IBM

Reference Manual

7030 Data Processing System

Programming Examples

IBM Reference Manual

7030 Data Processing System Programming Examples

CONTENTS

FOREWORD

The following programming examples are intended to illustrate the use of 7030 instructions as active tools in problem solving. It is believed that the serious reader, equipped with the 7030 Reference Manual (A22-6530) and a description of the STRAP assembler (say the Reference Manual, 704-709-7090 Programming Package for the IBM 7030 Data Processing System (C22-6531)), can obtain a dynamic knowledge of 7030 programming without extensive outside help.

Experience in computer programming, while certainly an asset, is not taken for granted.

The subject matter is divided into four main sections:

1. Instruction Arithmetic Unit Instructions,
2. Variable Field Length Instructions,
3. Floating-Point Arithmetic,
4. Special Problems.

No attempt is made to cover the entire instruction set, to define every term or to explain every programming step. There are however, a number of comments to assist the reader over rough spots or points of ambiguity. Frequently programming alternatives are brought to the attention of the reader to emphasize the fact that there are many ways of doing the same problem. Efficiency in computer problem solving involves the balancing of the following factors:

1. Accuracy of results,
2. Analysis effort,
3. Programming time,
4. Debugging time,
5. Production run time,
6. Effectiveness in repeated use of program (possibly by a stranger).

The relative merits of these factors vary from problem to problem, individual to individual and organization to organization.

In the design of the programming examples a seventh factor, pedagogical value, has received the primary stress, and no claim is made for efficiency in terms of the other six.

1 INSTRUCTION ARITHMETIC UNIT INSTRUCTIONS

PROBLEM 1.1 TRANSMITTAL OF TWO FULL WORDS

Copy the contents of full words located in DOG, DOG +1.0 into full words located in CAT, CAT + 1.0 respectively.

Method 1. Use the immediate transmit instructions.

```
TI, 2, DOG, CAT
      or
TBI, 2, DOG +1.0, CAT + 1.0
```

Comments: (a) No more than 16 full words can be transmitted by TI or TBI. If 16 words are to be transmitted the J fields could be filled by either 16 or 0 in STRAP coding. (b) If the "source" and "sink" areas overlap, to insure that all the source words are transmitted properly, use TBI if CAT $>$DOG; use TI if CAT $<$ DOG. In the following we shall assume no overlap.

Method 2. Use an index register to control the number of words transmitted.

```
LCI, $1, 2.0
T, $1, DOG, CAT
```

Comments: (a) As many as 2^{18} (262,144) words can be specified this way. (b) The programmer should be cautioned that direct transmit type operations with J field referring to an index register with a zero count field means the maximum count possible.

Method 3. Use index instructions.

```
LX, $1, DOG
SX, $1, CAT
LX, $1, DOG + 1.0
SX, $1, CAT + 1.0
```

Comments: (a) Although data transmission is not the primary function of index registers, the two "unused" bits (bits 27 and 28) of each index register have been made available for this. (b) Two other ways are available: VFL load-store type operations and floating point (unnormalized) LWF-store. The latter is efficient but may turn on the $XPFP indicator. Further a "minus zero" exponent will be changed into a "plus zero" exponent.

PROBLEM 1.2 INTERCHANGE OF TWO WORD-PAIRS

Interchange the contents of full words DOG, DOG + 1.0 with full words CAT, CAT + 1.0.

Method 1. Use immediate swap instructions.

```
SWAPI, 2, DOG, CAT
      or
SWAPBI, 2, DOG + 1.0, CAT + 1.0
```

Comments: (a) The swapping of each word-pair involves two memory fetches followed by two stored into the "fetch" locations. (b) The J field in swap instructions is treated in exactly the same way as in transmit instructions.

Method 2. Use index instructions.

```
LX, $0, DOG
LX, $1, DOG + 1.0
LX, $2, CAT
LX, $3, CAT + 1.0
SX, $0, CAT
SX, $1, CAT + 1.0
SX, $2, DOG
SX, $3, DOG + 1.0
```

Comments: (a) Extensive use of this type of coding is clearly limited by the entailing tedium. Other alternatives are again, VFL and floating-point LWF-stores. (b) $0 may be used for any index purpose except address modification and progressive indexing. In address modification a zero I field specifies no modification.

PROBLEM 1.3 CYCLIC PERMUTATION OF A GROUP OF FULL WORDS

Given quantities A, B, C, D, E, F, G, H, I, in full words DOG through DOG + 8.0. Cyclically permute the information such that the new contents will be in the sequence DEFGHIABC.

Method 1.
```
TI, 3, DOG, 17.0          'store A,B,C, in $1,$2,$3, respectively
TI, 6, DOG + 3.0, DOG     'DEFGHIGHI
TI, 3, 17.0, DOG + 6.0    'DEFGHIABC
```

Method 2.
```
SWAPI, 8, DOG, DOG + 1.0  'cyclic left shift one unit
SWAPI, 8, DOG, DOG + 1.0  'shift another unit
SWAPI, 8, DOG, DOG + 1.0  'complete the 3 unit cyclic left shift
```

Method 3.
```
SWAPI, 3, DOG, DOG + 6.0  'place GHIDEFABC
SWAPI, 3, DOG, DOG + 3.0  'complete the permutation
```

Method 4.
```
SWAPI, 6, DOG, DOG + 3.0
```

Comments: (a) In order to permute N consecutive full words (say DOG through DOG + N-1), cyclically left K places, if K is a divisor of N, the single instruction

```
SWAPI, N-K, DOG, DOG + K
```

is adequate. If on the other hand N-K is a divisor of N, the situation is equivalent to that of cyclically permuting right N-K places, and a backward swap may be used:

```
SWAPBI, K, DOG + N-K-1        'N-K divides N
```

If neither K or N-K is a divisor of N, no single swap instruction will suffice.

