T H E U N I V E R S I T Y O F M I C H I G A N

Memorandum 20

AN ASSEMBLY LANGUAGE SYSTEM FOR DEC MINICOMPUTERS

V. Michael Powers
David L. Mills
Neal L. Laurance (Scientific Research Staff, Ford Motor Company, Dearborn, Michigan)

CONCOMP: Research in Conversational Use of Computers
ORA Project 07449
F.H. Westervelt, Director

supported by:

DEPARTMENT OF DEFENSE
ADVANCED RESEARCH PROJECTS AGENCY
WASHINGTON, D.C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

May 1969

ABSTRACT

A collection of programs, running in The University of Michigan Terminal System (MTS) on an IBM 360/67 computer system, accomplish assembly and linkage editing of relocatable programs written for DEC's PDP-1, PDP-5, PDP-8, PDP-7, and PDP-9. A link editor program can be used to merge the outputs from several different runs of the assembler and produce a single relocatable or absolute load module.

PREFACE

This memorandum describes the PDP-5/8 and the PDP-7/9 language assemblers and the PDP-8 Link-Editor/Loader which are currently running on the duplex IBM 360/67 system at the Computing Center of The University of Michigan under MTS (Michigan Terminal System). The programs are written in IBM System/360 OS Assembly Language, Level G. The memorandum serves both as a manual for the system user as well as a report on the system development.

Tim Swanson, of the Geophysics Laboratory, Institute of Science and Technology, effected the changes necessary to realize the PDP-1 version, and wrote the appendix.

V. Michael Powers

David L. Mills

Neal L. Laurance

TABLE OF CONTENTS

LIST OF TABLES

1. INTRODUCTION

Within the last few years, several assemblers have been developed at The University of Michigan to service the ever-increasing number of PDP-8s located on the campus[1,2] These assemblers were designed to be run on a large machine and produce code which was executed on the PDP-8. Such systems also served the PDP-5s, which are logically identical to the PDP-8. The recent addition to the PDP-8 family, the PDP-8S, the PDP-8I, the PDP-8L, as well as the PDP-8 part of the LINC-8 and LAB-8 computers can also be served by such assemblers.

The present assembler represents several departures in construction from earlier systems. In the first place, it is designed more for a timesharing environment both in its facilities and its record formats. It takes advantage of the virtual memory of the IBM 360/67 for intermediate file storage, thereby greatly increasing its speed and reducing the cost of assembly. But a more important difference is in the language structure itself. An assembly language statement is composed of three fields, a label field, an operation field, and an operand field. (We neglect comments since they play no role in the language structure.) The difference between this assembler and conventional machine assemblers arises from the fact that the operation field and the operand field are both sentences in a language which can be described by a modified operator precedence grammar.

This means that the operation or the operand part of an assembly input line, or both, can be complex expressions, much in the style of a compiler language. Indeed, the syntax used in the expression analyzer is quite similar to the syntax in the MAD/I compiler language[3].

Another major departure from the earliest PDP-8 assemblers is the inclusion of relocatability within the language structure. To effect complete relocatability, a link-editor and synthetic loader have been written which run within MTS, as well as several loaders which load relocatable code within the PDP-8. Relocation has been carefully implemented so that an absolute assembly is a simple subset of a relocatable assembly, which is otherwise compatible with relocatable assemblies. The relocation structure used was described in an earlier memorandum[4], but the interpretation of some terms has been somewhat modified from the description in that document.

The assembler has also been enlarged to include assemblies for the PDP-7 and PDP-9 class of computers, but relocation facilities have not been implemented for these machines. Throughout the report we will be discussing the PDP-8 assembler, but most remarks, except those specific to relocation, apply also to the PDP-7/9 assembler. A minor modification of the assembler has produced a version which assembles for the PDP-1; pertinent details are included as an appendix to this report.

2. ASSEMBLING IN MTS

The assembler can be invoked in MTS by the $RUN command (see the MTS manual for a full description), with the object module specified as *8ASR (for PDP-8 assemblies), *9ASR (for PDP-9 assemblies), and *1ASR (for PDP-1 assemblies). Logical I/O units used are:

SCARDS assembly input file (defaults to *SOURCE*)

SPRINT assembly printed listing (defaults to *SINK*)

SPUNCH binary object module

SERCOM (optional) error listings

In addition, the programmer may specify a set of print options by means of the PAR=(string) specification. The options available are identical to the options described under the OPTIONS pseudo-op below.

The input lines to the assembler are variable-length records in standard MTS format. The lines may be from 1 to 256 characters in length. Because of this long line available within the system, no continuation card convention was implemented within the assembler; each input line is processed by itself. The assembler has the ability to interpret lines containing tabulation characters, and to expand a compressed line for output. The expansion is not performed until the line is to be printed, thereby saving storage space in most source files.

The assembler scans the input line for three fields:

the label field, the operation field, and the operand field. If the first character of the line is an asterisk "*", the entire line is treated as a comment; it is printed but otherwise ignored. Fields are separated from each other by one or more blanks or by a single tabulation character. If the first character of the line is a blank or a tabulation character, the label field is considered empty; otherwise the label field is collected. The operation field of the line is then scanned. The result is used to determine how to interpret the label field and whether or not to scan the operand field. After the operand field is scanned (or the operation field, if an operand field is not present) the remainder of the line is treated as a comment.

The result of the operation field scan may be a machine operation code or an assembler pseudo-operation indicating that some fixed procedure is to be executed. If the result is a machine instruction, the label is defined to be the current value of the instruction location counter; otherwise the interpretation depends on the pseudo-operation.

The printer listing is produced in one of two formats depending on the setting of a print options switch. One of these formats is designed for a 128-column line printer while the other is designed for a 72-column teleprinter or similar device. For control of this option, refer to the OPTIONS pseudo-op below.

The binary output is produced in variable-length records designed for MTS internal storage. They can be copied onto paper tape, but more usually they are used as the input to some utility program which alters their format for the need at hand. A binary card format has been specified for this language[4], but is not produced by the assembler directly.

3. ASSEMBLY LANGUAGE

3.1 LANGUAGE STRUCTURE

Besides being a working tool for the many users of PDP-8s within the academic community, the assembly language also represents the results of some experiments in applying the techniques of operator precedence grammars to the construction of assemblers. The grammatical analysis involved within the assembler is very similar to that used in the construction of the MAD/I translator, and the syntax of the expressions allowed within the assembler is very nearly a subset of the syntax found within the MAD/I system. The assembler syntax is not properly an operator precedence grammar, but is transformed into one by means of terminal context transformations. For example, the minus sign (-), when found in a unary context, is "transformed" into the unary negation operator. In this way one variously interprets the asterisk (*) as the ILC, the multiplication operator, or the indirect addressing flag. The operator precedence grammar has many processing advantages including speed

of parsing and excellent error recovery. For a further discussion of the syntactic structure of the assembler as well as that of MAD/I, the reader is referred to a previous technical report, The Syntactic Structure of MAD/I[3].

3.2 FIELDS

An input line may have four fields, delimited by one or more blanks or by a horizontal tab character. The first field, or label field, may contain a single symbol. This field must begin with the first character of the input line. If this first character is blank, the label field is considered empty. The operation field (which is always present) is an expression formed according to the grammar mentioned above. The basic terms involved in this expression are machine operation code mnemonics or assembler instructions (pseudo-ops) and constants. The resulting value of this expression is used to determine whether or not to expect a third field, called the operand field, to follow on the input line. The operand field, if present, is also an expression formed in the assembler grammar, which is normally interpreted as the machine address to be combined with the operation code which resulted from the evaluation of the instruction field. (If the instruction field contained a pseudo-op, the interpretation of the operand field varies with the pseudo-op.) The last part of the input line is the comment field and is printed on the assembly listing. If the input line begins with an asterisk as its first

character, the entire line is treated as a comment field.

