UNIVAC® 494 REAL-TIME SYSTEM

ASSEMBLER

PROGRAMMERS REFERENCE

UP-4133 Rev. 1

UNIVAC 494

ASSEMBLER
UP-4133 Rev. 1

UNIVAC SYSTEMS PROGRAMMING LIBRARY SERVICES **RELEASE**

UNIVAC 494 Library Memo 12 releases and announces the availability of the "UNIVAC 494 Real-Time System Assembler Reference Manual," UP-4133 Rev. 1, covers and 144 pages. This is a Standard Library Item (SLI).

The UNIVAC 494 Assembler is an integrated program which converts source code programs (UNIVAC 494 SPURT or UNIVAC 494 Assembler (ASM) language) to relocatable binary (RB) elements for use by the UNIVAC 494 Operating System. The Operating System collects all the independent RB elements required to produce an absolute object code program for execution.

The assembly language is a set of mnemonic statements which are directly converted to relocatable binary code. Also contained in the assembly language are directives which are instructions to the assembler to permit the user to define symbols for complex operations and to control the assembly process.

This revision reflects changes in the 494 Assembler, adds a description of pseudo-ops, and includes a description of the following assembly directives: END, BLOCK-DATA, XREF, EDEF, Expression (SLEUTH, BITARRAY), INPUT or INPUTFORM, LET, UNLIST, and LIST. In addition, three appendices have been added. These appendices describe the error flags that may appear in the listing at assembly time, depict the order in which various sources are searched in order to define symbols used in the operation field, and list the available options on the #ASM card.

A listing of the contents of this manual follows: 1. Introduction; 2. Computer Formats; 3. Source Language Format Requirements; 4. Basic Assembler Language Instructions; 5. Assembly Directives; 6. PROC, FUNC, and Associated Directives; Appendix A, Abbreviations and Special Symbols; Appendix B, Fieldata and Card Codes for Character Representation; Appendix C, Assembler/SPURT Function Codes; Appendix D, Error Flags; Appendix E, Operation Field Hierarchy; and Appendix F, #ASM Options.

Destruction Notice: This manual, UP-4133 Rev. 1, supersedes and replaces "UNIVAC 494 Real-Time System Assembler Reference Manual," UP-4133, issued on Library Memo 7 dated October 21, 1966. Please destroy all copies of UP-4133 and/or Library Memo 7.

(See Reverse)

TO LISTS: 10U, 217, 630, and 692 Library Memo 12 only.	ATTACHMENTS: UP-4133 Rev. 1 plus Library Memo 12 to S.P.L.S. lists 43 and 44.	THIS SHEET IS: 494 Real-Time System Library Memo 12
		DATE: November 10, 1967

Distribution of this manual, UP-4133 Rev. 1, and/or the Library Memo 12 is being made as indicated below. Additional copies may be ordered via a Sales Help Requisition through your local Univac Manager, from Holyoke, Massachusetts.

MANAGER
Systems Programming Library Services

CONTENTS

FIGURES

1. INTRODUCTION

The UNIVAC 494 Assembler is an integrated program which converts source code programs (UNIVAC 494 Assembler (ASM) language) to relocatable binary (RB) elements for use by the UNIVAC 494 Operating System. The operating system collects all the independent RB elements required to produce an absolute object code program for execution. This manual describes use of the assembly language. (For references in this manual to SPURT assembly language, see "UNIVAC 494 SPURT Reference Manual," UP-4090 (current version).)

The assembly language is a set of mnemonic statements which are directly converted to relocatable binary (RB) code which is accepted as instruction input by the UNIVAC 494 Operating System. Also contained in assembly language are directives which actually are instructions to the assembler to permit the user to define symbols for complex operations and greatly expand the power of the assembler. The term "relocatable binary" refers to the values assigned by the assembler to symbols representing storage areas, instructions, and constants. These values will later be changed by the operating system to provide the absolute machine addresses required for execution. Thus,the relative addressing feature of the central processor which optimizes use of core storage in conformance with the requirements of a real-time system is fully utilized.

A side-by-side listing of basic source language instructions and machine instructions is provided. Directives may not require code generation. Errors detected by the assembler in the use of source language are flagged.

2. COMPUTER FORMATS

2.1. GENERAL

This section describes the computer data and instruction formats of interest to the programmer. (See "UNIVAC 494 Central Processor General Reference Manual," UP-4049 (current version) for a more detailed discussion of central processor hardware.)

2.2. DATA FORMATS

Data (operand) formats are of four distinct types: (1) single precision integer, (2) double precision integer, (3) Fieldata or decimal, and (4) exponential or floating point.

2.2.1. Single Precision Integer Word

The fundamental level of storage is the single precision (30-bit) integer word. This word contains 30 binary bit positions as shown in Figure 2–1. Each of these bit positions represents a binary value of 1 or 0. The highest order bit (bit 29) uses a 0 bit to represent a positive value; a 1 bit for a negative value.

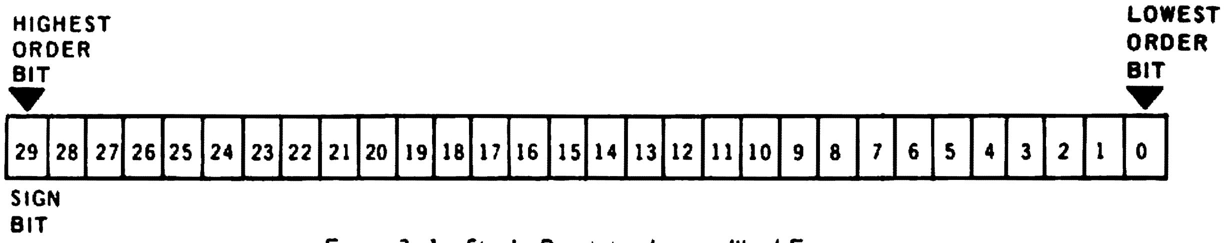


Figure 2–1. Single Precision Integer Word Format

Values may be expressed in binary notation for which the base is 2 instead of 10. The following equivalence exists:

BINARY	DECIMAL
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10
1011	11
. . .	. . .
10110101	181

The use of binary digits to represent large values is cumbersome. The use of octal notation for which the base is 8 is used for convenience. The following equivalence exists:

BINARY	OCTAL
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	10
. . .	. . .
111111	77
1100101	145

Binary values may be converted to octal notation by starting from the least significant (rightmost) digit. Each group of three binary digits is expressed as a digit from 0 to 7. By this method:

$$1\ \underbrace{100}\ \underbrace{101} = 145 \text{ (octal)}$$

$$\underbrace{111}\ \underbrace{101}\ \underbrace{000} = 750 \text{ (octal)}$$

A computer word containing 30 binary bits could be expressed in octal notation as:

7 7 7 7 7 7 7 7 7 7

Negative integer numbers are represented as the ones complement of positive numbers. A value of –3 is represented as:

7 7 7 7 7 7 7 7 7 4

The assembler will accept both octal and decimal numbers. To indicate a decimal number, a "D" is placed at the right end of the number; otherwise, it is assumed to be octal. Thus, 11D and 13 are equal. When the contents of a computer word are displayed, or if reference is made to a computer instruction word, octal notation will be assumed.

Most arithmetic instructions permit use of a half-word (15-bit) operand. If the lower half is specified, then bits 14 through 0 make up the operand with bit 14 used as a sign indicator. If the upper half is specified in the instruction, bits 29 through 15 make up the operand with bit 29 used as the sign indicator.

2.2.2. Double Precision Integer Word

The double precision integer word requires two successive memory addresses for storage or two 30-bit arithmetic registers combined for the 60 bit positions required. The format of this operand is shown in Figure 2–2.

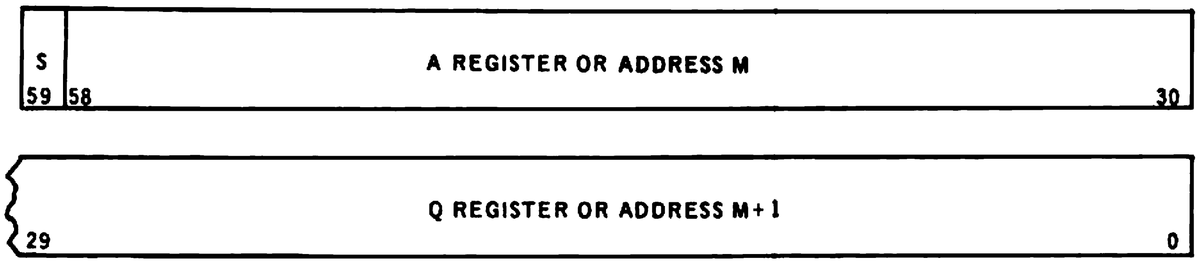


Figure 2–2. Double Precision Integer Word Format

The double precision format permits arithmetic operations upon operands having 59 significant bits where the single precision format permits operations upon operands having 29 significant bits.

2.2.3. Decimal (BCD) Word

The decimal or binary coded decimal (BCD) format permits arithmetic operations upon digits which are BCD encoded within a six-bit character code such as Fieldata code. The decimal word requires two successive memory addresses for storage or two arithmetic registers combined for the 60 bit positions required. The format of the decimal word (Figure 2–3) permits use of ten decimal digits, each of which represents a decimal digit 0 through 9. This format, just as the two preceding word formats, represents a fixed point number – no provision is made for a decimal point.

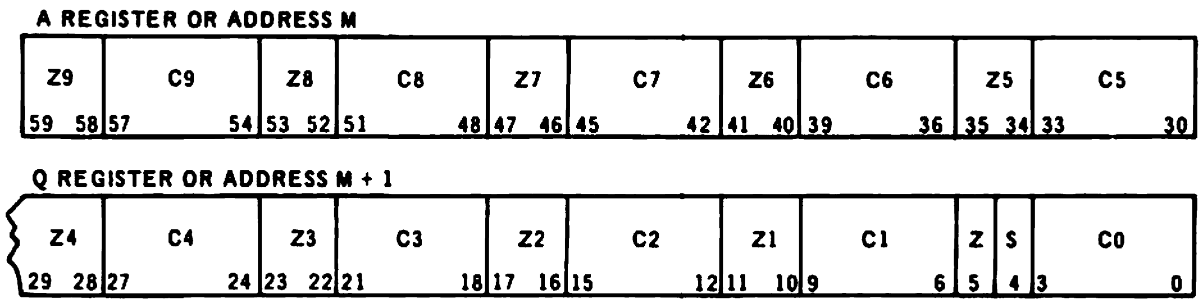


Figure 2–3. Decimal Word Format

Decimal operands are used when inputs arrive as a succession of 10 six-bit characters in conformance with a code similar to the Fieldata code. The "Z" (zone) bits shown in the format are arbitrary and are determined by the code itself, playing no part in the arithmetic operation. They are unchanged by the arithmetic operation. However, the lowest-order digit must indicate the sign in its fifth bit: a 1 for positive, a 0 for negative. The "C" (character) fields, which actually represent the binary-coded decimal (BCD) digit, must be encoded as shown in Table 2–1. No other encoding is acceptable for the BCD arithmetic instructions in the instruction repertoire. A positive decimal operand is exactly the same as the negative decimal operand (having the same absolute value), except for the sign bit.

DIGIT	CODING	DIGIT	CODING
0	0000	5	0101
1	0001	6	0110
2	0010	7	0111
3	0011	8	1000
4	0100	9	1001

Table 2–1. BCD Coding

2.2.4. Exponential (Floating Point) Word

The exponential format (Figure 2–4) is used if the computer must "remember" where the decimal point is located in a series of operands, when each of the operands may have the decimal point in a different position. This format permits storage of numbers of high magnitude. The operand is expressed as a fraction (fixed point part) multiplied by 2^n, where n is the exponent shown in the format. This exponent is always an integer.

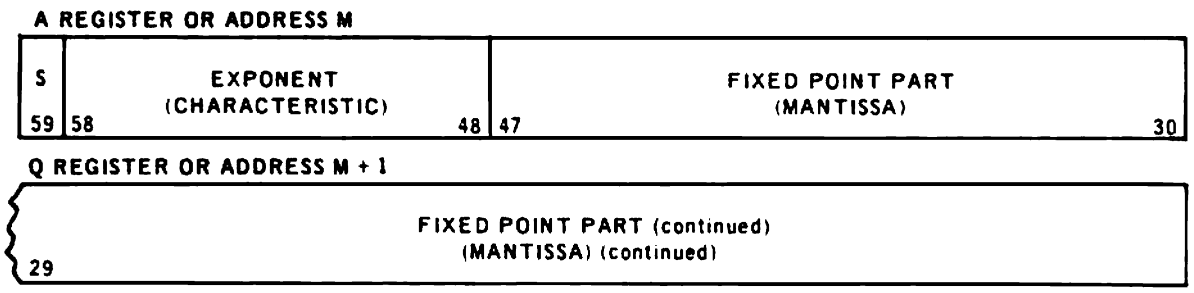

A REGISTER OR ADDRESS M
S
EXPONENT (CHARACTERISTIC)
FIXED POINT PART (MANTISSA)
59
58
48
47
30
Q REGISTER OR ADDRESS M + 1
FIXED POINT PART (continued) (MANTISSA) (continued)
29

Figure 2–4. Exponential (Floating Point) Word Format

Two successive memory addresses and/or the AQ register are required for floating point operations. The sign bit represents the sign of the fixed point part and is a 0 for positive, a 1 for negative. A 1 bit is arithmetically added to the highest order bit of the 11-bit exponent, thereby biasing the exponent by 2^{10}. This bias eliminates the need for a sign but limits the exponent to a value less than $+1024_{10}$ and greater than -1025_{10}. The fixed point part is usually "normalized" (shifted left until its highest order bit is different from the sign bit, with the exponent adjusted accordingly) so that it represents a fraction between 0.5 and 1.0. The number zero is represented as all zeros in both characteristic and mantissa.

2.3. ADDRESSING

Each word within the computer has a unique address. If it is a word that requires successive 30-bit words in storage,it is addressed by the first of the consecutive addresses. Addresses available to worker programs range from 00000 to 77777_8. This leaves a maximum of 15 bits for worker program addressing. However, the relative addressing feature of the central processor adds an increment – making for a 17-bit address – so that a program or parts of a program can be allotted to different memory areas for optimum use of core memory. This relative addressing is under executive control and in no way alters the program as written by the programmer.

2.3.1. Data Addressing

Data is addressed by instructions that are themselves contained in the memory of the computer. When it is required to access data to complete an instruction, the instruction will contain an address portion capable of containing a maximum value of 77777.

2.3.2. Instruction Addressing

A basic computer instruction is contained in one computer word. An instruction is accessed from memory, analyzed by the computer,and then executed. The next instruction is then accessed at the next sequential location unless a new sequence is specified.

2.3.3. Standard (Fixed) Locations

Some memory locations serve as entrances to input/output control, fault procedures, and executive routines not available to worker programs. Access to these addresses is gained by interrupts. The contents of these addresses are loaded at computer initialization time and supervised by the operating system.

2.4. INSTRUCTION FORMATS

Three different instruction formats are used for worker programs. The normal instruction format and the 77 (extended repertoire) instruction format are shown in Figure 2–5. Discussion of the third type – the I/O instruction – is beyond the scope of this manual.

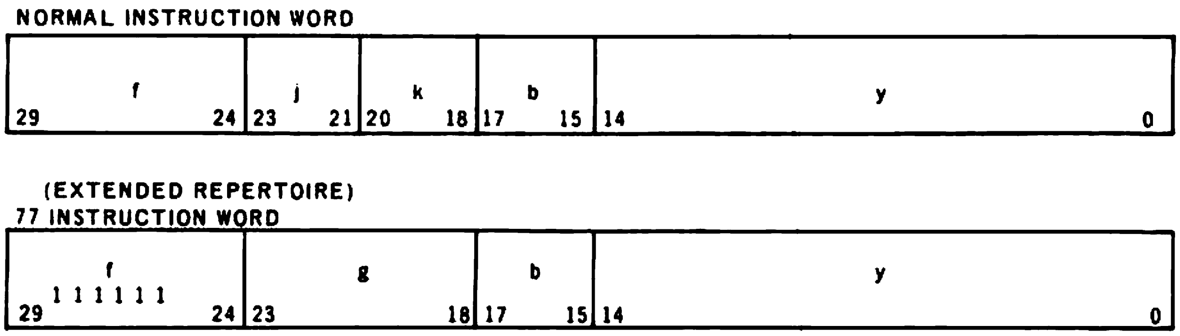
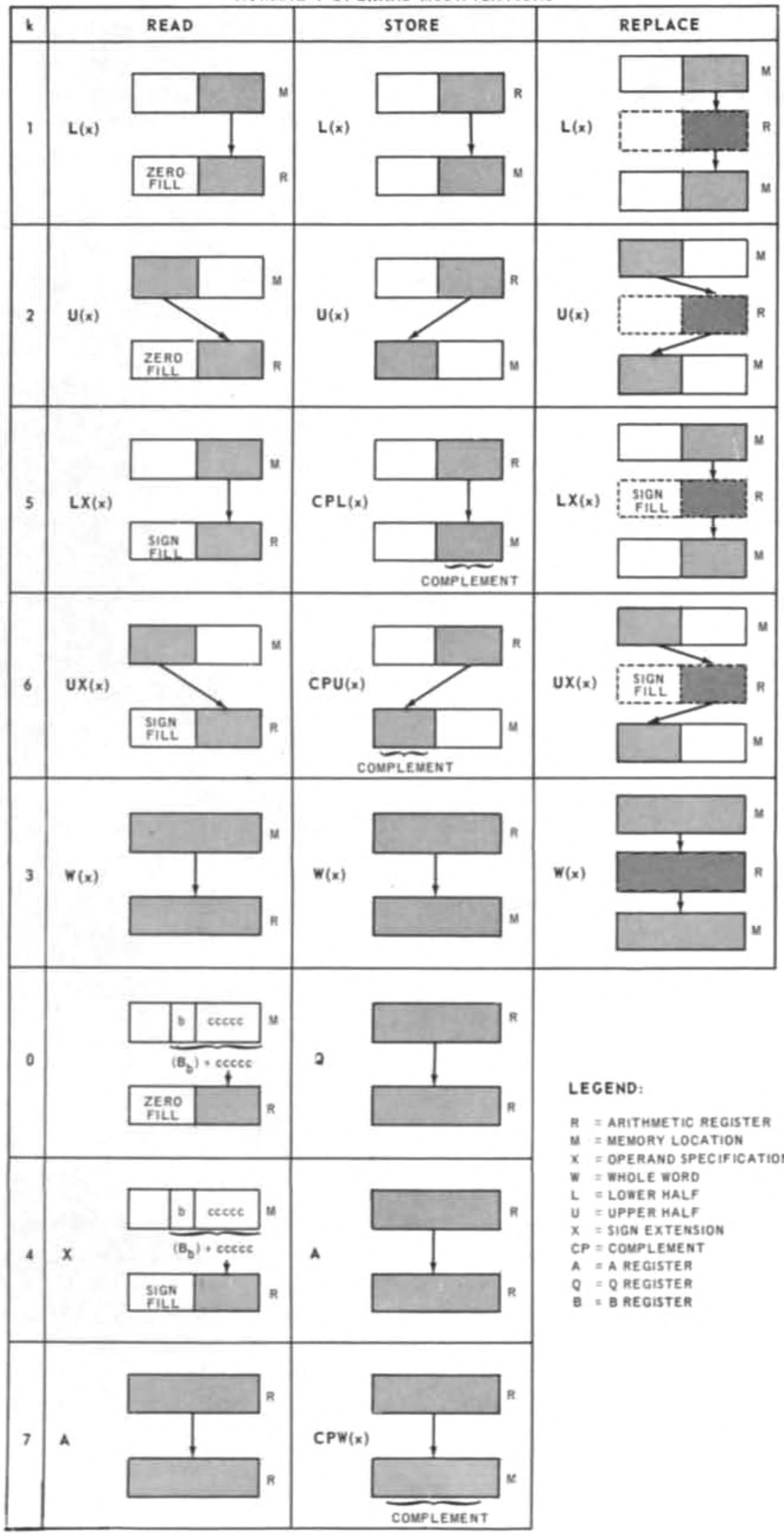


Figure 2–5. Instruction Word Formats

2.4.1. Normal Instruction Word

The normal instruction word is applicable to all instructions of the machine instruction repertoire except the 77 instructions. The term "normal" has no particular significance except to indicate that its f designator (described in the next paragraph) may be any octal number between (but not including) 0 and 77. These instructions are further subdivided into three classes: 1) read class instruction, which transfers data from core storage to an appropriate register, 2) store class instruction, which transfers data between registers or stores the contents of a register in core storage, and 3) replace class instruction, which replaces the data from core storage with the result of an operation performed upon the data. A replace class instruction is actually a combination of read class and store class instruction. The class of an instruction conditions interpretation of its designators, as shown in Table 2–2.

NORMAL Y OPERAND MODIFICATIONS



k	READ	STORE	REPLACE
1	L(x)	L(x)	L(x)
2	U(x)	U(x)	U(x)
5	LX(x)	CPL(x)	LX(x)
6	UX(x)	CPU(x)	UX(x)
3	W(x)	W(x)	W(x)
0		Q	
4	X	A	
7	A	CPW(x)	

LEGEND:

R	= ARITHMETIC REGISTER
M	= MEMORY LOCATION
x	= OPERAND SPECIFICATION
W	= WHOLE WORD
L	= LOWER HALF
U	= UPPER HALF
X	= SIGN EXTENSION
CP	= COMPLEMENT
A	= A REGISTER
Q	= Q REGISTER
B	= B REGISTER

Table 2–2. Designator Interpretation

The different designators of the instruction word are:

f A six-bit code (two octal digits) that specifies the basic operation to be performed.

j A three-bit code (one octal digit) which further defines the operation to be performed, thus extending the power of the particular operation. Depending on the basic operation to be performed, the j designator is interpreted as a skip designator, register designator, or repeat modification designator. Its most common interpretation is as a skip designator. For example, an add to A (20) with the inclusion of a j designator becomes an add to A with a test for the condition indicated by the j and a possible skip of the next instruction if the condition specified is met.

k A three-bit code (one octal digit) that further defines, together with the class of instruction, the source, form, size, and destination of the operand used by the instruction. The interpretation of the k designator is different for the read, store, and replace class categories of instructions. The k designator specifies an operand to be in the upper or lower 15 bit positions of a computer word, the entire 30 bit positions of a computer word with or without additional modification, or lower 15 bits of the instruction word itself. As an example: a read class instruction with a k designator of 0 or 4 uses the effective operand (see $\overline{y}$, following) in the instruction; a read class instruction with k designator of 1,2, or 3 uses, respectively, the lower 15 bits, the upper 15 bits, or the entire 30 bits of the word (or instruction) at an address; a read class instruction with k designator of 7 uses the word in the A register (accumulator) as the operand.

b A three-bit code (one octal digit) in the range of 0 thru 7 which specifies the B (index) register containing the value that is added to y to form the effective operand or operand address. It is a nondestructive modification of the y portion of an instruction forming an effective address but the y of the instruction remains unchanged. Fourteen addressable B registers (index registers) are included in the hardware for operand address modification, index code, and modifier incrementation. Of these, seven are available to worker programs and seven are reserved for exclusive use by executive control. The B registers can operate in one of two modes as designated by the internal function register (IFR). B registers are generally used in the 15-bit operational mode by worker programs. An additional 17-bit operational mode is available for B registers 4,5,6, and 7 of each group, specified under executive control, but most worker programs will only be allowed the 15-bit operational mode; the 17-bit operational mode being reserved for real-time programs and common subroutine communications.

In general usage, B registers 1 thru 7 are 15-bit registers that can be incremented or decremented under program control. Register B7 has the additional function of holding the repeat count during execution of the repeated instruction. Register B6 has the additional function of modification of the y portion of the repeated instruction when the instruction is a replace class instruction.

B0 is not a hardware register but functions as a B register containing the number +0. It can never be entered with a value but can be used in store and compare operations.

y A fifteen-bit value (five octal digits) used to form the effective operand or operand address of the instruction. It can also be used to form the shift count, repeat count, and compare value. It is the portion of the instruction modified when B register modification is specified.

$\bar{y}$ The relative address (relative to an RIR of zero) or effective operand, formed by the addition of y to the contents of the B register specified by the b designator. Both numbers are treated as unsigned positive numbers. The addition is always performed in the end-around-carry mode: any carry generated at the highest order bit position is carried around for addition at the lowest order bit position. This addition is normally a 15-bit addition but may, under executive control, be a 17-bit addition. It is not possible to generate an effective address or operand of all binary zeros unless y is zero and the contents of the referenced B register is zero (or B0 is referenced). The following examples illustrate operation of this addition (using octal notation):

```
                 15-BIT ADDITION          17-BIT ADDITION

   y                  77777                     77777
 (Bb)                 00001                    300001
                      -----                    ------
                      00000                    000000
                          1                         1
                      -----                    ------
   ȳ                  00001                    000001
```

Table 2–2 presents a general summary of the uses of the k designators; Table 4–2, the j designators. Designators are shown together with their corresponding mnemonics in source code. (A more detailed description of designator application is presented in Section 4.)

2.4.2. 77 (Extended Repertoire) Instruction Word

In the format for the "77" instruction word, the f designator is 77_8. Following the 77 is a two-digit (octal) g designator, which, together with the f designator, actually defines the function to be performed. The b and y designators are similar to their counterparts in the normal instruction word. No j and k designators can be used in the 77 instruction word.

3. SOURCE LANGUAGE FORMAT REQUIREMENTS

3.1. GENERAL

This section describes assembly language elements and format requirements for coding program lines on the standard coding form (see Figure 3–1). The symbolic language fields are described in 3.2; line control and comments in 3.3; data word generation in 3.4; and basic elements, expressions, and operators in 3.5.

