)

lex-var-name    = identifier

own-var-name    = identifier

fluid-var-name  = identifier
```

APPENDIX (Cont.)

PSEUDO-INSTRUCTION EQUATIONS

pseudo-inst = (ENTRY identifier displacer)
(BEGIN)
(ARG)
(CALL function-name map)
(PUSH {{A|P} [count]|AD})
(POP {A|P} [count])
(BLOCK item$_{*}$ declare item$_{*}$)
(RETURN)
(END)
(DECLARE bound-declaration$_{*}$)
(SYMBOL (identifier . integer)$_{*}$)
(ORG [integer])
(CASEGO reg label$_{*+1}$)
(ALIGN a [b])

function-name = variable

map = {undefined}

bound-declaration = {undefined}

26 April 1967

TM-3417/400/00

S.D.C.
DISTRIBUTION

	Room
J. Barnett (5)	9721
E. Book	2322
R. Bosak	2328
J. Burger	9919
D. Crandell	9731
E. Ehrich (30)	2225
S. Feingold	9525
Donna Firth	9722
H. Howell	9912
Aiko Horman	9717
K. Hinman	2032
A. Irvine	1139
E. Jacobs	2344
B. Jones	2231
S. Kameny	9310
C. Kellogg	9636
R. Long	9716
E. Myer	9413A
M. Perstein	2344
W. Schoene	9923
V. Schorre	2330
J. Schwartz	2105
R. Simmons	9439
S. Shapiro	2413
E. Stefferud	2620
M. Spierer	2109
A. Vorhaus	2213
C. Weissman (10)	2314
R. Wolfson	2368