3.3 TERMS

As the expressions in the instruction and operand field are analyzed, the assembler attempts to compute some result which it ascribes as the "value" of the expresssion. We are using the word value here in a broad sense as being the result of some computation. This result is obtained by combinations of the "values" of the primitive terms within the expressions. A term in an expresssion may be a variable, a constant, or the special term * (asterisk) which stands for the indirect flag in the operation field, or the ILC in the operand field.

Each variable is a string of one to eight letters and digits, the first of which must be a letter. A label is a variable which is defined by its appearance in the label field of an input line; its value, unless the instruction field is one of certain pseudo-ops, is the value of the ILC corresponding to the line in which it appears. An opcode is a variable which represents a machine instruction, such as JMP or CLA. Many opcodes are predefined by the assembler, and others may be defined by the user through the use of the OPD or OPDM pseudo-ops. A pseudo-op, or assembler instruction, is a variable predefined by the assembler, which represents a call on a procedure internal to the assembler, such as setting the value of the ILC (ORG)

or performing some formatting operation (EJECT) or definition (EQU). Lists of predefined pseudo-ops and opcodes appear in Section 5. An external symbol is a variable, appearing within the current assembly, which names a value which may be defined or used in another assembly; it is intended as a means of communication among the assembler and programs which operate on the assembled object module, such as a linkage editor or a loader. Definitions of variables are separated into three internal tables by the assembler: the labels in one, the opcodes and pseudo-ops in another, and the external symbols in a third. Thus, the same alphanumeric string can variously represent three different labels: a label, an opcode or pseudo-op, and an external symbol. The value of an expression containing one such label is unambiguous, since a variable in the instruction field must be an opcode or pseudo-op, a variable in the operand field must be a label, and external symbols are not used in expressions.

Constants can be numeric or alphabetic. A string of decimal digits is converted as a number with the appropriate radix (8, if none other is specified). If the numeric constant is immediately followed by the letter B, K, D, or X, then the radix is taken as 2(B), 8(K), 10(D), or 16(X). A string of characters between apostrophes may be a numeric or an alphabetic constant, depending on what letters immediately follow the last apostrophe. An A means the last

character of the string is converted to its ASCII equivalent; if an N immediately follows the A, the high-order (parity) bit is forced to 0. A P immediately following the term's final apostrophe forces 6-bit packed ASCII conversion: the low-order 6 bits of the ASCII translation of the last two (or three, for PDP-9) characters of the constant are packed in order to form the value of the constant. If no letter appears after the last apostrophe, an A or P is assumed, respectively, depending on whether or not the constant contains fewer than two characters. (In order for the constant to contain an apostrophe, the character string must contain two adjacent apostrophes.) If the letter trailing the character string is B, K, D, or X, the string is taken as a numeric constant (of radix 2, 8, 10, or 16, respectively). Table 1 lists some possible constants and their corresponding octal PDP-8 conversions.

3.4 EXPRESSIONS

The productions governing the analysis of the expressions found in the assembly language are fully described in Reference 3. We can summarize the language here by describing an expression as a term or a combination of terms separated by operators which obey certain straightforward precedence relations. The assembler scans an expression from left to right, associating terms in the expression according to the precedence table given in Table 2. Expressions may be parenthesized as required to an indefinite

Table 1. Examples of Constant Conversion

character string	converted value (octal)
1	0001
101B	0005
10D	0012
'A'	0101
'C'	0303
'E'A	0305
'F'AN	0106
'G'AN	0107
'CC'	0303
''''	0047
''''''	4747
'FFF'X	7777
'1'K	0001
'2'X	0002
'11C'X	0434
'11'D	0013

Table 2. Operators in Order of Decreasing Precedence

Operator	Semantics
¬	unary logical inversion
ε	bitwise logical AND
|	bitwise inclusive OR
¬	bitwise exclusive OR
-	unary negation
*,/	arithmetic multiplication, division
+,-	arithmetic addition, subtraction

degree. Two's complement arithmetic is used in all calculations.

Although all the operators listed in Table 2 are recognized syntactically by the assembler scan, not all combinations of operators are semantically valid. Within the instruction field, especially, the only operators recognized are the plus sign +, and the * which signifies indirect addressing. The plus sign is taken as meaning the logical OR function of two opcodes. The assembler checks to see that operate group operation codes belonging to different subgroups are not combined. Thus, for example, a line containing an IOT instruction ORd with a microinstruction will be flagged as having an operator error.

Within the operand field, any of the operators listed in Table 2 may be used with the semantics indicated. In order to implement relocatable code, each term and each expression is associated with two numbers, one called the value and another which will be called the CSID. The instruction field also has these two values, but the CSID component is not used. A discussion of the CSID is deferred until the next section; here we merely remark that the value is computed to 18 bits and the CSID to 9 bits, regardless of target machine. These values are further truncated, depending on the target machine, when the binary records are produced.

The assembly of the machine word is governed by the

result of the operation field. If the operation field is such that it did not require an operand field, then its value is the final machine word. If the operation requires an operand, then the value of the operation is ORd with some appropriate subfield of the operand value to form the final machine word. Which subfield is used is dictated by the operation field. In the PDP-8 case, four types of operand fields are recognized. For an adcon (DC), the low-order 12 bits of the operand value become the machine word. For a memory reference instruction, the low-order 7 bits of the operand value are used. The "same page" bit is also inserted if necessary, or a page flag may be generated if the operand value differs in page from that of the ILC. Two new instructions, IDF and IIF, have been added to the repertoire. They have the same operation codes as CDF and CIF respectively, but they require operand addresses. For these instructions, the high-order 3 bits of the 15-bit address are selected and ORd into bits 6-8 of the machine word. This allows some very convenient code. For example, if one wishes to refer to a location TAB which is in core bank 1, one may write the instruction

```
CDF+10
```

which will change the data field to core bank 1. More transparently, one can now write

```
IDF    TAB
```

which changes the data field to correspond to TAB. Note

that the CDF (like other micro-instructions and IOTs) does not use an operand field; the entire instruction must be described in the operation field. Finally, two instructions peculiar to the 338 display are predefined within the PDP-8 assembler because they also require the 3 high-order bits of the 15-bit address. These instructions are PJMP and JUMP. For these instructions, the 3 high-order bits of the operand value are ORd into the low-order bits of the instruction.

3.5 RELOCATABLE EXPRESSIONS

It is sometimes convenient to be able to assemble a program before it is known what addresses the program will occupy when it is loaded. In the loading of such a program, the adjusting of its contents to specify the correct addresses is known as relocation; the assembled object module of such a program, containing the information necessary for such adjustment, is known as a relocatable object module. Construction and use of relocatable programs are discussed in Section 5. Here, we will make use of a few concepts, which are more fully explained later, in order to discuss relocation information as it relates to expressions.

Any term whose value is an address within a relocatable segment of a program is a relocatable term; such a term has assigned to it by the assembler not only the value of the address but also a second, nonzero number known as the CSID, which indicates something about its relocation properties. In fact, it is convenient to regard every term as having a

complex value which has two parts: CSID and (simple) value. Those terms which are not relocatable (called absolute terms), such opcodes, pseudo-ops, alphabetic and numeric constants, and terms whose values are not addresses in relocatable program segments, have assigned a special CSID value, zero. Operationally, then, a term is relocatable iff its CSID is nonzero; it is absolute iff its CSID is zero.

Every expression also has two numbers, CSID and value. An expression consisting of a single term has the CSID and value of that term. An expression with more than one term has a CSID and a value each formed by combinations of the CSIDs and values of the terms, as determined by the operators of the expression.

