COMPUTATION CENTER
Massachusetts Institute of Technology
Cambridge 39, Massachusetts

TO: All programmers
FROM: J. H. Saltzer
DATE: September, 1964
SUBJECT: A BRIEF INTRODUCTION TO THE FAP LANGUAGE(1)

1. FAP is an assembly language for the 7094 developed at the Western Data Processing Center, at UCLA. It was originally conceived as an aid in writing FORTRAN-compatible machine language programs, and the FAP assembly program works within the FORTRAN monitor system. However, it is a complete assembly program in its own right, and has the independent subroutine ability which has proved valuable even when not writing FORTRAN subroutines.

Throughout this writeup it will be assumed that the reader is unaquainted with any assembly language, but is familiar with the operation of the 7094 computer and some of its instructions. Only an essential subset of the full FAP language is discussed here, but enough is said to permit writing complete accurate programs.

2. What is an assembler? An assembly program belongs to the class of programs known as system programs, that is, it is a program commonly used to aid in operating or programming the computer. Its purpose is to take as input a shorthand symbolic notation for a machine language program, and produce as output the binary machine language program for which the symbolic notation was a shorthand. (Note the similarity between figure 1 and figure 2.) For example, the 7094 binary machine instruction to add the contents of location 104 into the accumulator is (abbreviated here in octal):

```
040000000104
```

With the aid of an assembly program, it is possible instead to punch into a card the letters

```
ADD     ALPHA
```

The assembly program will look up in an operation table the binary machine operation code which corresponds to the symbolic mnemonic "ADD". It will take that binary code, insert in it the address obtained by evaluating the symbol "ALPHA" and punch out the resulting binary instruction on a card in a standard format which can be read back into the computer as an instruction.

---

(1) Reference Manual, IBM 7090/7094 Programming Systems, Fortran Assembly Program (FAP), C28-6235, April, 1964

In the early days of computers, assembly programs were not available, and programmers had to write out long strings of numbers to represent the instructions they were using.