Needless to say, if the number of full words to be swapped exceeds 16, the immediate swap instructions should be replaced by equivalent direct swap instructions.

PROBLEM 1.4 REPLACEMENT OF FULL WORDS BY ZEROS

Replace the contents of full words DOG through DOG + 24.0 with zeros.

Method 1. Set up a small loop using CB+ instructions.

```
        LX, $3, XW3
A       Z, DOG($3)
        CB+, $3, A
B       BEW, B
XW3     XW, 0.0, 25, XW3
```

Comments: (a) The address field of the BEW instructions and the refill field of the index word are being used for identification purposes. While the system is "waiting," the numeric equivalent of B, being a branch address, is in the instruction counter. During and after the execution of the program, one can examine the refill field of $3 to find out the source of the index information. These identification tags can be useful debugging aids. (b) It is good practice to use a decimal point in the value field of an index word.

Method 2.

```
        LX, $3, XW3A
        Z, 0($3)
        CB-, $3, $-0.32
        BEW, $
XW3A    XW, DOG + 24.0, 25, $
```

Comments: (a) The use of $ to mean "the location of this very instruction" is an efficient symbolic programming device. Instruction insertion and/or deletion in the vicinity of a symbolic instruction containing $, however, has to done with some care. For instance, the insertion of a half word instruction between the Z and CB- instructions without corresponding change in the CB- instruction will cause branches to this new instruction rather than to the Z instruction.

Method 3. During a transmit instruction execution, storing of the Kth "sink" word precedes the fetch of the (K+1)th "source" word. This makes the following concise program possible.

```
Z, DOG
TI, 12, DOG, DOG + 1.0
TI, 12, DOG, DOG + 13.0
```

Comments: (a) The execution sequence is:

Zeros $\longrightarrow$ C(DOG) $\longrightarrow$ C(DOG + 1.0),
C(DOG + 1.0) $\longrightarrow$ C(DOG + 2.0), etc.

C(Q) means the contents of location Q. (b) Programming convenience in this case means somewhat inefficient use of machine hardware. For instance, none of the memory fetches are really needed.

PROBLEM 1.5 REPLACEMENT OF ISOLATED FULL-WORD GROUPS BY ZEROS

Replace the following full words by zeros: DOG through DOG + 24.0, CAT through CAT + 15.0. CHICK through CHICK + 34.0

Method 1. Use chain indexing.

```
          PRNID, JOE BLOWE, DEPT. 333
          PUNID, J. BLOWE
          SLC, 1000.0
          LCI, $1, 3.0
          LX, $2, LINK 1
          Z, 0($2)
          CBR+, $2, $-0.32
          CB, $1, $-1.0
          BEW, $
LINK 1    XW, DOG, 25, LINK 2
LINK 2    XW, CAT, 16, LINK 3
LINK 3    XW, CHICK, 35, $
          END, 1000.0
```

Comments: (a) The PRNID, PUNID, SLC, and END pseudo-instructions should be included in every program intended for assembly. They are given here as an example of correct usage. (b) This is a simple demonstration of the utility of the automatic refill feature in the 7030.

Method 2. Use chain indexing and an XF to terminate the sequence.

```
          LX, $2, LINK 1
          Z, 0($2)
          CBR+, $2, $-0.32
          BZXF, $-1.0
          BEW, $
LINK 1    XW, DOG, 25, LINK 2
LINK 2    XW, CAT, 16, LINK 3A
LINK 3A   XW, CHICK, 35, $, 4
```

Comments: (a) The use of the index flag to terminate a sequence is especially important when the exact length of the indexing chain is unknown or variable. The number in the fourth subfield in LINK 3A concerns the setting of bits 25, 26, 27 of the index word. The number 4 means that only bit 25 (XF) is a 1. (b) Remember that the setting of the index flag indicator is done prior to the refill.

Method 3. Use transmit instructions.

```
          Z, DOG
          TI, 12, DOG, DOG + 1.0
          TI, 12, DOG, DOG + 13.0
          TI, 16, DOG, CAT
          TI, 12, DOG, CHICK
          TI, 12, DOG, CHICK + 12.0
          TI, 11, DOG, CHICK + 24.0
```

Method 4. Use transmit and index refill.

```
          LX, $2, XW2
          Z, CHICK
          T, $2, CHICK, CHICK + 1.0
          R, $2
          T, $2, CHICK, DOG
          TI, 16, CHICK, CAT
```

```
XW2     XW, 0.0, 34, XW2A
XW2A    XW, 0.0, 25, $
```

Comments: (a) The refill instruction operand is not limited to index registers. It is possible for example to write

```
R, XW2
```

and after its execution XW2 will have the same contents as XW2A.

PROBLEM 1.6 SUBTRACTION OF VALUE FIELDS

Subtract the value field $1 from that of $14 and put the result in the value field of $14. It is permissible to destroy $1 in the process.