3.2. SYMBOLIC LANGUAGE FIELDS

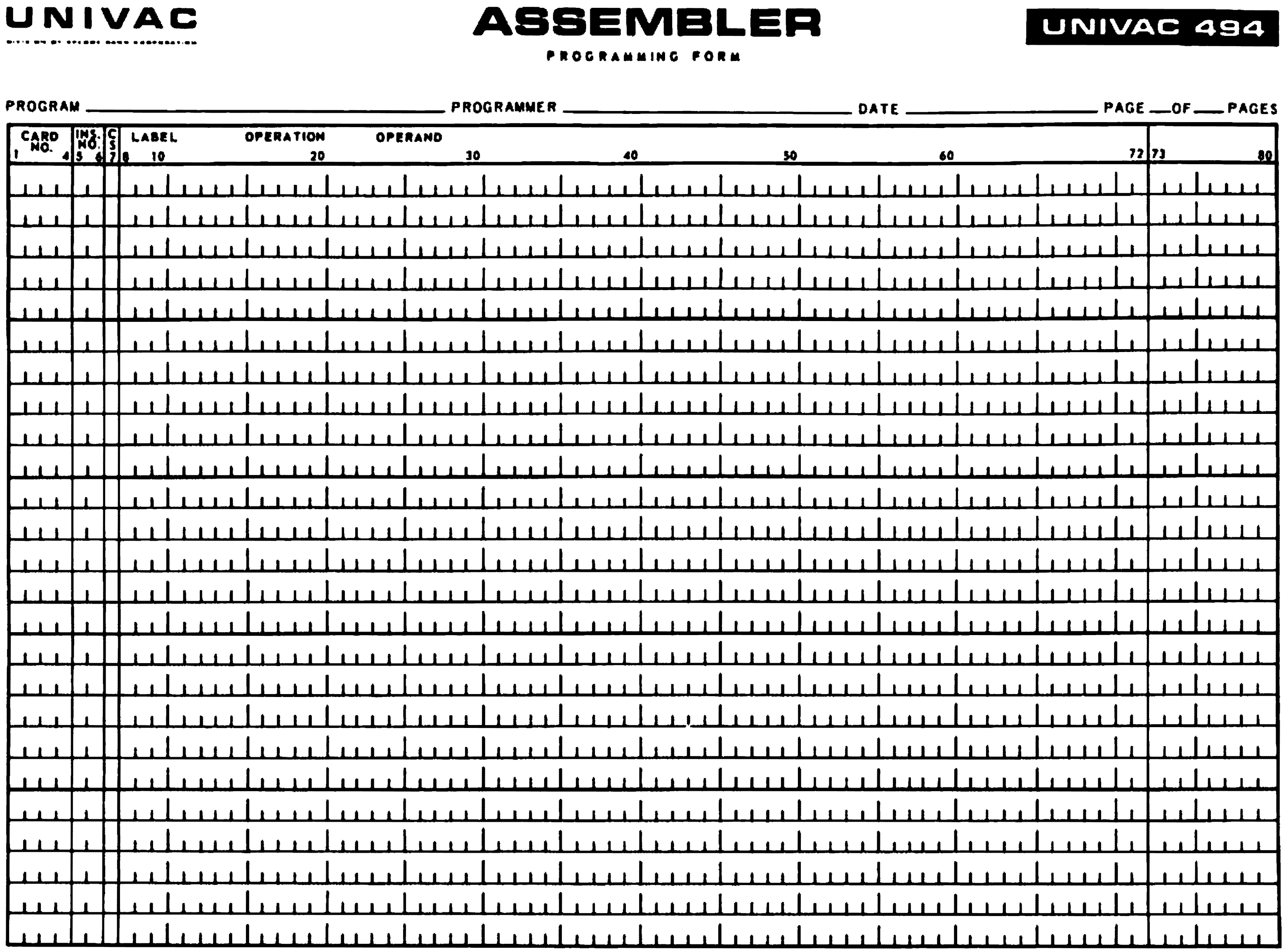
The basic line of coding is made up of three or fewer fields. These are the label field (not always required and sometimes prohibited), the operation field (always required), and the operand field (not always required). Fields are separated from each other by at least one space. Each field may be subdivided into subfields. A subfield is an expression which is terminated by a comma (followed by one or more spaces), except if it is the last subfield. In this case, a space terminates both the subfield and field.

Columns 1–4 represent the card number and provide an external sequencing criterion. Columns 5–6 represent the insertion number and are treated as the low order digits of the card number to enable sequential insertion. Normally these are not initially assigned. Column 7 contains a minus (–) to indicate that this line represents a continuation of the preceding line of coding. If not a continuation, column 7 of the line is usually left blank. Column 8 is the start of the label field. From this point on, the assembly language uses a free field format for operation and operand field(s). A space followed by a period (before column 80) after the last subfield terminates the scan of the line or column 80 terminates the scan of line (except if it is continued to the next line). Notes may be printed between the period and column 80. These notes will be presented on a printout of the program and in no way affect execution of the program. Notes may be used at significant points in the program as an aid to debugging. Additional notes may be written after column 80 for aid to the programmer but these will not be printed out. Notes may be continued to the next line if a period and blank are placed in columns 8 and 9, respectively,of the next line.

For the basic instructions which are described in Section 4, the correspondence of the subfield information with the designator information required in machine code (Figure 2–5) is shown in the following table.

	OPERATION FIELD		OPERAND FIELD
General Instruction	f,k	ƀ	y,b,j
B Register Instruction	f,k	ƀ	j,y,b
77 Instruction	fg	ƀ	y,b

NOTE: Both General and B Register instructions use Normal machine instruction format (Figure 2–5).



UNIVAC
DIVISION OF SPERRY RAND CORPORATION

ASSEMBLER
PROGRAMMING FORM

UNIVAC 494

PROGRAM ____ PROGRAMMER ____ DATE ____ PAGE __ OF __ PAGES

CARD NO. 1–4	INS. NO. 5–6	CS 7	LABEL 8 10	OPERATION 20	OPERAND 30 40 50 60 72	73 80

Figure 3–1. UNIVAC 494 Assembler Coding Form

The f (or fg) designator must always be written as the mnemonic representation of the operation to be performed (Section 4).

The j designator can be written as a number 0 through 7, a valid mnemonic representation (Section 4), an expression which will be evaluated as a number 0 through 7 (Section 3.5), or (for B register instructions only) as B0 through B7(B_j).

The k designator can be written as a number 0 through 7, an expression which will be evaluated as 0 through 7, or as the mnemonic W, X, A, L, U, LX, Q, CPW, CPL, or CPU.

The b designator may be written as a number 0 through 7 or as B0 through B7 (B_b).

The y designator is an item or expression (see 3.5) representing a 15-bit number. This y designator value is added to the contents of B_b to form either the effective operand or relative operand address, $\bar{y}$. Thus, the B registers can be used for indexing. The operating system will then add a relative index to all relative operand addresses of a program to form the absolute operand addresses for the program.

The f designator mnemonic is the only designator required of all instructions. When other designators are not explicitly defined by the programmer or are blank, they are evaluated as numerical zeros. The following are examples of the three general types of instructions, where the first column is column 8.

```
1  LAQ,L  TAG,B3,APOS
2  LB,U  B2,TAG,B0
3  DPS  TAG,B0
4  LAQ    ,,3,APOS
```

Line 1 is an example of a general type instruction. The instruction is: add the number in the lower half of the memory word at location TAG + contents of B3 (where, for purposes of description, TAG is 12345) to the number in the Q register and retain this sum in the A register; then skip the next sequential instruction if this sum is positive. In machine code this instruction would appear as 3061312345. The second line is an example of a B register instruction. The instruction is: load index register 2 with the number contained in the upper half of the memory word at location TAG. The machine code for this instruction is 1222012345. The third line is an example of a 77 instruction. The instruction is: store the contents of the AQ register at locations TAG and TAG + 1; the A portion at TAG, the Q portion at TAG + 1. In machine code, this instruction would appear as 7725012345. In all of these instructions, it must be remembered that the actual operand addresses would be biased by the relative index by the operating system. The instruction in line 4, similar to that of line 1, is: add the contents of B3 to the number in the Q register and skip if sum is positive. This would be machine coded as 3060300000 and the operands would not be affected by the relative index of the operating system (except, possibly, the contents of B3, from some prior instruction).

3.2.1. Label Field

The label field may contain two subfields, separated by a comma. The first subfield is used for location counter declaration. The second subfield is used as identification (for reference purposes) of a symbolic line of coding representing data or an instruction. This second subfield is commonly called the label of an instruction (as distinct from "label field"). If either subfield is not entered, no comma should be present in the label field. If the label field is not entered in a line of coding, column 8 must be blank unless a previous line is being continued on this line. The label field must start in column 8.

3.2.1.1. Location Counters

There are 64 location counters, numbered 0 through 77_8. They are used to control assembly sequence of the lines of coding by assigning sequential relative addresses (starting from 00000) under each location counter. Thus,at assembly time, all lines of coding controlled by location counter 0 are assigned in sequence. The same is done for those lines controlled by location counter 1, etc. In the source coding, the location counters may be used in any sequence and this enables regrouping and segmentation of programs as desired. If there is no location counter declaration in a program, all lines are assumed to be under control of location counter 0.

Location counter declaration has the form $(e) where e is any valid entry with a value of 0 through 31. A specified location counter will control its line of coding and all succeeding lines until a new location counter is declared. Each initial location counter declaration begins coding from zero for that location counter. Coding under a previously specified location counter will start from the last value used plus one for that location counter. The following illustration depicts use of the location counter.

```
1  $(1)  LB,U  B2,TAG
2  $(2)  LAQ,L TAG,,APOS
3        LB,U  B2,TAG+2
4  $(1)  LB,U  B1,TAG+1
5  $(4),TAG3   LB,TAG
6  TAG4  LB,U  B4,TAG
```

Lines 1 and 4 will be given sequential addresses under location counter 1.

Lines 2 and 3 are given sequential addresses under location counter 2.

Reference to the current location counter in a line of coding requires only the $ symbol. Reference to any other location counter requires $(e) where e is a valid expression with a value of 0 through 31. The following illustration shows use of the value in a location counter.

```
1  $(2)  J  $+3
2   LB,U  B2,TAG
3   LAQ,L  TAG,,APOS
4   DPS  TAG
5   J  $(1)-1
```

When line 1 is executed, it will cause a jump to line 4. When line 5 is executed, it will cause a jump to the last line (up to this point) controlled by location counter 1. It is dangerous to use such jumps (involving the $ in the operand), because such a jump may cause a jump into part of a procedure or function (Section 6), or cause erroneous skipping where any directive which generates more than one line of code may appear between the jump instruction and its intended operand.

3.2.1.2. Label

The label is the second subfield of a label field. It identifies either a symbolic line of instruction or data. Any name made up of no more than ten alphanumeric characters may be used in a label, and it must start with an alphabetic character. A label may be subscripted (up to two dimensions) to indicate that it is a unique element of the array bearing the same name, but the subscript is not counted as one of the ten (max.) characters permitted. Subscripts within subscripts (see last line of example following) are permitted provided that the subscripted item or expression is evaluated by execution time. A label is usually given a value determined by the current location counter and the number of the location counter.

A label in one program unit can be referenced by another separately assembled program unit only if the label has an asterisk immediately after its last character. In this case, the label is "externally defined." To avoid ambiguity, the programmer should avoid terms as A, Q, B0 through B7, and labels beginning with the alphabetic characters O or X. Labels are shown in the following:

```
CAMP  JBD  B6,TINY
$(2),A2$E  LA,05,6
$(1),CAMP(1)*  SA,W  CAMP
BOB(BAT(1,2))  .  VALID LABEL
```

3.2.2. Operation Field

The operation field is the first field following the label field. If no label field is used or there is no continuation of the preceding line, the first non-blank character (except period or apostrophe) is considered the start of the operation field.
The operation field may be:

- an instruction mnemonic possibly followed by a k designator as a subfield

- + or – to indicate a data word of octal, decimal, or alphabetic designation. In this case, a space is not necessary to terminate the operation field. The operand may immediately follow the + or –. As an example, +2 and +b2 are identical. If the operation field is a number, the + may be omitted for positive numbers.

- an assembly directive (Sections 5 and 6)

- a label previously defined as an entry point into a procedure or function (Section 6)

In all of these cases, except the second, a space following any character except a comma ends the operation field. If the operation field contains a directive (other than RES or DO), the location counter is not affected. In all other cases (other than the RES or DO directives) the location counter is incremented after the line has been generated. The following shows examples of valid operation fields.

```
1  D1 J HORSE
2   RES 01000 .
3  HORSE RI  15D,B2
4   LA,U TABLE1,0,BZERO .
5  BETA 1234512345
6  NU -1234512345
7   +'AB'
```

Line 2 increments the location counter by 512D. Line 5 contains a data word. At compilation time the octal 1234512345 will be generated at address BETA. Line 6 indicates that the octal 6543265432 will be generated at address NU (due to the ones complement representation of negative numbers). Line 7 indicates that the alphabetic characters AB will be generated, right justified, in their Fieldata code (Appendix B) representation. The apostrophes indicate that the characters are alphabetic. At compilation time, the address following NU will contain 0000000607. Data generation is more fully described in 3.4 and 3.5.1.5.

3.2.3. Operand Field

The operand field follows the operation field and must be separated from it by at least one space not following a comma. An unlimited number of blanks may be used to separate fields. The subfields of the operand field represent information necessary to generate the type of line indicated by the operation field. The maximum number of subfields is determined by the operation entry.

The subfields (except for the first and last) are enclosed within commas. A comma at the end of a subfield indicates that more subfields are present and scanning of the line will continue. Any number of spaces (or none at all) may precede the first character of the next subfield. If the first subfield is to be omitted,a zero followed by a comma must indicate this or, alternatively, this zero may be replaced by a space. If the last subfield or subfields are omitted, they are simply left blank but the field must not end with a comma. If any subfield other than the first or last is to be omitted, two contiguous commas or comma zero comma must indicate this. When subfield information is not explicitly defined by the programmer, the parameter represented by this subfield is assumed to be zero. Format requirements demand only that subfields or the lack of a subfield definition be indicated as described in this paragraph. The following lines are applications of these rules.

```
1  BOB LA,W NUMB,,AZERO
2  CLEAR SB B0,TAG,B0
3  CLEAR SB  ,TAG
4  CLEAR SB  0,TAG,0
```

In line 1, the b designator is zero. Lines 2, 3, and 4 are B register instructions and are equivalent to each other. Any one of these has the effect of clearing the 30-bit word at address TAG.

3.3. LINE CONTROL AND COMMENTS

A line may contain an instruction, data word, or assembler directive, followed by comments or the line may contain only comments. Further operand information is not interpreted after the maximum number of subfields required by the operation has been scanned or by the recognition of 80 characters, whichever occurs first. However, a line may be continued and comments may be provided by the programmer for printout as desired.

3.3.1. Continuation

A line is continued by the insertion of a minus (–) character in column 7 of the continuation, so long as the line has not been terminated by a period. If a line is broken with a subfield, the next character should begin in column 8 of the next line. The following illustrates use of the continuation feature.

```
1  $(3), MAIN
2 -LA,
3 -U
4 -TABLE1,,, AZERO
5  $(3), MAIN LA, U TABLE1,, AZERO .
```

Lines 1, 2, 3, and 4 contain the same instruction shown without continuations on line 5.

3.3.2. Comments

Comments may be freely written by the programmer to aid in programming and debugging. A comment is separated from the last field on the line by space, period, and space. The comment may be continued to the next line, or a comment may be started on the next line if a period is placed in column 8 followed by a space. If additional subfields are required by the operation, they are assumed to be zero. The following is an example.

```
 AQ, U  LABEL,,  QNOT  .  ADD  THE  NUMBER
.  IN  THE  UPPER  HALF  OF  THE  WORD  T
.  AT  LABEL  TO  NUMBER  IN  Q  AND  SKIP
.  NI  IF  SUM  IS  NOT  ZERO.
```

3.4. DATA WORD GENERATION

Data words may be either 30 bits in length (single word) or 60 bits in length (double word), and are generated as depicted below.

3.4.1. Single Word (30 Bits)

A list of one, two, three, or five expressions (see 3.5) may be used as arguments to generate one computer word, as follows:

SOURCE CODE	GENERATED CODE				
$\pm exp_1$	30 bits				
$\pm exp_1, exp_2$	15 bits		15 bits		
$\pm exp_1, exp_2, exp_3$	10 bits		10 bits		10 bits
$\pm exp_1, exp_2, exp_3, exp_4, exp_5$	6 bits	6 bits	6 bits	6 bits	6 bits

Note that the first expression must be signed for this application.

Examples:

LABEL ␢ OPERATION ␢ OPERAND	
−2	7777777775
+ 10D, 1=1	00012 00001
−1+1,−4,0	0000 1773 0000
+'A', 'B', 1,2,3	06 07 01 02 03

3.4.2. Double Word (60 Bits)

Sixty-bit constants may be specified by the use of the DLD directive (Section 5). These constants may be:

- Octal
- Decimal
- Floating Point
- Internal Decimal

3.4.3. Variable Length Word

A string of characters enclosed by apostrophes can be inserted into the generated code as a string of 6-bit Fieldata characters (Appendix B). Since the apostrophe is a control character in this case, the apostrophe cannot be one of the characters in the string. The number of computer words occupied by the string depends upon the number of characters. The characters are left justified into consecutive computer words, five characters to a word. Any remaining areas in the last word will be filled with Fieldata spaces (05_8). (This feature should not be confused with the one described in 3.5.1.5, which refers to character strings *in an expression* and uses only one word of computer storage.)

Examples:

```
LABEL ᵬ OPERATION ᵬ OPERAND

        'ABC'                       0607100505
        'ABCDE'                     0607101112
        '-123ABCD'                  41616263060710110505
```

3.5. EXPRESSIONS

An expression is the combination of items or expressions by a logical, relational, or arithmetic operator. It most commonly appears in the operand field of a symbolic line as an entry in a subfield. Blanks are not permitted within an expression. Double word items may not be used to form expressions except with double word operations or, in the operation field, as literals preceded by DLD.

3.5.1. Items

This section describes the various items which may appear in an expression.

3.5.1.1. Label

Any label may be used as an item. Whenever a label is encountered within an expression, the value equated to the label is substituted for the label within the expression.

Example:

```
LABEL ᵬ OPERATION ᵬ OPERAND

VARI    EQU        01000
        LB  B1,    VARI,B2              12 1 0 2   01000
```

3.5.1.2. Location

A location may be used as an item by reflexive addressing. Reflexive addressing may be achieved by referencing the current location counter, or a specific location counter, within a symbolic line. This feature has already been described in 3.2.1.1.

Example:

```
LABEL ƀ OPERATION ƀ OPERAND

BOB       J          $+2               61 0 0 0 00502
          LA         BOB,6             11 0 0 6 00500
          SA         0,B2              15 0 0 2 00000
```

The first line transfers control to the third line.

3.5.1.3. Octal Values

Octal digits (0–7) may appear as an item within an expression. The assembler will create a binary equivalent of the item value. The binary representation of the value will be right justified in a signed field.

Examples:

```
LABEL ƀ OPERATION ƀ OPERAND

          +17                      0000000017
          -074                     7777777703
```

3.5.1.4. Decimal Values

Decimal values may be represented as an item by following the desired digits with an alphabetic D. A decimal value, containing the characters 0–9 will be represented by a right justified and signed binary equivalent within the object field, sign filled.

Examples:

```
LABEL ƀ OPERATION ƀ OPERAND

           6D                      0000000006
          +6D                      0000000006
          +8D                      0000000010
          –6D                      7777777771
```

3.5.1.5. Character Strings in Expression

An item within an expression can be source coded by up to five 6-bit Fieldata characters (Appendix B). The desired characters are enclosed with apostrophes. The leftmost apostrophe may be preceded by a plus or minus sign. Since the apostrophe is a control character in this case, it may not be one of the characters in the string. Because this item is an expression, it will be treated as an arithmetic or logical value.

If the leftmost apostrophe is preceded by a plus sign (or no sign at all), the characters will be right justified in the computer word and empty spaces will be filled with 0 bits; if preceded by a minus sign, the ones complement (of the string preceded by a plus sign) will be the resultant value. Any characters to the left of the rightmost five characters in the string will be disregarded.

Examples:

```
LABEL ƀ OPERATION ƀ OPERAND

            +'00CAT'                    6060100631
            'CAT'                       0000100631
            -'CAT'                      7777677146
            '-CAT'                      0041100631
```

3.5.1.6. Double Word Constants (60 Bits)

Double length constants may be floating point, octal, decimal, and internal decimal (Fieldata). Only symbols which reference DLD constants are allowed in the operand field.

Examples:

```
LABEL ƀ OPERATION ƀ OPERAND

         DPL         BOB,6           77 21 6 03000
         FA          BOB,6           77 01 6 03000

BOB      DLD          16384.0        2017400000
                                     0000000000
```

In this example, the contents of BOB are loaded into the A and Q registers and then (via the Floating Point Add instruction) added to itself.

A floating point number is a number with a decimal point. The number is stored with a normalized mantissa and biased characteristic to conform to the format shown in Figure 2–4. For example: the number 16384.0 is a floating point number and, in octal form, corresponds to 40000. A shift of 15D or 17_8 is required to normalize this mantissa, so that the unbiased characteristic is 0017_8. The biased characteristic is then 2017_8 and the complete floating point number is stored as 2017400000 0000000000_8 in two consecutive words. Since two words are required, the DLD directive is used (see 5–8).

An internal decimal number is an integer appended with the alphabetic character I. It is stored as its Fieldata coded equivalent, right justified into the second word with Fieldata coded zero fill (see 5.8), requiring two words. The number 666I is stored as 6060606060 6060666666. The label refers to the first portion (6060606060).

3.5.1.7. Literals

A literal is a device for indicating operand constants requiring up to 30 bits with just one line of coding. For example, if it is desired to load the A register with the octal value 1234512345, the programmer could do this in two lines of coding without the use of a literal. The first line of the following example would load the A register with the entire word in location X. The second line of the figure is needed to store the data word 1234512345 at location X. Instead, with the aid of a literal, only one line of coding would be required. The load A instruction would have as the operand, the literal 1234512345. When this instruction is scanned, the literal 1234512345 is automatically assigned a 15-bit address in the literal pool for the location counter and it is this address that is automatically inserted into the operand portion of this instruction. Thus, one line of coding serves the purpose of two lines. All literals are placed in the literal table under control of location counter zero unless inserted under a LIT directive (see 5.4). Since most instructions enable coding of operand constants requiring 15 bits or less, it is not necessary to use literals for such constants.

A literal may be represented on an instruction line by immediately preceding the constant with a colon (:) and immediately ending the constant with a semicolon (;) in the y subfield to generate a word containing the constant. Literals are assigned addresses at the end of the program, and duplicates will be eliminated. When location counters are employed, these literals will be assigned to the end of the coding associated with a particular counter (see 5.4. LIT). In this case, duplicates are only eliminated for the counter itself.

In addition to a constant data word, the line item may be an instruction word (line 4 in the example). This will be assembled in the constant area at the end of the coding. Line items within line items, e.g., cascaded addresses, are permitted.

In the case of an instruction word line item, a label is not permitted. The first character following the colon must be the start of the operation field.

If a literal is split by a comma (see line 7 following) the two portions will constitute the upper and lower halves, right justified with sign fill, of the literal. Line 8 shows nested literals. Line 9 shows use of a double length literal (see Section 5 for DLD directive).

```
1   LA,W  X
2  X   1234512345
3   LA,W  :1234512345;
4   LA,W  :NOP;  .  THE A
5  .  REGISTER WOULD
6  .  CONTAIN 1200000000
7   LA,W  :123,-45;  .  LIT IS 0012377732
8   LA,W  :LB,U B3,:732;;
9   DPL DLD :731876D;
```

3.5.1.8. Parameters

A PROC or FUNC parameter may also be an item. Parameters will be discussed in detail in the section pertaining to PROC's and FUNC's (Section 6).

3.5.2. Operators

An operator is a mathematical, logical, or relational symbol representing an operation to be performed.

Items may be combined into expressions by means of operators.

There are 15 operators which designate the method and, implicitly, the sequence to be employed in combining items or expressions within a subfield. Blanks are not permitted within an expression. Evaluation of an expression begins with substitution of values for each element. The operations are then performed from left to right in order of hierarchy (see Table 3–1). No two operators may appear in immediate succession, but parentheses may be used for separation.

Each operator has a position in the precedence hierarchy which determines the sequence (highest precedence first) of evaluation of an expression: consecutive operations with the same precedence are executed from left to right. Interruption of the normal precedence is accomplished by using parentheses. If an item or an expression is enclosed in parentheses and an operator appears adjacent to the parentheses, the function of the parentheses in this instance is that of algebraic grouping. The value of this quantity is the algebraic solution of the items or expression enclosed in parentheses. There is no restriction on the number of nested parentheses within an expression. (See 5.2 for a description of the EQU directive used in the following examples.)

3.5.2.1. Arithmetic Product *

The value of the first item is used as the multiplicand, the value of the second is used as the multiplier; the value of the expression is the product obtained by the multiplication of the two items or expressions.

Example:

```
N     EQU  17D
        N*010
```

The value of the expression is 136_{10}.

PRECEDENCE	OPERATOR	DESCRIPTION
6	*/	a*/b is equivalent to a*2^b (see text)
5	*	arithmetic product
5	/	arithmetic quotient
5	//	covered quotient (a//b) is equivalent to (a+b–1)/b
4	+	arithmetic sum
4	–	arithmetic difference
3	**	logical product
2	++	logical sum
2	––	logical difference
1	=	a=b has the value 1 if true, 0 otherwise
1	>	a>b has the value 1 if true, 0 otherwise
1	<	a<b has the value 1 if true, 0 otherwise
1	<=	$a \leq b$ has the value 1 if true in either case, otherwise 0
1	>=	$a \geq b$ has the value 1 if true in either case, otherwise 0
1	/=	$a \neq b$ has value 1 if true and 0 otherwise

NOTE: Operations with the highest precedence are executed first.

Table 3–1. Operator Priority Within Expressions

3.5.2.2. Equal =

The "equal" operator compares the value of two items or expressions. If the two values are equal, the assembler will assign a value of 1 to the combined expression. If the values are not equal, the value of the combined expression is 0.

A = 1

If A is equal to 1, the value of the expression is 1 (true). If A is unequal to 1, the value of the expression is 0 (false).

Example:

```
DO   A = 3 ,    RES 3
```

If A is equal to 3, the controlling location counter will be incremented by 3; if not, the line will be skipped.

3.5.2.3. Greater Than >

The "greater than" operator compares two items or expressions. If the value of the first operand is greater than the value of the second operand, the combined expression is assigned the value 1 (true). If the value of the first is equal to or less than that of the second, a value of 0 (false) is assigned.

B > 2

If B is greater than 2, the expression value is 1. If B is not greater than 2, the expression value is 0.

Example:

```
(A>2) *5
```

If A is greater than 2, the value of the expression is 5, otherwise the expression is 0. Note that the parentheses interrupt the normal order of precedence. If the parentheses are omitted, the result is a 1 (if A is greater than 10_{10}) or a 0 (if A is not greater than 10_{10}).

3.5.2.4. Less Than <

The "less than" operator compares the values of two items or expressions. If the first is less than the second, the combined expression is assigned the value 1 (true); otherwise, a value of 0 (false) is assigned.

C<1

If C is less than 1, the expression value is 1. If C is not less than 1, the expression value is 0.

Example:

```
(A<2) *2
```

If A is less than 2, the expression value is 2, otherwise the value is 0

3.5.2.5. Less Than or Equal <=

The "less than or equal" operator compares two items or expressions. If the value of the first operand is less than or equal to the value of the second operand, the combined expression is assigned the value 1 (true); otherwise, a value of 0 (false) is assigned.

D <= 2

If D is less than or equal to 2, the expression value is 1. If D is not less than or equal to 2, the value of the expression is 0.

Example:

(B<=2) *2

If B $\leq$ 2, the combined expression value is 2, otherwise the expression value is 0.

3.5.2.6. Greater Than or Equal >=

The "greater than or equal" operator compares two items or expressions. If the value of the first operand is greater than or equal to the value of the second operand, the combined expression is assigned the value 1 (true); otherwise,a value of 0 (false) is assigned

E >= 4

If E is greater than or equal to 4, the expression value is 1. If E is not greater than or equal to 4, the value of the expression is 0.

Example:

(F>=5) *3

If F $\geq$ 5, the combined expression value is 3, otherwise the expression value is zero.

3.5.2.7. Not Equal To /=

The "not equal to" operator compares two items or expressions. If the two values are not equal, the assembler will assign a value of 1 (true) to the combined expression. If the values are equal, the value of the expression is 0 (false).

A /= 2

If A is not 2, the value of the expression is 1, if A is 2, the value of the expression is 0.

Example:

(C/=D) *4

If C is not equal to D, the value of the expression is 4. If C is equal to D, the value of the expression is zero.

3.5.2.8. Logical Sum ++

The "logical sum" operator provides the logical sum of values of two items or expressions. The assembler will produce the logical sum and use it as the value of the combined expression. The operation tests the bits in corresponding bit positions of both operands. The result will contain a 1 bit where either, or both, bits is a 1 bit. Thus, the logical sum corresponds to a bit-by-bit OR operation.

Example:

```
        A      EQU    3
            A++5
```

The value of the expression is 7.

3.5.2.9. Logical Difference --

The "logical difference" operator produces the logical difference between the values of two expressions or items. The operation compares the bits in corresponding bit positions of the two expressions or items. Where the bits are unlike, the logical difference will contain a 1 bit; if alike, a 0 bit. Thus, the logical difference corresponds to a bit-by-bit Exclusive OR operation.