General Computer Use

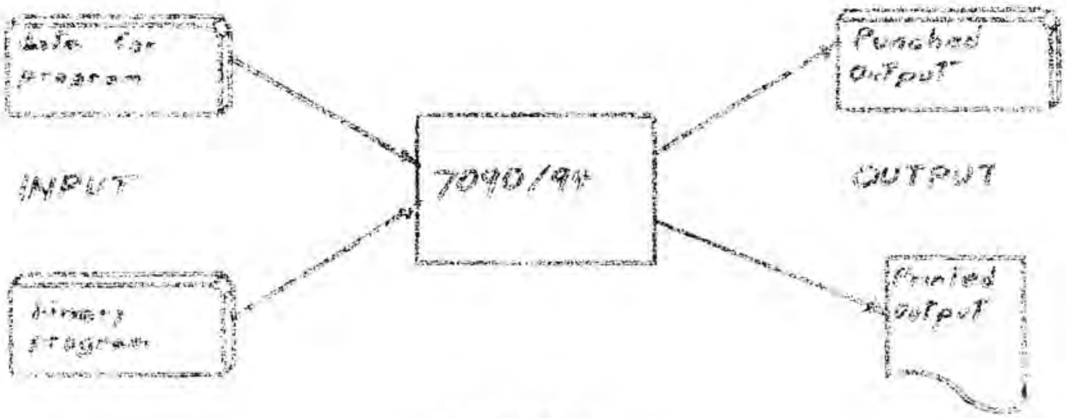

FIGURE 1

Use of Computer for Assembly

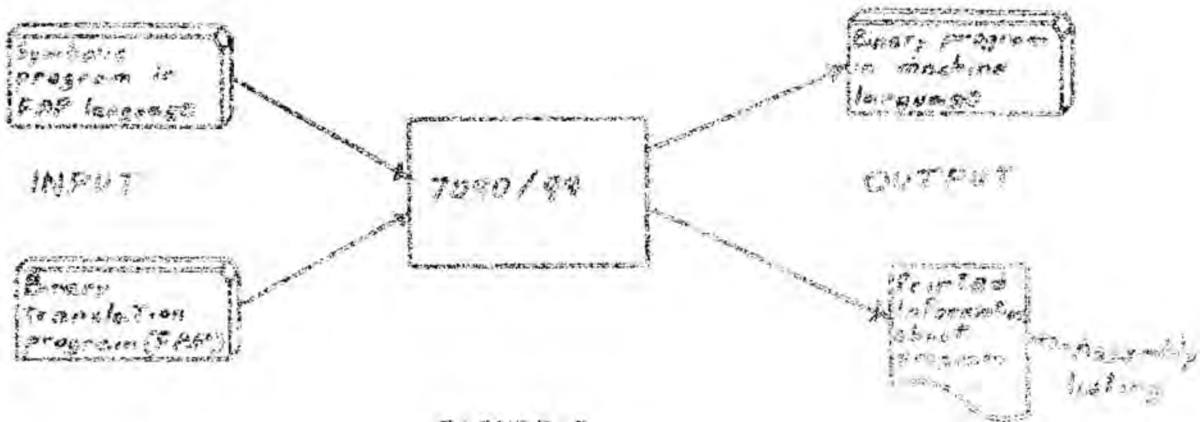

FIGURE 2

Since a string of numbers has very little mnemonic value for most people, the programmers of those days actually invented mnemonic names for their instructions and programmed in terms of these symbols. When they had completely written the program and were satisfied with it, they then rewrote it in terms of the binary numbers required by the computer, and these binary numbers were punched into paper tape or cards so they could be read into the computer.

The assembler, then, takes over this tedious, errorprone, second step of writing a program. It reads in symbols which have a mnemonic value to the programmer, and translates them into the binary machine instructions required by the computer.

However, once an assembler is available it can do other things beyond the simple substitution of binary machine codes for symbolic mnemonics. In fact, it can take over many of the tedious procedures which normally fall to the programmer. As commonly happens some of the other procedures which were assigned to the assembler rapidly became the most important reason for using it.

The particular procedure which can claim this merit is that of assigning values to symbols other than operation mnemonics. In writing a program, the programmer must decide as he writes each instruction into which location in memory it should go. Similarly, for each piece of data his program uses, he must assign a place in storage for the piece of data. Then, when an instruction should refer to that data, he can write in the proper numeric address for the instruction.

The difficulty with this procedure is not obvious at first, but comes to light almost immediately while actually writing a program. The difficulty is that if while studying a section of the program it is discovered that there is an error, and another instruction must be added to the sequence, there is no room for the the additional instruction. To make space for it all the instructions (and perhaps data) after that instruction have to be moved down one position in memory. While this move constitutes no great problem, another problem now arises. What of all the instructions which refer to instructions and data which have now been moved? Their addresses must be changed to correspond to the new locations of the instructions and data that they refer to.

The way around this difficulty is to leave the problem of assigning memory locations to the assembly program. The programmer simply writes his instructions down, one after the other in the order he desires them. Similarly, he places his data in the program in the desired order with respect to the instructions. When an instruction is to make reference to a piece of data (or another instruction), the programmer (since he now has no idea what the location of the piece of data will be) invents a symbol, and names the location of the piece of data with this symbol. He then <u>uses</u> this symbol as an address of the instruction which is making reference to the piece of data. The <u>value</u> of the symbol is unknown to the programmer, and it will remain unknown until the assembler begins working on the program.

The assembly program, then, is given the additional task of assigning each of the instructions and pieces of data to a memory location, thereby establishing the values of the symbols which may appear as names of the locations of various instructions and data. Then it may proceed with the process of replacing the operation mnemonic with the correct binary machine code, and it may evaluate the address of each instruction in terms of the values of the symbols it has previously established.

The Pseudo-operation

One more special feature of assembly programs will complete our discussion of them. The assembly program can, while in the process of looking up the proper binary machine code for each of the machine instruction mnemonics, check for certain special mnemonics intended to convey information to the assembler itself, rather than to be translated into a binary machine instruction. For example, the programmer may type the letters END into the operation field of a card, and make this card the last one in his program deck. The assembler then will examine each card it processes, to check for this special card. When it finds the END card, the assembler knows that there are no more instructions to follow in the program. The END card itself does not cause any instructions to be generated in the object program; it simply acts as a "note" to the assembler.

The letters END in the operation field are known as a pseudo-operation mnemonic; END is but one example. Eight other pseudo-operation mnemonics are described in section four; their effects on the assembly process are noted there. Some of these pseudo-operations do cause the generation of words in the assembled program in some special format; others more similar to the END card are simply notes to the assembler on some particular aspect of the assembly.