The value of the expression is determined straightforwardly as the sum, difference, negation, inclusive OR, etc. of the individual term values. The CSID of the result, however, is calculated differently in some cases. In contrast to some assemblers (see Reference 5, especially p. 22), the PDP-8 Assembler allows liberal mixtures of relocatable and absolute terms in its expressions. Thus, for example, division of a relocatable term by an absolute term is allowed without complaint, but with full warning that the relocated value of the expression may not be the relocated address divided by the constant. In this vein, the CSID of the sum or difference of two terms is the sum or difference, respectively, of their CSIDs. Any number of relocatable

and absolute terms are allowed in an expression. (Note that the CSID of the difference of two relocatable terms with equal CSIDs is zero; this implements the convention that the difference of two relocatable terms from the same program segment is absolute.) For the other binary operations (&, |, ¬, *, /), the CSID of the result is formed in the following, not entirely arbitrary, manner: if both operands are absolute, the result is absolute; if only one is relocatable, the result is relocatable and has the nonzero CSID of the relocatable one; if they are both relocatable, the result has a zero CSID (absolute), a flag is generated indicating a probable relocation error, and evaluation of the expression continues.

4. DOCUMENTATION AND DEBUGGING AIDS

As mentioned above, the assembly listing contains much useful information. The basic format, constructed for a 128-character printer, lists assembly flags, ILC (address) value, machine word value, pseudo-op parameter value, MTS source file line number, and input line image for each line. The assembly flags, as discussed below, are single characters denoting probable errors in the input line. The three value fields are different sizes for the PDP-5/8 and 1/7/9; although the PDP-8 is a 12-bit machine, the assembler treats addresses as 15 bits (5 octal digits) to allow for assemblies spanning more than one core bank. The machine word value is the width appropriate for the target machine (4 or 6 octal digits), and

displays the value of the text word which appears in the object module, and which corresponds to the address value. Certain pseudo-ops which do not produce machine words may be associated with an important value, such as a number of storage locations, an address, or the value of an expression; such a value is displayed in the third field, whose width is the same as the ILC value field. Some different language processors assign line numbers to statements arbitrarily; this assembler uses the file-device line number obtained from MTS. Maintenance and change of source text from indexed files can thus be immediately effected by use of the line numbers appearing on the assembly listing. The text of the input line is reproduced following the line number, except that horizontal tabulation occurs at tab characters, if the TAB pseudo-op has appeared.

Section 5 lists and discusses pseudo-ops, among them TAB, SPACE, EJECT, and TITLE, which affect space and page control of the listing; and OPTIONS which can, among other things, cause an alternate, short-line format suitable for teletype listing, or force the production or omission of the reference tables described below.

The operand cross-reference table is normally printed after the last source line. It contains, for each label appearing in the program, information about every definition and use of the label. Any assembly flags assigned to the label appear at the left, followed by the 8-character label.

Next are the CSID and value assigned to the label (both zero for undefined labels). The line number of the definition (if any) appears next, followed by a list of references, or occurrences of the label in expressions. Each reference is given as two numbers, the CSID and value of the ILC before the processing of the line in which it appeared. One special label, #ERROR, appears in the operand cross-reference listing only if an assembly flag appears in the listing; references listed under #ERROR indicate the locations, in the assembly listing, of the offending lines.

An operator cross-reference listing can be produced, if desired. This table lists the occurrences of each of the opcodes and pseudo-ops, in the same format as that of the operand cross-reference listing (the line number is omitted for predefined variables).

A relocation dictionary appears next. For every external symbol in the assembly, an entry appears, containing the name, type code (CSECT, EXTRN, or ENTRY type), CSID, value or address, and length of the symbol, along with a list of "RLD items." An item appears for each time a relocatable symbol occurred in a relocatable operand expression in such a way as to force the generation of relocation information. The item contains a relocation flag (one digit) and the CSID and address of the occurrence.

In all three of these tables, the operand and operator cross-reference listings, and the relocation dictionary,

variables which are not defined, and variables which are not used in the input text do not appear. Variables which acquire assembly flags are forced to appear, even though the rest of each table may be suppressed.

The alternate short format suppresses printing of much of this information. Any sections of the listing which would otherwise appear as above are printed in an abridged format which reduces line length by suppressing such things as multiple assembly flags, CSIDs, line numbers, and the high-order digit of PDP-8 addresses. The short format was designed with slow-speed, narrow-carriage printers such as teletypes in mind.

The assembly flags mentioned above are produced in those cases where the input text presents the assembler with an anomaly which cannot be resolved simply and unambiguously. These flags are usually said to refer to "assembly errors," although the assembler may occasionally produce, except for the flag, exactly what was intended by the programmer. A list of the assembly flags and some typical interpretations follows. Any or all of these flags may appear for an input line.

M - multiply defined symbol.

1. A variable (label, opcode or pseudo-op, or external symbol) is defined more than once.
2. The value to be used in the definition of the symbol results from an expression containing a symbol already

flagged with M.

In any case, the most recent definition establishes the value of the symbol used in an expression.

U - undefined symbol.

1. A variable appears in an expression, but has no definition. The variable in this case is assigned the value 0. (Note that some pseudo-ops require that all variables in their expressions be predefined.)
2. The expression defining a variable contains a variable marked U.

S - syntax error.

1. A name (usually a variable name) is more than 8 characters long.
2. A syntax error has been detected in an expression, such as:

```
A+*B
```

or incorrect parenthesization.

C - invalid character

1. A character found in an expression cannot be interpreted, at least in the given context.
2. The character string of a constant is too long, is not terminated by an apostrophe, or is followed by a letter which does not denote one of the permissible modes of conversion.

O - invalid operation code

1. The opcode expression includes at least one variable which is not defined as an opcode or pseudo-op.
2. The expression is an illegal combination of opcodes and/or pseudo-ops. (For example, in the PDP-8, RTL from microinstruction group 1 cannot be combined with SZA from group 2.)

P - page reference error

1. For the PDP-8, the direct address of a memory-reference instruction is neither in the current page nor in page 0.
2. The direct address of a memory-reference instruction is outside the current core bank.

R - relocation error

1. Some information about the relocating of the value of an expression is being lost (i.e., in division of a relocatable term by another relocatable term).
2. A CSECT (relocatable program segment) seems to have a negative length; part of the CSECT is being assembled at an address below the declared start.

L - missing or invalid label

The label field is blank or contains an invalid character string when it probably should have a legal variable name.

The final line of the assembly listing contains decimal numbers for the following quantities:

ERRORS - the number of input lines which caused generation of at least one assembly flag. Multiple flags on a line are counted as one, and appearances of flags in the reference tables are not counted.

SCARDS - the number of input lines read.

SPRINT - the number of lines of assembly listing printed.

SPUNCH - the number of records (lines) written as the object module.

STORAGE - the maximum number of virtual pages (4096 bytes per page) acquired for storage of tables and text.

5. PREDEFINED SYMBOLS

There are several classes of symbols which are defined within the assembler. Two special symbols have been mentioned before: *, which serves either as a multiplication sign, as an indirect addressing flag (in the instruction field), or as the name of the ILC (in the operand field), depending on context; and #ERROR, a symbol used as a label to reference occurrences of assembly flags. The rest of the predefined symbols are pseudo-ops and opcodes. The PDP-8, PDP-1, and PDP-9 all use the same set of pseudo-ops, but need different sets of opcodes.

5.1 PSEUDO-OPS

The pseudo-ops of the assembler are used for such purposes as setting and changing the ILC value, setting and changing global assembly parameters, defining internal and external symbols, controlling the format of the object module, and controlling the format of the assembly listing. In the list below, the pseudo-ops will generally be discussed with reference to the format,

LAB CODE EXP,

where LAB is a label (optional, except where noted), CODE is the mnemonic for the pseudo-op, and EXP is the designation for the contents of the operand field. Wherever the format of the pseudo-op shows a field enclosed in apostrophes, e.g., LAB ORG 'EXP' it means the expression in that field must be a predefined expression; the terms of the expression may only be constants and predefined symbols such as opcodes pseudo-ops, the special symbol *, and variables which have been defined in the assembly before their appearance.