Example:

```
        A     EQU     6
              A--5
```

The value of the expression is 3.

3.5.2.10. Logical Product **

The "logical product" operator produces the logical product of the values of two expressions or items. The two operands are compared, bit-by-bit. Where both are 1 bits, the logical product will have a 1 bit; otherwise, a 0 bit. Thus, the logical product corresponds to a bit-by-bit AND operation.

Example:

```
        N     EQU     17D
            N**3D
```

The value of the combined expression is 1.

3.5.2.11. Arithmetic Sum +

The "arithmetic sum" operator produces the algebraic sum of the values of two items or expressions. The value of the combined expression will be the sum of the value of the items or expressions.

Example:

```
        A     EQU     74D
           A + (A**061)
```

The value of the combined expression is 74D.

3.5.2.12. Arithmetic Difference –

The "arithmetic difference" operator produces the algebraic difference between the values of two items or expressions. The assembler will subtract the value of the second operand from the value of the first, and the difference is the value of the combined expression.

Example:

```
B    EQU    0662
     (B--0511)-B
```

The value of the combined expression is -0267.

3.5.2.13. Arithmetic Division /

The value of the first item or expression is the dividend, the value of the second item or expression is the divisor; the result of the operation is the quotient. The remainder is discarded by the assembler.

Example:

```
B    EQU    17D
   (B**3=1)*B/4
```

The value of the combined expression is 4. Note that the remainder is discarded.

3.5.2.14. Covered Quotient //

The covered quotient operates in the same fashion as the arithmetic quotient with this modification: if a remainder greater than 0 is created during the division, the quotient is increased by 1.

Example:

```
A    EQU    3
   (A--3=0)*A//2
```

The value of the combined expression is 2.

3.5.2.15. Shift Exponent */

The shift exponent indicates that the operand (preceding the shift exponent) shall be shifted by the number of positions denoted by the value immediately after the shift exponent. The shift is left or right according to the sign of the exponent (negative will produce a right shift). a*/b is equivalent to $a*2^b$, where b must be an integer and no sign or significant bits are shifted out. "a" will be shifted as though it were a binary quantity, with zero fill.

Examples:

```
A    EQU    2
     A*/3           (Result is 2x2^3 or 16D)
B    EQU    16D
     B*/(-3)        (Result is 16D x 2^-3 or 2)
```

3.5.2.16. Comments Within Expressions /. ./

Comments can be inserted within expressions by means of the two delimiters /. and ./ as shown in the examples. Insofar as processing is concerned, these comments will be ignored.

Examples:

```
+  TAPE=5/.TAPE = 5 IS SYSTEM./-1
+  TAPE=5-1
```

The same data will be stored for the two examples.

3.5.3. Absolute and Relocatable Labels and Expressions

The value assigned to a label is either relocatable or absolute. If the values assigned to label entries in the source statement are later displaced by a constant number from their originally assigned relative addresses, the labels are termed "relocatable" items. (This displacement normally occurs when the object program is loaded into storage locations other than those originally assigned by the assembler program.) If the values assigned to labels are unaffected by relocation procedures and remain set equal to a constant absolute value, the labels are termed "absolute" items. As examples, a label may be set absolute by an EQU or a LET directive (see Sections 5 and 6 for descriptions of these directives).

When plus (+) or minus (–) operators alone are used to form an expression, the expression is absolute if: 1) it has an even number of relocatable items (zero is considered an even number), and 2) each relocatable item is paired with a relocatable item of opposite sign (not necessarily contiguous), controlled by the same location counter. An expression is relocatable if: 1) it has a positive value, and 2) it has an odd number of relocatable items, and 3) the relocatable items (except one) are paired under the same location counter, with opposite signs.

When the other operators (all but plus and minus) are used to form an expression, the immediate result is an absolute item. (This absolute item can be used to form a relocatable item in the same line of coding.) If either, or both, of the items used in forming this absolute item are relocatable, the result will be flagged with a relocation error.

A literal is also a relocatable item since, in the source coding, it is represented by an assembler-assigned address. However, literals cannot be combined, by means of operators, to form expressions. A reference to a location counter with the symbol $ is also a relocatable item.

Examples of absolute and relocatable expressions are shown in Table 3–2.

EXPRESSION	TYPE	RELOCATION ERROR FLAG?
A+B	absolute	no
A+B–3	absolute	no
A+R12	relocatable	no
A+R12–R23–R22+R13	absolute	no
B+R12+R13–R22	relocatable	no
A//B	absolute	no
R12–A//B	relocatable	no
R22*A	absolute	yes
R12–R22*A	relocatable	yes
R22*1	relocatable	no
(R12–R22)*A	absolute	no
R12–R22*A	relocatable	yes
R12/R22	absolute	yes

NOTE: A and B are absolute items (unequal to 1).
R12 and R22 are relocatable items under location counter 2.
R13 and R23 are relocatable items under location counter 3.

Table 3–2. Absolute and Relocatable Expressions

4. BASIC ASSEMBLER LANGUAGE INSTRUCTIONS

4.1. GENERAL

The basic instructions are those instructions used by "worker" programs. Where the same instruction may operate differently under executive control, indication is provided in text. Basic assembler instructions consist of:

- *Transfer instructions*, to move data between memory storage and registers.
- *Shift instructions*, to move the contents of a selected register to the right or left as many bit positions as specified.
- *Comparison instructions*, to compare two operands either arithmetically or alphanumerically and skip (as determined by the comparison) the next instruction.
- *Jump instructions*, to transfer control of the program to another area within memory storage.
- *Special sequence-modifying instructions*, to enable repetitions, conditional skips, programmed interrupts, or use of programmed "electronic switches".
- *Arithmetic instructions*, to perform arithmetic operations in fixed point, floating point, or binary coded decimal (BCD) mode.
- *Logical instructions*, to operate upon and edit selected portions of a word.

In some cases,the classification may seem arbitrary, since an instruction may perform more than one function. For example: the RANQ (Replace Y–Q) incorporates both an arithmetic and a transfer function but, since its greatest utility lies in the arithmetic function, it is listed only under arithmetic instructions.

The format of each instruction is followed by an example written in assembly language. The executable instruction that is generated from the assembler mnemonics is shown at the right.

When the label is used in the examples, it is written as LABEL. The assembler-assigned computer address represented by LABEL will be 01234. This will maintain a consistent reference in the examples. For actual coding, any valid label could be used.

4.2. DESIGNATOR INTERPRETATION

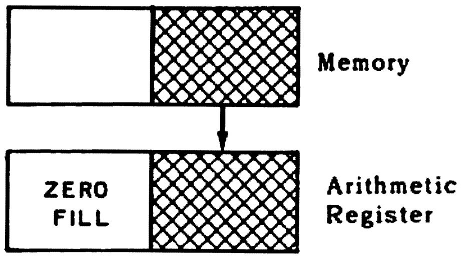
This subsection contains a detailed description of the k and j designators used in the instructions and is to serve as a reference in the descriptions of the instructions which follow:

4.2.1. Interpretation of k Designators

The first three subsections which follow deal with standard k designator interpretation for read class, store class, and replace class instructions, in that order. The fourth subsection presents a table showing the deviations from standard interpretation.

4.2.1.1. Standard Read Class k Designators

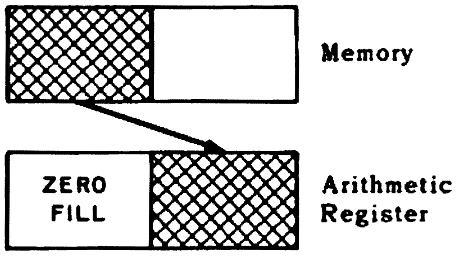
k = 1
<u>L</u>
indicates that the value is obtained from the lower 15 bits of the memory location specified by the operand and is transferred into the lower 15 bits of an arithmetic register. The remaining bits, i.e., the upper 15 bits, are filled with zeros.



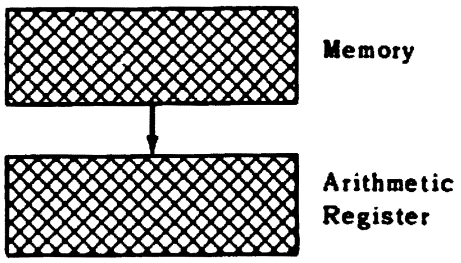
In the 17-bit B register operational mode, bits 15 and 16 of the value are lost.

k = 2
<u>U</u>
indicates the value is obtained from the upper 15 bits of the memory location specified by the operand and placed in the lower 15 bits of the register.

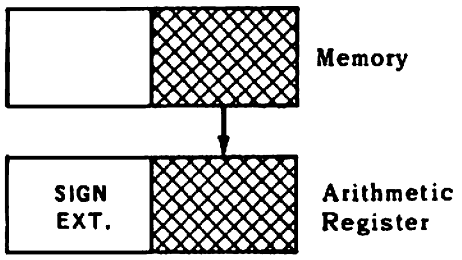
Note: The transfer is to the lower 15 bits of the arithmetic register. The upper 15 bits are zero filled.



k = 3
<u>W</u>
specifies that the entire 30 bits of the value at the memory location indicated by the operand are to be transferred to an arithmetic register.

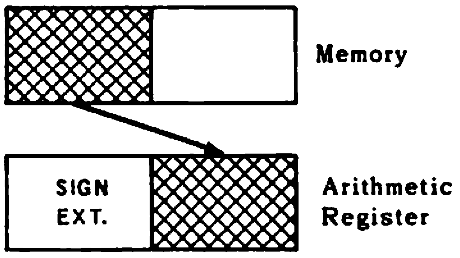


k = 5
<u>LX</u>
indicates transfer of the lower 15 bits of the value in the memory location specified by the operand to the lower 15 bits of an arithmetic register. The difference between L and LX is that with an LX specification, a sign extension is obtained in the arithmetic register, i.e., the highest order bit (the sign bit) of the 15-bit value is extended through the upper 15 bit positions of the register.

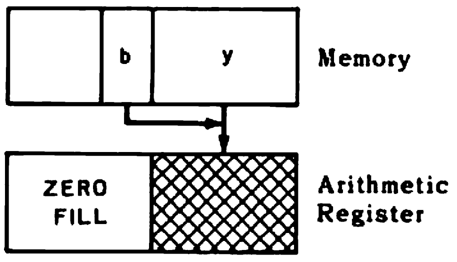


NOTE: An X in the alphabetic notation for the k designator always indicates sign extension.

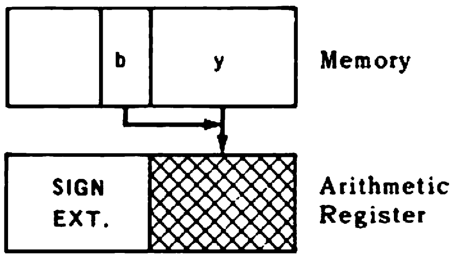
k = 6 / UX — indicates transfer of the upper 15 bits of the value at the specified memory location to the lower 15 bits of an arithmetic register. With X present in the alphabetic notation, there is a sign extension in the upper 15 bits of the register, otherwise it is identical to a k of 2.



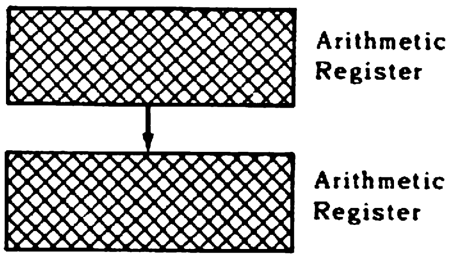
k = 0 / 0 — indicates transfer of the 15 bits of the operand portion of the instruction. The 15 bits contained in the instruction word after B register modification are transferred into the low order 15 bits of an arithmetic register and the remainder of the register is zero filled. In this case, the operand is a datum rather than the address of a datum.



k = 4 / X — indicates transfer of the 15 bits contained in the instruction word after B register modification into the low order 15 bits of an arithmetic register. The upper 15 bits of the arithmetic register are filled with the sign bits, i.e., sign extension occurs.



k = 7 — A — indicates transfer of the entire 30 bits in the A register into the A or Q register.

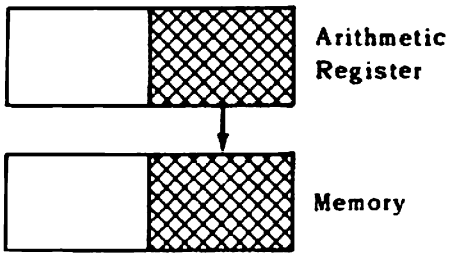


NOTE: For Load B Register instructions, only the low order 17 bits are transferred.

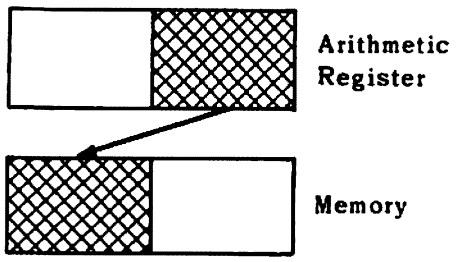
4.2.1.2. Standard Store Class k Designators

A store class instruction involves transfer of a value from an operational register to a register or memory address. Again there are eight possible k designations for this class.

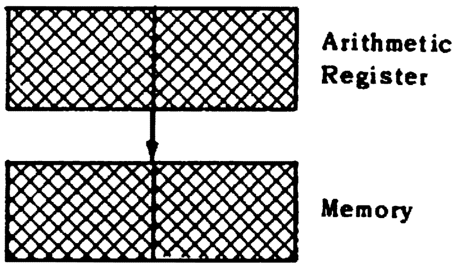
k = 1 — L — indicates that the lower 15 bits from an arithmetic register are to be stored as the lower 15 bits of word at memory address $\bar{y}$. The upper 15 bits at location $\bar{y}$ are undisturbed.



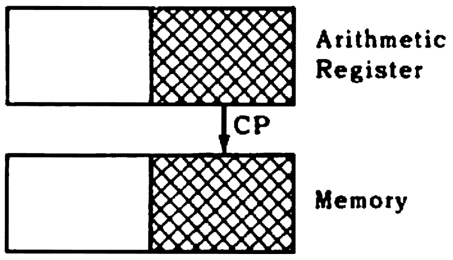
k = 2 — U — indicates storage of the lower 15 bits from an arithmetic register as the upper 15 bits of the word at memory address $\bar{y}$. The lower 15 bits at location $\bar{y}$ are undisturbed.



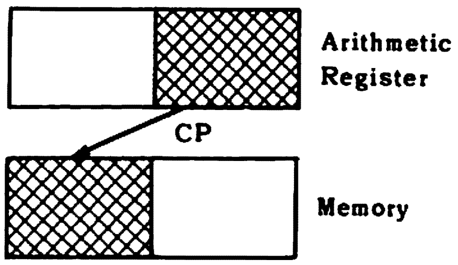
<u>k = 3</u> indicates storage of the entire 30 bits of the register as the word at the memory address $\bar{y}$.
<u>W</u>



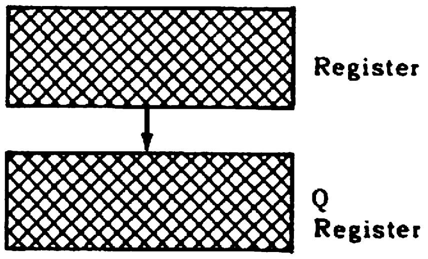
<u>k = 5</u> indicates that the ones complement of the lower 15 bits of an arithmetic register is to be stored as the lower 15 bits of the word at memory address $\bar{y}$; the upper 15 bits are not changed. It is identical to a k of 1 except that the value is ones complemented.
<u>CPL</u>



<u>k = 6</u> indicates that the ones complement of the lower 15 bits of an arithmetic register is to be stored as the upper 15 bits of the word at memory address $\bar{y}$; the lower 15 bits of the word are unchanged.
<u>CPU</u>

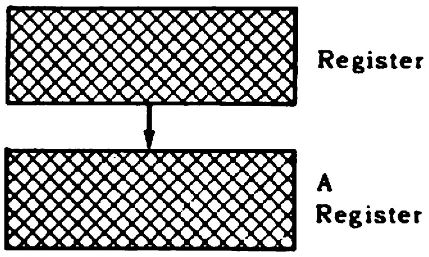


k = 0 / **Q** indicates that there is to be an entire word transfer from a register to the Q register.



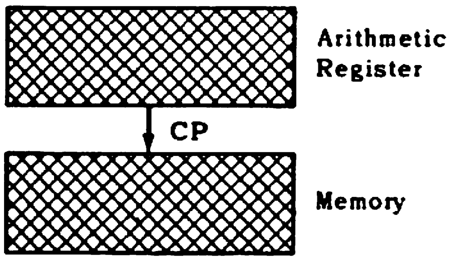
NOTE: Contents of the B register (B_j) are stored in the lower 17 bits of Q; the upper 13 bits of Q will be zero filled.

k = 4 / **A** indicates that there is to be an entire word transfer from a register into the A register.



NOTE: Contents of the B register (B_j) are stored in the lower 17 bits of A; the upper 13 bits of A will be zero filled.

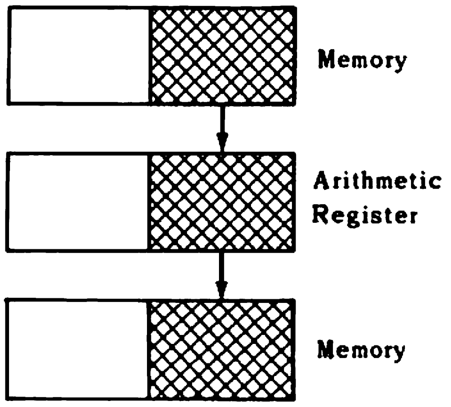
k = 7 / **CPW** indicates transfer of the ones complement of the entire 30 bits of a register into the 30-bit word of the specified memory location.



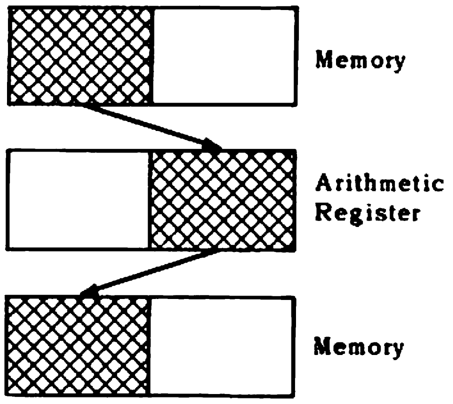
4.2.1.3. Standard Replace Class k Designators

A replace class instruction involves transfer of a value from the address in memory specified by the operand in the instruction to an arithmetic register where the operation designated by the operator field takes place. The result then replaces the initial contents of the memory location. The k designators of 0, 4, and 7 are invalid for use with replace class instructions.

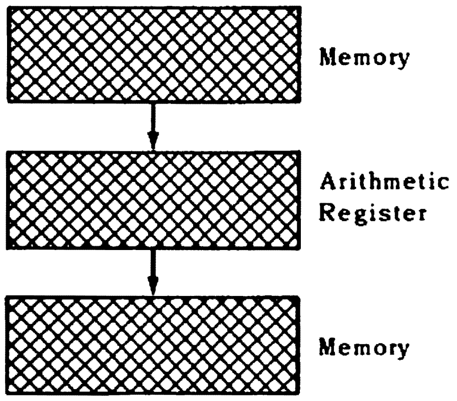
k = 1 indicates transfer of the lower 15 bits of the word at the specified memory address to the lower 15 bits of the arithmetic register where the operation takes place. The result is then transferred from the lower 15 bits of the arithmetic register into the lower 15 bits of the memory location, i.e., the original contents of the memory location are destroyed and the value obtained as a result of the computation replaces the original value in the memory address.
L



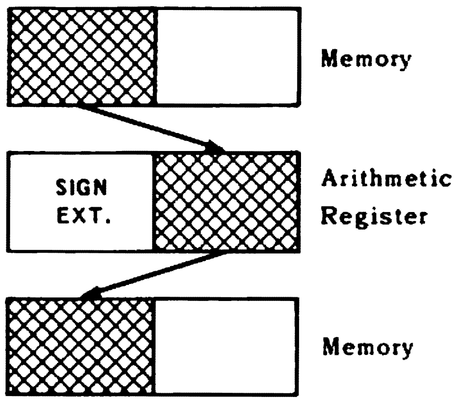
k = 2 indicates transfer of the upper 15 bits of the word at the specified memory address into the lower 15 bits of the arithmetic register where computation specified by the operation field of the instruction takes place. After computation, the result then replaces the original 15 bits of the memory location. The lower 15 bits of the memory location remain unchanged.
U



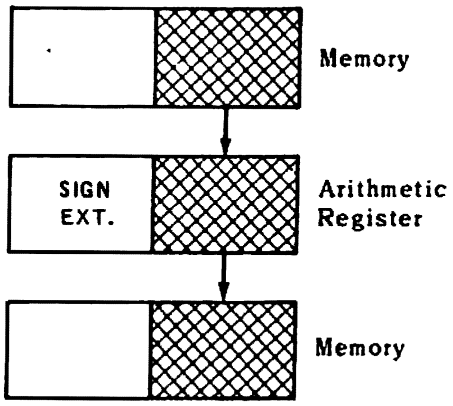
k = 3 W — indicates transfer of the entire 30 bits of the word in the specified memory location into the entire 30 bits of the arithmetic register, where the operation is performed. The entire 30 bits of the result will replace the previous contents of the memory address.



k = 6 UX — indicates transfer of the upper 15 bits of the word in the specified memory address to the lower 15 bits of the arithmetic register, with sign extension. When the logical or arithmetic operation is completed, the result, i.e., the lower 15 bits of the arithmetic register, replaces the upper 15 bits of the memory address.



k = 5 LX — indicates transfer of the lower 15 bits of the word in the specified memory address into the lower 15 bits of the arithmetic register, with sign extension, where the operation takes place. The result replaces the lower 15 bits of the memory location.



4.2.1.4. Exceptions to Standard k Designator Interpretation

Table 4–1 lists deviations from standard k designator interpretation, previously described, for particular instructions. These instructions are grouped by functions with function code and mnemonic code reference.

4.2.2. Interpretation of j Designators

The most common use of the j portion of the instruction word is to specify a skip condition. If the specified condition (such as a negative value or a value of zero) in an arithmetic register is present, the next instruction will be skipped. This permits the user to control program sequence based on the result of an operation. For those instructions that do not have skip conditions, this portion of the instruction word may have other uses.

In the majority of instructions, the normal j designator is used, i.e., the result in a particular register, the A or Q, is tested and a transfer of control takes place if the tested condition exists.

The following figures list the interpretation of the j designator for various instructions. Table 4–2 shows the "normal" j designator interpretation; Table 4–3, deviations from the normal interpretation for test (compare) instructions; Table 4–4, deviations for jump instructions; Table 4–5, the repeat instructions; Table 4–6, arithmetic and logical instructions. Abbreviations and special symbols used in these tables are explained in Appendix A.

TYPE	FUNCTION CODE	MNEMONIC	k DESIGNATOR	DEVIATION
Transfer	12	LB	0	17-bit $\bar{y} \rightarrow B_j$
			1	15-bit $(\bar{y})_L \rightarrow B_j$ with zero fill at high end of B_j
		j = 4, 5, 6, or 7	2	15-bit $(\bar{y})_U \rightarrow B_j$ with zero fill at high end of B_j
			3	17-bit $(\bar{y})_{0\text{-}16} \rightarrow B_j$
			4	15-bit $\bar{y} \rightarrow B_j$ with sign extension
			5	15-bit $(\bar{y})_L \rightarrow B_j$ with sign extension
			6	15-bit $(\bar{y})_U \rightarrow B_j$ with sign extension
			7	17-bit $(A)_{0\text{-}16} \rightarrow B_j$
			0	15-bit $\bar{y} \rightarrow B_j$
			1	15-bit $(\bar{y})_L \rightarrow B_j$
		j = 1, 2, or 3	2	15-bit $(\bar{y})_U \rightarrow B_j$
			3	15-bit $(\bar{y})_{0\text{-}14} \rightarrow B_j$
			4	15-bit $\bar{y} \rightarrow B_j$
			5	15-bit $(\bar{y})_L \rightarrow B_j$
			6	15-bit $(\bar{y})_U \rightarrow B_j$
			7	15-bit $(A)_{0\text{-}14} \rightarrow B_j$
		j = 0	NO OPERATION	
	14	SQ	0	$CP(Q) \longrightarrow Q$.
	15	SA	4	$CP(A) \longrightarrow A$.
	16	SB	0	$(B) \longrightarrow Q_L$ with zero fill.
			3	$(B) \longrightarrow (\bar{y})_L$
			4	$(B) \longrightarrow A_L$ with zero fill.
			7	$CP(B) \longrightarrow (\bar{y})_L$ with sign extension.
Shift	01, 02, 03, 05, 06, and 07	RSQ, RSA, RSAQ, LSQ, LSA, and LSAQ	1, 3, 5	Shift count is $(\bar{y})_{05\text{-}00}$.
			2, 6	Shift count is $(\bar{y})_{20\text{-}15}$.
			0, 4	Shift count is $\bar{y}_{05\text{-}00}$.
			7	Shift count is $(A)_{05\text{-}00}$.
Jump	60, 61, 64, and 65	JT, J, SLJT, and SLJ	1, 3, 5	Address of next instruction is $(\bar{y})_L$.
			2, 6	Address of next instruction is $(\bar{y})_U$.
			0, 4	Address of next instruction is $\bar{y}$.
			7	Address of next instruction is $(A)_L$.
Seq.-mod.	70	R	0	Repeat count is 17-bit $\bar{y}$.
			1, 4, or 5	Repeat count is 15-bit $(\bar{y})_L$.
			2 or 6	Repeat count is 15-bit $(\bar{y})_U$.
			3	Repeat count is 17-bit $(\bar{y})_{00\text{-}16}$.
			7	Repeat count is 17-bit $A_{00\text{-}16}$.
	72	JBD	1, 3, 5	Jump to $(\bar{y})_L$ for next instruction.
			2, 6	Jump to $(\bar{y})_U$ for next instruction.
			0, 4	Jump to $\bar{y}$ for next instruction.
			7	Jump to $(A)_L$ for next instruction.
Logical	47	SAND	0	$LP \longrightarrow Q$
			1	$LP_L \longrightarrow (\bar{y})_L$.
			2	$LP_L \longrightarrow (\bar{y})_U$.
			3	$LP \longrightarrow (\bar{y})$.
			4	$LP \longrightarrow A$.
			5	$CP(LP)_L \longrightarrow (\bar{y})_L$.
			6	$CP(LP)_L \longrightarrow (\bar{y})_U$.

NOTE: For abbreviations and symbols, see Appendix A.

Table 4–1. Exceptions from Standard k Designator Interpretation

MACHINE CODE (OCTAL) j DESIGNATOR	MNEMONIC j DESIGNATOR	RESULT
0		EXECUTE NI
1	SKIP	SKIP NI
2	QPOS	SKIP NI IF (Q) POS.
3	QNEG	SKIP NI IF (Q) NEG.
4	AZERO	SKIP NI IF (A) = 0
5	ANOT	SKIP NI IF (A) ≠ 0
6	APOS	SKIP NI IF (A) POS.
7	ANEG	SKIP NI IF (A) NEG.

