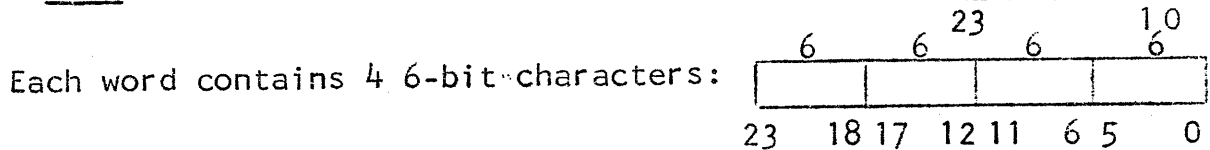
Mth 351 Notes

## CDC 3300 Computer

(The following information refers specifically to the CDC 3300 computer, and does not necessarily apply to other computers.)

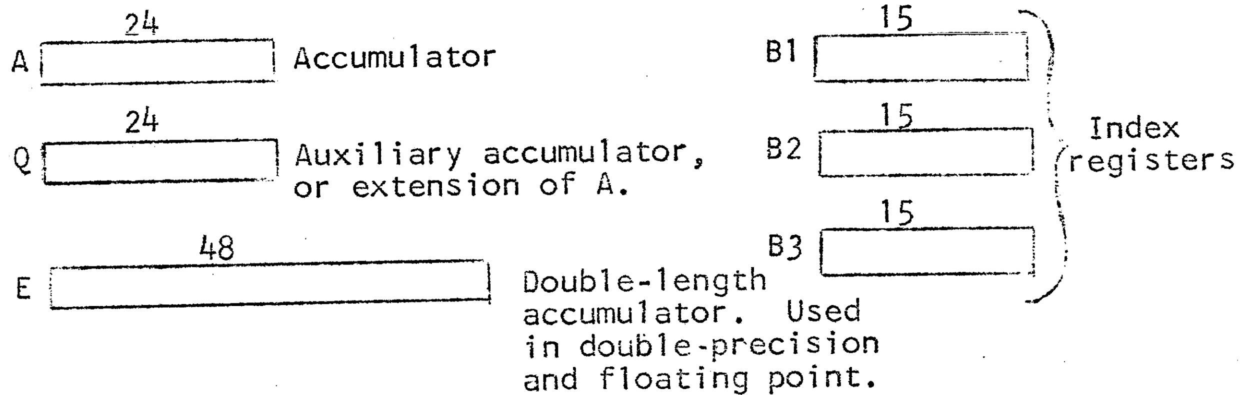
A word is a storage element composed of 24 bits:

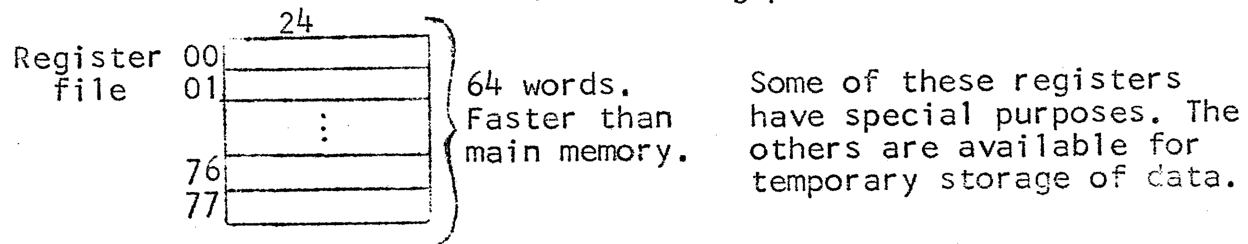
Each word contains 4 6-bit characters:



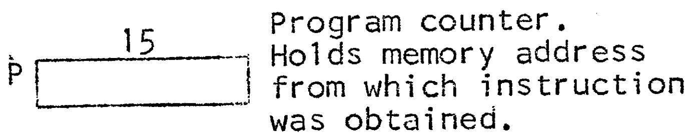

Most instructions refer to words or portions of words. A few instructions refer to characters.

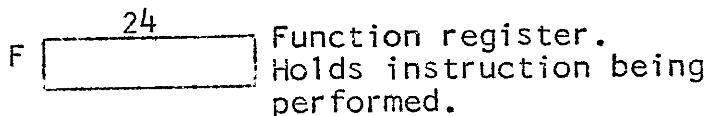
Registers. A register is a storage element which has special purposes. The CDC 3300 has the following registers:



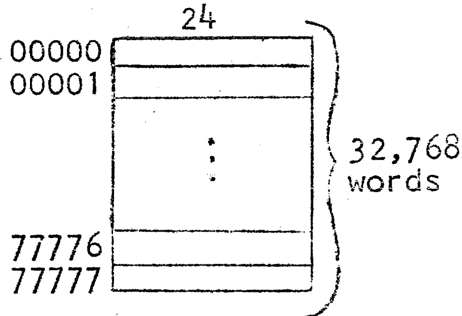




Control Registers: These registers are in the control unit.



Memory. The main memory consists of magnetic core storage with a 1.25$\mu$s cycle time.



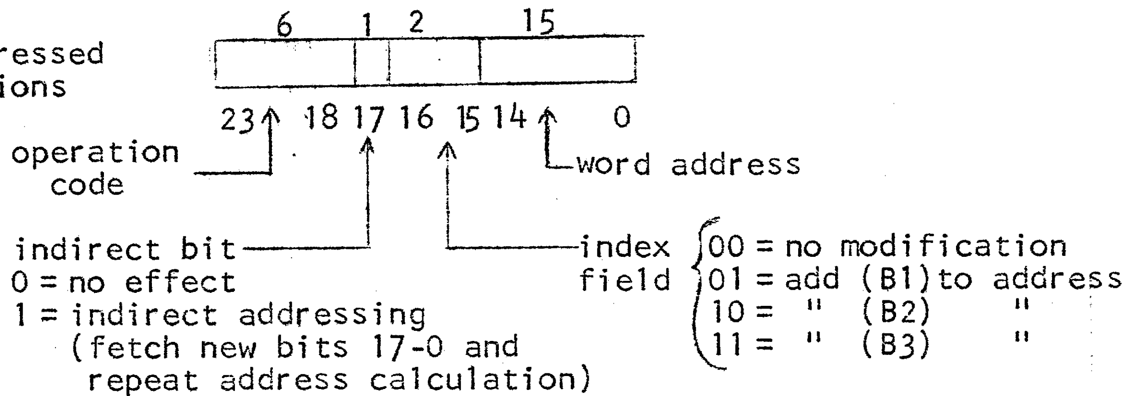

There are $2^{15}$= 32,768 words of 24 bits each, or $2^{17}$= 131,072 characters of 6-bits.

Memory can be expanded to as much as $2^{18}$= 262,144 words.

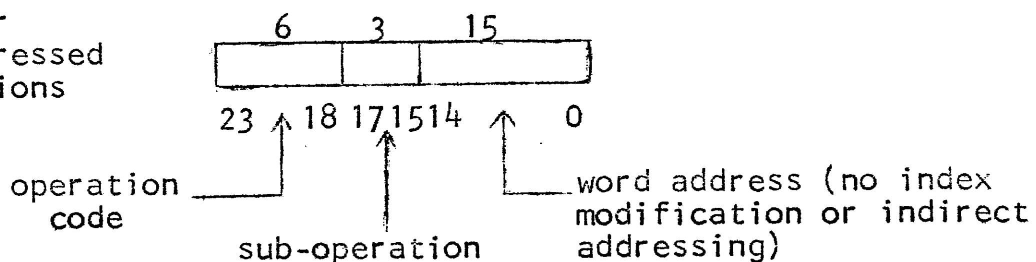
Main memory can hold instructions and data.