ORG — set instruction location counter (ILC)

format: LAB ORG 'EXP'

procedure: Set the ILC to the value of the expression EXP. If there is a label, define it with this new value. (The ILC is zero at the start of an assembly.) Set the CSID of the ILC to the CSID of EXP.

DC define constant

format: LAB DC EXP

procedure: Form a machine word from the value of EXP, and define LAB, then increment the ILC, in the normal manner. If EXP is relocatable, the word is an adcon and references are entered in the RLD.

DS define storage

format: LAB DS 'EXP'

procedure: Define LAB with the current ILC value, and increase the ILC by the value of EXP. Even if EXP is relocatable, only the "absolute" value is used. Any locations skipped by means of DS contribute to the length of the current CSECT.

PAGE start a new memory page

format: PAGE

procedure: Update the ILC value to the next PDP-8 page boundary address (no change if the ILC is already at a page boundary).

RADIX set global radix

format: RADIX KW

procedure: If the keyword, "KW", is "OCTAL", "DECIMAL", or "BINARY", set the radix appropriately for

converting numeric constants which have no modifier. The radix is initially 8.

OPD — opcode definition

format: OP OPD 'EXP'

procedure: OP is defined as an opcode, whose value is determined by the opcode expression EXP, and which can be combined in subsequent opcode expressions with opcodes of operate group 1, operate group 2, extended arithmetic group, or IOT group.

OPDM — memory-reference opcode definition

format: OP OPDM 'EXP'

procedure: The memory-reference instruction OP is defined from the opcode expression EXP.

EQU — label definition

format: LAB EQU 'EXP'

procedure: Define the label LAB with CSID and value from EXP.

CSECT — define a CSECT

format: LAB CSECT 'EXP'

procedure: Define LAB to be an external symbol of type 4 (CSECT) whose CSID is one more than the previous external symbol defined. Its address (the start of the CSECT) has the value of EXP.

The CSECT defined extends from this address to the highest address which is either filled with data or reserved by the pseudo-op DS, up to the start of the next CSECT or DSECT. At the appearance of the CSECT pseudo-op, the symbol, *, for the ILC becomes relocatable and is assigned the CSID at the CSECT. The CSECT object record is produced in pass 2 at the occurrence of this definitional pseudo-op.

EXTRN — define an EXTRN

format: LAB EXTRN LAB2

procedure: Increment the current CSID for use in defining LAB2 as an external symbol of the type 6 (EXTRN). Define LAB as a label (an internal symbol) whose value is zero, with the CSID of the external symbol LAB2. The address and length of LAB2 are 0.

ENTRY — define an ENTRY

format: LAB ENTRY EXP

procedure: Define LAB as an external symbol of type 5 (ENTRY), with its address and CSID taken from EXP.

BREAK — write a BREAK record

format: LAB BREAK EXP

procedure: Define the label LAB at the current ILC value, and produce a BREAK record in the object module, using the value of EXP.

PCS write checksum

format: PCS

procedure: Produce a checksum record, and start accumulating a new checksum, starting with the first byte of the next record.

START write transfer record

format: START EXP

procedure: Produce an END record with its address taken from the value of EXP, but continue the assembly.

END end assembly

format: END EXP

procedure: End the assembly. Print the reference listings and summary line, and write RLD, checksum, and END records to finish the object module. If EXP ends with a comma, the address of the END record is zero; if not, the address is the value of EXP (or zero, if EXP is empty). If no END pseudo-op appears at the end of the assembly the object module ends with a checksum but no END record.

DESIST cease object code production

format: DESIST

procedure: Suspend object code production. Continue to define labels and other symbols and to process pseudo-ops, but refrain from scanning operand fields of memory-reference instructions. This pseudo-op allows inclusion of dummy sections which are assembled in another assembly, some of whose internal symbols are needed in the current assembly. The code processed under the influence of this pseudo-op may be regular assembly language lines, but no flags are generated for such conditions as undefined symbols in memory-reference instruction operands.

RESUME resume normal assembly processing

format: RESUME

procedure: Cancel the DESIST command and resume assembling.

ICTL input format control

format: ICTL 'C1','C2' where 'C2' is optional

procedure: Use the values of the two expressions to set limits on the scan of the input lines. From now on, the assembler will only scan from byte C1 through C2 of each input line.

COPY copy from another source

format: COPY FDNAME

procedure: Start taking input lines from the MTS file/device FDNAME (starting and ending line numbers may be given). COPY is recursive, in that FDNAME may contain a COPY giving some other file/device, and so on. As each such file/device is exhausted, the assembler resumes reading input from the next most recently named "copy source."

OPTIONS set optional parameters

format: OPTIONS P1,P2,...

procedure: If any of the character strings P1,P2,...match the following key words, set flags to accomplish the appropriate formatting:

LONG -Print long format: 128-character lines, with full 15-bit addresses (PDP-8) and reference listings containing CSIDs.

SHORT -Print short format: 72-character lines, with short 12-bit addresses (PDP-8) and condensed reference listings, omitting CSIDs and MTS line numbers.

ON -Resume printing of assembly listing.

OFF -Suspend printing of assembly listing.

REF -Print normal reference listings; predefined symbols other than #ERROR do not appear.

FULREF -Print full reference listings: all referenced symbols appear.

NOREF -Omit reference listings.

ERR -Write on SERCOM a copy of each assembly listing line which has assembly flags.

NOERR -Inhibit the SERCOM output mentioned above.

NODECK -Suspend output of object code to logical device SPUNCH.

DECK -Resume SPUNCH output.

BATCH -The current assembly may be one of many. After the END card is processed, begin assembling again.

The initial assembler format flag settings are equivalent to the result of:

```
OPTIONS  LONG,ON,REF,NOERR,DECK
```

from a batch stream or

```
OPTIONS  LONG,ON,REF,ERR,DECK
```

from a terminal. (Unless SERCOM is specified, the ERR output appears on *MSINK*.)

TAB set tab stops

format: TAB T1,T2,...

procedure: Set the values of each of the expressions T1,T2,... as horizontal tab stops. When a tab character is found in an input line it terminates the field being scanned and causes

spaces to be inserted, in the assembly listing image of the input line, up to the position of the next tab stop.

SPACE skip

format: SPACE EXP

procedure: Insert the proper carriage control character in the next assembly listing line to effect a line skip. The expression values which work properly are empty, 0 and 1, skip 1 line; 2, skip 2 lines; 3, no skip.

TITLE set new title

format: TITLE TE ... XT

procedure: The first 69 characters following the first blank after the TITLE pseudo-op are saved as part of the page header line. A new page is not forced at this time, but the next new page will be headed by this title, unless the pseudo-op appears again before the new page is started.

EJECT skip to new page

format: EJECT

procedure: The next line of assembly listing will appear at the top of a new page.

5.2 PDP-8 OPCODES

The following opcodes are predefined for the PDP-8

corresponding to the codes listed in Reference 5 except where noted:

Memory Reference Instructions

AND	0000
TAD	1000
ISZ	2000
DCA	3000
JMS	4000
JMP	5000

Floating-point Mnemonic Pseudoinstructions

FEXT	0000
FADD	1000
FSUS	2000
FMPY	3000
FDIV	4000
FGET	5000
FPUT	6000
FNOR	7000

Operate Group 1 Microinstructions

IAC	7001	
RAL	7004	
RTL	7006	
RAR	7010	
RTR	7012	
CML	7020	
CMA	7040	
CIA	7041	(CLA+CMA)
CLL	7100	
STL	7120	(CLL+CML)
GLK	7204	(CLA+RAL)
STA	7240	(CLA+CMA)

Operate Group 2 Microinstructions

HLT	7402	
OSR	7404	
SKP	7410	
SNL	7420	
SZL	7430	
SZA	7440	
SNA	7450	
SMA	7500	
SPA	7510	
LAS	7604	(CLA+OSR)

Operate Extended Arithmetic Element Microinstructions

MUY	7405	
DVI	7407	
NMI	7411	
SHL	7413	
ASR	7415	
LSR	7417	
MQL	7421	
SCA	7441	
MQA	7501	
CAM	7601	(CLA+MQL)

Combined Microinstructions

NOP	7000
OPR	7000
CLA	7200

Note: Microinstructions from different groups may not be legitimately combined, but any of the "Combined Microinstructions" may be used with a set from any group above. For example,

CML+HLT is incorrect, but

SNA+HLT+CLA is legitimate.