Method 1. Change the sign bit of the value field of $1, then add value fields.

```
        BBN, 17.24, NEXT
NEXT    LVS, $14, $1, $14
```

Comments: (a) In the LVS instruction the index registers to be added together must all be different from each other. The J field, however, may refer to any index register. (b) A "V+, $14, 17.0" could also be used as an instruction at location NEXT. (c) The conditional branch is being used unconditionally. The computer nevertheless still makes the tentative assumption that the branch will be unsuccessful while preparing the BBN instruction. Some time is lost if the assumption proves incorrect during execution time. (d) The program above is therefore efficient if the bit 17.24 is probably zero. If this bit is probably 1, BBN should be changed to BZBN. (e) The machine preparation of the following conditional branch instructions involves the tentative assumption that the branch will not be successful:

All BB type of instructions (no exceptions)
All branches on indicator bits except the following:
- XF (11.38)
- XCZ (11.48)
- XVLZ (11.49)
- XVZ (11.50)
- XVGZ (11.51)
- XL (11.52)
- XE (11.53)
- XH (11.54)

Note that branches on index results or index register conditions do not involve tentative guesses. For example, CBRH does not behave like a true conditional branch. (f) A more efficient way is to use the connective instruction CM1100(BU, 1), 17.24 in place of the BBN instruction.

PROBLEM 1.7 INTERRUPTION MEASURE

$IA contains the address 1000.0. It is desired that when a $TS interruption occurs the instruction counter contents should be stored in the first 19 bits of location 2000.0 and the main program is to be continued. Write a code to effect this.

Method 1.

```
        SLC, 1000.0 + 4.0
TSFIX   SIC, 2000.0; BR, 0
```

Comments: (a) The SLC pseudo-instruction indicates the instruction TSFIX is to start at 1004.0. Since $TS is bit position 4 of the indicator register, a $TS interruption will lead to an automatic execution of the free instruction at C($IA) + 4.0 = 1004.0. (b) The instruction counter is not changed during the execution of the "free instruction," hence the "branch relative to zero" instruction will return to the main program. (c) The interrupt system is not disabled during the execution of the "free instruction." In fact during the interruption only the $IF monitoring is relaxed temporarily to allow the fetching of the "free instruction." (d) The SIC action is not performed unless the ensuing branch is successful, and even then it is performed after the execution of the branch. Instructions such as SIC, $+0.32; B, ANYWH will lead to a branch to ANYWH if the branch is executed. The instruction counter will not have time to alter the branch address before execution.

PROBLEM 1.8 SIMULATION OF RENAME INSTRUCTION

Create the effect of the instruction RNX, $1, DOG ($3). Do not simulate the indicator settings.

Method 1.

```
RNAME   SX, $2, X2              'save $2
        SR, $0, 18.0
STOX    SX, $1, 0($2)
        LX, $2, X2              'restore $2
        LVE, $1, LOX
        LR, $0, 17.0
LOX     LX, $1, DOG($3)
        BEW, $
X2      XW, 0
```

Comments: (a) It would seem that the SR instruction could be altered such that the refill field of $0 is stored directly into the address field of STOX, and the use of $2 would be avoided. This is not possible because in the SR operation of the refill field concerned is right appended by zero bits to create a 25 bit value field. The latter is then stored. The STOX instruction would be seriously altered if a direct SR operation is used.

PROBLEM 1.9 TRANSPOSITION OF A SQUARE MATRIX WITH FULL WORD ELEMENTS

An N x N matrix has full word elements and is stored row-wise beginning at LOC. Create the transpose of this matrix and store it in the same area.

Method 1. Interchange rows and columns starting from the north and west borders of the matrix.

```
TPOSE   LX, $2, XW2; SX, $2, XW22
        LX, $3, XW3; SX, $3, XW33
SWAPI   SWAPI, 1, 0($2), 0($3)
        V+ICR, $3, N
        CBR+, $2, SWAPI
        V+IC, $2, N+1.; SX, $2, XW22
        V+IC, $3, N+1.; SX, $3, XW33
        BZXCZ, SWAPI
```

```
          BEW, $
XW2       XW, LOC + 1., N-1, XW22
XW3       XW, LOC + N, N-1, XW33
XW22      XW, 0
XW33      XW, 0
N         SYN, 100.0                    'if 100 x 100 matrix
LOC       SYN, 32768.0                  'if matrix starts at 32768.0
```

Comments: (a) The program is written in such a way as to be reusable. Otherwise the temporary index word storages XW22 and XW33 could be omitted by a slight change of the program. (b) Relatively error-free instructions can be packed together in the same line. This enables the programmer to focus attention on the rest of the program during debugging.

Method 2. Start from the upper and lower co-diagonals of the matrix and proceed through the exchange of the northeast-most and the southwest-most elements.

```
TPOSE2    LX, $2, XW2; SX, $2, XW22
          LX, $3, XW3; SX, $3, XW33
SWAPI     SWAPI, 1, 0($2), 0($3)
          V+ICR, $2, N+1.
          V+ICR, $3, N+1.
          BZXCZ, SWAPI
          V+IC, $2, 1.0; SX, $2, XW22
          V+IC, $3, N; SX, $3, XW33
          BZXCZ, SWAPI
          BEW, $
XW2       XW, LOC+1., N-1, XW22
XW3       XW, LOC+N, N-1, XW33
XW22      XW, 0
XW33      XW, 0
N         SYN, 100.                     'size of matrix
LOC       SYN, 32768.0                  'starting location
```

2 VARIABLE FIELD LENGTH INSTRUCTIONS

PROBLEM 2.1 CYCLIC BIT SHIFTING

Cyclic left shift a full word in DOG by 7 bit positions.