Table 4–2. Normal j Designator Interpretation

MACHINE CODE (OCTAL) j DESIGNATOR	TA (04)		TQ (04)		TR (04)	
	j MNEMONIC	RESULT	j MNEMONIC	RESULT	j MNEMONIC	RESULT
0	XX	XX	XX	XX	XX	XX
1	SKIP	SKIP NI	SKIP	SKIP NI	SKIP	SKIP NI
2	X	X	YLESS	IF Y ≤ Q SKIP NI	X	X
3	X	X	YMORE	IF Y > Q SKIP NI	X	X
4	X	X	X	X	YIN	IF Q ≥ Y > A SKIP NI
5	X	X	X	X	YOUT	IF Q < Y ≤ A SKIP NI
6	YLESS	IF Y ≤ A SKIP NI	X	X	X	X
7	YMORE	IF Y > A SKIP NI	X	X	X	X

NOTE: XX indicates illegal (invalid) use.
X indicates nonexistence (not used); causes incorrect, but valid shift

Table 4–3. Special j Designator Interpretation for Test (Compare) Instructions

MACHINE CODE (OCTAL) j DESIGNATOR	J(61)/SLJ(65)		JT(60)		SLJT(64)	
	j MNEMONIC	RESULT	j MNEMONIC	RESULT	j MNEMONIC	RESULT
0	b̸ OR UNDEFINED	ALWAYS JUMP	RIL	RELEASE INTERRUPT LOCKOUT	SIL	SET INTERRUPT LOCKOUT
1	KEY 1	JUMP IF KEY 1 SET	RILJP	RELEASE INTERRUPT LOCKOUT AND JUMP	SILJP	SET INTERRUPT LOCKOUT AND JUMP
2	KEY 2	JUMP IF KEY 2 SET	QPOS	JUMP IF (Q) POS.	QPOS	JUMP IF (Q) POS.
3	KEY3	JUMP IF KEY 3 SET	QNEG	JUMP IF (Q) NEG.	QNEG	JUMP IF (Q) NEG
4	STOP	ALWAYS STOP	AZERO	JUMP IF (A) = 0	AZERO	JUMP IF (A) = 0
5	STOP 5	STOP IF KEY 5 SET	ANOT	JUMP IF (A) ≠ 0	ANOT	JUMP IF (A) ≠ 0
6	STOP 6	STOP IF KEY 6	APOS	JUMP IF (A) POS.	APOS	JUMP IF (A) POS.
7	STOP 7	STOP IF KEY 7 SET	ANEG	JUMP IF (A) NEG.	ANEG	JUMP IF (A) NEG.

Table 4–4. Special j Designator Interpretation for Jump Instructions

MACHINE CODE (OCTAL) j DESIGNATOR	R (70)	
	j MNEMONIC	RESULT
0		NE: $\bar{y}=\bar{y}$
1	ADV	NE: $\bar{y}=\bar{y}+1$
2	BACK	NE: $\bar{y}=\bar{y}-1$
3*	ADDB	NE: $\bar{y}=\bar{y}+(B_b)$
4	R	NE: $\bar{y}=\bar{y}$
5	ADVR	NE: $\bar{y}=\bar{y}+1$
6	BACKRNE	NE: $\bar{y}=\bar{y}-1$
7*	ADDBR	NE: $\bar{y}=\bar{y}+(B_b)$

NOTES:

1. *For the R (70) instruction j designators 0, 1, 2 and 3 are exclusive to read and/or store class instructions and 4, 5, 6 and 7 exclusive to replace class instructions. In repeat of replace class instructions, the result is stored at $\bar{y}$ (of NE) plus (B_6).*

2. *NE refers to execution of next repetition in terms of previous $\bar{y}$.*

* *For Mth execution; $\bar{y}=\bar{y}+M\times(B_b)$.*

Table 4–5. Special j Designator Interpretation for Repeat Instructions

MACHINE CODE (OCTAL) j DESIGNATOR	AQ(26)/ANQ(27)		LLP(40)/RLP(44)		D(23)	
	MNEMONIC	RESULT	MNEMONIC	RESULT	MNEMONIC	RESULT
0		EXECUTE NI		EXECUTE NI		EXECUTE NI
1	SKIP	SKIP NI	SKIP	SKIP NI	SKIP	SKIP NI
2	APOS	SKIP NI IF (A) POS.	EVEN	SKIP NI IF (A) EVEN NO. OF 1 BITS	NOOF	SKIP NI IF NO OVERFLOW
3	ANEG	SKIP NI IF (A) NEG.	ODD	SKIP NI IF (A) ODD NO. OF 1 BITS	OF	SKIP NI IF OVERFLOW
4	QZERO	SKIP NI IF (Q)=0	AZERO	SKIP NI IF (A)=0	AZERO	SKIP NI IF A=0
5	QNOT	SKIP NI IF (Q)≠0	ANOT	SKIP NI IF (A)≠0	ANOT	SKIP NI IF (A) ≠ 0
6	QPOS	SKIP NI IF (Q) POS.	APOS	SKIP NI IF (A) POS.	APOS	SKIP NI IF (A) POS.
7	QNEG	SKIP NI IF (Q) NEG.	ANEG	SKIP NI IF (A) NEG.	ANEG	SKIP NI IF (A) NEG.

Table 4–6. Special j Designator Interpretation for Arithmetic and Logical Instructions

4.3. DATA TRANSFER INSTRUCTIONS

Transfer instructions are used to move data within the central processor: core memory locations to registers, registers to core memory locations, registers to registers. All transfers are nondestructive in that the original source of data remains unchanged except in replace class instructions. Transfers may consist of 15 bits, 30 bits, or in character packing and unpacking, 6 bits, as determined by the k designator or the instruction itself. Transfers may consist of the original bits or their ones complements, as specified.

4.3.1. Load Q (10) LQ

Transfer the operand, $\bar{y}$, as determined by k, to the Q register.

Operation	k	Space	y	b	j
LQ	NORMAL	ƀ	READ CLASS	Note	NORMAL

Examples:

```
LQ,W    LABEL                10 0 3 0   01234
LQ,A                         10 0 7 0   00000
```

NOTE: If the k designator is 7, y is effectively zero and no B register modification is possible; Q and A registers will be the same after instruction execution.

4.3.2. Load A (11) LA

Transfer the operand , $\bar{y}$, as determined by k, to the A register.

Operation	k	Space	y	b	j
LA	NORMAL	ƀ	READ CLASS	Note	NORMAL

Examples:

```
LA,0    2,B2                 11 0 0 2   00002
LA,U    LABEL + 3, B2        11 0 2 2   01237
LA,A                         11 0 7 0   00000
```

NOTE: If the k designator is 7, y is effectively zero and (B_b) has no effect; the A register remains unchanged.

4.3.3. Load B_j (12) LB

Transfer the operand, $\bar{y}$, to the active (executive or worker) B register (1-7) specified by the j designator.

Operation	k	Space	j	y	b
LB	See Table 4-1	ƀ	Bj (Note 1)	READ CLASS	B_b

Examples:

```
LB,W   B7,00005              12 7 3 0  00005
LB,L     6,LABEL+1,B2        12 6 1 2  01235
```

NOTE 1: The j designator specifies the selected B register; consequently, a skip condition cannot be programmed in this instruction. The j of the Bj notation may be 0,1,2,3,4,5,6, or 7, to specify the B register of the operation. With a j designator of 0, no operation is performed; the program advances to the NI.

NOTE 2: Transfers of data into and out of 17-bit B registers usually are full words (k of 0, 3, or 7), but can be half words (k of 1, 2, 4, 5, or 6) if desired. The lower 17-bit value is stored in the lower 17 bits of a 30-bit location. When half-words are transferred into a B register, the upper two bits of a 17-bit B register are 0's. Half-word transfers out of a 17-bit B register result in the loss of the upper two bits.

NOTE 3: This instruction may not be used immediately following an Enter IFR (7761) or Enter RIR (7766) instruction, each of which is used under executive control.

4.3.4. Clear B_j (12) ZB

The contents of a specified B register are cleared to zero. The particular B register to be cleared is specified by the j designator. This is a variation of the Load B_j instruction, where B_b is B0 and y is zero.

Operation	k		j	y	b
ZB	none	ƀ	B_j	none	none

Example:

```
LB, 0       B3        12 3 0 0 00000
ZB          B3        12 3 0 0 00000
```

4.3.5. No Operation (12) NOP

No operation is performed and the program advances to the next instruction. This is actually a variation of the Load B_j instruction where B_j is B0.

Operation	k		j	y	b
NOP	none	b̸	none	optional	none

Example:

```
LB      B0                         12 0 0 0 00000
NOP                                12 0 0 0 00000
```

See 4.10.2.1 for examples using y portion.

4.3.6. Double Precision Load (7721) DPL

This operation loads the AQ register.

Operation		y, b
DPL	b̸	base address

The double length register AQ will be loaded. The contents of $\overline{y}$ are placed in A and the contents of $\overline{y}$+1 are placed in Q.

Examples:

```
DPL     LABEL                      77 21 0 01234
DPL     LABEL, B3                  77 21 3 01234
```

4.3.7. Store Q (14) SQ

Store the contents of the Q register in the k designated portion of the storage location specified by $\overline{y}$.

Operation	k	Space	y	b	j
SQ	See Table 4–1	b̸	STORE CLASS	B_b	NORMAL

Examples:

```
SQ,W    LABEL                      14 0 3 0  01234
SQ,1    01234,3,2                  14 2 1 3  01234
SQ,1    LABEL,B3,QPOS              14 2 1 3  01234
SQ,1    LABEL,B3,2                 14 2 1 3  01234
```

4.3.8. Clear Q (16) ZQ

The contents of the Q register will be cleared to zero by this instruction. This is a variation of the Store B_j instruction where B_j is B0 and the transfer is to the Q register because the k designator is 0 (see Table 4–2).

Operation	k		j	y	b
ZQ	none	b̸	none	none	none

Example:

```
SB, Q      B0              16 0 0 0 00000
ZQ                         16 0 0 0 00000
```

4.3.9. Negate Q or Complement Q (14) NQ

The contents of the Q register will be ones complemented as a result of this operation and stored in a memory location. This is a variation of the Store Q instruction where k is 0.

Operation	k		y	b	j
NQ	none	b̸	READ CLASS	B_b	NORMAL

Example:

```
SQ         TAG             14 0 0 0 01234
NQ         TAG             14 0 0 0 01234
```

4.3.10. Store A (15) SA

Store the contents of the A register in the storage location specified by $\bar{y}$, per k.

Operation	k	Space	y	b	j
SA	See Table 4–1	b̸	STORE CLASS	B_b	NORMAL

Examples:

```
SA,2    LABEL              15  0  2  0  01234
SA,W    LABEL-2,3          15  0  3  3  01232
```

4.3.11. Negate A or Complement A (15) NA

The contents of the A register are ones complemented as a result of this operation and stored in a memory location. This is a variation of the Store A instruction where the k designator is 4.

Operation	k		y	b	j
NA	none	␢	STORE CLASS	B_b	none

Example:

```
SA,A      TAG           15 0 4 0 01234
NA        TAG           15 0 4 0 01234
```

4.3.12. Clear A (21) ZA

The contents of the A register will be cleared to zero by this operation. This is a variation of the Subtract A instruction where the A register is subtracted from itself due to a k designator of 7 (see 4.8.3.1).

Operation	k		y	b	j
ZA	none	␢	none	none	NORMAL

Example:

```
AN,A                    21 0 7 0 00000
ZA                      21 0 7 0 00000
```

4.3.13. Store Bj (16) SB

Store the contents of a selected B register, indicated by the j designator, in the k designator-modified portion of the storage location specified by $\bar{y}$.

Operation	k	Space	j	y	b
SB	Note 2	␢	B_j (Note 1)	STORE CLASS	B_b

Examples:

```
SB,1   B7,LABEL             16 7 1 0 01234
SB,1   B7,LABEL-3,B2        16 7 1 2 01231
SB,1   B0,LABEL             16 0 1 0 01234
```

(L(LABEL) is cleared to zero by the preceding instruction.)

```
SB,CPL B0,LABEL             16 0 5 0 01234
```

(L(LABEL) is set to all ones by the preceding instruction.)

NOTE 1: The j designator specifies the selected B register; consequently, a skip condition cannot be programmed in this instruction.

The j of the B_j notation may be 0, 1, 2, 3, 4, 5, 6, or 7 to specify the B register of the operation. With a j designator of 0, all zeros are stored at the $\bar{y}$ specified; program advances to NI.

NOTE 2: It is necessary to use a full word (30-bit) transfer k of 0, 3, 4, or 7 in order to store the contents of an active 17-bit index register.

4.3.14. Clear Y (16) SZ

The k determined portion of a storage location specified by $\bar{y}$ is cleared to zero. This is a variation of the Store B_j instruction where B_j is B0.

Operation	k		j	y	b
SZ	See 4.3.13	ƀ	none	STORE CLASS	B_b

Example:

```
SB,W      B0,TAG        16 0 3 0 01234
SZ,W      TAG           16 0 3 0 01234
```

4.3.15. Double Precision Store (7725) DPS

The contents of the AQ register will be stored in $\bar{y}$ and $\bar{y}+1$; (A) in $\bar{y}$ and (Q) in $\bar{y}+1$.

Operation		y, b
DPS	ƀ	base address

Examples:

```
DPS    LABEL              77 25 0 01234
DPS    LABEL, B4          77 25 4 01234
```

4.3.16. Character Pack Lower (7731) CPL

This operation packs five 6-bit characters into the A register.

Operation		y, b
CPL	␢	base address

Initiate a transfer to A of five 6-bit characters located in bits 5-0 of the addresses specified by $\bar{y}$ through $\bar{y}+4$. $(\bar{y})_{5\text{-}0}$ will enter $A_{29\text{-}24}$; $(\bar{y}+1)_{5\text{-}0}$ will enter $A_{23\text{-}18}$; etc.

Examples:

```
        CPL    LABEL                    77 31 0 01234
        CPU    LABEL, B2                77 31 2 01234
```

4.3.17. Character Pack Upper (7732) CPU

This operation packs five 6-bit characters into the A register.

Operation		y, b
CPU	␢	base address

Initiate a transfer to A of five 6-bit characters located in bits 20-15 of the addresses specified by $\bar{y}$ through $\bar{y}+4$. $(\bar{y})_{20\text{-}15}$ will enter $A_{29\text{-}24}$; $(\bar{y}+1)_{20\text{-}15}$ will enter $A_{23\text{-}18}$; etc.

Example:

```
        CPU    LABEL                    77 32 0 01234
        CPU    LABEL, B4                77 32 4 01234
```

4.3.18. Character Unpack Lower (7735) CUL

This operation unpacks the A register into $\bar{y}$ through $\bar{y}+4$.

Operation		y, b
CUL	␢	base address

Five 6-bit characters are taken from the A register and distributed into bits 00 through 05 of the five locations, $\bar{y}$ through $\bar{y}+4$. $A_{24\text{-}29}$ is transferred to $(\bar{y})_{00\text{-}05}$; $A_{18\text{-}23}$ to $(\bar{y}+1)_{00\text{-}05}$; etc.

Examples:

```
        CUL    LABEL                    77 35 0 01234
        CUL    LABEL, B3                77 35 3 01234
```

4.3.19. Character Unpack Upper (7736) CUU

This instruction unpacks the A register into the upper half of addresses $\bar{y}$ through $\bar{y}+4$.

Operation		y,b
CUU	ъ	base address

Five 6-bit characters are taken from the A register and distributed successively into bits 15 through 20 of the five locations $\bar{y}$ through $\bar{y}+4$. $A_{24\text{-}29}$ is transferred to $(\bar{y})_{0\text{-}5}$, $A_{18\text{-}23}$ to $(\bar{y}+1)_{0\text{-}5}$, . . . , $A_{0\text{-}5}$ to $(\bar{y}+4)_{0\text{-}5}$

Examples:

```
CUU    LABEL              77 36 0 01234
CUU    LABEL, B5          77 36 5 01234
```

4.3.20. Load B-Worker (7771) LBW

This instruction transfers the lower 15 (or 17) bits from seven successive memory locations to the seven worker B registers. The first transfer is from the base address to B1; the second, from the next address to B2, etc. Transfers to B1, B2, and B3 are 15 bits; to B4, B5, B6, and B7, 17 bits. The initial contents of the B registers are available for base address modification.

Operation		y, b
LBW	ъ	base address

Example:

```
LBW       LABEL, B0        77 71 0 01234
```

4.3.21. Store B-Worker (7775) SBW

This instruction transfers 15 (or 17) bits from the seven worker B registers to seven successive memory locations. The first transfer is from B1 to the base address; the second, from B2 to the next address, etc. Transfers from B1, B2, and B3 are 15 bits; from B4, B5, B6, and B7, 17 bits. Transfers are made to the lower 15 (or 17) bits of the memory word with the upper 15 (or 13) bits zero filled. The B registers are unchanged and are available for base address modification.

Operation		y, b
SBW	ъ	base address

Example:

```
SBW       LABEL, B2        77 75 2 01234
```

4.4. SHIFT INSTRUCTIONS

Shift instructions move the contents of a selected register to the right or left as many positions as indicated. In a right shift, all bits shifted out at the right are lost. In the basic right shift, vacated bit positions are sign extensions; in the logical right shift, zero filled. All left shifts are circular, i.e., bits shifted out at the left are returned at the right. Except for the Scale Factor Shift (SFS) instruction, the number of positions to be shifted (the shift count) is determined by the number formed by the lowest order six bits of the operand specified under $\bar{y}$. A shift count greater than 59_{10} (73_8) will cause an incorrect shift. In all logical right shifts the first subfield (if used) of the operand must be a number; the second subfield (if used), a B register.

The k designators operate as follows for all shift instructions:

- k of 1,3, or 5 – the shift count is the low order six bits contained in the lower half of the word at address $\bar{y}$.
- k of 2 or 6 – the shift count is the low order six bits contained in the upper half of the word at address $\bar{y}$.
- k of 0 or 4 – the shift count is the low order six bits contained in the instruction word after B register modification – the number $\bar{y}$.
- k of 7 – the shift count is the low order six bits contained in the A register.

4.4.1. Right Shift Q (01) RSQ

Shift contents of the Q register to the right the number of positions specified by the shift count with sign extension. If the shift count is equal to 29_{10} or ranges between 29_{10} and 59_{10} all bit positions of the Q register will be filled with the original value of the sign position.

Operation	k	Space	y	b	j
RSQ	See Figure 4–1	ƀ	READ CLASS	B_b	NORMAL

Examples:

```
RSQ,U    LABEL                          01 0 1 0 01234
      Initial (Q) =100100101111111011100001110101
      Shift Count =8
      Final (Q)   =111111111001001011111110111000
RSQ  29D                                01 0 0 0 00035
      Initial (Q) =011111111111111111111111111111
      Shift Count =29₁₀
      Final (Q)   =000000000000000000000000000000
```

4.4.2. Right Shift A (02) RSA

Shift contents of A register to the right the number of positions specified by the shift count, with sign extension. If the shift count is between 29_{10} and 59_{10} all bit positions of the A register will be filled with the original sign.

Operation	k	Space	y	b	j
RSA	See Table 4–1	ƀ	READ CLASS	B_b	NORMAL

Examples:

```
RSA,L     LABEL                 02 0 1 0 01234
```

4.4.3. Right Shift AQ (03) RSAQ

Shift contents of the AQ register to the right the number of positions specified by the shift count, with sign extension. Both A and Q may be considered as a single combined register, AQ, containing 60 bit positions.

Bits that are shifted off the right end of the Q register are lost; bits that are shifted off the right end of the A register replace the shifted high order positions of the Q register. The sign value (bit position 29 of the A register) will be extended through the shifted high order positions of the A register and into the Q register. If the shift count is between 29_{10} and 59_{10} all bit positions of the A register will contain the initial sign value. If the shift count is 59_{10} all bit positions of both the A and Q registers will contain the initial sign value of the A register.

Operation	k	Space	y	b	j
RSAQ	See Table 4–1	ƀ	READ CLASS	B_b	NORMAL

Examples:

```
RSAQ          12D                    03 0 0 0  00014

Initial contents of AQ:
             (A)                                    (Q)
111010011111111111011010001100       001110101101010010100011001001

Final contents of AQ:

             (A)                                    (Q)
111111111111111010011111111111         011010001100001110101101010010

RSAQ,2        00035,B2               03 0 2 2 00035
```

The shift count will be obtained from the sum of the value in the upper half of word 00035 modified by B2. A shift count 59_{10} is assumed.

```
Initial contents of AQ:
            (A)                                   (Q)
101111000011101010000000000001      111111111100000101010101010101
Final contents of AQ:
            (A)                                   (Q)
111111111111111111111111111111      111111111111111111111111111111
```

4.4.4. Left Shift Q (05) LSQ

Shift contents of the Q register to the left, circularly, the number of positions specified by the shift count. If the shift count is 30_{10}, the Q register will be restored to its initial condition.

Operation	k	Space	y	b	j
LSQ	See Table 4–1	ƀ	READ CLASS	B_b	NORMAL

Examples:

```
LSQ,L       LABEL+1               05 0 1 0 01235

         Initial (Q) = 001110101101010010100011001001

         Shift Count= 15₁₀

         Final (Q)   = 010100011001001001110101101010

 LSQ     30D               05 0 0 0 00036

         Initial (Q) = 010111111100000110000010011100

         Shift Count= 30₁₀

         Final (Q)   = 010111111100000110000010011100
```

4.4.5. Left Shift A (06) LSA

Shift contents of the A register to the left, circularly, the number of positions specified by the shift count. If the shift count is 30_{10}, the A register will be restored to its initial condition.

Operation	k	Space	y	b	j
LSA	See Table 4–1	ƀ	READ CLASS	B_b	NORMAL

4.4.6. Left Shift AQ (07) LSAQ

Shift contents of the AQ register to the left, circularly, the number of positions specified by the shift count. For this instruction, the A and Q registers function as a single 60-bit register, AQ, in which high order bit positions are contained in the A register. The bit positions shifted off the left end of the A register replace the bit positions vacated from the right end of the Q register. Bit positions shifted off the left end of the Q register replace the bit positions vacated from the right end of the A register. If the shift count is 30_{10} the contents of the A register and the Q register will be interchanged.

Operation	k	Space	y	b	j
LSAQ	See Table 4–1	ƀ	READ CLASS	B_b	NORMAL

Examples:

```
LSAQ      6D                          07 0 0 0 00006

Initial (AQ):

               (A)                                (Q)
111010011111111111011010001100    001110101101010010100011001001

Final (AQ):
               (A)                                (Q)
011111111111011010001100001110    101101010010100011001001111010

LSAQ        30D,,ANEG                 07 7 0 0 00036

Initial (AQ):

               (A)                                (Q)
101111101010001100010011101101    000000011111111010101010011001

Final (AQ):
               (A)                                (Q)
000000011111111010101010011001    101111101010001100010011101101
```

4.4.7. Logical Right Shift Q (7751) LRSQ

This operation shifts the Q register right by the shift count with zero fill.

Operation		y, b
LRSQ	ƀ	Number

The shift count N, must be a number and may be modified by a B register. It must not be an address.

Examples:

```
LRSQ      11D                      77 51 0 00013
LRSQ      0, B1                    77 51 1 00000
LRSQ      6, B4                    77 51 4 00006
```

4.4.8. Logical Right Shift A (7755) LRSA

This operation shifts the A register right by the shift count with zero fill.

Operation		y, b
LRSA	ƀ	Number

Example:

```
LRSA      20                       77 55 0 00020
```

4.4.9. Logical Right Shift AQ (7756) LRSAQ

This operation shifts the AQ register right by the shift count with zero fill.

Operation		y, b
LRSAQ	ƀ	Number

Example:

```
LRSAQ     20                       77 56 0 00020
```

4.4.10. Scale Factor Shift (7730) SFS

This instruction shifts the contents of the A register to the left until the two highest order bits are unlike and records the number of shifts required in the Q register as the six lowest order bits. All bits shifted out at the left are returned, in turn, at the right in a circular shift. If all bits in the A register are alike, a count of 28_{10} will be recorded in the Q register, the A register will remain unchanged, and the program will proceed to the next instruction. This instruction is used to "normalize" the fixed point part (mantissa) of a number in the exponential (floating point) word format (Figure 2–4), and is used in floating point arithmetic (see 4.8.6).

Operation		y, b
SFS	ƀ	none

Example:

```
SFS                                77 30 0 00000
```

4.5. TEST (COMPARISON) INSTRUCTIONS

The following test (comparison) instructions extend the ability to perform comparisons over that already available in most instructions by use of the j designator. These comparisons may be either alphanumeric or arithmetic. In an arithmetic comparison, the sign bit is recognized as such, so that a negative number is always treated as less than a positive number, thereby using the following scale of values: any negative number < -0 < +0 < any positive number. In an alphanumeric comparison, no sign bit is recognized as such; the rule is that a 1 bit is greater than a 0 bit. Therefore, as an example, in an alphanumeric comparison, -0 > +0.

4.5.1. Test A (04) TA

Compare the signed value of the operand with the signed (A) and skip the next instruction as determined by the j designator. This comparison can be used on fixed point or floating point binary operands but may not be used for operands in the zoned BCD mode.

Operation	k	Space	y	b	j
TA	NORMAL	␢	READ CLASS	B_b	See Table 4–3

Example:

```
TA,W   LABEL,,YMORE        04 7 3 0 01234
TA,L   LABEL-2,B3,6        04 6 1 3 01232
```

4.5.2. Test Q (04) TQ

Compare the signed value of the operand with the signed (Q) and skip the next instruction as determined by the j designator. This comparison can be used on fixed point or floating point binary operands but may not be used for operands in the zoned BCD mode.

Operation	k	Space	y	b	j
TQ	NORMAL	␢	READ CLASS	B_b	See Table 4–3

Examples:

```
TQ,W   LABEL,,YMORE        04 3 3 0 01234
TQ,1   01234,3,YLESS       04 2 1 3 01234
```

4.5.3. Test Range (04) TR

Compare the signed value of the operand with the signed (A) and (Q) and skip the next instruction as determined by the j designator. It is a range test; the operand must be within a specified range in order to skip NI. This comparison can be used on fixed point or floating point binary operands but may not be used for operands in the zoned BCD mode.

Operation	k	Space	y	b	j
TR	NORMAL	␢	READ CLASS	B_b	See Table 4–3

Example:

```
TR,W    LABEL,,YIN          04 4 3 0 01234
TR,1    LABEL-2,B3,5        04 5 1 3 01232
```

4.5.4. Test Logical Product (43) TLP

Compare the contents of the A register to a masked operand. The comparison is made by forming the logical product (LP) of the Q register and the operand specified as $\overline{y}$. (See "Logical Product" under LOGICAL INSTRUCTIONS.) The logical product is subtracted from the contents of the A register to form a difference. Skip the NI if conditions denoted by the j designator are met. The logical product is then added back to the A register. There is no change in the contents of any of the operational registers as a result of this instruction.

Operation	k	Space	y	b	j
TLP	NORMAL	␢	READ CLASS	B_b	Note

Example:

```
TLP,W    LABEL,,AZERO          43 4 3 0 01234

Operand contents  =  100100001100111000101101110111
Contents of Q     =  000000000000000000000000111111

LP (Y and Q)      =  000000000000000000000000110111
Contents of A     =  000000000000000000000000111111
A–LP(Y and Q)     =  000000000000000000000000001000
```

The jump condition is not present, i.e., A is not zero. The result of the logical operation is then added back to A to return the register to the initial condition. Note that the difference is stored in the A register for j sensing before the A register is returned to its original state.

NOTE: The normal j designator is interpreted as follows:

j = 0:	No skip.
j = 1:	Skip.
j = 2:	Skip if sign in (Q) is positive.
j = 3:	Skip if sign in (Q) is negative.
j = 4:	Skip if difference is + 0.
j = 5:	Skip if difference is not + 0.
j = 6:	Skip if difference has positive sign.
j = 7:	Skip if difference has negative sign.