Review

We have seen, then, that the assembly program does three jobs for the programmer. First, it assigns his instructions locations in storage and defines symbols he has used. Second, using these symbol definitions and a standard table of operation table mnemonics, it translates each of his symbolic instructions into binary machine instructions and punches them out on cards in a format suitable for reading directly into the computer. Third, it looks for and recognizes several pseudo-operation codes which appear in the operation field, and considers these to be special notes to itself from the programmer; its operation is modified accordingly.

Now that we know what operations an assembly program is expected to perform, we may proceed with a discussion of how programs are written in a form suitable for FAP translation.

3. FAP-The language. In this section we will discuss the details of the FAP language, and the format of instructions written in the FAP language.

a. Symbolic Card Format

Instructions are punched one to a card in the following format:

1 — 6	7	8 — 14	15	16 — 72		73 — 80
symbolic location field		operation field		variable field	comment field	sequence field

Card columns 1-6 comprise the symbolic location field. Column 7 is always blank. Columns 8-14 are known as the operation field. Column 15 is blank, and the variable field starts in column 16. The variable field continues from column 16 until the first blank column is reached. After this first blank column may appear an arbitrary comment extending up to column 72. Columns 73-80 are commonly used for labelling and sequence numbering programs.

The contents of each symbolic card are copied onto the output <u>assembly listing</u>, along with the octal equivalent of any binary word generated by that card and the location assigned the binary word. Note that not all of the symbolic cards in a FAP program generate binary words; in case no binary word is generated, only the symbolic card is listed on the assembly listing.

<u>The Symbolic Location Field</u>

The location of an instruction or a piece of data may be named by placing a symbol in the symbolic location field of some card in the symbolic program. If a symbol appears on a card containing a machine operation, its value will be the location to which that machine operation has been assigned by the assembler. If it appears in the location field of a pseudo-operation, the discussion of the pseudo-operation must be read to determine which location has been named.

A symbol consists of one to six characters, which may include letters, numbers, parentheses, and the period. At least one of the characters must not be a number. The programmer is free to invent any names he likes within these restrictions. (Names are usually chosen for their mnemonic value.) He must be careful, however, to make sure he does not attempt to define the same symbol twice, by having it appear in the symbolic location field of two different instructions.

<u>The Operation Field</u>

The operation field may contain any one of the mnemonic codes corresponding to machine instructions described in the 7094 manual. Or, it may contain any of the pseudo-operation mnemonic codes described in section four. If it contains a 7094 instruction mnemonic the assembler will look up the proper binary operation code and insert it into the word assembled for that instruction. The operation of the assembler for the pseudo-operations should be checked in the description of the appropriate pseudo-operation.

The assembler recognizes a blank operation field as equivalent to 7094 machine instruction "HTR" and assembles a word with a zero operation code. The other fields are treated as in any other machine instruction. In this connection note that a blank card will cause the generation of a word of all zeroes in the assembled program.

<u>The Variable Field</u>

As its name implies, both the contents and the interpretation of the variable field change from instruction to instruction. For example, the variable field of a 7094 instruction mnemonic is interpreted as the name of a location in core storage. On the other hand the variable field of some pseudo-operations is interpreted as a piece of data for inclusion in the program.

In most cases, the variable field contains an expression and is intended to be interpreted as the name of a core storage location. (The interpretation of the variable field for the pseudo-operations is described in the descriptions of the individual pseudo-operations.) An expression consists of either a symbol, a decimal integer, a symbol plus a decimal integer or a symbol minus a decimal integer. The symbol, if present, must be the name of the location of some instruction.

An expression such as

```
ALPHA+5
```

is interpreted as the fifth location after the location named ALPHA. (Such expressions should be used with care and only in context-related situations, as when two related pieces of data are in consecutive locations.)

An expression such as

```
4
```

is interpreted as the fourth (absolute) location within the computer.

The special symbol "*" may be used in an expression in place of a defined symbol. Its value is taken to be the location of the instruction being assembled. Therefore, the * has a different value in each instruction which uses it. It may be used, for example, when an instruction refers to the next instruction in the program as in the following example.

```
STA    *+1
CLA    **
```

The second instruction in the example above illustrates the use of another special symbol, **. This symbol is taken to have the value zero, and is used when the programmer does not know the name of the location he wishes to operate upon. Instead his program will insert the correct location name into his CLA instruction before executing it. This operation is known as program modification. Obviously any legal expression could be used here, since the program will change it anyway; the double asterisk is a signal for the reader that program modification will occur. The special symbol ** is used primarily for the convenience of another person reading the program, so he may recognize those parts of the program which may be changed by the program itself.