Method 1

```
        L(BU, 64-7), DOG+.7, 7          'leave room for DOG thru DOG +0.6
        +(BU, 7), DOG
        ST(BU,64), DOG
```

PROBLEM 2.2 LENGTH OF AN UNKNOWN FILE

Information of unknown length is written in consecutive 7-bit bytes beginning at INFO. Its end is signified by the first appearance of a special character consisting of seven binary 1's. Write a program to find the file length (including the special character) in bits, and put the answer in the value field of $1.

Method 1. Byte-by-byte Compare.

```
        LVI, $1, 0.0
LOAD    L(BU, 7, 8), INFO ($1)
        K(BU, 7,8), ENDB
        V+, $1, SEVN
        BZAL, $+1.0
        B, LOAD
        BEW, $
ENDB    DD(BU, 7,8), (2)1111111         'or decimal 127
SEVN    VF, 0.07
```

Comments: (a) The use of a number in its own natural radix is convenient and can be a powerful aid in debugging.

Method 2. Put end byte in $R with the compare and use progressive indexing.

```
        LV, $1, VFIELD
        LI(BU,7), 127                   'or (2)1111111
COMP    K(BU, 7) (V+I), 0.07($1)
        BAE, $+1.0
        B, COMP
        L(BU, 7) (V-I), INFO($1)
        BEW, $
VFIELD  VF, INFO
```

Comments: (a) The last VFL instruction serves mainly to perform the (V-I) operation, for an alternative technique see Method 3. (b) Except for logical connectives, the result of a binary unsigned operation is independent of the byte size. The 7030 actually

uses a byte size of 8 internally for speed. This is true even for the numeric portion of a binary signed operation.

The byte size specification in an instruction is therefore important only for (1) decimal operations (signed or unsigned), (2) sign byte in binary signed operations and (3) logical connectives.

Unless specified otherwise, STRAP assumes a byte size of 8 for all binary unsigned and logical operations, a byte size of 1 for all binary signed operations and a byte size of 4 for all decimal operations.

(c) A numeric bit address is signified by the appearance of a "point" (whatever the radix). A number in the address field without the "point" is said to be an integer address. The latter is acceptable to STRAP, but STRAP must translate it into the equivalent numeric bit address before the program can be executed directly by the machine.

The bit address equivalent of an integer address is determined by the environment, which defines a subfield. The integer address is treated as an integer of the subfield (e.g., the non-zero bit of the integer 1 would occupy the rightmost position), then the left margin of the subfield is placed in juxtaposition with the leading bit of the address field, leading to a bit-address identification.

Where the environment seems to suggest more than one subfield, the smallest subfield is to be used.

A VFL instruction normally implies a subfield of 24 bits. In the second instruction of the present program, the "immediate" nature, plus the field length suggests a smaller (7-bit) subfield. The latter is adopted during the STRAP assembly as the defining subfield, and the bit address equivalent is therefore:

$$0.\ (127*2^{-7}*2^{24}) = 0.\ (127*2^{17})$$

$$= (127*2^{11}).0 = 260096.0$$

The convenience entailed by the use of integer addresses is apparent: 260096.0 is not only difficult to obtain, but does not contribute to understanding.

Method 3. Use connective and branch on $RZ.

```
        LVE, $1, VF1
        LI(BU,7), (2)1111111
CONT    CT0110(V+I) (BU, 7), 0.07($1)
        BRZ, $+1.0
        B, CONT
        V+, $1, VF1
        BEW, $
VF1     SIC, INFO
```

Comments: (a) The LVE instruction loads the magnitude of the dummy SIC instruction. (b) CT0110 will lead to $RZ=1 if the memory field and the accumulator field are equal. In reality the 7-bit memory field is left-appended with a zero bit and is connected with eight bits left of the offset. (c) The V+,$1, VF 1 instruction in

reality performs a subtract since bit 24 of the SIC instruction is a 1. (d) The progressive indexing secondary operation can precede the (dds).

PROBLEM 2.3 DELETION OF EVERY FIFTH BIT IN A FIELD

Given a string of 60 bits starting at FIELD, delete every fifth bit starting at FIELD +0.04 and put the 48 bit result consecutively starting at FIEL. Assume that there is no overlap between (FIELD - FIELD +0.59) and (FIEL - FIEL +0.47).

Method 1. Load 5 signed bits and store 4 unsigned bits at a time.

```
        LV, $2, VFIELD
        LX, $3, VFIEL
LOAD    L(B, 5, 1) (V+I), 0.05($2)
        ST(BU,4) (V+IC), 0.04($3)
        BZXCZ, LOAD
VFIELD  VF, FIELD
VFIEL   XW, FIEL, 12, $
```

Comments: (a) BZXCZ is not considered to be a conditional branch instruction since the instruction arithmetic unit knows the index conditions during decoding time.

Method 2. Load 5 unsigned bits and store 4 bits with offset 1.

```
        LV, $2, VFIELD
        LX, $3, VFIEL
LOADA   L(BU, 5) (V+I), 0.05($2)
        ST(BU, 4) (V+IC), 0.04($3), 1
        BZXCZ, $-1.0
        BEW, $
VFIELD  VF, FIELD
VFIEL   XW, FIEL, 12, $
```

Method 3. Other variations of the same theme. Instead of LOADA and LOADA +1.0 above, one may write any of the following instruction pairs:

```
L(BU, 4) (V+I), 0.05($2)
ST(BU, 4) (V+IC), 0.04($3)
        or
L(B, 5, 2) (V+I), 0.05($2)
ST(B, 4,1) (V+IC), 0.04($3)
        or
LWF(B,5,4) (V+I), 0.05($2)
ST(B,4,3) (V+I), 0.04($3)
```