4.5.5 Double Precision Test Equal (7723) DPTE

Compare the signed contents of the AQ register with the signed contents of the designated double word memory location. (The sign and most significant bits are in the A register and in the first word of the double word location.) If the numbers are equal, the next instruction will be skipped; if unequal, the next instruction will be performed. The AQ register remains unchanged.

Operation		y, b
DPTE	b̸	base address

Examples:

```
DPTE    LABEL                 77 23 0 01234
DPTE    LABEL, B7             77 23 7 01234
```

4.5.6. Double Precision Test Less (7727) DPTL

This instruction will cause a skip if the signed number in the AQ register is less than the signed number in the designated double word memory location. The AQ register remains unchanged.

Operation		y, b
DPTL	b̸	base address

Examples:

```
DPTL    LABEL                 77 27 0 01234
DPTL    LABEL, B7             77 27 7 01234
```

4.5.7. Masked Alphanumeric Test Equal (7753) MATE

This instruction causes a skip if the masked A register is equal to the masked operand, where the Q register is the mask, in an alphanumeric comparison. (The masked A register is the logical product of A and Q. The masked operand is the logical product of $\bar{y}$ and Q.)

Operation		y, b
MATE	b̸	address

Example:

```
MATE    LABEL                 77 53 0 01234
MATE    LABEL, B7             77 53 7 01234
```

4.5.8. Masked Alphanumeric Test Less (7757) MATL

This instruction causes a skip if, in an alphanumeric comparison, the masked A register is less than the masked operand.

Operation		y, b
MATL	ƀ	address

Example:

```
MATL    LABEL               77 57 0 01234
MATL    LABEL, B6           77 57 6 01234
```

4.6. JUMP INSTRUCTIONS

Instructions are normally executed in sequential order. Jump instructions are used to depart from this sequential order and may also specify a point in the program at which the sequential order will again be resumed (return jump). The jump may be unconditional or it may be based on various conditions. Manual jumps (depending on manual key settings) should be used with caution for programs under control of the Executive. Table 4–1 describes interpretation of the k designator for jump instructions; Table 4–4, the j designator.

A read class operand may be specified with the following restrictions:

- If the A register is specified, only the 15 low order bit positions will be meaningful.
- If a B register is specified, no sign extension is permitted.
- If an actual computer address is used, it cannot exceed 77776 or the decimal equivalent.

4.6.1. Jump (61) J

A jump instruction may be unconditional or manual (depending upon a manual key setting). If the jump condition is not satisfied, control proceeds to the next sequential instruction.

Operation	k	Space	y	b	j
J	See Table 4–1	ƀ	See 4.6.	B_b	See Table 4–4

Examples:

```
J, U    LABEL,,KEY 1    61 1 2 0 01234
J, U    LABEL           61 0 2 0 01234
```

4.6.2. Jump on Test (60) JT

This jump may be conditioned by an arithmetic test of the A or Q registers or it may condition the state of interrupt lockout.

Operation	k	Space	y	b	j
JT	See Table 4–1	␢	See 4.6	B_b	See Table 4–4

Examples:

```
JT,L      LABEL,,Q POS      60 2 1 0 01234
JT        LABEL,,ANOT       60 5 0 0 01234
```

4.6.3. Store Location and Jump (65) SLJ

This return jump may be unconditional or manual (depending on a manual key setting). If an unconditional jump is specified, or if the jump condition exists, a jump is made to the address specified in $\overline{y}$ incremented by 1, i.e., $\overline{y}+1$. The address of the instruction immediately following this jump is stored in the lower portion of the storage location at the address specified in $\overline{y}$. If the jump condition is not satisfied, the instruction immediately following the return jump instruction is executed.

Operation	k	Space	y	b	j
SLJ	See Table 4–1	␢	See 4.6	B_b	See Table 4–4

Examples:

```
SLJ      LABEL,,KEY      65 3 0 0 01234
SLJ      LABEL           65 0 0 0 01234
```

If the SLJ instruction shown above is located at address 23344, location 01234 will appear as follows:

```
LABEL                    00000 23345  (ȳ)
                         ..........   (ȳ + 1)
```

A jump instruction could terminate the sequence of instructions and bring the program back to the instruction following the SLJ instruction as follows:

```
J, L     LABEL
```

4.6.4. Store Location and Jump Test (64) SLJT

This instruction is arithmetic; i.e., it depends upon the contents of an arithmetic register. If the jump condition exists, a jump is made to the address specified by $\overline{y}$ incremented by 1, i.e., $\overline{y}+1$. The address of the instruction immediately following is stored in the lower portion of the storage location at the address $\overline{y}$. If the jump condition is not satisfied, the next sequential instruction is executed.

Operation	k	Space	y	b	j
SLJT	See Table 4–1	␢	See 4.6	B_b	See Table 4–4

Example:

```
SLJT    LABEL,,ANEG     64 7 0 0 01234
```

4.6.5. Execute Remote (7737) ER

Jump to the instruction at location $\bar{y}$. If that instruction does not skip or jump, return to the instruction following the Execute Remote after its execution.

Operation	Space	y, b
ER	␢	remote address

Examples:

```
ER      LABEL           77 37 0 01234
ER      LABEL, B6       77 37 6 01234
```

4.6.6. Enter B_x and Jump (7740–7747) LBPJ

Load the B register specified by the x designator (where x = B0 through B7) with the current setting of the P register (of NI) and jump to $\bar{y}$. If x = B0, an unconditional jump occurs. The x designator specifies the B register to be loaded; the b designator, the B register used for address modification.

Operation	Space	x,	y, b
LBPJ	␢	B_b	address

Examples:

```
LBPJ    B1, LABEL           77 41 0 01234
LBPJ    B4, LABEL, B2       77 44 2 01234
```

4.7. SEQUENCE-MODIFYING INSTRUCTIONS

These instructions permit modification of the normal sequential execution of instructions and include:

- Repeat instruction – enabling repetition of an instruction.
- Test B and/or Increment instruction – enabling the contents of a specified B register to condition a skip.
- Jump on B and Decrement instruction – enabling the contents of a specified B register to condition a jump to a specified location.
- Test and Set instruction – enabling generation of an interrupt and a jump to address 00030 as conditioned by bit 14 of the selected operand.
- Executive Return instruction – enabling generation of an interrupt and a jump to address 00007.

4.7.1. Repeat (70) R

This instruction causes the instruction immediately following it to be repeated the number of times specified by a 15-bit or 17-bit value in the $\bar{y}$ operand. The value may be equal to or greater than zero but cannot be greater than $131{,}071_{10}$. The value determined by $\bar{y}$ is placed in B7. If the value is zero the instruction immediately following the Repeat instruction is skipped. If the value is not zero, the repeat mode which is determined by the j portion of the instruction is initiated. Any modifications to the initial instruction are performed in transient registers; the instruction as it is stored in the computer is not altered. If a conditional skip is performed over a storage location containing a Repeat instruction, the next instruction (following the Repeat instruction) will be executed once. If the repeated instruction specifies a skip condition with its j designator, this designator may cause termination of the repeat mode (a skip of the Repeat instruction) when the skip condition is satisfied, even though the repeat count is not satisfied.

Operation	k	Space	y	b	j
R	See Table 4–1	ƀ	See Table 4–1	B_b	See Table 4–5

Example:

The following example shows how the Repeat instruction can be used to clear 30_{10} successive locations in storage.

```
ZA                    21 0 7 0 00000
R       30D,,ADV      70 1 0 0 00036
SA, W   BUFFER        15 0 0 3 BUFFER
```

4.7.2. Test Bj and/or Increment (71) TBI

This operation tests the contents of a specified B register. If the value in the B register is equal to the value in a memory location specified by $\bar{y}$, the B register is cleared and the next operation is skipped. If the value in the B register is not equal to the operand location, the value in the B register is incremented by 1 and the normal sequence of operations continues.

Operation	k	Space	j	y	b
TBI	See Table 4–1	ƀ	B_j	Note	B_b

j may be 0, 1, 2, 3, 4, 5, 6, or 7 to specify the B register being tested.

NOTE: A read class operand. The form TBI,W refers to the low order 15 bits of the operand location.

Examples:

```
TBI, L   B2,LABEL+3        71 2 1 0 01237
TBI      B2,,B2            71 2 0 2 00000
(This instruction will clear B2 and skip.)
```

Either of the above instructions will result in the low order 15 bits of 01237 being referenced as $\overline{y}$.

```
TBI,W    B1,LABEL+3        71 1 3 0 01237
TBI,L    B2,LABEL+3        71 2 1 0 01237
```

4.7.3. Jump on Bj and Decrement (72) JBD

This instruction tests the content of a specified B register. If the value in the register is zero, the normal sequence of operations continues. If the value in the register is not zero, the register is decremented by 1 and a new sequence of operations begins at the address specified by the $\overline{y}$ operand.

Operation	k	Space	j	y	b
JBD	Note	␢	B_j	Note	B_b

j may be 0, 1, 2, 3, 4, 5, 6, or 7 to specify the B register being tested. If j is 0, a No Op results.

NOTE: A read class operand. The form JBD,W refers to the low order 15 bits of the operand location.

Examples:

```
JBD,W  B4,LABEL,B2        72 4 3 2 01234
JBD,L  B4,LABEL,B2        72 4 1 2 01234
```

4.7.4. Test and Set (7752) TSET

Test bit 14 of $\overline{y}$. If this bit is a 0, set bits 0 through 14 to 1 and proceed to the next instruction; if bit 14 is already a 1, interrupt to location 30_8.

Operation	Space	y, b
TSET	␢	address

Examples:

```
TSET   LABEL              77 52 0 01234
TSET   LABEL, B4          77 52 4 01234
```

4.7.5. Executive Return (7754) EXRN

This instruction interrupts the computer to a fixed address (00007) in memory enabling the executive program to capture the P register value of the program which is interrupting.

Operation	Space	y, b
EXRN	ƀ	Note

NOTE: The y portion of this instruction may contain a constant or an address, with or without B register modification, as needed by the user. It has no bearing on the executable instruction.

Examples:

```
EXRN                     77 54 0 00000
EXRN      10             77 54 0 00010
EXRN      LABEL          77 54 5 01234
EXRN      LABEL+10,B5    77 54 5 01244
```

4.8. ARITHMETIC INSTRUCTIONS

This section describes the various arithmetic operations provided by the instruction repertoire of the assembler – addition, subtraction, multiplication, division, and (for decimal instructions) testing of results.

4.8.1. General

Arithmetic operations may be performed upon different types of operands – fixed point binary, floating point, and BCD (binary coded decimal). The following paragraphs summarize assembler features for each of the different operand modes.

4.8.1.1. Integer (Fixed Point) Addition and Subtraction

The actual mechanics of the arithmetic operations are beyond the scope of this manual but the following characteristics of these operations are of interest to the programmer:

- The programmer must guard against overflow conditions. For single word operations, the absolute value of the operands and results should not exceed $2^{29}-1$; for double word operations, $2^{59}-1$. In the event of an overflow the result will be incorrect.

- A sum of negative zero cannot be generated unless both addend and augend are negative zeros. A difference of negative zero cannot be generated unless the minuend (the A register) is a negative zero and the subtrahend (Y) is a positive zero. In all other cases involving an operand of negative zero, the same result is obtained as if a positive zero were used in its place. These cases are shown in the following:

Generation of negative zero:

```
  77777      77777
 +77777     -00000
 ------     ------
  77777      77777
```

4.8.1.2. Integer (Fixed Point) Multiplication and Division

Fixed point multiplication and division are performed as a series of additions and/or subtractions. The following characteristics of these operations are of interest to the programmer:

- The result of multiplication will have the correct algebraic sign. Where signs of operands are alike, the product will be positive; where unlike, negative. In division, the sign of the quotient (which will be in the Q register) will have its sign determined in the same manner. The remainder (which will be in the A register) will have the same sign as the quotient.

- In multiplication, the entire product *will* be in the Q portion of the AQ register if bit position 28-n of the multiplier contains a sign bit, where n is the most significant bit position of the multiplicand. (The most significant bit position is the highest order bit position containing a 1 in a positive number or a 0 in a negative number.) The entire product *may* be in the Q portion if bit position 28-n has the most significant bit. The entire product *will* spill over into the A portion of the AQ register if bit position 29-n does not contain a sign bit. No product can be generated that will overflow the AQ register. The maximum positive product is 17777777770000000001_8; the maximum negative product, 60000000007777777776_8.

- In division, the quotient is retained in the Q portion of the AQ register. The dividend in the AQ register may have up to 59 significant bits while the divisor may have as few as 1. In these cases, a quotient may be generated that has as many as 59 significant bits. Since the Q register has a 30-bit capacity, an overflow situation will result when a quotient is generated that has more than 29 significant bits. An overflow *will not* occur if the dividend has no significant bits past bit position n + 28 where n is the most significant bit position of the divisor. An overflow *will* occur if the dividend has a signficant bit past bit position n + 29. An overflow *may* occur if the most significant bit of the divident is in bit position n + 29. If overflow does occur, the quotient will appear as +0 or −0 (depending upon the similarity of signs in divisor and dividend). For j interpretation, the Q register will contain −0 if an overflow occurs.

A negative zero in the A or Q portion of the AQ register may have an adverse affect on further calculations and will be caused by the following conditions:

- The dividend is an integral multiple of the divisor (within the limits of resolution), the signs of both are different, and both values are not 0 (+ or −). The quotient will be correct but (A) will be −0. For j sensing, (Q) will appear as the absolute value of the quotient and (A) will be +0.

- The absolute value of the divisor is greater than the absolute value of the dividend, signs are different, and both are nonzero. In this case, (Q) will be −0 and (A) will be the ones complement of the absolute value of the dividend. For j sensing, (Q) is +0 and (A) is the absolute value of the dividend.

Division by +0 or −0 has the following results:

- If a positive number is divided by +0, (Q) will be −0 and the remainder in (A) will be the initial (Q). For j sensing, these final (Q) and (A) are used.
- If a negative number is divided by +0, (Q) will be +0 and the remainder in (A) will be the initial (Q). For j sensing, (Q) will be −0 and (A) will be the complement of the initial (Q).
- If a positive number is divided by −0, (Q) will be +0 and the remainder in (A) will be the ones complement of the initial (Q). For j sensing, (Q) will be −0, and (A) will be the initial (Q).
- If a negative number is divided by −0, (Q) will be −0 and the remainder in (A) will be the ones complement of the initial (Q). For j sensing, these final (Q) and (A) are used.

4.8.1.3. Floating Point Arithmetic

In floating point arithmetic, the following are of interest to the programmer:

- During execution, a floating point overflow interrupt is generated (turning control over to the executive) if an exponent of an operand is greater than 1023_{10} or when division by a ±0 (floating point) is attempted.
- During execution, a floating point underflow interrupt is generated if the exponent is less than -1023_{10}.

4.8.1.4. Decimal (BCD) Arithmetic

BCD (binary coded decimal) addition and subtraction can be programmed directly, where the BCD digits are present in the form of zoned BCD digits – six bits per character – such as Fieldata code, for example. The zone bits are disregarded during the arithmetic operation but are returned in the result. These decimal operations are available for single precision (ten or less BCD digits per operand and result) or for multiprecision operations. Instructions are present to test for an overflow borrow or carry after an operation.

4.8.2. Fixed Point Single Word Addition

These instructions consist of the following:

- Add A
- Add Q
- Load Y + Q
- Store A + Q
- Replace A + Y
- Replace Y + Q
- Replace Y + or Increment Y

4.8.2.1. Add A (20) A

This instruction adds a specified operand to the contents of the A register and retains the sum in the A register.

Operation	k	Space	y	b	j
A	NORMAL	ƀ	READ CLASS	B_b	NORMAL

Examples:

```
A,LX    LABEL,2        20 0 5 2 01234
A       773            20 0 0 0 00773
```

4.8.2.2. Add Q (26) AQ

Add a specified operand to the contents of the Q register and retain the sum in the Q register.

Operation	k	Space	y	b	j
AQ	NORMAL	ƀ	READ CLASS	B_b	See Table 4–6

Example:

```
AQ,U    LABEL,,QNOT    26 5 2 0 01234
```

4.8.2.3. Load Y + Q (30) LAQ

Add a specified operand to the contents of the Q register and retain the sum in the A register. The contents of the Q register and operand are undisturbed by this instruction.

Operation	k	Space	y	b	j
LAQ	NORMAL	ƀ	READ CLASS	B_b	NORMAL

Example:

```
LAQ,W   LABEL          30 0 3 0 01234
```

4.8.2.4. Store A + Q (32) SAQ

Add the contents of the A and Q registers, retain the sum in the A register, and store the sum in the storage location specified.

Operation	k	Space	y	b	j
SAQ	NORMAL	ƀ	STORE CLASS	B_b	NORMAL

Example:

```
SAQ,U   LABEL,B3       32 0 2 3 01234
```

4.8.2.5. Replace A + Y (24) RA

Add a specified operand to the contents of the A register. Retain this sum in the A register and replace the original operand with this sum.

Operation	k	Space	y	b	j
RA	NORMAL	␢	REPLACE CLASS	B_b	NORMAL

Example:

```
RA,UX   LABEL + 3         24 0 6 0 01237
```

4.8.2.6. Replace Y + Q (34) RAQ

Add the specified operand to the contents of the Q register, retain the sum in the A register, and store the sum in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RAQ	NORMAL	␢	REPLACE CLASS	B_b	NORMAL

Example:

```
RAQ,W   0,B7              34 0 3 7 00000
```

4.8.2.7. Replace Y + 1 or Increment Y (36) RI

Increment the specified operand by 1, retain the sum in the A register, and store this sum in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RI	NORMAL	␢	REPLACE CLASS	B_b	NORMAL

Example:

```
RI,UX   LABEL,B3,AZERO        36 4 6 3 01234
```

4.8.3. Fixed Point Single Word Subtraction

These instructions consist of the following:

- Subtract A
- Subtract Q
- Load Y–Q
- Store A–Q
- Replace A–Y
- Replace Y–Q
- Replace Y–1 or Decrement Y

4.8.3.1. Subtract A (21) AN

Subtract a specified operand from the contents of the A register and retain the difference in the A register.

Operation	k	Space	y	b	j
AN	NORMAL	␢	READ CLASS	B_b	NORMAL

Examples:

```
AN,W    LABEL,,ANOT    21 5 3 0 01234
AN,3    LABEL,3,5      21 5 3 3 01234
```

4.8.3.2. Subtract Q (27) ANQ

Subtract a specified operand from the contents of the Q register and retain the difference in the Q register.

Operation	k	Space	y	b	j
ANQ	NORMAL	␢	READ CLASS	B_b	See Table 4-6

Examples:

```
ANQ     12D,,QNOT      27 5 0 0  00014
ANQ     12D,4,4        27 4 0 4  00014
```

4.8.3.3. Load Y-Q (31) LANQ

Subtract the contents of the Q register from a specified operand and retain the difference in the A register. The contents of the Q register are not disturbed by this instruction.

Operation	k	Space	y	b	j
LANQ	NORMAL	␢	READ CLASS	B_b	NORMAL

Example:

```
LANQ,L     LABEL              31 0 1 0 01234
LANQ,U     LABEL,3,2          31 2 2 3 01234
LANQ,W     LABEL,5,AZERO      31 4 3 5 01234
```

4.8.3.4. Store A–Q (33) SANQ

Subtract the contents of the Q register from the A register, retain the difference in the A register, and store this difference in the storage location specified.

Operation	k	Space	y	b	j
SANQ	NOTE	ƀ	STORE CLASS	B_b	NORMAL

Examples:

```
SANQ,6        LABEL,3       33 0 6 3 01234
SANQ,CPL      LABEL         33 0 5 0 01234
```

NOTE: The k designator governs storage in both the memory and A register.

4.8.3.5. Replace A–Y (25) RAN

Subtract a specified operand from the contents of the A register, retain the difference in the A register, and store this difference in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RAN	NORMAL	ƀ	REPLACE CLASS	B_b	NORMAL

Examples:

```
RAN,L  LABEL           25 0 1 0 01234
RAN,U  LABEL,,AZERO    25 4 2 0 01234
```

4.8.3.6. Replace Y–Q (35) RANQ

Subtract the contents of the Q register from a specified operand, retain the difference in the A register, and store this difference in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RANQ	NORMAL	ƀ	REPLACE CLASS	B_b	NORMAL

Example:

```
RANQ,L        LABEL         35 0 1 0 01234
```

4.8.3.7. Replace Y−1 or Decrement Y (37) RD

Decrement a specified operand by 1, retain the difference in the A register, and store this difference in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RD	NORMAL	␢	REPLACE CLASS	B_b	NORMAL

Examples:

```
RD,UX        LABEL,,AZERO  37 4 6 0 01234
RD,6         LABEL+2,B3    37 0 6 3 01236
```

4.8.4. Fixed Point Double Word Arithmetic

The fixed point double word arithmetic instructions use ones complementation arithmetic for addition, subtraction, and ones complementation (or negation) upon 60-bit operands. In fixed point double word arithmetic, the sign and high order bits are retained in the A register or stored in the first of two consecutive addresses referenced by y. The Q register retains the low order 30 bits; the next consecutive address stores the low order 30 bits. These instructions include:

- Double Precision Add
- Double Precision Subtract
- Double Precision Complement

4.8.4.1. Double Precision Add (7722) DPA

The contents of $\overline{y}$ and $\overline{y}$ + 1 are added to the contents of the double length AQ register. The sum will be in AQ.

Operation	Space	y, b
DPA	␢	base address

Examples:

```
DPA    LABEL          77 22 0 01234
DPA    LABEL, B2      77 22 2 01234
```

4.8.4.2. Double Precision Subtract (7726) DPAN

The contents of $\overline{y}$ and $\overline{y}$ + 1 are subtracted from the contents of the double length register AQ. The difference will be in AQ.

Operation	Space	y, b
DPAN	␢	base address

Examples:

```
DPAN   LABEL          77 26 0 01234
DPAN   LABEL, B3      77 26 3 01234
```

4.8.4.3. **Double Precision Complement (7724) DPN**

The contents of the AQ register are converted to its ones complement.

Operation	Space	y, b
DPN	ƀ	none

Example:

```
        DPN                              77 24 0 00000
```

4.8.5. Fixed Point Multiplication and Division

The fixed point multiplication and division instructions use ones complementation arithmetic, and include:

- Multiply
- Divide

4.8.5.1. **Multiply (22) M**

This instruction multiplies the contents of the Q register by the operand specified in the instruction. The product is formed in the 60 bit positions of the combined AQ register.

Operation	k	Space	y	b	j
M	NORMAL	ƀ	READ CLASS	B_b	Note

The following rules apply for multiplication:

- If a positive number is multiplied by a positive number or a negative number by a negative number, the product will be positive.
- If a positive number is multiplied by a negative number or a negative number by a positive number, the product will be negative.
- If positive 0 is multiplied by positive 0 or negative 0 by negative 0, the product will be positive 0.
- If positive 0 is multiplied by negative 0 or negative 0 by positive 0, the product will be negative 0.

No product can be generated which will overflow AQ. The maximum positive product is:

$$\underbrace{17777\ 77777}_{A} \qquad \underbrace{00000\ 00001}_{Q}$$

The maximum negative product is:

$$\underbrace{60000\ 00000}_{A} \qquad \underbrace{77777\ 77776}_{Q}$$

Example:

```
        M, W    LABEL                  22 0 3 0 01234
```

The result of operations for various values contained in the Q register and the A register (initially the location defined by LABEL) follows:

```
                A                   Q                          AQ

        00000  00012  X  00000  00010  =  00000  00000  00000  00120
        77777  77767  X  00000  00012  =  77777  77777  77777  77657
        77777  77765  X  77777  77767  =  00000  00000  00000  00120
```

NOTE: The skip condition is tested prior to any final sign conversion. The significance of the normal skip condition for a multiple operation may be outlined as follows:

j Machine	J MNEMONIC	SKIP CONDITION
0	(absent)	No skip.
1	SKIP	Skip next instruction.
2	QPOS	Skip next instruction if there is no overflow into the A register. If a skip does not occur, a double length product is indicated since there is a significant bit in bit position 29 of the Q register. (The highest order bit that is different from the sign bit is the most significant bit.)
3	QNEG	Skip next instruction if there is an overflow into the A register. If a skip occurs, a double length product is indicated since there is a significant bit in bit position 29 of the Q register.
4	AZERO	Skip next instruction if the product is entirely within the Q register. If a skip occurs, it indicates that the product has 30 or less significant bits, and that the A register contains only sign bits. This does not mean the Q register contains the correct product, since bit position 29 of the Q register may contain a significant bit of the product, thus making bit position 0 of the A register the first sign bit. If a skip does not occur, it indicates that significant bits of the product are in the A register.
5	ANOT	Skip next instruction if product overflows. If a skip occurs, it indicates that significant bits of the product are in the A register. If a skip does not occur, it indicates the same condition that exists when a skip occurs with AZERO.
6	APOS	Skip next instruction.
7	ANEG	Do not skip next instruction.

4.8.5.2. Divide (23) D

This instruction divides the contents of the combined AQ register by the operand specified in the instruction and retains the quotient and remainder in the Q and A registers, respectively.

Operation	k	Space	y	b	j
D	NORMAL	ƀ	READ CLASS	B_b	See Table 4–6

The following rules apply for division:

- If a positive number is divided by a positive number or a negative number by a negative number, the quotient and remainder will be positive numbers.
- If a positive number is divided by a negative number or a negative number by a positive number, the quotient and remainder will be negative numbers.

Negative Zero Quotients and Remainders

Division, if handled improperly, may generate a negative 0 quotient or remainder that can have an adverse affect on further calculations. This situation can occur in the following four cases:

- Remainder is zero, the dividend and divisor have unlike signs, and are both nonzero.
- Absolute value of divisor is greater than the absolute value of the dividend, signs are unlike, and both are nonzero.
- Division by positive or negative zero
- Division of positive or negative zero by nonzero divisor with an unlike sign.

(1) Remainder is zero, the dividend and divisor have unlike signs and are both nonzero:

The result of such a division is that the correct quotient will be in the Q register and the remainder in the A register will be a negative 0.

When the j designator is interpreted, the Q register will appear as the absolute value of the quotient and the A register will appear as a positive 0. For example:

```
000000000000000000000000000000  000000000000000001010011100101  (dividend)

                                111111111111111110101100011010  (divisor)
--------------------------------------------------------------
                                000000000000000000000000000010  (quotient)
     At j
 interpretation                 000000000000000000000000000000  (remainder)

                                111111111111111111111111111101  (quotient in the
 Final Result                                                    Q register)
                                111111111111111111111111111111  (remainder in
                                                                 the A register)
```

(2) Absolute value of the divisor is greater than the absolute value of dividend, signs are unlike, and both are nonzero:

When division is performed in this case, the quotient in the Q register will be a negative 0 and the remainder in the A register will be the ones complement of the absolute value of the dividend. When the j designator is interpreted, the Q register will appear as a positive 0 and the A register will appear as the absolute value of the dividend. For example:

```
000000000000000000000000000000 000000000000000000000000000011   (dividend)

                               111111111111111111111111111010   (divisor)
-------------------------------------------------------------
                               000000000000000000000000000000   (quotient)
          At j
     interpretation            000000000000000000000000000011   (remainder)

                               111111111111111111111111111111   (quotient in the
     Final Result                                                 Q register)
                               111111111111111111111111111100   (remainder in
                                                                  the Q register)
```

(3) Division by positive or negative zero:

- If a positive number is divided by positive 0, the quotient in the Q register will be a negative 0 and the remainder in the A register will be the initial contents of the Q register. For j designator interpretation, the final contents of the Q and A registers are used.

- If a negative number is divided by a positive 0, the quotient in the Q register will be a positive 0 and the remainder in the A register will be the initial contents of the Q register. For j designator interpretation the Q register will appear as negative 0 and the A register will appear as the ones complement of the initial contents of the Q register.