Input-Output Microinstructions

Processor	IOT	6000
	ION	6001
	IOF	6002
	SMP	6101
	SPL	6102
	CMP	6104
33ASR Keyboard	KSF	6031
	KCC	6032
	KRS	6034
	KRB	6036
33ASR Printer	TSF	6041
	TCF	6042
	TPC	6044
	TLS	6046

High-speed Tape Reader	RSF	6011
	RRB	6012
	RFC	6014
High-speed Tape Punch	PSF	6021
	PCF	6022
	PPC	6024
	PLS	6026
Memory Extension	CDF	62n1
	IDF	62n1
	CIF	62n2
	IIF	62n2
	RDF	6214
	RIF	6224
	RIB	6234
	RMF	6244

Note: IDF and IIF generate the same instructions as CDF and CIF, respectively, except that n, the new data or instruction field is taken as the core bank of the expression, rather than its value.

338 Display	PJMP	2010
	JUMP	2000

5.3 PDP-9 OPCODES

The following opcodes are predefined for the PDP-9:[6]

Memory Reference Instructions

LAC	200000	
DAC	040000	
DZM	140000	
ADD	300000	
TAD	340000	
AND	500000	
XOR	240000	
SAD	540000	
ISZ	440000	
JMP	600000	
JMS	100000	
CAL	000000	
XCT	400000	
LAW	760000	(treated as a memory reference instruction to allow assembly of the "immediate" portion of the word from an address expression.)

Operate Group Microinstructions

OPT	740000	
NOP	740000	
CMA	740001	
CML	740002	
OAS	740004	
RAL	740010	
RAR	740020	
HLT	740040	
XX	740040	
SMA	740100	
SZA	740200	
SNL	740400	
SML	740400	
SKP	741000	
SPA	741100	
SNA	741200	
SZL	741400	
SPL	741400	
RTL	742010	
RTR	742020	
CLL	744000	
STL	744002	(CLL+CML)
CLL	744002	
RCL	744010	(CLL+RAL)
RCR	744020	(CLL+RAR)
CLA	750000	
CLC	750001	(CLA+CMA)
LAS	750004	(CLA+OAS)
LAT	750004	
GLK	750010	(CLA+RAL)
LAM	777777	(LAW 17777)

Extended Arithmetic Element Microinstructions

OSC	640001
OMQ	640002
CMQ	640004
LACQ	641002
LACS	641001
CLQ	650000
ABS	644000
GSM	664000
LMQ	652000
EAE	640000
LRS	640500
LRSS	640500
LLS	640600
LLSS	660600
ALS	640700
ALSS	660700

NORM	640444
NORMS	660444
MUL	653122
MULS	657122
DIV	640323
DIVS	644323
IDIV	653323
IDIVS	657323
FRDIV	650323
FRDIVS	654323

I/O Group Microinstructions

CAF	703302

(more may be added later)

6. RELOCATION

6.1 DEFINITIONS AND METHODS

The basic concepts of PDP-8 program relocation, and their implementation,were discussed previously.[4] This section describes the current implementation of the relocation machinery in the assembler and link editor. At the present writing, relocation facilities are implemented only for the PDP-5/8; some modification will probably be needed for efficient relocation with PDP-7/9 programs. PDP-7/9 programs will not, at the present time, assemble properly unless they are absolute. Further discussion in this section will be in terms of PDP-5/8 programs, and sizes and addresses will be in octal notation.

The present system centers around page-relocatable programs. In the PDP-8, direct memory reference instruction addresses refer to locations by using an offset from a page boundary. PDP-8 pages are blocks of 200 locations, beginning

at page boundaries 0, 200, 400, 600, 1000, ... 7600 in each core bank. A section of programs containing only direct memory references, IOT instructions, operate instructions, and numeric constants can thus be placed in almost any page of a given bank without changing its operation, if each word is placed at its proper page address, or offset from the page boundary. However, if one program segment contains an indirect reference to a second segment and the second segment is moved, the adcon (address constant, effective address value, or pointer) which appears in the first segment and which contains an address in the second segment must be changed to the proper 12-bit address. A program segment which can be loaded on any page within a core bank, while keeping its page addresses fixed and changing only the (relatively few) values such as adcons, is known as a page-relocatable segment. Because of the special properties and usages of page 0, regular page-relocatable program segments are usually to be loaded on some other page, while page 0 information such as pointers and constants are usually assembled specifically for page 0.

Programs typically consist of a number of segments, each of which may be relocated differently. A CSECT (control section) is such a program segment; it is a contiguous block of assembled instructions and/or data. A CSECT is up to a core bank in length and may start at any address in a core bank, but must end within the same bank. A module is an

assembly of some number of CSECTs. An object module is a loadable collection of data defining the structure and contents of a module, including text, CSECT definitions, relocation information, and other information, as will be mentioned shortly. The binary information written by *8ASR on SPUNCH during one assembly is an object module; several object modules can be merged to form a single one by use of the link editor.

Linkages between modules are maintained through the use of CSECTs, ENTRYs, and EXTERNs. They are three different types of external symbols; each has a name which is lexically a variable, an address, a length, and an identification number or CSID. The address of a CSECT is the ILC value at which the assembly of the program segment started, and the length is the number of locations it occupies. An ENTRY, or entry point, is the definition of an external symbol which can be used by other modules to refer to a location within a CSECT. Its address is the assembled address within the CSECT in which the definition appeared, and its length is zero. An EXTRN or external symbol reference is an external symbol, presumably defined outside the current module, whose relocated value might be used within the current module. Its address and length are zero. Any name appearing as an EXTRN in one of a loadable collection of modules should supposedly appear (be defined) as the name of a CSECT or ENTRY in another module of the collection.

The process of merging several modules and resolving the definitions of, and references to, external symbols is known as link editing; the link editor program described later in this section does this task and more. It produces, given the proper modules as input, a single output module with EXTRNs resolved and removed.

A page-relocatable program or module includes a number of CSECTs. Typically, most of the contents of these CSECTs will be insensitive to relocation, as mentioned early in this section. Such absolute text need merely be placed by the loader, without change, at the appropriate spot. Frequently, however, a value appears which must change depending on where the several CSECTs are loaded (an adcon, for example). Such relocation is enabled through use of a relocation dictionary, or RLD, produced by the assembler near the end of each load module. This dictionary lists, for each CSECT, every occurrence of a label from that CSECT in a context which implies relocation is necessary (this is essentially the same information as is contained in the relocation dictionary of the assembly listing). CSECTs are identified, for loading purposes, by identification numbers, or CSIDs, assigned by the assembler. The special CSID of zero is used for absolute segments. These segments are recognized by the assembler from the input assembly language when the ILC is set by means of an absolute expression and no CSECT is defined. Such segments are to be loaded at the

addresses where they are assembled, and occurrences of labels from these segments in expressions within other segments do not cause generation of relocation (RLD) items. Relocatable adcons appearing in such a section are properly referenced, however.

An absolute segment of a relocatable assembly might typically be used to specify the positions and contents of pointers and constants in page zero. Indeed, an absolute assembly consists entirely of absolute program, with no CSECTs, ENTRYs, or EXTRNs defined, and no RLD produced.