Method 4. Remembering decimal information is processed in the accumulator in 4-bit bytes, it is possible to write just two instructions to solve this problem under restrictions stated below. The decimal load operation behaves like a decimal "add to zero" operation.

```
L(DU, 60, 5), FIELD -0.01
ST(BU, 48), FIEL
        or
LWF(D, 60, 5), FIELD -0.01
ST(B, 48, 4), FIEL
```

Comments: (a) The lead bits in the 5-bit bytes are deleted to five 4-bit bytes. (b) In the decimal load the 4-bit bytes will not be altered if they contain what appears to be decimal information. Otherwise, carry propagation and assimilation will occur. The byte $(1\ 1\ 1\ 1)_2$, for instance, will become $(0101)_2$ with a carry to the higher byte. (c) The method fails if FIELD -0.01 happens to be in a protected memory area. To avoid this difficulty, use say, L(DU, 59, 5), FIELD instead.

```
Method 5.   LX, $1, XW1; LVI, $2, 56
            L(BU, 60), FIELD
STORE       ST(BU, 4) (V+I), 0.04($1), 0($2)
            V-I, $2, 5
            BZXVLZ, STORE
            BEW, $
XW1         XW, FIEL, 0, $
```

Comments: (a) The integer 5 in the V-I instruction means 5 units in the 19 bit address subfield of the instruction half-word. (b) This is an example of offset indexing.

```
Method 6.   Use logical connectives
            C0011 (BU, 60, 5), FIELD            'LF
            CM0101 (BU, 48, 4), FIEL, 1         'SF
```

Comments: (a) The accumulator always uses 8-bit bytes. Each memory byte is left-appended by enough zeros to become 8-bit bytes for the connect operation. In the LF operation true memory bytes are expanded to 8-bit bytes; in the SF operation the 8-bit bytes are truncated to the specified byte size (in the dds). (b) For operations Cabcd, CMabcd, CTabcd (abcd can be any combination of 0's and 1's) the result of the operation can be seen from the truth table:

m \ a	0	1
0	a	b
1	c	d

Cabcd: result goes to the accumulator
CMabcd: result goes to memory
CTabcd: result discarded

Where m refers to a memory bit and a refers to an accumulator bit.
If m=1 and a=0, for instance, the result would be c. If the instructions for this case was C0010, c equals 1.

(c) Valuable by-products of the connective operations are, among others;

```
$RZ "Is the result zero? Or, does the result contain no ones?"
$AOC "How many ones are there in the result?"
$LZC "Where is the leading one bit?"
```

The CTabcd operation allows the user to examine these by-products without affecting the accumulator or the memory. (d) The only acceptable entry mode for connective operations is BU. B, D, and DU are considered illegal by the STRAP assembler.

PROBLEM 2.4 BIT REVERSAL

The 64-bit full word starting at WORD contains a binary message which would be easily interpretable when every bit in the word is reversed (WORD +0.63 becomes WORD +0.0, etc.). Perform the bit reversal and put the result in DROW.

Method 1. Load the entire word and store a bit at a time.

```
            TI, 1, WORD, $R
            LX, $1, XW1
            LX, $2, XW2
STOR        ST(BU,1) (V+I), 0.1($1), 0($2)
            CBH, $2, STOR
            BEW, $
XW1         XW, DROW, 0, $
XW2         XW, 0, 64, $
```

Comments: (a) An advance of a half-word in the $2 value field leads to an effective offset change of one unit.

Method 2. Load a bit at a time and store the entire word.

```
            LX, $1, XW1
            LX, $2, XW2
LO          LF(BU,1) (V+I), 0.1($1), 0($2)
            CBH, $2, LO
            ST(BU, 64), DROW
            BEW, $
XW1         XW, WORD, 0, $
XW2         XW, 0, 64, $
```

Comments: (a) VFL stores are slower than VFL loads, and the present program is to be preferred over that in Method 1.

PROBLEM 2.5. REMOVAL OF KEY WORDS

Given a string of 100 6-bit bytes beginning at DATA, remove any 4 consecutive bytes which match a given "key word" KEY. Pack the result starting at ANSW.

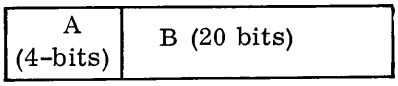
Method 1.

```
            LX, $1, XW1; LX, $2, XW2
LODE        L(BU, 24) (V+I), 0.6($1)
            K(BU,24), KEY
            BAE, AE
            ST(BU,6) (V+I), 0.6($2), 18
CAB         CB, $1, LODE
            ST(BU,18), 0($2)              'store remaining 3 bytes
            BEW, $
AE          V+, $1, X18                   'skip 3 more bytes
            C-I, $1, 3                    '3 means 3.0 here
            B, LODE
XW1         XW, DATA, 100-3, $
XW2         XW, ANSW, 0, $
X18         VF, 0.18
```

Comments: (a) The integer 3 in the C-I instruction means 3 units in a subfield of 18 bits (size of count field).

PROBLEM 2.6 SORTING ON THE BASIS OF SUBFIELDS

Given 16 consecutive fields beginning at DATA, each of the following appearance:



The four-bit subfield "A" may contain any integer number from 0 through 15. Assume all A subfields are different in content, sort on the basis of A subfields and put the correspondent B subfields together in a string beginning at ANS.