- If a positive number is divided by negative 0, the quotient in the Q register will be a positive 0 and the remainder in the A register will be the ones complement of the initial contents of the Q register. For j designator interpretation, the Q register will appear as a negative 0 and the A register will appear as the initial contents of the Q register.

- If a negative number is divided by negative 0, the quotient in the Q register will be a negative 0 and the remainder in the A register will be the ones complement of the initial contents of the Q register. For j designator interpretation, the final contents of the Q and A registers are used.

The following examples illustrate these rules:

A positive number divided by positive 0.

```
000000000000000000000000000001  000000000000000000000000000001  (dividend)

                                000000000000000000000000000000  (divisor)
________________________________________________________________
                                111111111111111111111111111111  (quotient)
      At j
 interpretation                 000000000000000000000000000001  (remainder)

                                111111111111111111111111111111  (quotient in the
  Final Result                                                   Q register)
                                000000000000000000000000000001  (remainder in
                                                                 the A register)
```

A negative number divided by a positive 1.

```
111111111111111111111111111111  111111111111111111111111111110  (dividend)

                                000000000000000000000000000000  (divisor)
________________________________________________________________
                                111111111111111111111111111111  (quotient)
      At j
 interpretation                 000000000000000000000000000001  (remainder)

                                000000000000000000000000000000  (quotient in the
  Final Result                                                   Q register)
                                111111111111111111111111111111  (remainder in
                                                                 the A register)
```

A positive number divided by a negative 0.

```
010111111111111111111111111111  110111111111111111111111111111  (dividend)

                                111111111111111111111111111111  (divisor)
________________________________________________________________
                                111111111111111111111111111111  (quotient)
      At j
 interpretation                 110111111111111111111111111111  (remainder)

                                000000000000000000000000000000  (quotient in the
  Final Result                                                   Q register)
                                001000000000000000000000000000  (remainder in
                                                                 the A register)
```

A negative number divided by a negative 0.

```
100111111111111111111111111111   110111111111111111111111111111   (dividend)
                                 111111111111111111111111111111   (divisor)
-----------------------------------------------------------------
                                 111111111111111111111111111111   (quotient)
        At j
   interpretation                001000000000000000000000000000   (remainder)

                                 111111111111111111111111111111   (quotient in the
   Final Result                                                    Q register)
                                 000000000000000000000000000000   (remainder in
                                                                   the A register)
```

(4) Division of positive or negative zero by a nonzero divisor with an unlike sign:

When division is performed in this case, the quotient in the Q register and the remainder in the A register will be a negative 0. When the j designator is interpreted, both the Q and A register will appear as a positive 0. The following examples will illustrate this:

```
000000000000000000000000000000   000000000000000000000000000000   (dividend)
                                 111111111111111111111111111110   (divisor)
-----------------------------------------------------------------
                                 000000000000000000000000000000   (quotient)
        At j
   interpretation                000000000000000000000000000000   (remainder)

                                 111111111111111111111111111111   (quotient in the
   Final Result                                                    Q register)
                                 111111111111111111111111111111   (remainder in
                                                                   the A register)

111111111111111111111111111111   111111111111111111111111111111   (dividend)
                                 000000000000000000000000000000   (divisor)
-----------------------------------------------------------------
                                 000000000000000000000000000000   (quotient)
        At j
   interpretation                000000000000000000000000000000   (remainder)

                                 111111111111111111111111111111   (quotient in the
   Final Result                                                    Q register)
                                 111111111111111111111111111111   (remainder in
                                                                   the A register)
```

Divide Overflow with Nonzero Divisor and Dividend

In division, the dividend in the AQ register may have up to 59 significant bits while the divisor may have as few as 1. In these cases, a quotient may be generated that has as many as 59 significant bits. Since the Q register has a 30-bit capacity, an overflow situation will result when a quotient is generated that has more than 29 significant bits. If overflow does occur, the quotient in the Q register will be a positive 0 if the divisor and dividend have unlike signs, or it will be a negative 0 if the signs were the same. At the time the j designator is interpreted, the Q register will appear as a negative 0.

The following rules govern the occurrence of a divide overflow:

- If the most significant bit of the divisor is in bit position n, a divide overflow will not occur if the dividend has no significant bits beyond bit position n+28.
- If the most significant bit of the divisor is in bit position n, a divide overflow will occur if the dividend has a significant bit in bit position n+30 or beyond.
- If the most significant bit of the divisor is in bit position n, a divide overflow may occur if the most significant bit of the dividend is in bit position n+29.

The following examples illustrate these rules:

- No overflow.

```
000000000000000000000000000000   000000000000000010110001011100  (dividend)
                                 000000000000000000001010011100  (divisor)
-----------------------------------------------------------------
                                 000000000000000000000000010001  (quotient)
         At j
    interpretation               000000000000000000000000000000  (remainder)

                                 000000000000000000000000010001  (quotient in the
    Final Result                                                  Q register)
                                 000000000000000000000000000000  (remainder in the
                                                                  A register)
```

- Overflow occurs.

```
000000000000000100011001100011   010011110111100111100110101011  (dividend)
                                 000000000000001000000001000000  (divisor)
-----------------------------------------------------------------
                                 111111111111111111111111111111  (quotient)
         At j
    interpretation               010011110111100001100111101011  (remainder)

                                 111111111111111111111111111111  (quotient in the
    Final Result                                                  Q register)
                                 010011110111100001100111101011  (remainder in the
                                                                  A register)
```

- Overflow may occur.

```
111111111111111011100101011100  101100001000011010001100111111  (dividend)

                                000000000000000010000001000000  (divisor)
--------------------------------------------------------------
                                111111111111111111111111111111  (quotient)
         At j
    interpretation              010011110111100111110011000000  (remainder)

                                000000000000000000000000000000  (quotient in the
    Final Result                                                 Q register)
                                101100001000011000001100111111  (remainder in the
                                                                 A register)
```

(In this example, overflow occurs).

```
000000000000000000000000000001  000000000000000000000000000000  (dividend)

                                000000000000000000000000000011  (divisor)
--------------------------------------------------------------
                                000010001000100010000111101101  (quotient)
         At j
    interpretation              000000000000000000000000000001  (remainder)

                                000010001000100010000111101101  (quotient in the
    Final Result                                                 Q register)
                                000000000000000000000000000001  (remainder in the
                                                                 A register)
```

(In this example, overflow does not occur.)

The remainder in overflow division is difficult to determine and the value of such information, when obtained, is questionable. The rules that are stated below are valid at least in the above examples. They should not, however, be considered universal rules.

- If the dividend and divisor are positive numbers, add the dividend and divisor. The remainder in the A register will be the low order 30 bits of the sum that is formed. At the time the j designator is interpreted, the final contents of the A register will be used.

- If the dividend and divisor are negative numbers, complement the dividend and divisor, and then add them. The remainder in the A register will be the low order 30 bits of the sum that is formed. For interpretation of the j designator, the final contents of the A register will be used.

- If the dividend is a positive number and the divisor is a negative number, the divisor should be ones complemented and then added to the dividend. The final remainder in the A register will be the ones complement of the low order 30 bits of the sum that is formed. For interpretation of the j designator, the contents of the A register will appear as the low order 30 bits of the sum that is formed.

– If the dividend is a negative number and the divisor is a positive number, the dividend should be ones complemented and then added to the divisor. The final remainder in the A register will be the ones complement of the low order 30 bits of the sum that is formed. For j designator interpretation the contents of the A register will appear as the low order 30 bits of the sum that is formed.

Examples:

Examples are for normal division, where all results are shown following final sign correction, if correction is required.

- 00000 00000 00000 26134 ÷ 00000 01234

  quotient in Q = 00000 00021
  remainder in A = 00000 00000
  (A) and (Q) used for j interpretation

- 00000 00000 00000 26152 ÷ 00000 01234

  quotient in A = 00000 00021
  remainder in A = 00000 00016
  (A) and (Q) used for j interpretation

- 777777 777777 777777 51625 ÷ 77777 76543

  quotient in Q = 00000 00021
  remainder in A = 00000 00016
  final (A) and (Q) used for j interpretation

- 02400 21166 21233 52654 ÷ 5400 16354

  quotient in Q = 67777 03046
  remainder in A = 54733 20156
  (A) and (Q) appear as their ones complements for j interpretation

- 75377 56611 56544 25123 ÷ 23777 61423

  quotient in Q = 67777 03046
  remainder in A = 54733 20156
  (A) and (Q) appear as their ones complements for j interpretation

- 00000 00000 00000 12345 ÷ 77777 65432

  quotient in Q = 77777 77776
  remainder in A = 77777 77777 (even division)
  (A) and (Q) appear as their ones complements for j interpretation

4.8.6. Floating Point Arithmetic

Floating point arithmetic enables uses of the exponential (floating point) format (Figure 2–4) for arithmetic with operands having the decimal point in different positions. These instructions include:

- Floating Point Add
- Floating Point Subtract
- Floating Point Multiply
- Floating Point Divide
- Floating Point Pack
- Floating Point Unpack

in addition to the Scale Factor Shift (see 4.4.10).

4.8.6.1. Floating Point Add (7701) FA

The signed floating point number contained in $\bar{y}$ and $\bar{y}$ + 1 is added to the floating point number in the AQ register. The adjustment to AQ is made by a comparison of the characteristics involved. The contents of $\bar{y}$ and $\bar{y}$ + 1 are added to AQ after the comparison. The sum will be contained in AQ.

Operation	Space	y, b
F A	b̸	base address

Examples:

```
FA     LABEL, B1        77 01 1 01234
FA     LABEL            77 01 0 01234
```

4.8.6.2. Floating Point Subtract (7702) FAN

The signed floating point number contained in $\bar{y}$ and $\bar{y}$ + 1 is subtracted from the floating point number in the register. The adjustment to AQ is made by a comparison of the characteristics involved. The contents of $\bar{y}$ and $\bar{y}$ + 1 are subtracted from AQ after the comparison. The difference will be contained in AQ.

Operation	Space	y, b
FAN	b̸	base address

Example:

```
FAN    LABEL, B5        77 02 5 01234
FAN    LABEL            77 02 0 01234
```

4.8.6.3. Floating Point Multiply (7703) FM

The signed floating point number contained in the AQ register is multiplied by the signed floating point number in $\overline{y}$ and $\overline{y}$ + 1. The product is contained in AQ. This product will be normalized and correct only if both numbers were originally normalized.

Operation	Space	y, b
FM	ƀ	base address

Examples:

```
FM    LABEL        77 03 0 01234
FM    LABEL, B2    77 03 2 01234
```

4.8.6.4. Floating Point Divide (7705) FD

The signed floating point number in the AQ register is divided by the signed floating point number in $\overline{y}$ and $\overline{y}$ + 1. Both numbers must be normalized prior to the divide sequence. The normalized quotient will be in the AQ register; any remainder will be discarded.

Operation	Space	y, b
FD	ƀ	base address

Examples:

```
FD    LABEL        77 05 0 01234
FD    LABEL, B1    77 05 1 01234
```

4.8.6.5. Floating Point Pack (7706) FP

This operation forms a floating point number in A and Q using as its mantissa, the value in AQ, and as its characteristic, the biased value represented by $\overline{y}$.

Operation	Space	y, b
FP	ƀ	base address

The mantissa in the combined AQ register is normalized. A biased characteristic is then taken from $\overline{y}$ and inserted in bits A29 – A18 of the AQ register.

Example:

```
FP    LABEL        77 06 0 01234
FP    LABEL, B6    77 06 6 01234
```

4.8.6.6. Floating Point Unpack (7707) FU

This operation separates the characteristic and mantissa of the floating point number in AQ.

Operation	Space	y, b
FU	b̸	address

A positive, biased characteristic is taken from the AQ register and stored at $\overline{y}$. Bits A29 – A18 of the AQ register are sign filled.

Examples:

```
FU      LABEL             77 07 0 01234
FU      LABEL, B4         77 07 4 01234
```

4.8.7. Decimal Arithmetic

These operations expect fixed point, zoned binary-coded-decimal, double precision operands which consist of 10 six-bit characters conforming to a predetermined code similar to the Fieldata code. The "Z" (zone) bits shown in Figure 2–3 are arbitrary and have no effect on arithmetic operations. However, the fifth bit of the lowest order digit is a sign bit. A positive zoned BCD operand is the same as a negative zoned BCD operand (having the same absolute value), except for the sign bit. (See Figure 2–3.)

Decimal arithmetic instructions include:

- Decimal Test AQ
- Decimal Add
- Decimal Add With Carry
- Decimal Subtract
- Decimal Subtract With Borrow
- Decimal Complement AQ
- Decimal Test Less
- Decimal Test Equal
- Convert Lower
- Convert Upper

4.8.7.1. Decimal Test AQ (7710) DT

This operation tests the decimal contents of AQ for one or more of the options listed below.

Operation	Space	y, b
DT	b̸	See list which follows

Wherever there is a 1 bit in the operand, perform the test indicated in the following table for that bit position. If a condition is satisfied in one or more of the tests (more than one test may be specified), skip the next sequential instruction. If none of the conditions is satisfied, or no tests are indicated, advance to the next sequential instruction.

If there is a 1 in bit position	and	then
0	overflow designator = 1 (on)	skip next sequential instruction
1	overflow designator = 0 (off)	
2	(AQ) ≠ 0 and sign is +	
3	(AQ) = 0 (neglecting sign)	
4	(AQ) ≠ 0 and sign is –	
5	sixth decimal digit in AQ ≠ 0	
6	seventh decimal digit in AQ ≠ 0	
7	eighth decimal digit in AQ ≠ 0	
8	ninth decimal digit in AQ ≠ 0	
9	tenth decimal digit in AQ ≠ 0	
10	(AQ) ≠ 0 (neglecting sign)	

NOTES:
1. All tests on (AQ) assume decimal format. Zone bits are not tested.
2. Bit positions 5 through 9 may be used to detect a decimal number exceeding one word in length.
3. Bit positions 11 through 17 of the instruction word have no effect upon the instruction.

Examples:

```
DT     2000          77 10 0 02000
DT     1*/11D        77 10 0 02000
```

(Skip the next instruction if AQ ≠ 0)

4.8.7.2. Decimal Add (7711) DA

This operation adds two decimal numbers.

Operation	Space	y, b
DA	ƀ	base address

The ten-character decimal contents of AQ are added to the decimal contents of $\overline{y}$ and $\overline{y}$ + 1. The sum will be left in the AQ register. The zone bits of AQ will not be changed.

Examples:

```
DA     LABEL          77 11 0 01234
DA     LABEL, B6      77 11 6 01234
```

4.8.7.3. Decimal Add With Carry (7715) DAC

This operation adds two decimal numbers and a carry if present.

Operation	Space	y, b
DAC	ƀ	base address

The contents of $\overline{y}$ and $\overline{y}$ + 1 are added to the AQ register. If the carry has been generated from a previous decimal operation, the carry is added into the least significant position to enable multiprecision operations.

Examples:

```
DAC    LABEL          77 15 0 01234
DAC    LABEL, B1      77 15 1 01234
```

4.8.7.4. Decimal Subtract (7712) DAN

This operation subtracts two decimal numbers.

Operation	Space	y, b
DAN	ƀ	base address

The ten-character decimal contents of $\overline{y}$ and $\overline{y}$ + 1 are subtracted from the contents of the AQ register. The signed result will be in AQ. The zone bits which were in AQ will not be changed.

Example:

```
DAN    LABEL          77 12 0 01234
DAN    LABEL, B3      77 12 3 01234
```

4.8.7.5. Decimal Subtract With Borrow (7716) DANB

This operation subtracts two decimal numbers and a "borrow" if necessary.

Operation	Space	y, b
DANB	ƀ	base address

The contents of $\overline{y}$ and $\overline{y}$ + 1 are subtracted from the contents of the AQ register. If a borrow has been generated from a previous decimal operation, the borrow is subtracted starting from the least significant position. If this operation requires a borrow, the borrow is stored for a succeeding decimal operation to enable multiprecision operations.

Examples:

```
        DANB   LABEL             77 16 0 01234
        DANB   LABEL, B1         77 16 1 01234
```

4.8.7.6. Decimal Complement AQ (7714) DN

The decimal number in the AQ register is converted to its decimal complement. If $\bar{y}$ is odd, the nines complement results (each digit is replaced by its difference from 9); if even, the tens complement (the nines complement plus one). The sign is unchanged.

Operation	Space	y, b
DN	ƀ	Number

Example:

```
        DN     12345             77 14 0 12345
```

4.8.7.7. Decimal Test Less (7717) DTL

Skip the next instruction if the decimal number in AQ is less than the decimal number in $\bar{y}$ and $\bar{y}$ + 1. Zone bits are ignored.

Operation	Space	y, b
DTL	ƀ	base address

Examples:

```
        DTL    LABEL             77 17 0 01234
        DTL    LABEL, B5         77 17 5 01234
```

4.8.7.8. Decimal Test Equal (7713) DTE

Skip the next instruction if the decimal contents of AQ equals the decimal contents of $\bar{y}$ and $\bar{y}$ + 1. Zone bits are ignored.

Operation	Space	y, b
DTE	ƀ	base address

Example:

```
        DTE    LABEL             77 13 0 01234
        DTE    LABEL, B7         77 13 7 01234
```

4.8.7.9. Decimal Convert Lower (7733) DCL

This operation converts decimal to binary.

Operation	Space	y, b
DCL	␢	base address

Initiate a transfer and conversion of $(\bar{y})$i by converting the decimal numbers in bits 3–0 of $\bar{y}$ through $\bar{y}$ + 4 to binary in AQ. The AQ register must have been initially cleared.

Examples:

```
DCL     LABEL           77 33 0 01234
DCL     LABEL, B6       77 33 6 01234
```

This operation is a successive convert-and-shift-left operation into the AQ register. Only the first 34 bit positions of the AQ register can be used for this instruction. Thus, it is possible to convert 9,999,999,999 to binary by two successive conversions of 99999 since this does not require more than 34 bit positions. The result of this operation is not affected by any sign bit in a decimal number; only the absolute decimal digits are converted.

4.8.7.10. Decimal Convert Upper (7734) DCU

This operation converts decimal to binary.

Operation	Space	y, b
DCU	␢	base address

Initiate a transfer and conversion of $(\bar{y})$i by converting the decimal numbers in bits 18–15 of $\bar{y}$ through $\bar{y}$ + 4 to binary in AQ. The AQ register must have been initially cleared. As described for the DCL operation (see 4.8.7.9) only 34 bits are available in the AQ register for this convert-and-shift-left operation.

Examples:

```
DCU     LABEL           77 34 0 01234
DCU     LABEL, B1       77 34 1 01234
```

4.9. LOGICAL OPERATIONS

Logical instructions provide the programmer with the means of operating upon specific bits of a word. These logical operations are the logical product (LP), the OR operation, the NOT operation, the Exclusive OR operation, and the selective substitute. The logical operation is performed upon the bits in the same corresponding bit positions of each of the words to form the resulting word. For all j interpretations which use the contents of a register to determine a skip, it is always the final state of the register which is used for j interpretation.

The logical product is generally used for "masking" (lifting the selected bits of a word and using 0 bits for unselected positions). This is accomplished by placing 1's in the mask to select bits and 0's for the other bits. Wherever there is a 1 in the mask, the corresponding bit of the operand will appear in the logical product. Wherever there is a 0 in the mask, a 0 will appear in the logical product. Thus, the logical product corresponds to the AND function – the logical product is a 1 when the mask *and* the operand are both 1's; otherwise it is a 0. The following example illustrates the logical product:

```
Mask       111 000 001 010 011 100 101 110 111 000
Operand    010 100 110 000 001 011 110 111 101 100
           ---------------------------------------
LP         010 000 000 000 001 000 100 110 101 000
```

The OR operation (selective set) is used to force 1's into selected bits of the A register. Wherever there is a 1 in the operand a 1 is forced into the A register. If the A register bit is already a 1, it remains undisturbed. Wherever there is a 0 in the operand the A register bit remains undisturbed. Described differently, the result is a 1 if the A register bit is a 1 *or* the operand bit is a 1, or both. The following example illustrates operation of the selective set.

```
Operand                010 100 110 000 001 011 110 111 101 100
A register (initial)   111 000 001 101 011 100 101 110 111 000
                       ---------------------------------------
OR                     111 100 111 101 011 111 111 111 111 100
```

The NOT operation (selective clear) forces 0's into selected bits of the A register. Wherever there is a 1 in the operand a 0 will be forced into the A register. If the A register bit is a 0 it remains undisturbed. Wherever there is a 0 in the operand the A register bit remains undisturbed. The selective clear can also be regarded as a modified masking operation. Wherever there is a 0 in the operand (mask) the corresponding bit of the accumulator is lifted and placed in the final result. The following example illustrates operation of the NOT operation:

```
Operand                010 100 110 000 001 011 110 111 101 100
A register (initial)   111 000 001 101 011 100 101 110 111 000
                       ---------------------------------------
A register (final)     101 000 001 101 010 100 001 000 010 000
```

The Exclusive OR operation (selective complement) operates upon selected bits of the A register. Wherever there is a 1 in the operand, the A register bit is ones complemented. Described differently, if either of the two corresponding bits, but not both, is a 1, the result is a 1 bit.

```
Operand                010 100 110 000 001 011 110 111 101 100
A register (initial)   111 000 001 101 011 100 101 110 111 000
                       ---------------------------------------
Exclusive OR           101 100 111 101 010 111 011 001 010 100
```

The selective substitute replaces selected bits in the A register with the corresponding bit of the operand. Selection is performed by the Q register – for each 1 bit in the Q register the substitution is made. The following example illustrates operation of the selective substitute:

```
Q register            101 010 000 111 100 011 110 001 110 011
Operand               010 100 110 000 001 011 110 111 101 100
A register (initial)  111 000 001 101 011 100 101 110 111 000
                      ---------------------------------------
A register (final)    010 000 001 000 011 111 111 111 101 000
```

4.9.1. Load Logical Product (40) LLP

This instruction forms the logical product of the contents of the Q register and an operand and retains it in the A register.

Operation	k	Space	y	b	j
LLP	NORMAL	␢	READ CLASS	B_b	See Table 4–6

Example:

```
LLP,W   LABEL,,EVEN    40 2 3 0 01234
```

4.9.2. Store Logical Product (47) SAND

This instruction forms the logical product of the contents of the Q register and the A register and stores this product in a storage location.

Operation	k	Space	y	b	j
SAND	See Table 4–1	␢	STORE CLASS	B_b	NORMAL

Example:

```
SAND,L       LABEL,B5       47 0 1 5 01234
```

4.9.3. Replace Logical Product (44) RLP

This instruction forms the logical product of the contents of the Q register and an operand, retains the logical product in the A register, and stores this logical product in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RLP	NORMAL	␢	REPLACE CLASS	B_b	See Table 4–6

Example:

```
RLP,W        LABEL,,ODD     44 3 3 0 01234
```

4.9.4. Add Logical Product (41) ALP

This instruction adds the contents of the A register to the logical product of the contents of the Q register and an operand. The sum is retained in the A register.

Operation	k	Space	y	b	j
ALP	NORMAL	b̸	READ CLASS	B_b	NORMAL

Example:

```
ALP,X          77773          41 0 4 0 77773
```

4.9.5. Replace A + Logical Product (45) RALP

This instruction forms the logical product of the contents of the Q register and an operand, then adds this product to the contents of the A register. The sum is retained in the A register and is stored in the location from which the operand was obtained.

Operation	k	Space	y	b	j
RALP	NORMAL	b̸	REPLACE CLASS	B_b	NORMAL

Example:

```
RALP,LX        LABEL,B4       45 0 5 4 01234
```

4.9.6. Subtract Logical Product (42) ANLP

This instruction subtracts the logical product of the contents of the Q register and an operand from the contents of the A register. The difference is retained in the A register.

Operation	k	Space	y	b	j
ANLP	NORMAL	b̸	READ CLASS	B_b	NORMAL

Example:

```
ANLP,W         LABEL,,QPOS    42 2 3 0 01234
```

4.9.7. Replace A – Logical Product (46) RANLP

This instruction forms the logical product of the contents of the Q register and an operand, then subtracts this product from the contents of the A register. The difference is retained in the A register and is stored in the location from which the operand was obtained.

Operation	k	Space	y	b	j
RANLP	NORMAL	b̸	REPLACE CLASS	B_b	NORMAL

Example:

```
        RANLP,UX      LABEL,B3        46 0 6 3 01234
```

4.9.8. OR (50) OR

This instruction forces 1 bits into selected bit positions of the A register. For corresponding bit positions of operand and A register, a 1 bit will be forced into the final A where either, or both, the initial A register or the operand has a 1 bit; otherwise, a 0 bit will occur in the final A register.

Operation	k	Space	y	b	j
OR	NORMAL	b̸	READ CLASS	B_b	NORMAL

Example:

```
        OR,W          LABEL           50 0 3 0 01234
```

4.9.9. Replace OR (54) ROR

This instruction compares the bits of the initial A register with corresponding bits of the operand. Where at least one bit is a 1 bit, a 1 bit is forced into the final A register; otherwise, a 0 appears in the final A register. After the selective set operation is performed, the result is retained in the A register and is also stored in the location from which the operand was obtained.

Operation	k	Space	y	b	j
ROR	NORMAL	b̸	REPLACE CLASS	B_b	NORMAL

Example:

```
        ROR,UX        LABEL,1         54 0 6 1 01234
```

4.9.10. Exclusive OR (51) XOR

This instruction compares the bits of the operand with the corresponding bits of the initial A register to form the Exclusive OR function in the final A register. Wherever either, but not both, of the corresponding bits is a 1, a 1 is forced into the final A register; otherwise, a 0 occurs in the final A register.

Operation	k	Space	y	b	j
XOR	NORMAL	b̸	READ CLASS	B_b	NORMAL

Example:

```
        XOR           77777           51 0 0 0 77777
```

4.9.11. Replace Exclusive OR (55) RXOR

This instruction forms the Exclusive OR function of $\bar{y}$ and A as described for the previous instruction and also replaces the original $\bar{y}$ by the Exclusive OR function.

Operation	k	Space	y	b	j
RXOR	NORMAL	b̸	REPLACE CLASS	B_b	NORMAL

Example:

```
RXOR,U          LABEL          55 0 2 0 01234
```

4.9.12. NOT (52) NOT

This instruction will clear selected bit positions of the A register to zero. A 1 bit in an operand bit position clears the corresponding bit position in the A register.

Operation	k	Space	y	b	j
NOT	NORMAL	b̸	READ CLASS	B_b	NORMAL

Example:

```
NOT,W           LABEL,B6       52 0 3 6 01234
```

4.9.13. Replace NOT (56) RNOT

This instruction clears selected bit positions of the A register. The bit positions that are cleared are determined by the presence of 1 bits in the corresponding bit positions of the operand. After the NOT operation is performed, the result is retained in the A register and is also stored in the storage location from which the operand was obtained.

Operation	k	Space	y	b	j
RNOT	NORMAL	b̸	REPLACE CLASS	B_b	NORMAL

Example:

```
RNOT,L          LABEL,,SKIP    56 1 1 0 01234
```

4.9.14. Selective Substitute (53) SSU

This instruction replaces the contents of selected bit positions of the A register with the content of corresponding bit positions in an operand. The bit positions selected for replacement are determined by the presence of 1 bits in the corresponding bit positions of the Q register.

Operation	k	Space	y	b	j
SSU	NORMAL	ƀ	READ CLASS	B_b	NORMAL

Example:

```
SSU,X          11000          53 0 4 0 11000
```

4.9.15. Replace Selective Substitute (57) RSSU

This instruction replaces the contents of selected bit positions of the A register with the contents of selected bit positions of the operand. The bit positions that will be replaced are determined by 1 bits in the corresponding bit positions of the Q register. After the selective substitute operation is performed, the result is retained in the A register and is also stored in the location from which the operand was obtained.