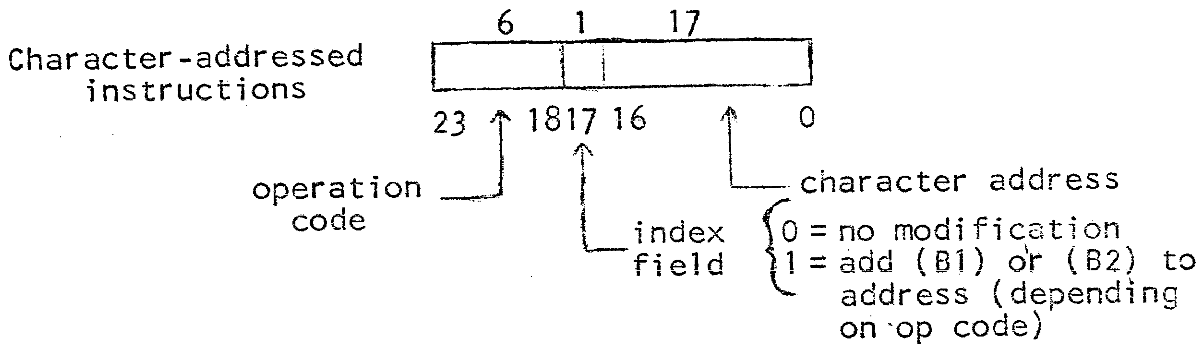
Instruction Formats. Most instructions occupy one word, but some occupy two or three words. There are many different formats. In all cases, the left-most six bits determine the operation (or group of operations). The format of the instruction depends on which operation code is specified in the first 6 bits. Here are some of the 1-word instruction formats:

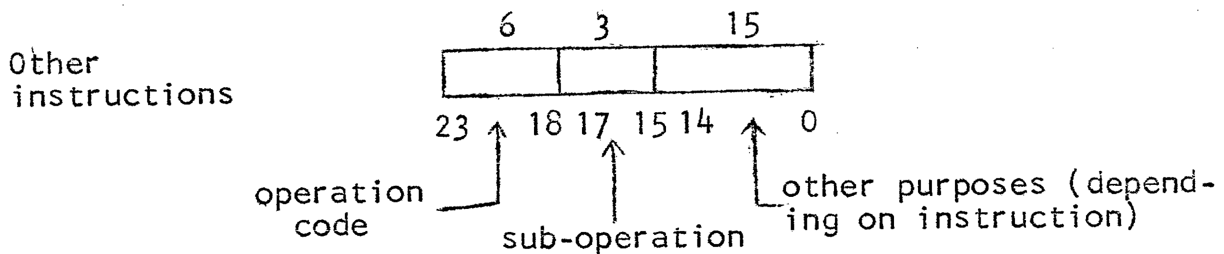
Most word-addressed instructions




Other word-addressed instructions






Arithmetic. There is an optional Business Data Processor (BDP) which can be added to the computer. This unit does decimal arithmetic and character handling. All other arithmetic in the 3300 is binary, one's complement arithmetic.

In one's complement notation, the left-most bit determines the sign of a number. If the left-most bit is 0, the number is positive; if 1, negative. To change the sign of a number, all bits are changed (complemented). All zeroes (0000) denotes +0, all ones (1111) denotes -0. A number can be lengthened by extending its sign bit: +5 = 0101 = 00000101, -5 = 1010 = 11111010. Addition is performed by simply adding two binary numbers, and if a carry occurs from the left, adding one to the right-most bit (end-around carry).

In the 3300, if the result of an arithmetic operation is -0, it is changed to +0. For example:

```
  111110   (-1)
+ 000001   (+1)
  ------
  111111   (-0)
changed to ↓
  000000   (+0)
```

There are 3 modes of binary arithmetic in the 3300. The numbers indicate the number of bits in the operands and results.

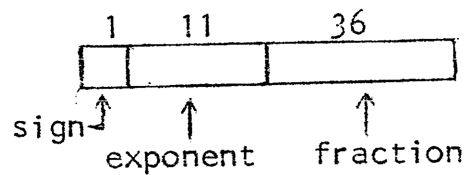
<u>Single-precision fixed point</u>

addition: (24)+(24)→(24) ⎫ Overflow sets
subtraction: (24)-(24)→(24) ⎭ an indicator
multiplication: (24)*(24)→(48)
division: (48)/(24)→(24)quotient,
(24)remainder

<u>Double-precision fixed point</u>

addition: (48)+(48)→(48) ⎫ overflow sets
subtraction: (48)-(48)→(48) ⎭ an indicator
multiplication: (48)*(48)→(96)
division: (96)/(48)→(48)quotient,
(48)remainder

<u>Floating point</u> addition, subtraction, multiplication, and division operate on 48-bit floating point numbers:



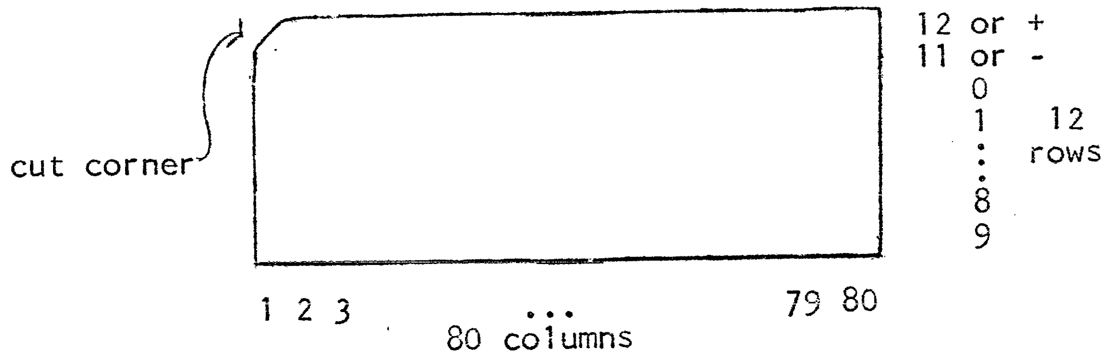
Character codes

| BCD code (octal) | Card code | Key Punch character | Line Printer character |
|---|---|---|---|
| 00 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 |
| 02 | 2 | 2 | 2 |
| 03 | 3 | 3 | 3 |
| 04 | 4 | 4 | 4 |
| 05 | 5 | 5 | 5 |
| 06 | 6 | 6 | 6 |
| 07 | 7 | 7 | 7 |
| 10 | 8 | 8 | 8 |
| 11 | 9 | 9 | 9 |
| 12 | 2,8 | | : |
| 13 | 3,8 | = | = |
| 14 | 4,8 | ' | ≠ |
| 15 | 5,8 | | ≤ |
| 16 | 6,8 | | % |
| 17 | 7,8 | FILE CARD | [ |
| 20 | 12 | + | + |
| 21 | 12,1 | A | A |
| 22 | 12,2 | B | B |
| 23 | 12,3 | C | C |
| 24 | 12,4 | D | D |
| 25 | 12,5 | E | E |
| 26 | 12,6 | F | F |
| 27 | 12,7 | G | G |
| 30 | 12,8 | H | H |
| 31 | 12,9 | I | I |
| 32 | 12,0 | | < |
| 33 | 12,3,8 | . | . |
| 34 | 12,4,8 | ) | ) |
| 35 | 12,5,8 | | ≥ |
| 36 | 12,6,8 | | ¬ |
| 37 | 12,7,8 | | ; |