6.2 IMPLEMENTATION: LOGICAL OBJECT DECK

Each load module consists of a number of records, written as binary MTS lines. There are eight types of records. The detailed format of these records varies depending on the target machine, but their logical format, i.e., their order and the meaning of their contents, is as described below. Each record type is identified in the load module by a single octal digit in the first byte of the record.

0 - Checksum record, containing the two's complement sum, masked to the memory word size of the target machine, of every byte since the last checksum record.

1 - TXT or text record, containing an address for the loading of the first word in the record, followed by text words to be loaded in successive locations.

2 - END record, containing a starting address for the

object module. If this address is 0, the program is not to be started when loaded.

3 - BREAK record, reserved for special communication to the loader concerning the loading address.

4 - CSECT

5 - ENTRY

6 - EXTRN

These records define respectively external symbols of type CSECT, ENTRY, and EXTRN and give a CSID, an address, a length, and a name for each.

7 - RLD or relocation dictionary record, giving the relocation information as described above.

CSECT, ENTRY, and EXTRN records are produced by the assembler in the order in which the pseudo-ops of the same names appear in the input text; therefore, a CSECT record appears before any TXT record with text for that CSECT. BREAK records and checksum records also appear in the order prescribed by the text. TXT records appear in the order assembled; text is buffered and punched when the record length approaches 255 bytes, when the ILC is reset by ORG or DS, or when a CSECT is defined. The END pseudo-op forces the production of RLD records, a checksum record and, usually, an END record.

A typical relocatable load module may therefore have the following records: a number of TXT records with CSID=0 for an absolute portion; a CSECT record, followed by EXTRN,

ENTRY, and TXT records pertaining to that CSECT; a few more sections of CSECT, EXTRN, and ENTRY records, each set concerned with the programming within a single CSECT and perhaps a number of checksum records anywhere among these; a block of RLD records, detailing the linkages and references among the external symbols; a final checksum record; and an END record giving the starting location for the module.

An absolute load module will have only TXT records, a checksum record, and an END record.

6.3 *8 LINK: THE LINK EDITOR

The PDP-8 Link Editor is an MTS program which serves to link edit programs assembled with the relocatable PDP-8 assembler. The link editor performs the tasks of resolving external references between two or more relocatable assemblies and producing an output which can be handled by a loader within the real PDP-8. Facilities are also provided for the maintenance and alterations of programs, some of which may prove useful even for absolute assemblies.

In order to allow the possibility of absolute code within a relocatable assembly, as well as to allow the possibility of absolute assembly, the convention is established that code written without CSECT definitions is in a special CSECT whose name is blanks and whose assigned number is zero. References to this CSECT never appear within the binary records, but the number zero as a CSECT identifier is always taken to mean absolute code.

Relocation (presently) does not apply to core banks; the core bank specified in the assembly is the core bank in which the code will be loaded. This core bank value, however, is available as part of the address value of an external symbol and it may be symbolically referenced. Actual relocation of code is based on page relocatability; that is, a given instruction is assumed to lie in the same word relative to a page boundary after it is loaded, as when it was assembled. This assumption greatly reduces the relocation problem, necessitating only the relocation of adcons. This page boundary loading may be suppressed when appropriate, as may be required for tabular data or display files.

The link editor is driven by a sequence of MTS-like commands; indeed, the MTS command interpreter is actually used. The initial commands are taken from *SOURCE*, not SCARDS, which implies that the program is normally interactive. The first command to the program must be an initialization command. This serves to establish program mode and to preset certain tables. Also in continuing operation, it serves to discard any intermediate file storage left from some previous link editing operation. In describing the program, we will focus on the first mode of operation, relocatable link editing. Other modes will be discussed later.

To establish the link edit mode, the first command must be $RELOCATABLE. There are no parameters to be specified with this command. Note that, like most MTS commands, only the first three characters are required. The link editor, to avoid confusion with MTS, uses % as a prompting character, and its input commands, like those for MTS, must begin with a $.

At this point, the link editor is ready to receive input. The principal command for this purposes is the LOAD command

$LOAD SOURCE=Fdname (ROUND=(ON,OFF))

The LOAD command has several effects. First, it sets the link editor to accept as input, *8ASR-produced binary files which it reads and processes. Second, it establishes the source of these records as the Fdname specified in the SOURCE parameter. Third, an optional parameter ROUND may be specified which effects the setting of a binary switch. If ROUND is ON, then each succeeding CSECT is forced to begin on a PDP-8 page boundary. The setting of this switch remains in effect until reset by another ROUND parameter. Note that if, while reading binary records from the indicated file-device, a record which begins with a $ is read, the link editor reverts to command mode and proceeds to execute the command. In this way, files may be constructed which are sequences of commands for the link editor. The link editor keeps track of all files read (within one link edit

operation) so that if it is asked to reread a certain file, it ignores the command and simply proceeds. The nesting of files which reference other files is arbitrary and limited only by an internal stack. By properly interspersing binary files and link-editor commands, one can construct a well-structured library so that a call for one specific routine will cause the automatic inclusion of all dependent routines in the program.

Other commands available during the input phase include the REPLACE function and the BREAK function. The first has the form $REPLACE add num1 num2 num3 ...
The parameters to be specified are the starting address of the replacement and the values to be stored at that and succeeding locations. The parameter "add" may be one of two forms. The first form is NAME.0000, where NAME is the name of some CSECT or ENTRY which has been previously read by means of a $LOAD command, and 0000 is four-digit octal number specifying a displacement from that symbolic location. Note the occurrence of the period separating the two parts of the address. The NAME field must conform to the rules for CSECT names, one to eight characters in length, alphanumeric, of which the first character must be alphabetic. The second form of this parameter is a five-digit octal number which is treated as an absolute address. The symbols "num1", "num2", etc., refer to four-digit octal numbers used to

specify the value to be written in the memory locations specified by "add." Note that to be effective, these commands must appear after the $LOAD for the code they are intended to override. Examples

```
$REPLACE   ALPHA.0230   7540 3615 2310
$REPLACE    21375   2310 6201 6212
```

The link editor assumes that the relocatable code will all be loaded into memory starting at location 200 within each core bank. Although this assumption is not used in the generation of output files, it is used in the memory maps produced. In order to change these values, one specifies a value for the program break for each core bank by means of the $BREAK command.

```
$BREAK  add
```

Here add is a five-digit octal number which specifies the value of the break field for the core bank indicated. If the break field is to be changed in more than one core bank, several such values may appear on this command. The link editor produces a break record for each core bank in which code appears for use of the relocatable loader. These break values come either from explicit specification of a break field by the command described above, from binary break records contained within the loaded programs, or from the default value of 200 built into the link editor. In any case, the last read value is effective.

The output phase is initiated by the $OUTPUT command.

$OUTPUT MAP=FD1 FILE=FD2 TAPE=FD3 ESD=FD4

Up to four file-device names may be specified as parameters as indicated above. If any of these names is omitted, the corresponding output is not produced (the default for all is *DUMMY*). In addition, the link editor produces certain error comments on SERCOM, principally if it finds references to an undefined external symbol within the programs loaded. The MAP parameter specifies the production of a loading map listing all the CSECT and ENTRY names loaded and their values. In addition, the internally assigned numbers for each of the CSECTs are given along with the length of the CSECT. ENTRYs are assigned the same number as the CSECT in which they are found. It is possible that an ENTRY be assigned the CSECT number 0, if its assembled value is absolute. If any unresolved externals still remain after the link-editing process, these will also be listed so that they can be identified by the programmer.

The file device specified by FILE will contain an output binary file which represents the link-edited records which have been input. This is a suitable form for containment within MTS files. The file produced by the TAPE command is like that of FILE except that it contains leader, trailer, and a checksum record. This file is suitable for punching on paper tape for direct input to the PDP-8 relocatable loader. Finally, the file produced as a result of

the ESD parameter is a file containing only the CSECT and ENTRY records produced. Such a file may be useful for later link editing with other files in which the symbol definitions are required (similar to the LCSYMBOL facility of the MTS loader).