Operation	k	Space	y	b	j
RSSU	NORMAL	ƀ	REPLACE CLASS	B_b	NORMAL

Example:

```
RSSU,LX        LABEL          57 0 5 0 01234
```

4.9.16. Application of Logical Instructions

The uses of logical instructions are varied. An individual example is meaningless unless seen in its context. The portion of coding presented below is designed to add two numbers in Fieldata code to produce a sum in Fieldata code.

```
MASK       6060606060
ADJ        5252525252
           LQ,W      FD1     (1)
           ANQ,W     ADJ     (2)
           AQ,W      FD2     (3)
           LLP,W     MASK    (4)
           LSAQ      30D     (5)
           LSQ       27D     (6)
           OR,W      MASK    (7)
           SANQ,W    FDSUM   (8)
```

As an example: FD1 = 12345 and FD2 = 1234.

The result will be 24690 (FDSUM). The following operations are performed:

(1) The first number is entered in the Q register.

(2) An adjusting value (ADJ) is subtracted from this number, and

(3) the second number is added to the first number as follows:

```
6162636465  (FD1, in Fieldata)
5252525252  (ADJ)
----------
0710111213
6162636465  (FD2, in Fieldata)
----------
7072747700  (Contents of Q register)
```

(4) The logical product of the contents of the Q register and the operand MASK is formed and entered in the A register as follows:

```
111000  111010  111100  111111  000000   (Q)
110000  110000  110000  110000  110000   (MASK)
----------------------------------------
110000  110000  110000  110000  000000   (A)
```

(5) The contents of A and Q are interchanged by a 30-bit left shift of AQ, and

(6) the resulting contents of Q are shifted to account for a carry from one Fieldata digit to the next. The result of these operations appears in Q as follows:

```
000110  000110  000110  000110  000000
```

(7) An OR operation, and

(8) a storage of AQ results in the sum of the two Fieldata numbers as follows:

```
111000  111010  111100  111111  000000   (A)
110000  110000  110000  110000  110000   (MASK)
----------------------------------------
111000  111010  111100  111111  110000   (A)
000110  000110  000110  000110  000000   (Q)
----------------------------------------
110010  110100  110110  111001  110000   (FDSUM)
 ~~~~    ~~~~    ~~~~    ~~~~    ~~~~
  62      64      66      71      60     (24690 in Fieldata)
```

4.10. PSEUDO-OPS

The term "pseudo-ops" refers to that group of instructions which are actually specific variations of a basic instruction. A pseudo-op is a facility offered by the UNIVAC 494 Assembler to the programmer, whereby often-used instructions can be source coded without j and k designators and with implicit operands. This feature is similar to the single-ops of UNIVAC 494 SPURT. Each pseudo-op occupies one word of memory.

Pseudo-ops fall into two classes – data transfer and linkage pseudo-ops. Although the NOP (No Operation) pseudo-op is technically a data transfer pseudo-op, it merits description as a linkage pseudo-op because of its many applications in this area.

4.10.1. Data Transfer Pseudo-Ops

The pseudo-ops of this group have already been described in 4.3 and are listed here (Table 4–7) for reference purposes together with their counterparts in UNIVAC 494 SPURT.

494 ASSEMBLER	SEE SECTION	494 SPURT
ZB	4.3.4	CL*B_n
ZQ	4.3.8	CL*Q
NQ	4.3.9	CP*Q
NA	4.3.11	CP*A
ZA	4.3.12	CL*A
SZ	4.3.14	CL*Y

Table 4–7. Data Transfer Pseudo-Ops

4.10.2. Linkage Pseudo-Ops

Linkage pseudo-ops are used for block and subroutine control and comprise the NOP, ENTRY, and EXIT pseudo-ops.

4.10.2.1. NOP Pseudo-OP

The NOP pseudo-op (see 4.3.5) corresponds to the NO-OP of UNIVAC 494 SPURT. Applications of the NOP in subroutine linkage include its use as: (1) a logical switch to alter program flow; (2) a furnisher of arguments to a called subroutine; (3) an entry point of a subroutine. The following examples illustrate these uses.

Example as logical switch:

```
SWITCH2  NOP
          .
          .
          .
         LA,W      SWITCHX
         SA,W      SWITCH2
          .
          .
          .
SWITCHX  J         OUT
```

Example as argument list:

```
SLJ  SUBR
NOP  50D
NOP  10-A
NOP  B
```

In this example, each of the NOP's contains data in the rightmost portion of the word which can be as an argument in subroutine SUBR. When control is returned to the instruction following the SLJ instruction, the NOP's act as a "fall-through" sequence, and control will actually be returned to the first instruction following the last NOP.

Example as entry point:

```
SUBRN      NOP       A
            .
            .
            .
           J,L       SUBRN
```

The return address is represented by A.

4.10.2.2. ENTRY and EXIT Pseudo-Ops

The ENTRY and EXIT pseudo-ops are the standard entrance to, and exit from, a subroutine. Each of these pseudo-ops is a special form of the Jump instruction (see 4.6.1). The difference between the two is that a k designator of 0 is generated for the ENTRY and a k designator of 1 for the EXIT.

The ENTRY mnemonic is the first line of the subroutine and, therefore, should have a label which is the name of the subroutine. The jump to a subroutine is the SLJ instruction (see 4.6.3) which places the return address in the rightmost half (lower half) of the computer instruction generated by ENTRY.

An EXIT pseudo-op can appear wherever an exit from the subroutine is desired. More than one EXIT can be used in a subroutine if alternate paths are present. The computer instruction generated will refer back to the return address that was placed in the computer address associated with ENTRY. The use of j designators is permitted.

LABEL	OPERATION	y	j
name	ENTRY	none	none

OPERATION	y	j
EXIT	none	see 4.6.1

Example:

```
                                 Computer Code

CHKINPUT    ENTRY                02244     6100000000
              .                    .
              .                    .
            EXIT                 02277     6101002244
              .                    .
              .                    .
            SLJ  CHKINPUT        04322     6500002244
            (return address)     04323     ..........
```

5. ASSEMBLY DIRECTIVES

5.1. GENERAL

Symbolic assembly directives direct and control the assembly processor just as operation codes direct and control the central processor. Directives are represented by mnemonics written in the operation field of a line of symbolic coding. Their uses are varied: to equate expressions; to adjust the location counter value; to offer the programmer special controls over the generation of object coding. Directives are processed as encountered by the assembler.

Some directives do not cause generation of object code while others cause generation of more than one line of object code. Some directives may appear anywhere within an assembly while others are limited to PROC's and FUNC's. Those directives which are limited to PROC and FUNC directives are discussed in the next section (Section 6).

The directives, discussed in this section, not limited to PROC's and FUNC's are:

- EQU (Equate)
- RES (Reserve)
- LIT (Literal)
- FORM (Format)
- START
- END
- DLD (Double Length Data)
- UTAG
- DO
- COMMON
- BLOCK-DATA
- XREF
- EDEF
- EXPRESSION
- INPUT or INPUTFORM
- LET
- UNLIST
- LIST

5.2. EQU (EQUATE)

The EQU directive equates the label (in the label field) to the value of the expression or item in the operand field.

This value may be referenced in any succeeding line by use of the label equated to it. If a label is to be assigned a value by the programmer, it must appear in the label field of an EQU line before it is considered defined. Only then may it be used or referenced in subsequent lines of symbolic coding. If it is referenced prior to the EQU line in which it was equated, the label is considered undefined.

Example:

```
        T3      EQU       30D
        T1      EQU       0500
                .
                .
                .
                LA        T1           11 0 0 0  00500
                LA,1      T3,B2        11 0 1 2  00036
```

The EQU directive does not permit redefinition of a label. Thus,

```
        A           EQU           A+2
```

is invalid since it involves redefinition of a label.

5.3. RES (RESERVE)

The RES directive may be used to create work areas for data or to specify absolute location counter positioning to the assembler. If a label is placed on the coding line which contains a RES directive, the label is equated to the present value of the location counter, which is in effect the address of the first reserved word. Its immediate effect is to increment (or decrement) the controlling location counter by the number of words specified by an item or expression in the operand. The RES directive may not be used as the first code-generating statement of a program.

Example:

```
        TABLE      RES        50D
                   SA,W       TABLE + 5
```

The SA instruction will store the contents of the A register in the sixth word of the area reserved for TABLE.

5.4. LIT (LITERAL)

The LIT directive defines a class of literals which are placed under the control of a specific control counter. Only one LIT directive is allowed under each control counter. The directive may have a label.

Use of the label with a literal will place the literal generated in the table of literals associated with the control counter current at the time the related LIT directive was encountered. The origin of the literal table follows the last coding line of the specified control counter. Duplicate literals are discarded in each table, but may exist in separate literal tables.

Through the use of LIT directives, a number of separate literal tables can be created. In the absence of a LIT directive, all literals will be placed in the literal table under control of location counter zero. The entries in the label field of a LIT directive comply with the rules of labeling.

If a literal table which is not under the control of control counter zero is required, a LIT directive is used. If a LIT directive has no label, *all* literals which are not preceded by a label will be placed in the literal table designated by this LIT directive. There may be only one LIT directive in a program which does not have a label associated with it.

If a LIT directive has a label, all literals to be placed in this literal table must be preceded by the label associated with this LIT directive.

Example 1:

```
          LA,3        :05;
```

The octal literal 0000000005 will be placed in the literal table controlled by counter zero.

Example 2:

```
$(3)      LIT
          LA,W        :05;
```

The octal literal 0000000005 will be placed in the literal table controlled by counter three.

Example 3:

```
$(3),BILL LIT
          LA,3        :5D;
          LB,W        2,BILL:0100;
```

The octal literal 0000000005 will be placed in the literal table controlled by counter zero. The octal literal 0000000100 will be placed in the literal table controlled by counter three.

5.5. FORM (FORMAT)

The FORM directive is a means of describing a special word format designed by the user. This word format may comprise fields of variable length (within the word). The length in bits of each field is defined by the user through expressions in the operand field of a FORM line. The value of each expression specifies the number of bits desired in its respective field.

The number of bits specified by the sum of the values of the operand expressions cannot exceed 30 (the size of a UNIVAC 494 word). The assembler uses the values of the operand expressions within the FORM line to create a control pattern that dictates a word format.

A reference to the word format is accomplished by writing the label of the FORM directive in the operation field followed by a series of expressions in the operand field which specify the value to be inserted in each field of a generated word. A reference to a specific FORM label will always create a word composed of fields in the same format. Of course, the contents of the fields may vary according to the expression values in the referencing line.

Example:

```
1  INSTR     FORM      6,3,3,3,15
2            INSTR     014,0,4,0,05000
```

The relocatable instruction that will be generated alongside the second line will be 14 0 4 0 05000.

5.6. START

The START assembly directive defines the starting line entry point of a program portion. No label is used with this directive. The operation field is START; the operand field is the label of a line to which control will be transferred.

Example:

```
             START     BEGIN
```

5.7. END

The characters END in the operation field signal the end of an assembly. This end-of-assembly indicator may be omitted, but such a practice is not recommended. When this directive is omitted, the required OMEGA •END will indicate the final source card of an element.

5.8. DLD (DOUBLE LENGTH DATA)

The DLD assembly directive permits a two-word literal to be specified by one line of coding. The assembler will generate the literal and assign it to two consecutive memory locations. A label may be used, if required. The operation field is DLD. The operand is the literal and may be an octal, decimal, floating point, or internal decimal (Fieldata) value, as shown in the following:

```
AA          DLD          417736432176642
.  ADDRESS  AA  STORES   0000041773
.  ADDRESS  AA+1         6432176642
BB          DLD          536870912D
.  ADDRESS  BB  STORES   0000000000
.  ADDRESS  BB+1         4000000000
CC          DLD          16384.0
.  ADDRESS  CC  STORES   2017400000
.  ADDRESS  CC+1         0000000000
DD          DLD          6661
.  ADDRESS  DD  STORES   6060606060
.  ADDRESS  DD+1         6060666666
```

5.9. UTAG

The UTAG assembly directive enables division of a computer word into upper and lower halves. Any valid expression, constant, or variable can be supplied for the upper and lower portions. The assembler will evaluate each and generate the resultant values as the upper and lower portions of the word. This directive is particularly useful for the preparation of jump tables.

A label may be used, if required. The operation is UTAG. The operand consists of two subfields, each of which may be a valid arithmetic expression or a constant. The value of the first subfield is stored in the upper portion of the computer word; the second, in the lower portion. A maximum of 15 bits is available for each portion.

Examples:

```
TABLE     UTAG      ABLE, BAKER
.  THE  COMPUTER-ASSIGNED  ADDRESSES
.  ABLE  AND  BAKER  WILL  BE  SUBSTITUTED
.  IN  THE  SUBFIELDS
VALUES    UTAG      D+QU+10D, A1+C4
.  EVALUATION  OF  EACH  EXPRESSION  WILL
.  BE  SUBSTITUTED  IN  SUBFIELDS
```

5.10. DO

The DO directive is used within a procedure or function to generate a specified line of coding a number of times. The operation DO is followed by at least one blank and then the expression which conditions the number of times a line of coding is to be generated. This expression is followed by one blank, a comma, and then the coding which is to be done. If there are no intervening blanks between the comma and the first character of the second operand entry, the symbolic line to be produced is assumed to have a label. The line of coding associated with the DO directive starts with the first column following the comma as though this first column were the first column of a separately written line.

If the DO directive is labeled, the value of the label will be N the Nth time the line is coded. The label serves as a counter and not as a reference name. A typical example is the following:

```

I         DO  27 ,  LA, W     I, B3
```

The "I" is set to one immediately after the first operation DO and is available for use only in the coding line to be done. This "I" cannot be obtained outside the coding to be done unless the coding stores it (as in the example shown) at a point where it can be obtained.

The following illustrates the use of a DO directive with an arithmetic, relational, and/or logical condition determining coding of the instruction:

```
TAG1      DO A<2 ,  LA,W DOG
.  IF A IS LESS THAN 2, GENERATE THE
.  LINE OF CODE
          DO A<N* *3 , LB,W B3,BARB
.  IF THE RELATIONAL
.  EXPRESSION IS TRUE, THE INSTRUCTION
.  IS GENERATED. FOR EXAMPLE LET N=170.
.  IF A=0 THE INSTRUCTION IS GENERATED,
.  IF A=1 THE INSTRUCTION IS NOT
.  GENERATED.
```

5.11. COMMON

The COMMON directive defines an area of core to be shared by two or more independently compiled program units (main program, PROC's, and FUNC's), permitting these program units to communicate with each other. The format of this directive is:

- Label field:
  optional
- Operation field:
  COMMON
- Operand field:
  number of a location counter (0 through 31)

The COMMON directive must appear in each program unit which is to share a common area. If the label field contains a normal label subfield (up to ten characters, starting with an alphabetic character), the common area so defined is termed a labeled common block; if not, it is a "blank common block". The control counter assigned to a common block by the COMMON directive in one program unit need not be the same as the control counter assigned to the same block in another program unit. However, within a program unit, once a control counter has been assigned to a common block, all instructions and data under that control counter will form part of that common block. Similarly, all instructions and data to be shared by that common block must be governed by that control counter assigned in the COMMON directive as shown in the following example:

PROGRAM UNIT 1

```
BLK4          COMMON     25
BLK5          COMMON     23


$(23),TAG2    RES        9D
$(4)          R          9D,,ADV
```

PROGRAM UNIT 2

```
BLK5          COMMON     17


$(17),LBL3    RES        12D
$(3),INCR     +1
              LB,W       BG,INCR
              R          3,,ADVR
              RI,W       LBL3+8D
```

As can be seen from the example, all references to a given block, within the same program unit, use the name assigned to that block by the RES directive. Identification of the common blocks among program units is accomplished by the label (or blank) assigned to that block in the COMMON directive. In this example, if program unit 1 and program unit 2 are the only references affecting the contents of the labeled common block BLK5, this common block will consist of 12D words – the first nine will contain zeros; the tenth, 1; the eleventh, 2; the twelfth, 3.

5.12. BLOCK-DATA

This directive creates an element that is recognized by the UNIVAC 494 Operating System as a FORTRAN-compatible element (see "UNIVAC 490/491/492/494 FORTRAN IV Programmers Reference," UP-4087 (current version)). This element may not contain EDEF/XREF references or a start address.

OPERATION
BLOCK-DATA

5.13. XREF (EXTERNAL REFERENCE)

The XREF lists those symbols that are used in this assembled portion of the program (program element) but are defined in another program element. Since references to these symbols cannot be satisfied at assembly time, they must be satisfied (that is, the symbols must be defined) at collection time by a corresponding EDEF (see 5.14). No XREF may appear as a program label (excluding labels for PROC, FUNC, DO, and FORM).

The number of symbols in the list is limited only by card continuation requirements. A program element may contain more than one XREF.

Format of the XREF directive is:

- Label:
  none
- Operation:
  XREF
- Operand:
  list of symbols, each separated from the next by a comma

An alternate method of specifying external references is to pass on all undefined unsubscripted symbols to OMEGA to be satisfied at load time. However, the use of XREF's is preferred because it avoids the confusion caused by U (undefined) flags (see Appendix D). The use of the U option on the #ASM card (see Appendix F) will insure that only those symbols listed as XREF's will be considered valid external references. If a reference in this element cannot be satisfied by a label within the element and is not listed as an XREF, the U (undefined) flag will be generated (when the U option is used).

5.14. EDEF (ENTRY DEFINITION)

This directive lists all labels which are defined within this program element and may be used by other program elements. This directive, together with the XREF directive in the other program element(s), provides linkage between program elements. During the collection process of the operating system, each XREF is satisfied by an EDEF. The number of symbols in an EDEF directive is limited only by card continuation requirements. More than one EDEF directive may be used in an assembled program element.

Format of the EDEF directive is:

- Label:
  none
- Operation:
  EDEF
- Operand:
  list of labels, each separated from the next by a comma

An alternate method of specifying entry definitions is the use of an asterisk immediately after a label. This convention is derived from the use of entry points in PROC's and FUNC's (Section 6).

Example:

```
                                                     Error Flag
        EDEF      TAG1,TAG2,TAG3    .LINE 1              U
 TAG1   NOP                         .LINE 2
 TAG2   NOP                         .LINE 3
          .
          .
          .
```

The error flag was generated because, in this particular example, TAG3 has not been used in a source statement of this program element.

5.15. EXPRESSION

This directive permits changing conventions in source coding constants used in expressions. Two different versions of this directive are available. In the first, the operand field contains the characters SLEUTH; in the other, the characters BITARRAY.

5.15.1. Expression SLEUTH

This directive changes the method of specifying constants as follows: if the constant has a leading 0 (zero), it is assumed to be an octal number; if the leading digit is different from 0, the number is assumed to be a decimal number. If a string of digits is preceded by 0 and contains an 8 or 9, an error flag will be printed (see Appendix D).

Format of the directive is:

- Label:

  none

- Operation:

  EXPRESSION

- Operand:

  SLEUTH

Example:

```
     Source Code                        Generated Code

       EXPRESSION  SLEUTH
       +           10-010               0000000002
```

5.15.2. Expression BITARRAY

This directive permits partial word values to be used as items within assembly-time expressions.

Example:

```
     Source Code                                         Generated Code

     EXPRESSION BITARRAY
A    EQU   01010                                             0000001010
C    EQU   1                                                 0000000001
     LB    B5,2                                  00000       12 5 0 0 00002
     LB    B6,3                                  00001       12 6 0 0 00003
     +     A                                     00002       0000001010
     +     W(T1+B5)                              00003       0000000002
     +     L(T2-(C*1)+B6)                        00004       0000000012
     +     A+W(T1+B5)-L(T2-(C*1)+B6)             00005       0000001000
     .                                             .
     .                                             .
     .                                             .
T1   +     0                                     00100       0000000000
     +     1                                     00101       0000000001
     +     2                                     00102       0000000002
     .                                             .
     .                                             .
     .                                             .
T2   +     0000,010                              00200       00000 00010
     +     0111,011                              00201       00111 00011
     +     0222,012                              00202       00222 00012
     +     0333,013                              00203       00333 00013
```

5.16. INPUT OR INPUTFORM

The INPUT directive and the INPUTFORM directive are alternate names for the directive which is used to alter the format of source cards to be assembled.

Format of the directive is:

LABEL	␢	OPERATION	␢	OPERAND
none		INPUT or INPUTFORM		exp_1,exp_2,exp_3,exp_4,exp_5

where exp_1 through exp_5 are expressions which specify the following:

Expression exp_1 specifies the start of label field.

Expression exp_2 specifies end of card.

Expression exp_3 specifies the continuation column.

Expression exp_4 specifies start of sequence number.

Expression exp_5 start of operation field (fixed format).

Example:

CARD COLUMNS			
1 9	10 71	72	73 80
	INPUTFORM 1,71D,72D,73D,12		00000001
ABCDEFGHI	LA,W BOB		00000002
		-	00000003

5.17. LET (GENERAL)

The LET directive is actually a variation of the EQU directive (see 5.2) but its use is confined to variables which may require redefinition. It is intended to avoid the possible confusion that may arise when an EQUated variable is redefined, causing generation of a possible error indicator. Although use of the LET directive is not confined to PROC's and FUNC's, it is most generally used within a procedure or function. For this reason, it is described in detail in 6.4.3.

5.18. UNLIST

This directive completely suppresses output listing after the line containing the UNLIST directive is printed.

OPERATION
UNLIST

5.19. LIST

This directive will allow normal printout of the assembly listing from (and including) the line on which the LIST directive occurs.

OPERATION
LIST

6. PROC, FUNC, AND ASSOCIATED DIRECTIVES

6.1. GENERAL

PROC and FUNC directives are used to define often-used sequences of coding which are not necessarily identical but are similar enough so that repetition of the coding requires only the insertion of parameters or arguments when the sequence is called. For both directives, the lines of coding representing the definition must precede the reference (or references) to the sequence and this coding is saved when encountered. The PROC directive is different from the FUNC directive in that the PROC directive usually generates lines of object code at assembly time at its point of reference to be executed at object time. The FUNC directive is executed entirely at assembly time and stores its results into the program at this time. The FUNC directive calculates a value when referenced and does not cause generation of object code. For purposes of this discussion, the PROC (procedure) directive will be described first and the FUNC (function) directive will be described afterwards in terms of its differences.

The first line that defines a procedure is the PROC directive. The last line must be an END line to indicate its logical termination. Between the PROC directive and the END line, the special following directives may be used (in addition to the universal directives described in Section 5):

- NAME
- GO
- LET

6.2. PROC (PROCEDURE) DIRECTIVE

The PROC directive is the header for a procedure that is terminated by an END line. The following is a simple procedure:

```
LDZER*     PROC
           LA, 0
           END
```

Lines 1 and 3 are the limits of the procedure. LDZER is the label by which this PROC may be referenced. The asterisk after the label is necessary, to indicate that the label LDZER can be used to call this PROC. Each time this PROC is called by a source line containing the word LDZER, the code provided by line 2 will be generated. Thus, the sequence:

```
START      LDZER                 .CALL LINE
           SA        DOG,B2
           LDZER                 .CALL LINE
STOP       J         KAT
```

is equivalent to (expanded source code):

```
START      LA,0
           SA        DOG,B2
           LA,0
STOP       J         KAT
```

and each time the PROC LDZER is called, the A register is cleared.

6.2.1. PROC Directive Format

The label, operation, and operand of the PROC line are as follows:

- Label:

  Any normal unsubscripted label is acceptable as identification of the PROC sequence. Every PROC line must be labeled. This label can be used as an entry point for the sequence if the label is concluded with an asterisk.

- Operation:

  PROC

- Operand:

  Some expression or item may be given to indicate the number of lines of code that will be generated as a result of a call on this procedure. This value may only be supplied if the number of words generated will always be the same and only if the PROC contains no forward references. (See 6.2.7 for example.)

6.2.2. END Directive

The END directive signals the logical end of a sequence. In a procedure, its format is as follows:

- Label:

  None required since it serves no purpose.

- Operation:

  END

- Operand:

  None since it serves no purpose. (In a function, the operand field provides the value of the function.)

6.2.3. Symbolic Lines Within Procedure

The formats of the lines within a procedure (except NAME directives) are as follows:

- Label:

  Any normal label may be employed; however, its definitions will be restricted to the bounds of the PROC, unless it is an entry point. Any label may be available immediately outside the bounds of the PROC that contains it by appending an asterisk to the label; this label is then referred to as a reference point. Multiple reference points are permissible and multilevels are permissible. A label is raised one level for each asterisk following it. An asterisk following the label of a PROC or NAME directive indicates an entry point.

- Operation:

  Any mnemonic, meaningful special character, label of a PROC, or directive is permissible.

- Operand:

  Any operand appropriate to the operation code is acceptable. The operand may also be taken from the Call line by means of a paraform. (Paraforms are described in 6.2.5.)

The following example shows nesting of PROC's and use of multilevel reference points:

```
                         etc.
r------A*                PROC
|  r---B*                PROC
|  |  r-C*               PROC
|  |  |  W***            + 1            ·CARRIES 3 LEVELS
|  |  |  X**             + 2            ·CARRIES 2 LEVELS
|  |  |  Y*              + 3            ·CARRIES 1 LEVEL
|  |  |  Z               + 4            ·KNOWN HERE ONLY
|  |  L-----------------END
|  |                     C·
|  |                     + Z            ·UNKNOWN
|  |                     + W,X,Y        ·KNOWN
|  L--------------------END
|                        B·
|                        + Y, Z         ·UNKNOWN
|                        + W, X         ·KNOWN
L-----------------------END
                         A·
                         + X, Y, Z      ·UNKNOWN
                         + W            ·KNOWN
                         etc.
```

6.2.4. Call Line

A Call line is a symbolic line of code which uses an external label of a procedure in its operation field as an entry point into a procedure. (An external label of a procedure is an asterisked label of a PROC or NAME line. Any other asterisked labels are "reference points".) It informs the assembler that generation and modification of a procedure (or part of a procedure beginning at an entry point) should begin at this point. The Call line provides any required parameters for use in the procedure. A Call line must not appear in the program until the procedure has been defined by the PROC and END directives, including any optional NAME directives. The format of the Call line is as follows:

- Label:

  Any normal label is acceptable. It refers to the first line of code generated, unless an asterisk is located in column 1 of some part of the procedure. The asterisk is a flag indicating that the label is to be applied on this line of code.

- Operation:

  An entry point label of a procedure. (An entry point label of a procedure is an asterisked label prefixing a PROC or NAME directive within the procedure.) Additional subfields (separated by commas) may be added to furnish parameters to the procedure called.

- Operand:

  Any number of fields and subfields, in any sequence, is acceptable. Fields are separated by one or more blanks not preceded by a comma. The use of an asterisk preceding a subfield is explained in the evaluation of paraforms. An example, for a hypothetical procedure entry point "COMPAR" is shown in Figure 6–1.

```
A         COMPAR  X TO Y   .X, TO AND Y ARE 3 FIELDS
A         COMPAR  X,TO,*Y  .1 FIELD, 3 SUBFIELDS
```

Figure 6–1. Typical Procedure Call Lines

6.2.5. Paraforms

A paraform (parameter reference form) is the means whereby an operation within the procedure can obtain values of parameters used in the operation from the Call line or entry point. This enables the same procedure to be used many times with the Call line or point of entry furnishing a different set of parameters each time the procedure is called.

A paraform appears in the operand portion of the symbolic coding within the procedure. It consists of the name of the procedure called, followed by a set of parentheses. The parentheses enclose a double coordinate reference system expressed as n,e. The n refers to the nth operand field of the call line; the e, to the eth subfield within that field. A typical paraform, using the same hypothetical procedure COMPAR is shown in Figure 6–2, where the reference is to field 3, subfield 1 of the operand of the Call line. This paraform can be used with the first of the two Call lines shown in Figure 6–1, since the second has only one field in its operand portion.