| BCD code (octal) | Card code | Key Punch character | Line Printer character |
|---|---|---|---|
| 40 | 11 | — | — |
| 41 | 11,1 | J | J |
| 42 | 11,2 | K | K |
| 43 | 11,3 | L | L |
| 44 | 11,4 | M | M |
| 45 | 11,5 | N | N |
| 46 | 11,6 | O | O |
| 47 | 11,7 | P | P |
| 50 | 11,8 | Q | Q |
| 51 | 11,9 | R | R |
| 52 | 11,0 | | ∨ |
| 53 | 11,3,8 | $ | $ |
| 54 | 11,4,8 | * | * |
| 55 | 11,5,8 | | ↑ |
| 56 | 11,6,8 | | ↓ |
| 57 | 11,7,8 | | > |
| 60 | Blank | Blank | Blank |
| 61 | 0,1 | / | / |
| 62 | 0,2 | S | S |
| 63 | 0,3 | T | T |
| 64 | 0,4 | U | U |
| 65 | 0,5 | V | V |
| 66 | 0,6 | W | W |
| 67 | 0,7 | X | X |
| 70 | 0,8 | Y | Y |
| 71 | 0,9 | Z | Z |
| 72 | 0,2,8 | | ] |
| 73 | 0,3,8 | , | , |
| 74 | 0,4,8 | ( | ( |
| 75 | 0,5,8 | | → |
| 76 | 0,6,8 | | ≡ |
| 77 | 0,7,8 | | ∧ |

Punched Cards. (Also called tabulating cards, tab cards, Hollerith cards, and IBM cards.)

A punched card is a card 7 3/8" long and 3 1/4" wide which is used in various devices that can punch holes in such cards and/or sense the presence of holes. Usually, one side of the card is printed. When looking at the printed face of the card, one will find that either the upper left or upper right corner is cut. The other corners may be square or rounded. See figure below.




There are 80 columns and 12 rows, defining 960 positions where holes may be punched in the card. There are two common ways of representing information in terms of punched holes. One way is the "binary" card. In this mode, any combination of holes may be punched, all combinations are meaningful. This mode allows a maximum of information to be punched into the card, but it is difficult for humans to interpret such cards. (Also, the presence of a large number of holes tends to weaken a card.)

The other way of representing information is to use each column to represent a single character (digit, letter, or special character). This mode, which is called "Hollerith" or "BCD" uses a code of 1, 2, or 3 punches in a column to designate a character. Only certain combinations of holes are meaningful. There are 47 characters, plus blank (no punches), which are commonly used. See page 5 for a list of characters and the corresponding card codes.

Card-handling machines. There are many different kinds of machines which can use punched cards, such as sorters, printers, etc. Here we shall be concerned with the following:

Card reader. Attached to the CDC 3300 is a photoelectric card reader which can read 1200 cards per minute. This machine can read cards in either the binary or BCD modes, as commanded by the computer. However, if the first column of the card

contains a 7 and a 9 punch, the card is read as a binary card, regardless of which other holes may be punched. The table on page 5 gives the octal (so-called BCD) codes which the reader supplies to the computer when it reads a BCD card.

Card punch. Also attached to the CDC 3300 is a card punch which can punch cards in either binary or BCD mode at a rate of 250 cards per minute. This relatively slow device is used primarily to punch binary forms of assembled and compiled programs.

Key punch. A key punch has a keyboard with letters, digits, special characters, and various control buttons on it. The key punch can punch and/or read cards, feeding them from a hopper at the top right of the machine and stacking them at top left. There are a number of IBM key punches in the computer center, which are used by human operators to prepare punched cards. The key punches are used to punch programs and data for input to computers, or for use in other card machines.

Line Printer. The principal output device on the CDC 3300 is a line printer which can print as many as 136 characters on a line, at a rate of 1000 lines per minute. There are 63 different characters which can be printed, plus blank. The chart on page 5 shows the characters that are available, and the octal (BCD) codes which must be sent to the printer by the computer to print these characters.

The printer prints 10 characters per inch on a line, 6 lines per inch, 66 lines per page. (In ordinary usage, fewer than 66 lines are printed, to allow a margin at top and bottom of the page.)

Input-Output Programming. The programming for the computer to handle input from the card reader and output to the line printer is rather complicated. To avoid these complexities (for a while, at least), we shall use a set of subroutines for input and output which are easy to use. These subroutines are described on a later page.

## COMPASS

COMPASS (COMPrehensive ASSembler) is an assembly program for the CDC 3300 which enables us to write programs in a symbolic form. The assembler translates the symbolic form into the actual binary operation codes and addresses which the computer can interpret. Thus, we use ADA to denote "Add to A"; COMPASS translates ADA into the operation

code 30 (octal). We can use a symbol such as X as the name of a storage location in memory; COMPASS translates this into an actual machine address. We do not need to know what the actual address is.

In COMPASS, there is a set of pre-defined mnemonic operation codes (such as LDA, ADA, etc.) which represent machine operation codes. There are also some "pseudo-instructions" which tell COMPASS to reserve storage for data, or to do various other things. Some of these symbolic instructions are listed on following pages.

To refer to storage locations containing instructions or data, we use names of our own choosing. These symbolic names may consist of 1 to 8 letters, digits, or periods (.), of which the first character must be a letter. All symbols used in a program must be defined, either by appearing in the location field of some instruction, or by being declared as external to the program.

A COMPASS-language program consists of a sequence of symbolic instructions. These are written one per line, and punched one per card. The card is divided into several fields, as illustrated below.

| LOCATION | | OPERATION,MODIFIERS | ADDRESS FIELD | COMMENTS | IDENT |
|---|---|---|---|---|---|
| 1 8 | 9 | 10 19 | 20 40 | 41 72 | 73 80 |

<u>Location field</u>. Columns 1 to 8. May be blank, or may contain a 1 to 8 character symbol (see above) placed anywhere in the field.

<u>Column 9</u>. Must be blank.

<u>Operation field</u>. The mnemonic operation code or pseudo-instruction, and any modifiers following it, must start in column 10. It is terminated by the first blank column. One may also use a two-digit octal operation code in columns 10 and 11. (If column 10 is blank, the operation field is assumed to be absent and is assembled as zero.)

<u>Address field</u>. The address field may start anywhere after the blank that terminates the operation field, but must start no later than column 40. It is terminated by the first blank or column 73. The address field may consist of several sub-fields, separated by commas. It usually contains a symbol, a decimal integer (positive or negative), or an octal integer (positive or negative, consisting of 1 or more octal digits followed by "B"), or it may consist of several of these items connected by "+" or "-" signs. The asterisk (*)

may be used in the address field like a symbol, and represents the location of the instruction itself. If the address field consists solely of the notation ** , it will be assembled with the address field filled with 1-bits.

Comments field. Comments or remarks may be written after the address field, up to column 72.

Identification. Columns 73 to 80 are treated as a comment by COMPASS. This field is usually left blank, or used for identification, or sequence numbers.

Comment card. If column 1 contains an asterisk (*), the entire card is treated as a comment.

Note: It is recommended that the address field begin in column 20, and comments in column 41, for legibility of the printed listing.

## Operation Codes

Here we list some of the operation codes available in COMPASS. We include some jump, load, store, arithmetic, shift, and logical instructions. In the following table, the octal operation code is in the leftmost column, then the COMPASS mnemonic code, the form of the address (m means a memory address, k stands for a shift parameter), name and description of the operation. (A) denotes contents of A, (Q) denotes contents of Q, etc. The arrow → means "replaces the contents of". Thus, (m) → (A) means that the contents of m are copied into the A register. A bar over a quantity (for example: $(\overline{m})$) denotes "complement of" (all 0-bits changed to 1-bits and vice-versa). (m + 1) denotes contents of location m + 1 (it does not mean contents of location m, plus 1).