Tags and Decrements

Numbers may be inserted into the tag and decrement fields of those instructions which may have tags and decrements, adding subfields to the variable field. A subfield is indicated by typing a comma at the end of the variable field, followed by an integer. This integer is inserted into the tag part of the instruction being assembled. A second subfield may be indicated by a comma following the first one. Again, an integer (or the symbol **, to indicate program modification) may appear, and it will be evaluated and inserted in the decrement part of the instruction being assembled.

Examples:

```
CLA   ARRAY,1   tag is 1.
TXI   LOOP,4,1  tag is 4, decrement is 1.
```

It should be noted that FAP permits somewhat more complicated expressions for the variable field address, tag, and decrement; however, the correct construction of these expressions is somewhat difficult. Since the simpler expressions described here will suffice for almost all situations, the more general facility of FAP may be left for future study, or the advanced reader.

Assembly

FAP begins assembling the program as though it would start in location zero. It assigns instructions, data words, and space for arrays to ascending locations in core storage in the order they appear in the symbolic deck. The resulting binary instructions are punched in a relocatable column binary format suitable for loading into the 7094 computer by the FORTRAN monitor system and the BSS loader.

4. The Pseudo-operations

One of several pseudo-operation mnemonics may appear in the operation field of a card. These pseudo-operations can be placed in one of five classes: list control, data-generating, storage allocating, symbol defining, and organizational. These classes will be discussed in order.

a.) List Control Pseudo-operations

The list control pseudo-operations have no effect on the assembled program. Instead, they are used to control the printed assembly listing, to make it more understandable to the reader. Only one list control pseudo-op is of general enough interest to mention.

REM (arbitrary remark)

constituents:

1. The letters REM in the operation field.
2. An arbitrary remark starting after column 11.

The REM pseudo-op is used to introduce an arbitrary remark into the assembly listing. The entire card, with the exception of the operation field (which contains the letters REM) is printed on the output listing. No binary instructions are assembled, and no symbols defined.

Example:

```
REM    SECTION TO CALCULATE CORRELATION
```

Note that two different ways are provided for the programmer to introduce comments into his listing. This is because machine language instructions do not usually reveal anything about the techniques being used, and comments are needed to guide a reader (or even the author, after a couple of weeks) through the program. It is therefore worthwhile to scatter meaningful comments liberally throughout the program.

b.) Data Generating Pseudo-operations

The data generating pseudo-ops are used to introduce into the program registers containing those constants which are needed by the program.
DEC (Decimal data item)
constituents:

1. A name may appear in the symbolic location field.
2. The letters DEC in the operation field.
3. An integer or real constant in the variable field.

DEC is used to introduce integer and floating point (real) constants into a program. If the variable field contains an integer constant, a word would be assembled which contains that integer, in binary form. If the variable field contains a floating point (real) constant a word will be assembled which contains that floating point number, in the proper format for floating point machine operations. The definitions of integer and real constants are the same as in the MAD or FORTRAN languages.

Examples:

symbolic coding			octal result
TEN	DEC	10	000000000012
CONST	DEC	10.425	20[illegible]15463146
TWO	DEC	2.0	202400000000

OCT (Octal data item)

constituents:

1. A name may appear in the symbolic location field.
2. The letters OCT in the operation field.
3. An octal constant of up to 12 digits (preceeded by a sign, if desired) in the variable field.

OCT is used to introduce an octal constant into the program. A word is assembled which contains the value of the octal constant in the variable field. If the variable field contains fewer than 12 digits, the octal number is right justified within the word. The sign, if present, is assembled into the sign bit of the word, and is equivalent to a 4 in the high order octal digit. A symbol, if any, appearing in the symbolic location field, is the name of the location of the octal data item.

Example:

symbolic coding			octal result
SIZE	OCT	1756425	000001756425
LTH	OCT	-5	400000000005

BCI (Binary coded decimal information)

constituents:

1. A name may appear in the symbolic location field.
2. The letters BCI in the operation field.
3. The digit "1" followed by a comma, followed by six characters of alphanumeric information in the variable field.