One may switch freely between the input and output phases within a program run; thus, for example, it is possible to load several files, to produce output with only MAP specified to see what external symbols require inclusion, to load those files which define the necessary symbols, and to again generate output. The memory of all the files previously read is erased only by a new appearance of an initialization command, i.e., $REL.

In addition to this mode of operation, the link editor may be run in absolute mode. Absolute mode is established by the initialization command

$ABSOLUTE (no parameters)

In this mode, all the other commands have exactly the same form but some of them take on slightly different meaning. In particular, the break field now specifies the absolute locations for the relocatable loader which is an integral part of the program. The map produced by the MAP parameter also contains a relocation factor for each symbol. This factor, when added to the compiled address of the symbol gives the loaded address. (All values are octal.)

The FILE specification produces a file which is in relocatable format but contains only absolute records. The TAPE parameter, however, produces a file which is PAL format binary tape, and, if punched on paper tape, can be read directly by the BIN loader. The ESD file is as before.

7. WRITING RELOCATABLE PROGRAMS

In this section we will try to give some idea of how the relocatable machinery which we have described can be used in the construction of programs. One can imagine many configurations of programs and equipment to which relocatable programming could be applied; we will confine ourselves here to a very specific example of a system, and describe two different ways in which it could be programmed.

The system we envision contains a hard core of routines which comprise the operating nucleus, such things as interrupt processors, basic conversion subroutines, etc. To this, for a specific application we want to add some very particular routines, possibly to act like a desk calculator in one case, to service a Calcomp plotter in another, etc. We will construct the basic system as an absolute assembly. It will contain allocation for all the shared constants on page 0, and moreover, it will have an entry definition for every symbol within the system to which other programs must refer. For example, if the constant 7000 is contained on

page 0 with the symbolic name K7000, we would include the code

```
K7000   DC    7000
K7000   ENTRY K7000
```

The second statement defines K7000 to be an external symbol whose value is the same as the internal symbol K7000. In this fashion, every label within the system which must be referred to by other programs would be made an entry.

The other program segments would be assembled as relocatable programs. For example, if we have a program segment which we call 'HSR', it may start with the lines

```
        ORG     200
HSR     CSECT   *
```

The first instruction sets the ILC to 200 to begin the assembly. One normally begins relocatable assemblies at a page boundary other than 0 so that the assembler will insert the "this page" bit properly into the machine instruction. The second statement defines HSR to be an external symbol whose value is the current value of the ILC, and it simultaneously sets the CSID of the ILC to the CSID value of HSR. Note that the instructions ORG and CSECT interact in that both change the ILC. In the example given above, the ILC is relocatable with the CSID of HSR and the value 200. If we invert the instructions

```
HSR     CSECT   200
        ORG     200
```

the ILC is now absolute since the ORG pseudo-op changes both its value and its CSID to agree with the operand, 200, which is absolute.

Within this assembly we may refer to symbols in memory reference instructions which we know to be on page 0 although not in this assembly, by listing these symbols as EXTRNs. For example, if we wish to refer to the constant K7000 which is assembled in the main program, we could write

```
K7000   EXTRN   K7000
        TAD     K7000
```

The first statement declares that K7000, an internal symbol in this program, is the same as the external symbol K7000. The assembler will assume an address 0 for the instruction, and the link editor must insert the proper value.

When both programs have been assembled, the link editor can be called to link edit these programs into a running program. Assume that the files MAIN and HSR contain the binary object module associated with these assemblies. Then the following instructions will serve to produce a BIN format tape which can be loaded on the PDP-8 with the BIN loader.

```
#    $RUN *8LINK
%    $ABSOLUTE
%    $LOAD SOURCE=MAIN
%    $LOAD SOURCE=HSR
```

```
%     $OUTPUT  TAPE=BIN  MAP=*SINK*
%     $ENDFILE
```

This technique can produce large object modules because of the large number of symbol definitions involved. Another way of assembling these programs, which avoids this problem, is to use the copy facility. In this technique, we place all common code such as that found in page 0 in a separate file. By way of example, let us call this file LOWC. In the main program we will have a set of instructions as follows:

```
            ORG      0
MAIN        ENTRY    *
            COPY     LOWC
            ORG      200
               (etc.)
```

This will result in the contents of LOWC being assembled into the MAIN program as normal. MAIN becomes an external symbol defined as location 0. In the relocatable program HSR we write

```
MAIN        EXTRN    MAIN
            DESIST
DUMMY       CSECT    MAIN
            ORG      0
            COPY     LOWC
            RESUME
```

```
        ORG     200
HSR     CSECT   *
        (etc.)
```

The DESIST pseudo-op causes the assembler to stop producing code until the appearance of RESUME. This means that all the code read will result in symbol definitions but no code will be produced. The ORG 0 instruction appearing immediately after the CSECT sets the ILC to 0 absolute so that the code read in by the COPY pseudo-op will result in absolute definitions. In this way a common file can be shared by two components of a system. This method results in much smaller object decks because there are fewer EXTRNs required, but it means that any change in the LOWC file necessitates reassembling all components of the system that use it.

8. OBJECT MODULE FORMAT

Each record of a load module, as described in Section 6, is written as a binary MTS line of fewer than 256 bytes. Data are written in the low-order 6 bits of a byte only. The high-order 2 bits have 10 in the first byte of a record only; the high-order 2 bits have 00 in the rest of the bytes of the record. The data are therefore suitable for direct copying to paper tape. The rest of this description of the physical format of object modules will refer to <u>characters</u>, which, in this section only, means the low-order 6

bits of the bytes.

Every load module record begins with two (6-bit) characters identifying the record. The high-order 3 bits of the first character form the octal digit, mentioned in Section 6, identifying the record as being of type Checksum, TXT, END, BREAK, CSECT, ENTRY, EXTERN, or RLD. The low-order 3 bits of the first character, followed by the second character, comprise the 9-bit CSID known as the record CSID. In the code for the PDP-5 and -8 only, the convention has been adopted that the first 3 bits of the CSID (the low-order half of the first character) define the core bank of the CSID. Here, the assembler inserts the high-order 3 bits of its internal, 15-bit PDP-8 address.

The rest of the contents of a record vary depending on its type and the identity of the target machine. Each can be specified, however, in terms of four types of frames. A frame is the contents of several characters. An address frame is 12 bits (two characters) for the PDP-8 and 15 bits (the low-order 15 bits of three characters) for the PDP-7 and PDP-9. A data frame is similarly 12 bits (two characters) or 18 bits (three characters) for the different machines. A name frame is eight "trimmed EBCDIC" characters; each character consists of the low-order 6 bits of the corresponding EBCDIC character of an external symbol name. Finally, an RLD frame has a 3-bit relocation code and a 9-bit CSID as its first two characters, followed by an address frame

as described above, and comprises four characters (PDP-8) or five characters (PDP-7,-9).

After the first two characters, the various records contain:

0 - Checksum (CSID=0)

A data frame, containing the sum of all bytes of the module since the last checksum, up to but not including the checksum data frame.

1 - TXT

An address frame, giving the assembled address of the first data word on the record, followed by a sequence of data frames containing machine words to be loaded in sequential locations. The record CSID identifies the CSECT in which the text belongs.

2 - END

An address frame containing the assembled address of the starting location for the module (if the address frame is 0, no transfer is to be made after loading). The record CSID identifies the CSECT in which the transfer address appears.

3 - BREAK

An address frame.

4 - CSECT

5 - ENTRY

6 - EXTRN

Two address frames, followed by a name frame. The

first address is the starting address of a CSECT, the address within the most recently defined CSECT of an ENTRY, or 0 for an EXTRN. The second address is the length of the CSECT, or 0 for ENTRY or EXTRN. The name frame contains the name of the external symbol identified by the record CSID.