## Jump Instructions

| | | | | |
|---|---|---|---|---|
| 01 | UJP | m | Unconditional Jump. | Take next instruction from location m. (Jump to m.) |
| 00.7 | RTJ | m | Return Jump. | Store program counter (P) in address field (bits 14-0) at location m, and take next instruction from location m + 1. |
| 03.0 | AZJ,EQ | m | A Zero Jump, Equal. | If (A) = 0, jump to m. Otherwise, take next instruction in sequence (+0 and -0 are treated as 0.) |

| | | | |
|---|---|---|---|
| 03.1 | AZJ,NE m | A Zero Jump, Not Equal. | If (A) $\neq$ 0, Jump to m. |
| 03.2 | AZJ,GE m | A Zero Jump, Greater or Equal. | If (A) $\geq$ 0, jump to m. (Does not jump if (A) = -0.) |
| 03.3 | AZJ,LT m | A Zero Jump, Less Than. | If (A) < 0, jump to m. (Jumps if (A) = -0.) |
| 03.4 | AQJ,EQ m | AQ Jump, Equal. | If(A) = (Q), jump to m. (+0 = -0) |
| 03.5 | AQJ,NE m | AQ Jump, Not EQual. | If (A) $\neq$ (Q), jump to m. (+0 = -0) |
| 03.6 | AQJ,GE m | AQ Jump, Greater or Equal. | If (A) $\geq$ (Q), jump to m. (+0 > -0) |
| 03.7 | AQJ,LT m | AQ Jump, Less Than. | If (A) < (Q), jump to m. (+0 > -0) |

## Load and Store Instructions

| | | | | |
|---|---|---|---|---|
| 20 | LDA | m | Load A. | (m) $\rightarrow$ (A). |
| 21 | LDQ | m | Load Q. | (m) $\rightarrow$ (Q). |
| 24 | LCA | m | Load Complement A. | $(\overline{m}) \rightarrow$ (A). |
| 25 | LDAQ | m | Load AQ. | (m) $\rightarrow$ (A), (m + 1) $\rightarrow$ (Q). |
| 26 | LCAQ | m | Load Complement AQ. | $(\overline{m}) \rightarrow$ (A), $(\overline{m+1}) \rightarrow$ (Q). |
| 40 | STA | m | Store A. | (A) $\rightarrow$ (m). |
| 41 | STQ | m | Store Q. | (Q) $\rightarrow$ (m). |
| 45 | STAQ | m | Store AQ. | (A) $\rightarrow$ (m), (Q) $\rightarrow$ (m + 1). |
| 55.3 | EAQ | | E to AQ. | (E) $\rightarrow$ (AQ), (No address field.) |
| 55.7 | AQE | | AQ to E. | (AQ) $\rightarrow$ (E). " |

## Arithmetic Instructions

| | | | | |
|---|---|---|---|---|
| 30 | ADA | m | Add to A. | (A) + (m) $\rightarrow$ (A). |
| 31 | SBA | m | Subtract from A. | (A) - (m) $\rightarrow$ (A). |
| 32 | ADAQ | m | Add to AQ. | (AQ) + (m, m + 1) $\rightarrow$ (AQ). |
| 33 | SBAQ | m | Subtract from AQ. | (AQ) - (m, m + 1) $\rightarrow$ (AQ). |
| 50 | MUA | m | Multiply A. | (A) * (m) $\rightarrow$ (QA). |
| 51 | DVA | m | DIVIDE A. | (AQ) / (m) $\rightarrow$ (A), remainder to (Q). |
| 56 | MUAQ | m | Multiply AQ. | (AQ) * (m,m+1) $\rightarrow$ (AQE). |
| 57 | DVAQ | m | DIVIDE AQ. | (AQE)/(m,m+1) $\rightarrow$ (AQ), remainder $\rightarrow$ (E). |

## Shift Instructions

| | | | | |
|---|---|---|---|---|
| 12.0 | SHA | k | Shift A. | If k $\geq$ 0, shift (A) left end-around k places.<br>If k < 0, shift (A) right sign-extended -k places. |

## Shift Instructions (cont.)

12.4 SHQ k Shift Q. Same as SHA, except shift (Q).

13.0 SHAQ k Shift AQ. Same as SHA, except shift (AQ).

## Logical Instructions

27 LDL m Load Logical. $(Q)\wedge(m) \rightarrow (A)$.

35 SSA m Selectively Set A. $(A)\vee(m) \rightarrow (A)$.

36 SCA m Selectively Complement A. $(A)\not\vee(m) \rightarrow (A)$. (Exclusive OR)

37. LPA m Logical Product A. $(A)\wedge(m) \rightarrow (A)$.

## Pseudo Instructions

Here we list some of the more important pseudo instructions in COMPASS.

<blank> IDENT m

The first instruction of each subprogram must be an IDENT. The "m" here is the name of the subprogram (8 characters or less).

<blank> END m

The last instruction of each subprogram must be an END. The "m" is normally a symbolic location at which the subprogram is to be started. This symbol must also appear on an ENTRY card. The address field (m) should be left blank if this subprogram is not to receive control from SCOPE.

<blank> FINIS <blank>

This instruction immediately follows the END of the last subprogram to be assembled. It terminates assembly and causes COMPASS to return control to SCOPE.

<blank> ENTRY $m_1, m_2, \ldots, m_n$

The symbols $m_1, m_2, \ldots$ appearing in the address field of an ENTRY card are declared as entry points to the subprogram, which means that they can be referred to by other

subprograms. These symbols must be defined in this subprogram.

<blank> EXT $m_1, m_2, \ldots, m_n$

The symbols $m_1, m_2, \ldots$ are symbols used in address fields of instructions in this subprogram, which refer to entry points of other subprograms. They must not be defined in this subprogram.

<symbol or blank> BSS m

This instruction reserves m words of storage space in the subprogram. If a symbol appears in the location field, it is defined as the address of the first word of the block of words reserved.

<symbol or blank> OCT $m_1, m_2, \ldots, m_n$

The $m_1$, $m_2$, ... are signed or unsigned octal integers, of 1 to 8 octal digits. They are assembled into consecutive words in the program. If a symbol appears in the location field, it is defined as the address of the first octal integer.

<symbol or blank> DEC $m_1, m_2, \ldots, m_n$

Similar to OCT, except that decimal integers are converted to binary form and assembled into the program.

## Input-Output Subroutine

INCHAR (Entry points: INCHAR, LCI, CHEOF.)

INCHAR reads cards and supplies one character each time it is called. See page 5 for the codes supplied for various characters. Where there is a string of consecutive blanks on the card, INCHAR returns only one blank. INCHAR supplies a blank at the end of each card.

Calling sequence: RTJ INCHAR

Returns with next character in $(A)_{5-0}$ (rest of A zero).

The Last Character In is also in location LCI.

Clobbers Q. Restores B1, B2, B3.

To check for End-of-File, store an address in location CHEOF. When an end-of-file card is read, the subroutine will jump to this address. (For example, to cause jump to EOFCHK when EOF is read, do the instructions ENA EOFCHK,

SWA CHEOF .) This should be done before first call on INCHAR. If it is not done, an end-of-file will cause abnormal termination of job.

OUTCHAR (Entry Points: OUTCHAR,LCO, OUTLINE, OUTPAGE.)
OUTCHAR, OUTLINE, and OUTPAGE are three subroutines which provide for printing outputs on the line printer. Characters to be printed are supplied one at a time to OUTCHAR. OUTCHAR stores these characters in a buffer (an area of memory). When 100 characters have been stored, OUTCHAR causes the line to be printed, and resets, ready for another 100 characters. OUTLINE can be called to print whatever is in the buffer. OUTPAGE is called to eject the paper to the top of the next page.