The BCI pseudo-operation is used to encode letters and numbers in the standard BCD code, and insert those codes into the assembled program. The six characters (including blanks and commas) following the comma are converted to BCD and the resulting word is inserted in the program. A symbol, if any, appearing in the location field, is the name of the location of the BCD word.

Example:

symbolic coding			octal result
TOWN	BCI	1,NYC	457023606060
NAME	BCI	1, JIMMY	604131444470

c.) Symbol Defining Pseudo-operations

In addition to the usual procedure for assigning names to locations by placing them in the symbolic location field of some instruction, a name may be assigned by the SYN pseudo-operation.

SYN (Define synonymous symbol)

constituents:

1. A symbol in the symbolic location field.
2. The letters SYN in the operation field.
3. An expression in the variable field.

An expression in the variable field of the SYN pseudo-op is assumed to be the name of some location in the computer, say location A. The symbol in the symbolic location field is assigned as the name of location A. No binary words are generated or inserted in the program. SYN is commonly used to make two symbols (perhaps provided by different programmers) synonymous. Thus the same location may have two or more names.

Example:

symbolic coding			octal result
A	SYN	B	none
Q	SYN	ALPHA+25	

restriction:

Any symbol appearing in the variable field of an SYN pseudo-operation must be "previously defined". That is, it must appear in the symbolic location field of a card earlier in the deck.

Discussion: Note carefully the difference in the following two situations.

```
1.          CLA    ALPHA               2.          CLA    BETA


ALPHA       DEC    5                   BETA   SYN    5
```

In the first, ALPHA is the name of the location of the decimal integer "5". At execution time, the CLA instruction will therefore cause the decimal integer "5" to be brought into the AC. In the second, BETA is the name of location five, and if used as an address, will cause reference to location 5. The CLA instruction therefore will bring the contents of location 5 into the AC. This difference illustrates that one must carefully distinguish between the name of a storage location and the name of the contents of a storage location.

d.) Storage Allocating Pseudo-operations

In some programs, it is desirable to set aside a section of core storage for an array of numbers to be computed by the program. If this storage space is desired in a program, some way is needed to inform the assembler that it should not place any assembled instructions or data in the area. The storage-allocating pseudo-operations are used to accomplish this.

BSS (Block of storage started by symbol)

constituents:

1. A name may appear in the symbolic location field.
2. The letters BSS in the operation field.
3. A decimal integer in the variable field.

The BSS pseudo-operation causes a block of storage cells equal in length to the value of the integer in the variable field to be set aside. Any symbol in the symbolic location field is the name of the location of the first cell in the block.

"Setting aside" of a block of storage is evidenced by the fact that the next instruction after the BSS will be assigned a location after the block: the cells in between will have no particular binary number assigned to them.

Example:

symbolic instruction	octal result
ARRAY BSS 10	none

e.) Organizational Pseudo-operations

The organizational pseudo-operations are used to indicate important features of the program to the assembler, and pertain to the entire program rather than to a single instruction. As such, they must appear in specific places within the program.

END (End of the program)

constituents:

1. The letters END in the operation field.

The END pseudo-operation marks the physical end of the program, and therefore, must be the last card in the deck.

Example:

symbolic coding	octal result
END	none

COUNT (length of program)

constituents:

1. The letters COUNT in the operation field.
2. An integer in the variable field.

The assembly can be made more efficient if the assembly program knows the approximate number of cards to expect in the program being translated. Therefore, this optional pseudo-operation should appear as the first card of FAP programs. The number of cards indicated need not be exact. If no estimate is given, FAP will assume 2000 for the count, and give an appropriate comment on the assembly listing.

Example:

symbolic coding	octal result
COUNT 150	none

ENTRY (Entry point)

constituents:

1. The letters ENTRY in the operation field.
2. A symbol in the variable field.

In a subprogram which is to be used by another program, the ENTRY pseudo-operation indicates the first instruction which is to be executed in the subprogram when it is called.

The entry card has three functions:

1. It defines this program to be a subroutine.
2. It defines the name of the subroutine to be the symbol appearing in its variable field.
3. It indicates the location within the program at which the first instruction to be executed may be found.

The symbol appearing in the variable field must be a name which appears in the symbolic location field of some instruction within the program. If the ENTRY pseudo-operation appears, it must be placed at the beginning of a program and may be preceded only by a COUNT card. Two or more entry points to the same program can be indicated by two or more ENTRY cards following the COUNT card.