7 - RLD

An address frame, followed by several RLD frames. The record CSID identifies the external symbol (CSECT or EXTRN) to which an adcon referred. The RLD frames identify the positions of adcons which referred to that external symbol. Each frame has a 3-bit relocation flag, a 9-bit position CSID, and a position address. The loader is instructed, for example, to add the relocation factor of the external symbol whose CSID is the record CSID to the adcon which was loaded at the location given in the address portion of an RLD frame, relocated by the relocation factor of the CSID given in that RLD frame.

REFERENCES

1. "PDP-8 Simulator System," University of Michigan Executive System for the IBM 7090 Computer, Vol. 2, Computing Center, University of Michigan, Ann Arbor, September 1966.

2. Powers, V.M., PDP-8 Assembler, Memorandum 12, Concomp Project, Unviersity of Michigan, Ann Arbor, November 1967.

3. Mills, D.L., The Syntactic Structure of MAD/I, Technical Report 7, Concomp Project, University of Michigan, Ann Arbor, June 1968.

4. Mills, D.L., and Powers, V.M., PDP-8 Program Relocation: Concepts and Facilities, Memorandum 17, Concomp Project, University of Michigan, Ann Arbor, February 1968.

5. IBM System/360 Operating System Assembler Language, IBM Systems Reference Library Form c28-6514-4, San Jose, Calif., February 1968.

6. "PDP-8 User's Handbook," in Part 3 of The Digital Small Computer Handbook, Digital Equipment Corporation, Maynard, Mass., 1967.

7. PDP-9 User Handbook, Digital Equipment Corporation, Maynard, Mass., 1967.

APPENDIX: VERSION OF THE ASSEMBLER FOR THE PDP-1

The following is a list of the instruction op-codes for the PDP-1. The following points should be borne in mind when programming this machine:

(a) The PDP-1 is a one's-complement machine, whereas the Assembler does its arithmetic in two's complement.

(b) The shift-group instructions contain a shift field which specifies the shift count as the number of 1-bits in the field. For example, a 5-bit shift can be specified by any of the binary numbers 000011111, 101010101, 110011001, etc. There are nine predefined symbols for use in shift instructions:

```
#1 = 000000001
#2 = 000000011
     ....
#8 = 011111111
#9 = 111111111
```

Thus the instruction SCL #4 causes a 4-bit left shift of the AC and IO.

(c) There is only one operate-group, but many of the IOT instructions can be ORed together. The same three equivalence classes are used to partition various groups of instructions. For example, the class OPR1 contains both the operate-group and such IOT instructions as DRS, DLA, and DCF. Two instructions

of the same class and the same functional category may be combined meaningfully, but the Assembler will not flag such meaningless combinations as CAL+DLA (clear accumulator 760200 and display load address 720015).

Memory-Reference Instructions (Require operands) Operate Class

LAC	200000
DAC	240000
DAP	260000
DIP	300000
LIO	220000
DIO	320000
DZM	340000
XCT	100000
JMP	600000
JSP	620000
JFD	120000
CAL	160000
JDA	170000
SAD	500000
SAS	520000
ADD	400000
SUB	420000
MUS	540000
MUL	540000
DIS	560000
DIV	560000
IDX	440000
ISP	460000
AND	020000
XOR	060000
IOR	040000

Note that MUS/MUL and DIS/DIV are all meaningful to the Assembler, but only one out of each pair can be hardware implemented.

Augmented Instructions (Require operands)

LAW	700000

Shift Group

RAR	671000
RAL	661000
SAR	675000
SAL	665000
RIR	672000
RIL	662000
SIR	676000

Operate Class

SIL	666000	
RCR	673000	
RCL	663000	
SCR	677000	
SCL	667000	

Skip Group

SKP	640000	II
SZA	640100	II
SPA	640200	II
SMA	640400	II
SZO	641000	II
SPI	642000	II
SZS	640000	II
SZF	640000	II

Operate Group Instructions

CLI	764000	I
EAT	762000	I
LAP	760100	I
CMA	761000	I
HLT	760400	I
XX	760400	I
CLA	760200	I
CLFn	76000n*	I
STFn	76001n*	I
NOP	760000	I

Input/Output Transfer Instructions

IOT	720000
RPA	720001
RPB	720002
RRB	720030
PPA	720005
PPT	720006
TYO	720003
TYI	720004
ESM	720055
LSM	720054
CBS	720056
CKS	720033
EEM	724074
LEM	720074
DSC	720050
ASC	720051
ISB	720052
CAC	720053

*Note: n may be 1,2,3,4,5,6,7

		Operate Class
SCW	720057	
SCI	720157	
SRB	720021	
RKY	720035	
RLS	720026	
DRS	720115	I
DLA	720015	I
DCF	720215	I
DRA	720016	II
DRC	720116	II
DSE	721417	E
DSV	720417	E
DSH	721017	E
DSS	720217	E
DSP	720117	E
MRD	720501	
MWR	720601	
MSE	720301	
MLC	720401	
MRS	720701	
RCV	720031	
CAD	720040	
SMC	720047	
IMC	720060	
LDK	720216	
RDK	720037	
RSK	720316	
CSK	720616	
ECB	720416	
LCB	720516	

The production of character constants, e.g., DC 'ABC'P, is governed by a special translate-table which produces FIO/DEC 8-bit or 6-bit (concise) codes.

The following diagram gives the (rather arbitrary) correspondence that this table sets up between graphics and control functions peculiar to the FIO/DEC hardware and TTY graphics with their representations as System/360 "EBCDIC" codes.

FIO/DEC		CODE		TTY	MTS
Graphic	Meaning	FIO/DEC (octal)	Concise (octal)	Graphic	"EBCDIC" Code (hex)
~	('not')	203	03	!	5A
⊃	('implies')	004	04	$	5B
∨	('or')	205	05	#	7B
∧	('and')	206	06	&	50
↑		211	11	↑	AA
‾	(non-spacing overbar)	256	56	C-S-K	27
|	(non-spacing vertical)	256	56	C-S-K	27
.	(non-spacing dot)	040	40	@	7C
_	(non-spacing underline)	040	40	_	6D
×	(multiplication)	073	73	*	5C
Control characters:					
lower case		072	72	;	5E
upper case		074	74	:	7A
backspace		075	75	\ (S-L)	BA
tab		236	36	TAB (C-I)	05
carriage return		277	77	RETURN	15
tape feed		000	00	LINE FEED	25
stop code		013	13	EOM (C-C)	03
black		---	34	CN FM	2B
red		---	35	ALT MODE	22

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*

THE UNIVERSITY OF MICHIGAN
CONCOMP PROJECT

2a. REPORT SECURITY CLASSIFICATION

2b. GROUP

3. REPORT TITLE

AN ASSEMBLY LANGUAGE SYSTEM FOR DEC MINICOMPUTERS

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Memorandum 20

5. AUTHOR(S) *(First name, middle initial, last name)*

V. Michael Powers
David L. Mills
Neal Laurance

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1969	62	7

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
DA-49-083 OSA-3050 b. PROJECT NO.	Memorandum 20
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Qualified requesters may obtain copies of this report from DDC

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Advanced Research Projects Agency

13. ABSTRACT

This memorandum describes the PDP-5/8 and the PDP-7/9 language assemblers and the PDP-8 Link-Editor/Loader which are currently running on the duplex IBM 360/67 system at the Computing Center of The University of Michigan under MTS (Michigan Terminal System). The programs are written in IBM System/360 OS Assembly Language, Level G. The memorandum serves both as a manual for the system user as well as a report on the system development.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
assemblers						
small computer assembler						
PDP-1						
PDP-5						
PDP-8						
PDP-7						
PDP-9						
linkage editing						
relocatable assemblies						
assembling DEC machine programs on an IBM system						