Calling sequence:

Character $(A)_{5-0}$ (Rest of A does not matter.)

RTJ OUTCHAR

Stores character in buffer, prints line if 100 characters have been supplied since last line was printed. Last Character Out is also stored in LCO. Clobbers A and Q, restores B1, B2, and B3.

Calling sequence: RTJ OUTLINE
Causes a line to be printed, containing the characters which have been supplied to OUTCHAR since last line was printed. If no characters have been supplied, prints blank line. Clobbers A, Q, restores B1, B2, B3.

Calling sequence: RTJ OUTPAGE
If any characters have been supplied to OUTCHAR since last line was printed, causes a line to be printed. Then causes the paper to be ejected to the top of the next page. The next line will be printed about one inch from the top of the new page. Clobbers A, Q, restores B1, B2, B3.

INDEC (Entry points: INDEC, NODIGS.)
INDEC reads characters from INCHAR and converts a positive or negative decimal integer to binary form, allowing up to 48 bits in the binary form. (Maximum size of decimal integer: 140, 737, 488, 355, 327.) INDEC begins by examining the character in LCI (Last Character In). It ignores all characters until it gets a minus sign (-) or a decimal digit. Then it accepts decimal digits, converting to binary, until it receives a character from INCHAR that is not a decimal digit. If the number was preceded by a minus sign, the decimal integer is complemented (after converting to binary).

Calling sequence: RTJ INDEC

Returns with binary equivalent of decimal integer in AQ. The terminating character is in LCI. The number of digits that the number contained (counting leading zeroes, if any) is in location NODIGS. Clobbers E. Restores B1, B2, B3.

<u>OUTDEC</u> (Entry point: OUTDEC.)

OUTDEC accepts a 48-bit positive or negative binary integer, which it converts to a decimal integer with sign, and outputs it to OUTCHAR. The sign (+ or -) is printed first, then the decimal digits with leading zeroes suppressed (at least one digit is printed). A space (blank) is output after the number. If the current line does not have enough room left for the number, OUTDEC calls OUTLINE to print the current line, and the number will be printed on the next line. This avoids "splitting" a number on two lines.

Calling sequence: Binary integer → (AQ)
RTJ OUTDEC

Clobbers A, Q, E. Restores B1, B2, B3.

<u>Debugging Aids</u>. We have two subroutines which can be called to print contents of registers and of selected areas of memory. Calls to these subroutines can be inserted at various places in a program to obtain information which may be of help in determining what is wrong with the program. The subroutines are described below.

<u>REGDUMP</u> (Entry point: REGDUMP.)
Calling sequence: RTJ REGDUMP

REGDUMP first saves the contents of various registers. Then it calls OUTLINE to print any characters which may have been given to OUTCHAR. It next prints one line giving the contents (in octal) of P, A, Q, E, B1, B2, B3. (The value of P is the octal location of the instruction following the RTJ REGDUMP.) Finally, REGDUMP restores the contents of the registers and returns to the program.

(Be sure to declare REGDUMP as external if you insert calls to REGDUMP into a program.)

<u>MEMDUMP</u> (Entry point: MEMDUMP.)
Calling sequence consists of three words:

```
RTJ   MEMDUMP
00    FWA
00    LWA
```

Like REGDUMP, MEMDUMP saves the registers and calls OUTLINE. Then it prints a line giving the value of P in octal (which is the location of the word following the RTJ MEMDUMP.) Next it prints (in octal) the contents of all the memory locations from FWA (<u>F</u>irst <u>W</u>ord <u>A</u>ddress) to LWA (<u>L</u>ast <u>W</u>ord <u>A</u>ddress), inclusive. These are printed 8 words per line, with the octal location of the first word in each line printed at the left of the page. FWA and LWA can be any symbolic locations in the program. (Usual COMPASS address expressions may be used.) Finally, the registers are restored, and control is returned to the instruction following the 00 LWA.

Be sure to declare MEMDUMP as external. The memory map will be useful in determining the correspondence between symbolic addresses and the absolute (octal) addresses printed by MEMDUMP.

<u>COMPASS Error Messages</u>. If there is an error in a line of a COMPASS program, an error flag (a single letter) is printed at the left of the listing. Here are some of the error flags which you may encounter. (It is possible to

have more than one error in the same line.)

- A Address field error.
- D Duplicate symbol (symbol defined more than once).
- L Location field error.
- M Modifier subfield of operation code field is in error.
- O Operation code error.
- U Undefined symbol in address field.

If any errors are detected by COMPASS, the program will neither be loaded nor run.

## More Input-Output Subroutines

OUTDECF (Entry point: OUTDECF)

OUTDECF accepts a 48-bit positive or negative binary integer, which it converts to a decimal integer, and outputs it to OUTCHAR. The format of the output may be controlled by two numbers, n and c, which are specified in the word following the RTJ OUTDECF . OUTDECF will print n digits, or more if necessary to represent the number correctly. If c = 0, the sign (+ or -) will be printed in front of the number, leading zeroes (if any) are converted to blanks (and the sign is moved over accordingly), and a trailing blank is output. Other values of c have the following effects:

c = 1: does not convert leading zeroes to blanks (prints zeroes).
c = 2: does not print the sign.
c = 4: does not print the trailing blank.

Combinations of these effects may be obtained by adding the values given above. For example, c = 3 has the effect of both 1 and 2.

If the current line does not have enough room for the number, OUTDECF calls OUTLINE to print the current line, and the number will be printed on the next line.

Calling sequence:

```
Binary integer ⟶ (AQ)
  RTJ    OUTDECF
  00     n, c          (0 ≤ n ≤ 31, 0 ≤ c ≤ 7)
Clobbers A, Q, E.  Restores B1, B2, B3.
```

IND (Entry point: IND)

IND reads characters from INCHAR and converts a positive or negative decimal number to 24:24 fixed point binary form (24 bit integer portion, 24 bit fraction). IND begins by examining the character in LCI. It ignores

all characters until it gets a minus sign (-), a decimal digit, or a point (.). Then it accepts decimal digits, a point (if it hasn't already gotten one), and more digits. When a non-digit, non-point is received from INCHAR, it converts the number to a 24:24 binary number, and complements it if a minus sign preceded the number. Decimal numbers to be read by IND may start with a minus sign or no sign. (A plus sign would be ignored.) They may contain from 0 to 7 digits before the point and from 0 to 7 digits after the point. The point may be omitted if the number is an integer. The integer portion must not exceed 8388607 in magnitude, and the fraction portion must not exceed .9999999 . There must not be any blanks in the middle of the number, and the number must be entirely on one card.

Calling sequence: RTJ IND

Returns with 24:24 binary equivalent of decimal number in AQ. The terminating character is in LCI. Clobbers E. Restores B1, B2, B3.

## Another Output Subroutine

OUTD (Entry point: OUTD)

OUTD accepts a 24:24 fixed point binary number, which it converts to decimal form with sign, and outputs it to OUTDECF and OUTCHAR. The sign (+ or -) is always printed. 7 digits are printed before the point (leading zeroes are converted to blanks), and 7 digits are printed after the point. A trailing blank is output after the number. OUTLINE is called before printing the number if there is not enough room left on the current line for the number.

Calling sequence: 24:24 binary number → (AQ)

RTJ OUTD

Clobbers A, Q, E. Restores B1, B2, B3.