Method 1. Take advantage of the fact that there are exactly 16 A subfields and that these subfields have different contents.

```
ASORT   LX, $2, XW2
LOOP    L(BU, 4) (V+I), 0.04($2)
        *(BU, 24), VALF                 'answer at offset 20
        ST(B, 25, 1), 17.0, 20          'store into index register value field
        L(BU, 20) (V+I), 0.20($2)
        ST(BU, 20), ANS($1)
        CB, $2, LOOP
        BEW, $
XW2     XW, DATA, 16, $
VALF    VF, 20
ANS     DRZ(BU, 20), (16)
```

Comments: (a) If the A fields are not all different the stores will be incorrect.

Method 2. A slight modification of Method 1.

```
ASORT2  LX, $2, XW2A
LOOP    L(BU, 4), -0.4($2), 20 + 2
        +(BU, 4), -0.4($2), 20 + 4
        ST(B, 25, 1), 17.0, 20
        L(BU, 20) (V+IC), 0.24($2)
        ST(BU, 20), ANS($1)
        BZXCZ, LOOP
        BEW, $
XW2A    XW, DATA + 0.4, 16, $
ANS     DRZ(BU, 20), (16)
```

Comments: (a) The multiplication by 20 is replaced by judicious placement of data in the load and add operations. (b) The following sets of instructions lead to the same results, and other variations are possible.

```
($2 has X in value field)
L(BU, 4) (V+I), 0.04($2)
  .
  .
  .
L(BU,20)(V+I), 0.20 ($)
ST(BU,20), ANS($1)
CB, $2, LOOP
```

```
($2 has X in value field)
L(BU, 4) (V+I), 0.24($2)
  .
  .
  .
L(BU,20), -0.20($2)
ST(BU,20), ANS($1)
CB, $2, LOOP
```

```
($2 has X + 0.4 in value field)
L(BU, 4), -0.04($2)
  .
  .
  .
L(BU,20) (V+IC), 0.24($2)
ST(BU,20), ANS($1)
BZXCZ, LOOP
```

(c) A negative numeric address is assembled by STRAP as its two's complement, thus, -A will be assembled as 2**18-A.

Method 3. Repeated compares for minimum

```
            LX, $1, XW1; LX, $2, XW2
            B, STIX
VPC         V+C, $1, VF1                    'outer loop, restart with changed $1
STIX        SX, $1, XW11                    'save $1 contents for later refill use
            L(BU,24), 0($2)                 'load assumed minimum
KOMP        K(BU,4) (V+ICR), 0.24($1), 20   'inner loop, test against assumed min
            BAH, FIXMIN                     'usually unsuccessful
AGAIN       BZXCZ, KOMP
            SF(BU,24)(V+IC),0.24($2)        'store proven minimum
            BZXCZ, VPC
            LX, $1, XW1A; LV, $2, VF2
PACK        L(BU,20)(V+I), 0.24($2)         'skip A field
            ST(BU,20) (V+IC), 0.20 ($1)     'store sorted B field
            BZXCZ, PACK
            BEW, $
FIXMIN      LF(BU,24), -0.24($1), 24        'fixup routine, load new minimum
            SF(BU,24), -0.24($1)            'store old guess in its place
            LF(BU,24), 9.16                 'position new min. in accumulator
            B, AGAIN                        'return to inner loop
XW1         XW, DATA +0.24, 15, XW11
XW11        XW, 0                           'will be changed during computation
XW2         XW, DATA, 14, $
XW1A        XW, ANS, 16, $
VF1         VF, 0.24
VF2         VF, DATA +0.04
ANS         DRZ(BU, 20), (16)
```

Comments: (a) This method applies even if all the A fields are not different in content. (b) The original information will be permuted in the program. If this is deemed undesirable, one could transmit the information to a temporary area and do the permutation there, leaving the original information unaltered. (c) The code is written under the reasonable assumption that the provisional minimum stands a good chance of being no larger than an average entry. (d) For the sake of clarity the packing of the sorted fields is done separately at the end. By using an extra index register this packing action can be performed whenever a new proven minimum is found.

Method 4. Repeated compares for both maximum and minimum.

```
            LX, $1, XW1
            LX, $2, XW2
            LX, $3, XW3; SX, $3, XW33
LODE        L(BU,24), 0($1)
            LF(BU,24), 0($2), 64
            KF(BU,4), 0($2), 20
            BAH, SWICH
TEST        KF(BU,4)(V+ICR), 0.24($3),20    'test against assumed minimum
            BAH, FIXMIN
```

```
            KF(BU,4), -.24($3), 64+20          'test against assumed maximum
            BAL, FIXMAX
AGAIN       BZXCZ, $3, TEST
            ST(BU,24) (V+I), 0.24($1)          'store minimum
            ST(BU,24)(V-I), 0.24($2), 64       'store maximum
            V+C, $3, VF3; CB, $3, LODE-1
PACK        LV, $1, XW11;LX, $2, XW22
LOAD2       L(BU, 24)(V+I), 0.24($1)
            ST(BU,20)(V+IC), 0.20($2)
            BZXCZ, LOAD2
            BEW, $
SWICH       SWAPI, 1, $L, $R
            B, TEST
FIXMIN      LF(BU, 24), -0.24($3), 24
            ST(BU, 24), -0.24($3), 64
            ST(BU, 24), 9.40, 24               'new minimum
            B, AGAIN
FIXMAX      LF(BU, 24), -0.24($3), 64+24
            ST(BU, 24), -0.24($3), 64
            ST(BU,24), 8.40, 64+24             'new maximum
            B, AGAIN
XW1         XW, DATA, 16, $
XW2         XW, DATA +0.360, 0, $
XW3         XW, DATA + 0.24, 14, XW33
XW33        XW, 0
XW22        XW, ANS, 16, $
XW11        VF, DATA + .4
VF3         VF, 0.24
ANS         DRZ(BU, 20), (16)
```

Comments: (a) This method applies even if the A fields are not all different in content.