Example

symbolic coding	octal result
ENTRY COS ENTRY PHI	none

5. Error messages. One bonus which may be obtained when using an assembly program is that the assembler can look for certain standard types of errors and inform the programmer of them. FAP distinguishes between two types of errors, those which make

it impossible to assemble the program correctly, and those which can be assembled, but are probabbly slips by the programmer. All detected errors are indicated to the programmer by the presence of a letter at the left edge of his assembly listing adjacent to the instruction in question.

<u>Fatal error indicators</u>:

letter used	error made
U	An undefined name has been used in this instruction in the variable field. The assembler does not know what location corresponds to the name.
M	This instruction uses (or defines) a symbol which has been defined more than once in the program. The assembler does not know which definition to use.
O	The operation field of this instruction contains a mnemonic unknown to FAP.
E	The address field of this data-generating pseudo-operation contains an error.

<u>Non-fatal error indicators</u>:

F	This SYN pseudo-operation contains a symbol which has not yet appeared in a symbolic location field (e.g., it is not previously defined.) This error does not become fatal until another instruction attempts to use the symbol defined by the SYN pseudo-operation.
A	This instruction is expected to have an address and the programmer has not provided one. (Or it is not expected to have an address, and the programmer has provided one.)
T	Same as A except applies to the tag field.
D	Same as A, except applies to the decrement field.

Certain of the more sophisticated features of the FAP language are carefully checked, and appropriate error indicators are printed. Occasionally, an error when using a simple feature will appear to the assembler to be an error in use of one of its bells or whistles, and some rather obscure indication may be made. In these cases, the difficulty is usually obvious from an inspection of the instruction in question.

Sample program. Fills an array with a constant number.

```
*       FAP
        COUNT    8
        CLA      WORD           GET CONSTANT INTO AC.
        AXT      25,4           SET UP FOR LOOP 25 TIMES.
        STO      ARRAY+25,4     INSERT IN CURRENT ARRAY POSITION.
        TIX      *-1,4,1        INDEX, AND GO TO NEXT POSITION.
        HPR                     ALL DONE, STOP.
ARRAY   BSS      25             SPACE FOR NUMBERS.
WORD    DEC      15             CONSTANT.
        END
```

ASSEMBLED PROGRAM

LOCATION	OCTAL WORD
0	050000000036
1	077400400031
2	060100400036
3	200001400002
4	042000000000
.	.
.	.
.	.
36	000000000017

SYMBOL DEFINITIONS

SYMBOL	VALUE
ARRAY	5
WORD	36

GUIDE TO THE 7094 MANUAL (A22-6703-1)

For study purposes on the first pass through the 7094 manual, the following list of relevant sections may be helpful.

SYSTEM DESCRIPTION.

INSTRUCTION DESCRIPTIONS

CLA	LDQ	ALS	TOV
ADD	STQ	ARS	CAS
SUB	MPY	LLS	XCA
STO	DVP	LRS	

FLOATING POINT INSTRUCTIONS.

FAD FDP FMP FSB

INSTRUCTIONS FOR MANIPULATING LOGICAL WORDS.

CAL	LGL	RQL
SLW	LGR	LAS

TEST AND BRANCH INSTRUCTIONS.

TMI	TZE	TRA	PBT
TPL	TNZ		

INDEXING INSTRUCTIONS.

LXA	PAX	TIX	TXH
SXA	PXA	TXI	TXL
TSX			

OTHER USEFUL INSTRUCTIONS.

CHS	SSP	STA	STD
STL			

LOOK AT THE APPENDIXES TO SEE WHAT MATERIAL IS THERE.

BIBLIOGRAPHY

I.B.M., 7094 Principles of Operation,

I.B.M. Corp., file no. 7094-01, form no. A22-6703-1. (The 7094 manual, needed for machine language programming).

I.B.M., 7090/7094 Programming Systems FORTRAN II Assembly Program (FAP).

I.B.M. Corp., file no. 7090-21, form C28-6235-2. (A complete description of the FAP language).

Saltzer, J. H., A 7090 Machine Language Programming Workbook and Laboratory, M.S. thesis, September, 1963, M.I.T., Department of Electrical Engineering.

(Introduction to fundamentals of symbolic machine language programming. Available in the Document Room, 26-270).

Corbato, F. J., Poduska, J. W., and Saltzer, J. H., Advanced Computer Programming, M.I.T. Press, 1963.

(Operation and construction of assembly program).