## Fixed-point Arithmetic

One method of using a computer to calculate with numbers that are not integers is the technique known as "fixed point arithmetic". In this method, one assumes that each number has a point somewhere in it, and programs the computer to handle the numbers properly. In general, the technique requires shifting numbers to line up the points before adding or subtracting, and shifting operands in multiplication and division so as to get the proper results. Here we shall discuss a special case of fixed point arith-

metic for the CDC 3300 in which the point is in the same place in all numbers. This will simplify things, although it poses some restrictions on the range and accuracy of numbers we can handle.

In this fixed point scheme, we shall use 2 words (48 bits) for each number, and we shall suppose that the point is between the two words. This means we have 24 bits before the point (integer part), and 24 bits after it (fraction part), so we shall refer to this as "24:24 arithmetic". Of course, the left-most bit is the sign bit, so we really have only 23 bits for the magnitude of the integer portion. This allows our numbers to range from -8388607 to +8388607, and to have an accuracy of 24 bits in the fraction, which is roughly equivalent to 7 decimal places.

Let us now show how to do arithmetic operations with 24:24 numbers. In each case below, we shall assume we want to do our operation with two 24:24 operands X and Y, and store the 24:24 result in Z. In actual practice, one of the operands may already be in AQ, or we may wish to do further calculations with the result and not bother storing it.

<u>Addition</u> and <u>subtraction</u> are very simple, since the point is in the same position in both operands, and no shifting is required to line up the points.

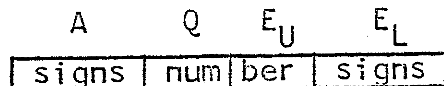
| add | subtract |
|---|---|
| LDAQ X | LDAQ X |
| ADAQ Y | SBAQ Y |
| STAQ Z | STAQ Z |

In multiplication, we multiply two 24:24 numbers, and get a 48:48 result (the number of places after the point in the result is the sum of the number of places after the points in the two operands). This result is in AQE with the point between AQ and E. It would be nice if we could simply shift AQE left 24 places to put the point between A and Q, where we want it to be. Unfortunately, the 3300 does not have any instructions that shift AQE. So, we have to do something like the following:

<u>multiply</u>

```
LDAQ X
MUAQ Y
EUA
SHAQ 24
STAQ Z
```

Division is a little more complicated. We want to place the numerator (dividend) in AQE as a 48:48 number, and then divide it by the denominator (divisor), which is a 24:24 number. This will give us a 24:24 quotient in AQ, which is just the result we want. The numerator must have its upper half placed in (Q) and its lower half in (E upper). (A) and (E lower) should be filled with sign bits. Here is one method of doing this:



| A | Q | $E_U$ | $E_L$ |
|---|---|---|---|
| signs | num | ber | signs |

divide

```
LDQ X     Upper half of X to (Q).
SHQ -23   Generate sign bits in (Q).
LDA X+1   Lower half of X to (A).
AQE       (AQ) to (E), sets up (E) properly.
LDA X     Upper half of X to (A).
SHAQ -24  Shift into (Q), filling (A) with sign bits.
DVAQ Y    Divide by Y (at last!)
STAQ Z    Store the quotient.
```

(A shorter method is shown below.)

Combined multiplication and division. We note that the result of a multiplication is a 48:48 number in AQE, and that this is exactly the form required for the numerator of a division. Hence, if we have to do a multiplication followed by a division, we can save a lot of trouble. Suppose we wish to compute W:=X*Y/Z, where X, Y, Z, and W are all 24:24 numbers. Here is how we can do it:

```
LDAQ X    X to (AQ).
MUAQ Y    X*Y in 48:48 form in (AQE).
DVAQ Z    X*Y/Z in 24:24 form in (AQ).
STAQ W    Store result.
```

This suggests a simpler way of doing division. To compute Z:=X/Y, we could rewrite it as Z:=1*X/Y and do the following:

```
ENA  1    These two instructions place a 1
ENQ  0      in 24:24 form in (AQ).
MUAQ X    1*X in 48:48 form in (AQE).
DVAQ Y    1*X/Y in 24:24 form in (AQ).
STAQ Z    Store result.
```

This requires fewer instructions than the method suggested above, but it might take a little more time, since the MUAQ instruction takes more time to do than instructions such as LDA and SHAQ.

Constants. One can use the DECD pseudo-instruction to cause COMPASS to assemble 24:24 constants and place them in a program. One must not use a decimal point in such constants, because COMPASS will assemble a floating point constant if a decimal point appears. We must express the number as an integer with a power of ten factor (D3, D-5, etc.), and binary factor B24. Here are a few examples. Suppose we wish to have the constants 3, 400, 5.87, -.0042, -12, and -83.5 stored in our program in 24:24 form. This could be done with the following instruction:

```
DECD   3B24,4D2B24,587D-2D24,-42D-4B24,-12B24,-835D-1B24
```

Each constant will occupy two words and they will be stored in consecutive locations in storage.

Square root. Here we give a square root subroutine for 24:24 numbers, to illustrate some programming techniques. The subroutine uses Newton's method, in which a new approximation XNEW is computed from the old one XOLD by the formula XNEW :=(XOLD + Y/XOLD)/2, where Y is the number whose square root is sought. This formula is applied repeatedly until the new and old approximations are nearly the same.

The calling sequence for the subroutine is:

```
24:24 number (Y)-->(AQ)
  RTJ   SQRT
24:24 result (√Y)-->(AQ)
```

If Y is negative, the subroutine simply returns with the original number Y in AQ.

```
SQRT   UJP      **
       AZJ,LT  SQRT     If Y < 0, forget it!
       STAQ    Y        Save Y.
       SCAQ    0        Scale Y to test for zero.
       AZJ,EQ  SQRT     If Y = 0, we are done already.
       LDAQ    INIT     Initial XOLD is 1.
       UJP     LOOP+1   Go to LOOP.
LOOP   LDAQ    XNEW     Pick up XNEW to use as XOLD.
       STAQ    XOLD     Set XOLD.
       LDAQ    Y+1      Set up Y.
       AQE                in (AQE) for
       LDAQ    Y-1        division.
       DVAQ    XOLD     Divide Y by XOLD.
       ADAQ    XOLD     Add XOLD to result.
       SHAQ    -1       Divide by 2.
       STAQ    XNEW     Store new approximation.
       SBAQ    XOLD     Subtract XOLD for comparison.
       AZJ,NE  LOOP     If XNEW-XOLD is not
       SHAQ    22        very small, do
       AZJ,NE  LOOP      another iteration.
       LDAQ    XNEW     It is small, return with
       UJP     SQRT       answer in (AQ).
```

(Square Root Subroutine, cont.)

```
        OCT    0
Y       BSS    2
        OCT    0
INIT    DECD   1B24
XOLD    BSS    2
XNEW    BSS    2
```

Input. Here we give the program and flow chart for the 24:24 input subroutine IND, described on page 16.