PROBLEM 2.7 SORTING INTO RESERVED TABLE AREAS

Given the same field description as in Problem 3 above, as well as reserved table area beginning at TABL 0,..., TABL 15, each of which is capable of holding the entire string (in this case 400 bits). Put the proper B fields in successive entry areas of the TABL areas as dictated by the contents of the A fields. Assume the A fields are not all different.

Method 1.

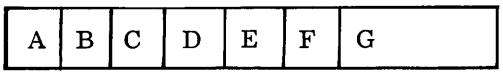
```
            LX, $2, XW2
LOAD        L(BU, 4)(V+I), 0.24($2), -18
            LX, $1, $L
            LVE, $3, MTABL($1)
            L(BU, 20), -0.20($2)
            ST(BU,20) (V+I), 0.20($3)
            SVA,$3, MTABL($1)
            CB, $2, LOAD
XW2         XW, DATA, 16, $
MTABL       SIC,TABL0;SIC, TABL1;SIC, TABL2;SIC, TABL3    'Master Table
            SIC, TABL4; SIC, TABL5;SIC, TABL6;SIC, TABL7
```

```
SIC, TABL8; SIC, TABL9; SIC, TABL10; SIC, TABL11
SIC, TABL12; SIC, TABL13; SIC, TABL14; SIC, TABL15
```

Comments: (a) The "master table" area is updated constantly to avoid conflicts in the storing of entries with equal A fields. (b) The SIC operation by itself is meaningless as an instruction. However, it specifies a 24-bit address, and this fact is noted by LVE and SVA instructions.

PROBLEM 2.8 PURCHASING LIST ARITHMETIC

A purchasing list consists of a string of fields, each of which has the following structure:



Subfield A is an eight-bit byte consisting of 1's.
Subfield B has two 8-bit bytes (item no.).
Subfield C has six 8-bit bytes (coded name of product).
Subfield D has three 8-bit bytes, and contains the number of units of the product desired in decimal (DU, 24, 8).
Subfield E has six 8-bit bytes, and contains the unit price in cents of the product in decimal (DU, 48, 8).
Subfield F has 12 8-bit bytes, and is blank (to be the total price field).
Subfield G is an unknown number of 8-bit bytes. It contains the remarks concerning the product and/or the entire purchase. The first three 8-bit bytes of the subfield G in the last "product field" contains the 8-bit IQS expression END. None of the 8-bit bytes in G are all 1's.

If the complete string begins at LIST, write a program to fill in the total price for each product in (DU, "96", 8). For simplicity of programming do repeated additions instead of decimal multiplications. Create the grand total also, and put it (DU, "128", 8) in the pseudo accumulator 13.0 through 14.0 ($RM and $FT).

Method 1.

```
        Z, $FT
LCON    LX, $2, XW2
LCON    LCV(DU, 24, 8)(V+I), 0.24+0.48($2),128-18   '$R cleared too
        LC, $1, $L; BXCZ, NEXT                      'binary count field
ADD     +(D, 48, 8),-0.48($2)
        CB, $1, ADD
NEXT    ST(DU, 64, 8) (V+I), 0.64($2), 16           'store total into F
        ST(DU, 32, 8) (V+I), 0.32($2)
        M+(DU, 64, 4), $FT                          'update grand total
        L(BU, 32), TESTW
        KF(BU, 24), 0($2),8                         'test for end of string
        BAE, LAST
KOMF    KF(BU,24)(V+I), 0.8($2)          'test for beginning of new field
        BZAE, KOMF
MORE    V+I, $2, 1.0                     'bypass 64 more bits to new D field
        B, LCON
```

```
LAST    L(DU, 64, 4), $FT
        ST(DU, 64, 8), $FT
        ST(DU, 64, 8), $RM, 64
        BEW, $
TESTW   (IQSQ)DD(DU,24,8),ENDQ          'end mark for string
        DD(BU, 8, 8), (2)11111111       'beginning mark for field
XW2     XW, LIST+0.72, 0, $
```

Comments: (a) Decimal quantities with more than one digit must be converted into binary before a binary arithmetical operation (say index count down) is attempted. (b) It is convenient to load one test quantity to be compared against many. This eliminates a number of memory fetch operations. In the present program two kinds of tests are performed, but the test quantities can be made adjacent to each other, and loaded simultaneously. Note the KF's cannot be replaced by simple K operations. (c) Special devices are used to deal with long field lengths. The 96-bit stores into F fields are performed in two separate instructions. The running grand total is kept in a packed (4-bit byte) form, to be expanded to fill both $FT and $RM at the end of the program. (d) Normally the STRAP assembler ignores all blanks in instructions and pseudo-instructions. An exception is made for pseudo-instructions specifying alphabetic characters, since "blank" is a character in its own right. In TESTW, therefore, any characters following the third comma will be assembled as bona fide characters, and the usual typographic practice of leaving a blank after a comma will prove unwise.

PROBLEM 2.9 EFFECTIVE ADDRESS CREATION

Find the effective address of the instruction beginning at the 19-bit address INST without using the LVE instruction. Put the answer in the value field of $1.