```
                      COMPAR(3,1)
```

Figure 6–2. Simple Paraform

In addition to the simple paraform just described, paraforms may be written in different forms to extend their applicability. The general rules regarding their interpretation are presented in Table 6–1 for a hypothetical procedure headed by a PROC directive with the label L. (All directives used within the procedure not previously described, such as the NAME directive, are described in 6.4.)
If a paraform is not supplied, its value is zero.

If the paraform is written as, or in the form	it is given the numerical value equal to
L	the number of fields in the operand of the Call line. If entry is by a NAME directive, this value is increased by 1.
L(0)	the total number of subfields, minus 1, in the operation field of the Call line. If the entry point is a PROC directive, this value is 0.
L(N)	the number of subfields of the Nth field in the operand of the Call line.
L(0,0)	the operand of the NAME line used for entry into the procedure. If the entry point is a PROC directive, this value is 0.
L(0,C)	the Cth subfield after the entry point label in the operation field of the Call line.
L(A,*B)	1 (on) if an asterisk precedes subfield B of field A in the operand of the Call line; if no asterisk, the value 0 (off).

Table 6–1. Evaluation of PROC Paraforms.

To illustrate use of these special paraforms, two Call lines are shown.

```
          A        7, G
          B, *14   8, *13  6
```

The procedure to which these call lines refer is

```
P*        PROC  X, Y  , SEE NOTE BELOW
A*        NAME  8
B*        NAME  10
                  . ETC, ANY PARAFORM IN THE
                  . TABLE BELOW MAY BE USED
```

The following table shows the values assigned to different paraforms within this procedure.

If the Call line is	and the paraform is	then its value is
A 7,G	P	2
	P(0)	0
	P(1)	2
	P(1,1)	7
	P(1,2)	G
	P(1,3)	0
	P(1,*2)	0 (off)
	P(0,0)	8
	P(0,1)	0
	P(0,*0)	0 (off)
B,*14 8,*13 6	P	3
	P(1,2)	13
	P(2,1)	6
	P(1,*2)	1 (on)
	P(2,2)	0
	P(0)	1
	P(0,0)	10
	P(0,1)	14
	P(0,2)	0
	P(0,*1)	1 (on)

6.2.6. Expanded Procedures

The following example illustrates the use of the NAME and GO directives.

```
1  P        PROC
2  LOAD*    NAME
3           LA,W  DOG
4           GO    FINISH
5  STORE*   NAME
6           SA,W  KAT
7  FINISH   NAME
8           END
```

Line 1 provides a label P which is the name of the procedure, but cannot be used to call the procedure. It must be used, however, when referencing parameters; this method is explained later.

Lines 2 and 5 contain the NAME directive and provides the labels: LOAD and STORE. Since each label is flagged, each may be used to call the procedure. These are external definitions or entry points.

Line 4 contains a GO directive; the operand of a GO must always be the label of a NAME line or PROC line. If the label appears in the same procedure as the GO, it *may* be flagged with an asterisk. If it appears in a different procedure, it *must* be flagged. It is not possible to reference an entry point which has not yet been encountered by the assembler. The effect of the GO directive is to skip over lines 5 and 6.

Assume the following calls:

```
START   LOAD
        STORE
        LOAD
FINIS   STORE
```

This would be equivalent to (expanded source code):

```
START   LA,W    DOG
        SA,W    KAT
        LA,W    DOG
FINIS   SA,W    KAT
```

Notice that each call generates only that code which is encountered in the procedure.

It is possible to provide parameters on the line calling the procedure, thus:

```
START   LOAD  START
        STORE START
        LOAD  START
END     STORE START+2
```

It is also possible to select the desired parameter via the parameter construction:

```
1  P         PROC
2  LOAD*     NAME
3            LA,W      P(1,1)
4            GO        BEND
5  STORE*    NAME
6            SA,W      P(1,1)
7  BEND      NAME
8            END
```

The construction P (1,1) on line 6 is a paraform. P is the label of the procedure and is the common reference. The coordinates in parentheses indicate field 1 of the operand, and subfield 1 of field 1, respectively.

This procedure call when expanded will appear as:

```
START     LA,W    START
          SA,W    START
          LA,W    START
END       SA,W    START+2
```

6.2.7. Efficient Use of Procedures

When a procedure is used many times, e.g., as a code generating method in compilation, care should be taken in its construction to avoid time consuming operations. Several simple devices may be used to advantage in this regard.

- Parameters

  If a parameter to a procedure is referenced many times, efficiency is increased by equating the parameter to a simple label. (This device should not be used, however, if the parameter may be preceded by an asterisk in the call, as the equivalence will not retain the indirect flag. However, when the same parameter preceded by an asterisk is called indirectly via several nested paraforms, each reference includes the asterisk.)

- Call lines

  Use of the period to terminate each call line saves time by stopping the scan of the line for possible parameters.

- Procedure levels

  Where practical, the depth of nesting of procedures should be limited. Use of distributed NAME lines and the GO directives may be helpful in decreasing the depth required.

- Specification of the number of object lines to be generated

  A feature has been included to greatly reduce assembly time for procedures which produce a predetermined number of lines of object code. This number will be indicated as the operand field in a PROC line, as follows:

```
COMPAR*     PROC  1
```

  Procedures which define an external label, or which make a forward reference to a label defined within the procedure, may not use this feature. Where possible, procedures should be constructed to take advantage of this feature.

- Summary

  PROC's may be nested, i.e., they may be included within each other. Nesting may be physical or it may be implied.

  - Physical nesting means that the procedure is physically located within the bounds of another procedure.

  - Implied nesting means that although a procedure is not physically contained within another, it may be temporarily considered so by implication, i.e., its reference line is contained within another PROC.

  The primary purpose of nesting procedures is to restrict labels should they interfere with labels from other procedures, or the main program. Another purpose is to re-equate labels for homogeneity. Nesting allows simpler block building techniques but requires longer assembly time.

  The Conditional DO statement allows symbolic lines to be created or negated. The GO may also be employed to include or skip lines of code.

  The NAME directive allows alternate entrances into a procedure. This is a method for qualifying a procedure.

  PROC's employ all directives. Since procedures allow the presence of all directives their power is enhanced. Of special value are the NAME, GO, DO, and EQU directives.

  PROC's are reflexive and may refer to themselves.

- Restrictions

While restrictions may help develop unique situations, they may also hinder general methods. Therefore, careful analysis should precede their usage.

Labels are local to procedures; they must be flagged to make them more universal by levels.

Nesting further restricts the locale of labels of inner procedures, but enlarges the locale of labels of the outer procedures.

The redefinition of labels is a restrictive process since it destroys previous values. It may not always be intentional.

6.3. FUNC (FUNCTION) DIRECTIVE

The FUNC directive enables the user to obtain a value at assembly time contingent upon a set of parameters. The function is a device which will cause certain predetermined lines of coding to be saved when encountered during assembly and, when referenced subsequently during the assembly, a computation will be made according to this coding. The evaluated quantity is then substituted for the reference call within the program.

The function is similar to the procedure in that the lines of coding representing the definition must precede any call (reference point) and this delineation of code is saved when encountered. The function is different from the procedure in that a value is calculated when a function is referenced and, unlike the procedure, no object lines of coding are ever generated. The procedure usually generates lines of object code at assembly time at its point of reference to be executed at object time. The function executes entirely at assembly time and stores its results into the program at this time.

The general rules of definition are similar to the PROC. A FUNC directive must start the function area. This line must have an unsubscripted label which may be flagged. If this line is an entry point into the function, it must be flagged. The delineation of code is terminated with an END directive which must have an operand. This operand field will be an expression whose evaluation will result in the proper quantity being substituted into the reference point in the program.

NAME lines may be alternate entry points into the function. The labels associated with these NAME lines must be flagged in this event. NAME lines may also be used as local reference points within the function. Forward references should be avoided.

6.3.1. Function Nesting

When a function is nested it is not necessarily assembled when the outer code is called; assembly occurs only if the function has been specifically called. Until a function has been called, no information contained in it, except the label and code level of its entry points, is available to the assembler. Following a call, information is available according to the standard code level rules.

6.3.2. Function Calls

A function is called by coding one of its entry points in any *operand* expression. This differs from a procedure which is called from the operation field. Parameters for the function may be specified by the programmer by following the call with a parenthesized single list of items or expressions. The items may not be logically forward referenced.

6.3.3. Function Paraforms

Parameters supplied with a function call can be substituted in programmer-designated places within the function coding through the use of paraforms. Unlike PROC, however, FUNC paraforms are, at most, *singly subscripted* labels, whose primary is the label of the FUNC directive.

The paraforms of a function are evaluated as shown in Table 6–2, where F is the label of the FUNC directive.

If the paraform is written as, or in the form	it is given a numerical value equal to
F	the number of parameters (items or expressions) supplied with the call, plus 1 if the entry line is a NAME directive.
F(0)	the value of the expression, if any, in the operand field of the NAME directive used for entry into the function. If the entry line is the FUNC directive, this value is 0.
F(K)	the Kth expression in the parameter list supplied in the call. If less than K expressions are supplied, this value is 0.

Table 6–2. Evaluation of FUNC Paraforms

Note that F is a true implicitly defined subscripted label, whose value is derived from the call line and the method of call. Paraforms differ from true subscripted labels in one respect: when a function is terminated, a paraform that was referenced, but not otherwise defined,is given the value of zero. Additional subscripted labels whose primary is the label of the FUNC directive may be defined within the function and will affect the appropriate previously defined paraform, if not externalized. Paraforms of a function are available to the function called by the FUNC.

An example of the function is the case where a certain average calculation is made throughout the coding. The programmer should keep in mind that this calculation could have been made by hand and is not dependent upon the execution of the object code. If "a" is the number of first type objects and "b" is its unit price and "c" is the number of second type objects and "d" is its unit price and it is necessary to calculate the average price of the combined number of objects, a mathematical expression which would calculate this value would be:

$$\text{Average cost} = \frac{ab + cd}{a + c}$$

If at assembly time, a, b, c, and d are known to have the values 1, 2, 3, and 4, respectively, the calculation can be accomplished by a FUNC directive as shown in Figure 6–3.

```
AVGCOS* FUNC .
A(1)    EQU AVGCOS(1)*AVGCOS(2)
B(1)    EQU AVGCOS(3)*AVGCOS(4)
C(1)    EQU A(1)+B(1)
D(1)    EQU AVGCOS(1)+AVGCOS(3)
        END C(1)/D(1)
. ALTHOUGH THE CALCULATION COULD BE
. DONE IN 1 STEP, IT IS MORE EXPEDI-
. ENT TO USE SEPARATE EXPRESSIONS
. AND THEN COMBINE


        LB  6,AVGCOS(1,2,3,4),2 .
. ABOVE IS CALL WHICH CAUSES GENER-
. ATION OF VALUE AT ASSEMBLY TIME
```

Figure 6–3. Simple Function

The next example shows a function which has a procedure embedded within it. The problem is to find the square root of the largest square which is less than or equal to a given number. Although it is not the most elegant method, it illustrates coding features of interest to the programmer.

```
 1  SQRT*     FUNC
 2  A(1)      EQU     0
 3  B(1)      EQU     0
 4  C*        PROC
 5  A(1)*     EQU     A(1)+2*B(1)+1
 6  B(1)*     EQU     B(1)+1
 7            END
 8  D         NAME
 9            C
10            DO      SQRT(1)>A(1) ,        GO  D
11            END     B(1)-(SQRT(1)<A(1))
12          + SQRT(64)          .  FIRST   CALL
13          + 2*SQRT(13)        .  SECOND  CALL
14            END
```

The calculation is based upon the equation $(a+1)^2 = a^2 + 2a + 1$. Line 6 supplies successive a's while line 5 contains successive a^2's. Lines 1 through 11 are the function with a nested procedure which, in turn, contains a nested DO. The first call upon the procedure, line 12, will produce the object code 0000000010. The second call upon the procedure, line 13, will produce the object code 0000000006. Line 14 terminates the assembly or program.

6.4. DIRECTIVES ASSOCIATED WITH PROCEDURES AND FUNCTIONS

This section describes the following directives, which are associated with procedures and functions: NAME, GO, and LET.

6.4.1. NAME Directive

The NAME directive has three functions: to act as a local reference point within a procedure or function, to act as an alternate entrance into a procedure or function, and to assign a value to a procedure. It must be located between the PROC (or FUNC) directive and its associated END line. Those variables in the procedure or function defined previous to the NAME directive are considered undefined. The operand portion of the NAME can be used to supply a parameter, P(0,0), to the procedure or function. (See Tables 6–1 and 6–2 for evaluation of paraforms, and 6.4.4 for an example.)

6.4.2. GO Directive

The GO directive is used within a procedure or function to transfer control of the assembler to the line whose label is in the operand field of this directive. This operand field must be one of the following:

- The label of a NAME directive in the same procedure. If this is a forward transfer, the label must end with an asterisk (in the NAME directive).
- An external label of a NAME directive of any procedure.
- The label of a PROC or FUNC directive which must be suffixed by an asterisk if in another program unit.

Examples:

A simple example is the following:

```

          GO  LABEL1



LABEL1    NAME
```

Another example is shown in 6.2.6.

6.4.3. LET Directive

A LET directive is generally used within a procedure or function (see 5.17). It is used to assign (and reassign, if required) values to assembly variables which may later require reassignment. The variables upon which it may operate must not have been defined elsewhere (e.g., by an EQU directive). The format of the LET directive is LET V =e where V is a LET-defined variable and e is any expression, as in the following:

```
P           PROC
DOG*        NAME        05
            LET         A=0
BACK        NAME
            LA,0        CAT(A)
            SA,W        DOG(A+2)
            LET         A=A+1
            DO  A<25 ,  GO  BACK
DOG1*       NAME
            LET         B=1
BK1         NAME
            LA,W        PCAT+B
            SA,W        P(1,B)
            LET         B=B+2
            DO  B<20 ,  GO  BK1
            END
```

APPENDIX A. ABBREVIATIONS AND SPECIAL SYMBOLS

The following is an explanation of special symbols and abbreviations used in text:

SYMBOL OR ABBREVIATION	MEANING
$\longrightarrow$	direction of data transfer.
(a)	the contents of a location or register, "a".
$(a)_n$	the n th bit of (a).
$(a)_L$	the lower 15 bits of (a).
$(a)_U$	the upper 15 bits of (a).
A	30-bit A register (accumulator).
AQ	60-bit register made up of A and Q.
ƀ	blank.
b designator	three-bit designator in an instruction word indicating an index register whose contents are added to the y designator to form the effective address or effective operand, $\bar{y}$, of an instruction (also termed "base address" or "base operand address").
B	index register.
B_b	index register determined by b designator.
B_j	index register determined by j designator
BCD	binary coded decimal
CP(x)	ones complement of the contents of x.
f designator	six-bit function designator in instruction word.
g designator	six-bit designator in 77 instruction word which supplements the f designator.
IFR	internal function register
j designator	three-bit designator in instruction word which usually defines a condition for skipping the NI.
k designator	three-bit designator in instruction word for defining source of operand and/or destination of result.
LP	logical product.
nines complement	value formed by subtracting each decimal digit from the number 9.
ones complement	value formed by subtracting each bit from the number 1.
NI	next sequential instruction.
P	Program register–containing address of next instruction during execution of current instruction.
Q	30-bit Q register
RB	relocatable binary.
RIR	relative index register.
tens complement	value formed by adding 1 to the nines complement (performing all carries).
y designator	lower 15 bits of an instruction word.
$\bar{y}$ designator	value formed by addition of contents of B_b to y to be used either as effective operand or relative (relative to RIR) address of operand.
Y designator	operand, from whatever source derived.

Table A-1. Symbols and Abbreviations

APPENDIX B. FIELDATA AND CARD CODES FOR CHARACTER REPRESENTATION

CODING SYMBOL	FIELDATA CODE (OCTAL)	HIGH SPEED PRINTER	CARD CODE	CODING SYMBOL	FIELDATA CODE (OCTAL)	HIGH SPEED PRINTER	CARD CODE
∇ (Master Space)	00	@	7-8	) (Right Parenthesis)	40	)	12-4-8
[(Left Bracket)	01	[	12-5-8	– (Minus)	41	–	11
] (Right Bracket)	02	]	11-5-8	+ (Plus)	42	+	12
# (Pound)	03	#	12-7-8	< (Less Than)	43	<	12-6-8
Δ (Caret)	04	Δ	11-7-8	= (Equal)	44	=	3-8
(Blank)	05	(Space)	(Blank)	> (Greater Than)	45	>	6-8
A	06	A	12-1	& (Ampersand)	46	&	2-8
B	07	B	12-2	$ (Dollar)	47	$	11-3-8
C	10	C	12-3	* (Asterisk)	50	*	11-4-8
D	11	D	12-4	((Left Parenthesis)	51	(	0-4-8
E	12	E	12-5	% (Per Cent)	52	%	0-5-8
F	13	F	12-6	: (Colon)	53	:	5-8
G	14	G	12-7	? (Question)	54	?	12-0
H	15	H	12-8	! (Exclamation)	55	!	11-0
I	16	I	12-9	, (Comma)	56	,	0-3-8
J	17	J	11-1	\ (Slant)	57	\	0-6-8
K	20	K	11-2	0	60	0	0
L	21	L	11-3	1	61	1	1
M	22	M	11-4	2	62	2	2
N	23	N	11-5	3	63	3	3
O	24	O	11-6	4	64	4	4
P	25	P	11-7	5	65	5	5
Q	26	Q	11-8	6	66	6	6
R	27	R	11-9	7	67	7	7
S	30	S	0-2	8	70	8	8
T	31	T	0-3	9	71	9	9
U	32	U	0-4	→ (End Statement)	72	'(Apos.)	4-8
V	33	V	0-5	; (Semicolon)	73	;	11-6-8
W	34	W	0-6	/ (Virgule)	74	/	0-1
X	35	X	0-7	. (Period)	75	.	12-3-8
Y	36	Y	0-8	⌑ (Lozenge)	76	□	0-7-8
Z	37	Z	0-9	Not Used	77	‡	0-2-8

NOTE: (NP) Signifies that the printers will substitute a space for this character.

Table B–1. Fieldata and Card Codes for Character Representation

APPENDIX C. ASSEMBLER / SPURT FUNCTION CODES

The following table (Table C-1) lists both Assembler and SPURT mnemonics by function code.

F	MNEMONIC DESCRIPTION	ASSEMBLER	SPURT
01	Right Shift Q	RSQ	RSH*Q
02	Right Shift A	RSA	RSH*A
03	Right Shift AQ	RSAQ	RSH*AQ
04	Test A, Test Q, Test R	TA, TQ, TR	COM*A*Q*AQ
05	Left Shift Q	LSQ	LSH*Q
06	Left Shift A	LSA	LSH*A
07	Left Shift AQ	LSAQ	LSH*AQ
10	Load Q	LQ	ENT*Q
11	Load A	LA	ENT*A
12	Load B_j	LB B_j	ENT*B_j
14	Store Q	SQ	STR*Q
15	Store A	SA	STR*A
16	Store B_j	SB B_j	STR*B_j
20	Add A	A	ADD*A
21	Subtract A	AN	SUB*A
22	Multiply	M	MUL
23	Divide	D	DIV
24	Replace A + Y	RA	RPL*A+Y
25	Replace A – Y	RAN	RPL*A–Y
26	Add Q	AQ	ADD*Q
27	Subtract Q	ANQ	SUB*Q
30	Load Y + Q	LAQ	ENT*Y+Q
31	Load Y – Q	LANQ	ENT*Y–Q
32	Store A + Q	SAQ	STR*A+Q
33	Store A – Q	SANQ	STR*A–Q
34	Replace Y + Q	RAQ	RPL*Y+Q
35	Replace Y – Q	RANQ	RPL*Y–Q
36	Replace Y + 1 (Increment Y)	RI	RPL*Y+1
37	Replace Y – 1 (Decrement Y)	RD	RPL*Y–1

Table C-1. Assembler/SPURT Function Codes (part 1 of 3)

F	MNEMONIC DESCRIPTION	ASSEMBLER	SPURT
40	Load Logical Product	LLP	ENT*LP
41	Add Logical Product	ALP	ADD*LP
42	Subtract Logical Product	ANLP	SUB*LP
43	Test Logical Product	TLP	COM*MASK
44	Replace Logical Product	RLP	RPL*LP
45	Replace A + Logical Product	RALP	RPL*A+LP
46	Replace A – Logical Product	RANLP	RPL*A–LP
47	Store Logical Product	SAND	STR*LP
50	OR	OR	SEL*SET
51	Exclusive OR	XOR	SEL*CP
52	NOT	NOT	SEL*CL
53	Selective Substitute	SSU	SEL*SU
54	Replace OR	ROR	RSE*SET
55	Replace XOR	RXOR	RSE*CP
56	Replace NOT	RNOT	RSE*CL
57	Replace Selective Substitute	RSSU	RSE*SU
60	Jump on Test	JT	JP
61	Jump	J	JP
64	Store Location and Jump on Test	SLJT	RJP
65	Store Location and Jump	SLJ	RJP
70	Repeat	R	RPT
71	Test B and/or Increment	TBI B_j	BSK*B_j
72	Jump on B and Decrement	JBD B_j	BJP*B_j
7701	Floating Add	FA	FADD
7702	Floating Subtract	FAN	FSUB
7703	Floating Multiply	FM	FMUL
7705	Floating Divide	FD	FDIV
7706	Floating Point Pack	FP	FPP
7707	Floating Point Unpack	FU	FPU
7710	Decimal Test	DT	DTEST
7711	Decimal Add	DA	DADD
7712	Decimal Subtract	DAN	DSUB

Table C–1. Assembler/SPURT Function Codes (part 2 of 3)

F	MNEMONIC DESCRIPTION	ASSEMBLER	SPURT
7713	Decimal Test Equal	DTE	DCME
7714	Decimal Complement AQ	DN	DCP
7715	Decimal Add with Carry	DAC	DADDC
7716	Decimal Subtract with Borrow	DANB	DSUBB
7717	Decimal Test Less	DTL	DCML
7721	Double Precision Load	DPL	DPENT
7722	Double Precision Add	DPA	DPADD
7723	Double Precision Test Equal	DPTE	DPCME
7724	Double Precision Complement	DPN	DPCP
7725	Double Precision Store	DPS	DPSTR
7726	Double Precision Subtract	DPAN	DPSUB
7727	Double Precision Test Less	DPTL	DPCML
7730	Scale Factor Shift	SFS	SFSH
7731	Character Pack Lower	CPL	CREL
7732	Character Pack Upper	CPU	CREU
7733	Convert Lower	DCL	DCVL
7734	Convert Upper	DCU	DCVU
7735	Character Unpack Lower	CUL	CRSL
7736	Character Unpack Upper	CUU	CRSU
7737	Execute Remote	ER	XQT
7740	Unconditional Jump	LBPJ B0	EBJP*B0
7741	Load B1 with contents P register and Jump	LBPJ B1	EBJP*B1
7742	Load B2 with contents P register and Jump	LBPJ B2	EBJP*B2
7743	Load B3 with contents P register and Jump	LBPJ B3	EBJP*B3
7744	Load B4 with contents P register and Jump	LBPJ B4	EBJP*B4
7745	Load B5 with contents P register and Jump	LBPJ B5	EBJP*B5
7746	Load B6 with contents P register and Jump	LBPJ B6	EBJP*B6
7747	Load B7 with contents P register and Jump	LBPJ B7	EBJP*B7
7751	Logical Right Shift Q	LRSQ	LRSQ
7752	Test and Set	TSET	TSET
7753	Masked Alphanumeric Test Equal	MATE	MACE
7754	Executive Return	EXRN	EXRN
7755	Logical Right Shift A	LRSA	LRSA
7756	Logical Right Shift AQ	LRSAQ	LRSAQ
7757	Masked Alphanumeric Test Less	MATL	MACL
7771	Load B-Worker Registers	LBW	EWB
7775	Store B-Worker Registers	SBW	SWB

Table C-1. Assembler/SPURT Function Codes (part 3 of 3)

APPENDIX D. ERROR FLAGS

D1. GENERAL

The following error flags may appear in the listing at assembly time.

FLAG	INDICATION
U	Undefined symbol
D	Doubly defined
R	Relocation
L	Level
T	Truncation
E	Expression error
I	Illegal operation
P	Parameter error

D2. U (UNDEFINED)

If a symbol has not been defined in its own program element, the U flag will result, even though the symbol may be listed as an external reference (in an XREF directive) Thus, this flag may not always indicate a logical error (see 5.13).

D3. D (DOUBLY DEFINED)

The D flag will result when a label has been defined more than once in a program element. In some cases, this may be desired by the programmer and the LET directive avoids any such confusion (see 5.17).

D4. R (RELOCATION)

The R flag indicates that a relocatable item in an expression has lost its relocatability property due to an arithmetic operation. (See 3.5.3 for those operations causing the R flag.)

D5. L (LEVEL)

The L flag indicates that some capacity of the assembler has been exceeded. Possible causes of the L flag are:

- nesting of PROC's and FUNC's beyond the maximum of 16
- nesting of DO-loops beyond the maximum of 8

D6. T (TRUNCATION)

The T flag indicates that a value is too large for its destined field. As an example, in the source-coded line ALPHA

```
ALPHA  + 'A', 'B', 1, 2, 67D
```

the decimal 67 is too large for the rightmost six bits of the word, and the T flag will be generated.

D7. E (EXPRESSION)

The E flag indicates an invalid expression. Possible causes are (but are not restricted to) the inclusion of a decimal digit in an octal number (e.g., 080) or a fractional exponent with the shift exponent operator (e.g., A*/3.5D).

D8. I (ILLEGAL)

The I flag indicates that the first source coded subfield does not contain the name of a directive, nor the name of an available procedure, nor a legitimate mnemonic. The one exception to this is the case where the operation field starts with a plus or minus sign to indicate data.

D9. P (PARAMETER)

The P flag indicates faulty specification of argument(s) for an operation. For example, the pseudo-op ZA W would result in a P flag because no operand (argument) is required for the instruction to clear the A register.

APPENDIX E. OPERATION FIELD HIERARCHY

A symbol in the operation field may be defined in different areas of the UNIVAC 494 Operating System. The following sources are searched, in the order shown, until the symbol is defined. As soon as it is defined, the search ends.

The sources are:

- FORM labels (see 5.5)
- Assembler directives
- PROC entry points
- Assembler mnemonics
- SPURT directives
- SPURT mnemonics
- SPURT macro labels
- SPURT poly-ops
- system operators (OMEGA packets)

APPENDIX F. #ASM OPTIONS

The following table lists available options on the •ASM card (see "UNIVAC 494 Operating System Programmer Reference Manual," UP-7504 (current version)).

OPTION	FUNCTION
B	Inhibit basic mnemonics
C	Check sequence numbers
D	Delete sequence numbers from listing
F	Inhibit PROC's and FUNC's
G	Allow extended language features
H	Reverse normal operation field hierarchy
I	Use single spaced listing
L	Print relocation information
M	Print outline of UNIVAC 494 Assembler Reference Manual
N	Suppress listing except for errors
O	Inhibit OMEGA packets
P	Punch deck
Q	Absolute assembly
R	Produce cross referencing
S	Place SDEF's (Symbol DEFinitions) with RB element
U	Only symbols listed in XREF directives are valid external references
W	Suppress error printout
X	Abort job on errors
Y	Do not indicate errors in TOC
Z	Abort task on errors

UNIVAC