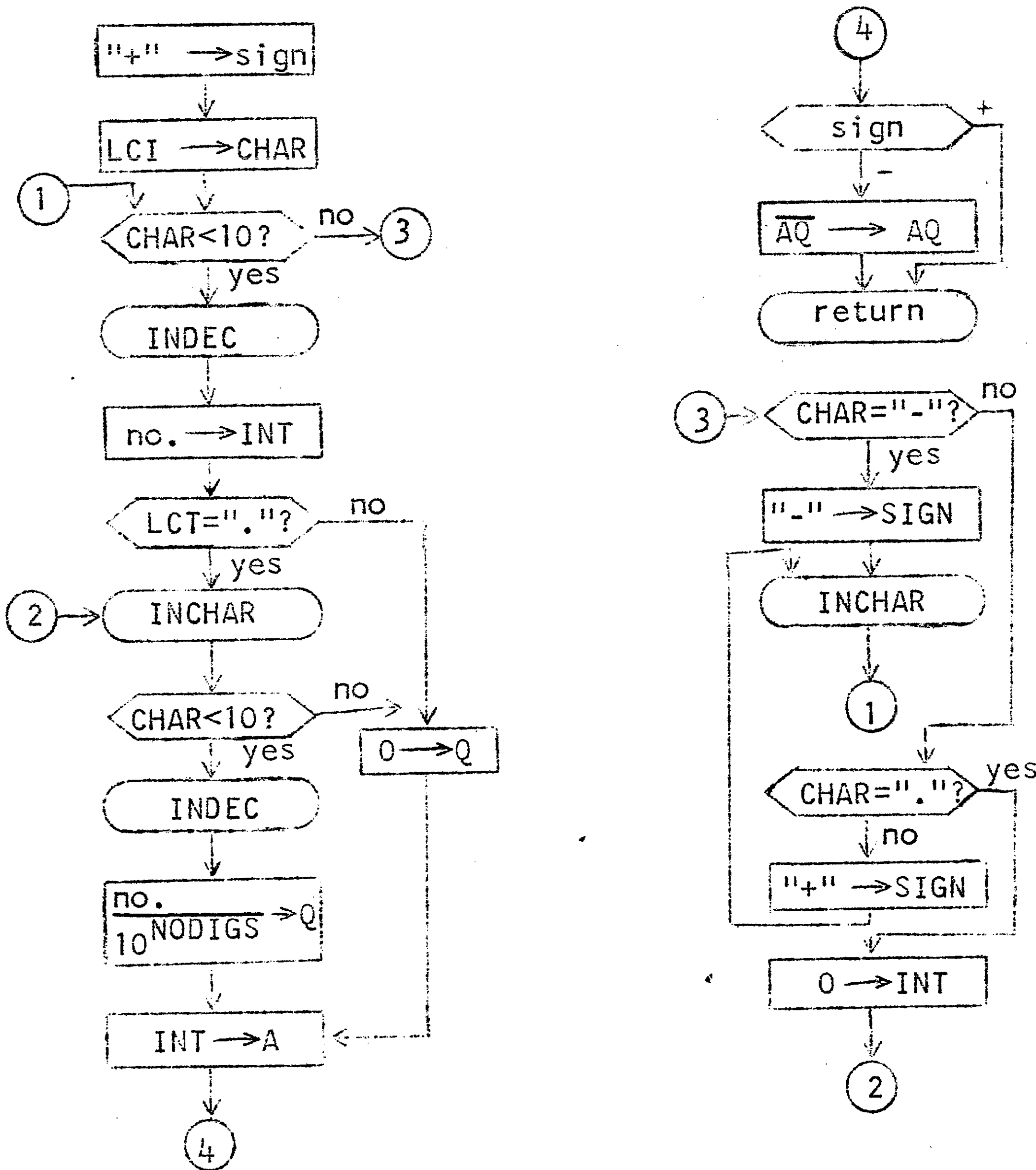
```
        EXT    LCI,INCHAR,INDEC,NODIGS

IND     UJP    **
        ENA    0
        STA    SIGN      Zero to Sign Flag.
        LDA    LCI       Pick up Last Character In.
TD      ASG    10        Is character a digit?
        UJP    DIG       Yes, go read integer part.
        ASE    40B       No.  Is it a minus sign?
        UJP    NM        No.
        ENA,S  -0        Yes, set sign flag to one's.
RD      STA    SIGN
        RTJ    INCHAR    Fetch next character.
        UJP    TD        Go and test it.
NM      ASE    33B       Is character a point?
        UJP    NP        No.
        ENA    0         Yes, put zero in
        STA    INT         integer part.
        UJP    RF        Go look for fraction part.
NP      ENA    0         Set sign flag to zero.
        UJP    RD
DIG     RTJ    INDEC     Read integer part
        STQ    INT         and store it.
        LDA    LCI       Check Last Char In.
        ASE    33B       Is it a point?
        UJP    NF        No, there is no fraction part.
RF      RTJ    INCHAR    Yes, read next character.
        ASG    10        Is it a digit?
        UJP    FR        Yes, go read fraction part.
NF      ENQ    0         No, zero to fraction part (Q).
        UJP    ITA
FR      RTJ    INDEC     Read fraction part
        STAQ   FRAC+1      and store it.
        LDA    NODIGS    Get no. of digits in fraction.
        SHA    1         Multiply by 2.
        ADA    DV        Add to divide instruction.
        STA    DIV       Store modified divide instr.
        LDAQ   FRAC+2    Put fraction part
        AQE                in (AQE) FOR
        LDAQ   FRAC        division.
```

(Input subroutine, cont.)

```
DIV  00                       Divide by 10↑NODIGS
ITA  LDA   INT                Pick up integer part.
     SSH   SIGN               Test sign flag.
     UJP   IND                Flag=0, number is positive.
     XOA,S -0                 Flag=1, change
     XOQ,S -0                   sign of number.
     UJP   IND                Return.
DV   DVAQ  PT-2               Divide instruction to be modified.
PT   DECD  10,100,1000,1D4        Table of powers
     DECD  1D5,1D6,1D7               of ten.
SIGN BSS   1                  Sign flag.
INT  BSS   1                  Storage for integer.
FRAC OCT   0,0,0,0            Storage for fraction.
```

FLOW CHART



## Floating Point Arithmetic

"Floating Point" is the name given to a method of doing calculations in a computer, which provides for a wide range of values of numbers with a reasonable amount of accuracy, and eliminates problems of shifting, etc., which are present in fixed point arithmetic. Floating point is analogous to the "scientific notation" which is often used in dealing with very large and very small numbers. For example, one writes $3.708*10^{20}$ instead of writing a 21-digit number. In a binary computer, it is more convenient to use a power of 2 as a factor instead of a power of 10. There are various ways to represent floating point numbers in computers. Here we shall discuss the scheme used in the CDC 3300, which is a fairly typical method.

In the 3300, a non-zero number $x$ is represented in floating point by finding a fraction $f$ and an integer $p$ such that $x = f * 2^p$, and $1/2 \le |f| < 1$. If p is in the range $-1023 \le p \le +1023$, the number can be represented in 3300 floating point. If not, the number is too large (overflow) or too small (underflow) to be represented. This means that $x$ itself must be in the (approximate) range $10^{-308} < |x| < 10^{308}$. A special case is the number 0, which is represented by taking $f = 0$ and $p = -1023$, which turns out to be "all 0" in the machine representation.

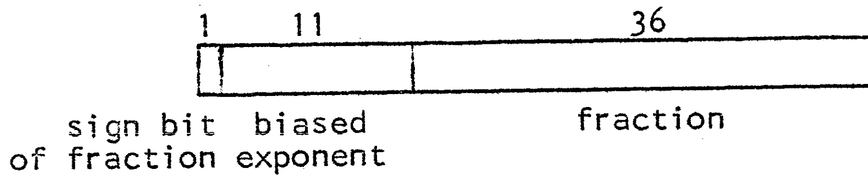
The machine representation of a floating point number in the 3300 occupies 2 words, of which 11 bits are used for the exponent (p) and 37 bits are used for the fraction (f). One problem is that both $f$ and $p$ need a "sign bit", since either may be positive or negative. This problem is solved by representing $f$ in usual one's complement notation, and by "biasing" $p$ so as to make it appear positive. If $p$ is negative, it is biased by adding $1777_8$ to it; if it is non-negative, the bias is $2000_8$. Here is a chart showing how various values of $p$ are represented:

| true exponent (decimal) | true exponent (octal) | biased exponent (octal) | true exponent (decimal) | true exponent (octal) | biased exponent (octal) |
|---|---|---|---|---|---|
| +1023 | +1777 | 3777 | - 0 | -0000 | 1777 * |
| +1022 | +1776 | 3776 | - 1 | -0001 | 1776 |
| ... | ... | ... | ... | ... | ... |
| + 2 | +0002 | 2002 | -1022 | -1776 | 0001 |
| + 1 | +0001 | 2001 | -1023 | -1777 | 0000 |
| + 0 | +0000 | 2000 | | | |

* see next page

* (From previous page) A biased exponent of "minus zero" (1777) is handled correctly by the machine's circuits, but is not generated as a result of a floating point operation. That is, a zero exponent will always come out "plus zero" (2000).

As can be seen, the biased exponents range from 0000 (representing -1023) to 3777 (representing +1023), with 1777 not used. It thus requires 11 bits to represent the exponent. These 11 bits are placed between the sign bit of the fraction and the other 36 bits, as shown below:




A positive number has a "0" in the sign bit, the biased exponent as described previously, and the 36 bits denoting the magnitude of the fraction. A negative number is the one's complement of the positive number of equal magnitude. That is, in changing the sign of a floating point number, all 48 bits (including the exponent) are complemented. Here are some example:

$+1 = .1_2 * 2^1 = .100_2 * 2^1 = .4_8 * 2^1 = 2001400000000000_8$

$+2.375 = 10.011_2 = .10011_2 * 2^2 = .100110_2 * 2^2 = .46_8 * 2^2$

$= 2002460000000000$

$+.1875 = .0011_2 = .110_2 * 2^{-2} = .6_8 * 2^{-2} = 1775600000000000$

$-1 = 5776377777777777$    $-2.375 = 5775317777777777$

$-.1875 = 6002177777777777$    $.5 = 2000400000000000$    $.25$

$= 1776400000000000$

As a result of this manner of representing floating point numbers, one can test a floating point number for positive, negative, non-zero, or zero by testing the upper half of it (AZJ,LT for example). If one "looks" at floating point numbers as if they were fixed point, larger numbers look larger than smaller ones.

Arithmetic with floating point numbers requires special actions. This can be done either by subroutines (software) or by special circuits (hardware). In the 3300, floating point hardware is available at extra cost (the OSU machine has this hardware). When this is present, one can use the floating point instructions (FAD, FSB, FMU, FDV) to operate on floating point numbers. These instructions all operate on two floating point numbers, one of which is in AQ, the other being in M and M+1 (two consecutive words in memory). The result is a floating point number in AQ. All four instructions use the E register. Indirect addressing and index modification can be used, if desired.

```
FAD M  (AQ) + (M,M+1) --> (AQ)     FMU M  (AQ)*(M,M+1) --> (AQ)

FSB M  (AQ) - (M,M+1) --> (AQ)     FDV M  (AQ)/(M,M+1) --> (AQ)
```

In working with floating point numbers, one uses LDAQ to load a number, LCAQ to load the negative (complement) of a number, and STAQ to store a number. One can use [XOA,S -0 ; XOQ,S -0] to change the sign of a number in AQ. To compare two floating point numbers X and Y , one can do LDAQ X ; FSB Y ; AZJ,LT .. (or AZJ,NE , AZJ,GE , or AZJ,EQ).

Constants. One can use the DECD pseudo-instruction to cause COMPASS to assemble floating point constants and place them in a program. Such constants must contain a decimal point. A power of ten scale factor (D) may be used, but not a binary factor (B). For example, to have the constants 3, 400, 5.87, -.0042, -12, -83.5, $-4*10^{-15}$, and $3.09*10^{32}$ stored in a program, we can write:

```
DECD    3.,4.D2,5.87,-.0042,-12.,-83.5,-4.D-15,3.09D32
```

Each floating point constant will occupy two words and they will be stored in successive loacations in storage.

Square root. Here is a square root subroutine for floating point numbers. It uses Newton's method, the same as the fixed point subroutine (page 20). The calling sequence is:

Floating point number (Y) --> (AQ)

RTJ FSQRT

Floating point result ($\sqrt{Y}$) in (AQ)

If Y is negative, the subroutine returns with the original number Y in AQ.

```
FSQRT   UJP      **
        AZJ,LT   FSQRT      If Y < 0, return.
        AZJ,EQ   FSQRT      If Y = 0, we are done.
        STAQ     FY         Store Y.
        SHA      -1         Shift right 1 and add 1/2 bias
        ADA      HBIAS       to divide exponent by 2.
        SSA      FB         Set left-most fraction bit to 1.
                            This is our initial approximation.
FLOOP   STAQ     XOLD       Set XOLD.
        LDAQ     FY         Fetch Y.
        FDV      XOLD       Divide by quotient.
        FAD      XOLD       Add XOLD to quotient.
        FDV      =2D2.0     Divide by 2.
        STAQ     XNEW       Store new approximation.
        SBAQ     XOLD       Subtract XOLD for comparison.
        AZJ,NE   NOTYET     If XNEW and XOLD are not almost
        SHAQ     22          the same, do another iteration.
        AZJ,EQ   DONE       If they are, we are done.
NOTYET  LDAQ     XNEW       Pick up XNEW and
        UJP      FLOOP       repeat loop.
DONE    LDAQ     XNEW       Pick up answer (XNEW)
        UJP      FSQRT       and return.

HBIAS   OCT      10000000   Half of bias (2000).
FB      OCT      4000       Left-most bit of fraction.
FY      BSS      2
XOLD    BSS      2
XNEW    BSS      2
```

<u>Float and Fix</u>. Here are subroutines for converting 24:24 fixed point numbers to floating point and vice-versa. The calling sequences are:

| | |
|---|---|
| 24:24 number →(AQ) | Float. pt. number →(AQ) |
| RTJ FLOAT | RTJ FIX |
| Float. pt. equiv. in (AQ) | 24:24 equiv. in (AQ) |

The FIX subroutine does not check for overflow or underflow; it will give an erroneous result if the floating point number is too large or too small to be represented in 24:24 form.

```
FLOAT   UJP     **
        STI     RB1,1       Save B1.
        SCAQ    23,1        Scale number.  23-(no. of shifts)-->B1.
        AZJ,EQ  RB1         If number is zero, we are done.
        SHAQ    -11         Position fraction.
        STA     X           Save upper half.
        TIA     1           B1-->A.  (True exponent.)
        INA     2000B       Add bias.
        ANA     3777B       Eliminate stray bits.
        SHA     12          Shift to position exponent.
        SCA     X           Assemble number.  Exponent is
                             complemented if number is negative.
RB1     ENI     **,1        Restore B1.
        UJP     FLOAT       Return.

FIX     UJP     **
        AZJ,EQ  FIX         If number is zero, return.
        STI     RX1,1       Save B1.
        STA     S           Remember sign.
        AZJ,GE  PLUS        If number is not positive,
        XOA,S   -0           complement it to
        XOQ,S   -0           make it positive.
PLUS    STA     X           Save upper half of number.
        SHA     -12         Shift exponent to right end of A.
        INA     76000B      Remove bias to get true exponent.
        TAI     1           Put it in B1.
        LDA     X           Pick up upper half of number.
        ANA     7777B       Wipe out the exponent.
        SHAQ    -12,1       Shift [(true exponent)-12] places
                             to position number.
        SSH     S           Test sign of original number.
        UJP     RX1         It was +, we are done.
        XOA,S   -0          It was -, we have to
                             complement the number.
RX1     ENI     **,1        Restore B1.
        UJP     FIX         Return.

X       BSS     1           Storage.
S       BSS     1              "
```