Method 1

```
EFFADR  L(BU, 32), INST
        KFI(BU,4), (2)0000              ' assume four-bit index field
        BAE, NOX                        'assume indexing needed
        ST(BU,4),SV+0.19                'store in J field of SV instruction
SV      SV, $0, 17.0                    'index value field now in $1
        B, TEST
NOX     Z, 17.0
TEST    KFI(BU, 2), (2) 10,4            'test if floating-point
        BZAE, NOTFP
FP      -(BU, 32-18), $R.32 + 0.18      'floating-point measure
MPLUS   M+(B, 25, 1), 17.0, 32-24       '25-bit add
        BEW, $
NOTFP   KFI(BU,4), (2)1000, 4           'test if VFL left address
        BZAE, NOTVFL
VFL     -(BU, 32-24), $R.32+0.24        'VFL measure
        B, MPLUS
NOTVFL  KFI(BU,3), (2)100,4             'test if K type indexing, CB, BIND
        BAE, KTYPE
        KFI(BU,9), (2)111000000,4       'test if K type indexing, BB
        BAE, KTYPE
        KFI(BU,5), (2)10000, 4          'test if immediate indexing
        BAE, IMMED
        B, MPLUS                        'otherwise 4 bit I field assumption valid
```

```
KTYPE     ST(BU, 1), KSV+0.22
KSV       SV, $0, 17.0
KMINUS    -(BU, 32-19), $R.32+0.19        '19-bit address
          B, MPLUS
IMMED     Z, 17.0
          B, KMINUS
```

Comments: (a) The effective address is the algebraic sum of the (positive) numeric address and the value field of the specified index register. In an instruction the numeric address is abbreviated into the numeric address field. The size of the numeric address field is determined by bits .24 through .27 of the instruction.

1000 means a 24-bit numeric address field;
XX10 means an 18-bit numeric address field;
otherwise a 19-bit numeric address field is meant.

The instruction may allow no indexing at all (immediate indexing instructions), may allow a one-bit K-type of indexing specification (CB, Bind, and BB) but generally allows a four-bit I-type indexing specification.

If bits 23-27 have 10000: no indexing allowed;
If bits 25-27 have 100: K-type (CB, Bind);
If bits 19-27 have 111000000: K-type (BB);
otherwise: I-type.

(b) The reader should write down the bit combination of several instructions and follow the program closely. (c) In many instances the symbolic instructions should be written for the convenience of the programmer. In the instruction FP, the field length 32-18 is evidently 14, but clarity is gained by retaining the longer expression. The same is true for the address field of TEST. The extra assembly time is trivial.

PROBLEM 2.10 FETCH (p,q)TH ELEMENT OF RECTANGULAR MATRIX

Given a matrix A of size M x N (M rows and N columns), stored row-wise in consecutive full words beginning with A_{11} in location MTRIX. Given also are binary integers p, q in the leading 18 bits of $1 and $2. Put the element A_{pq} in $R.

Method 1

```
LOCATE    V-I, $1, 1.0                    'p-1 generated
          L(BU, 18), 17.0                 '$1
          *(BU, 18), ENN                  'result has 20 offset
          ST(BU, 25), $3, 20-7            '(p-1)*N
          V+, $3, 18.0                    '(p-1)*N+q
          V+, $3, VF
          L(BU, 64), 0($3)
          BEW, $
ENN       DD(BU, 18), N                   'N is assumed defined elsewhere
VF        VF, MTRIX-1.0
```

Comments: (a) The element A_{p1} is in MTRIX+(p-1)N. The element A_{pq} is therefor in MTRIX+(p-1)N+(q-1) or MTRIX-1+(p-1)N+q. (b) After a binary VFL multiply the answer is placed in the cleared accumulator with offset 20.

PROBLEM 2.11 SIMULATION OF TWO-BIT ADDITION

If P, Q, R each define a two-bit non-overlapping field, using logical connectives only, create the lowest two bits of the sum C(P)+C(Q) and put it in $R. (C(X) means contents of X).

Method 1.

```
C0011(BU, 2,2), P
C0110(BU, 2,2), Q
CM0101(BU, 2,2), R
C0000(BU, 2,2), $Z                    'or any other address
C0011(BU,2,2), P, 1
C0001(BU,2,2), Q, 1
CM0110(BU,2,2), R
```

Comments: (a) This is actually a small-scale simulation of the parallel addition in binary digital machines.

PROBLEM 2.12 TRANSPOSITION OF RECTANGULAR MATRIX

Given an MxN matrix of floating-point words starting at location MATRX, with the elements stored row-wise. Create the transpose of the matrix, also stored row-wise, occupying the same area. Keep the number of temporary storage locations small for this purpose.

Analysis: Counting from the (1,1) element, if MATRX begins the storage area for a PxQ matrix, then we may say the location MATRX + L contains the (r, s)-element, if

$$L=(r-1)*Q + (s-1) \qquad r \leq P,\ s \leq Q.$$

The transpose of an MxN matrix is an NxM matrix. The (i,j)-element of this NxM matrix is in location, say, MATRX + K

$$K=(i-1)*M+(j-1) \qquad i \leq N,\ j \leq M$$

The contents of this location, however, has to be fetched from the original MxN matrix, the (j,i)-element. The fetch location is, say MATRX+K', with

$$K'=(j-1)*N+(i-1)$$

= integer remainder of $(K*N)/(M*N-1)$.

The algorithm is therefore to save one element (the lead element) from location MATRX+K, fill the latter with the contents of MATRX+K', then fill the latter with the contents of MATRX+K'' etc., until the fetch location is the same as that of the lead element. The last store is performed with the lead element to complete the permutation cycle. As the cycle invariably has fewer elements than the matrix itself, care must be exercised to avoid altering elements which have already been permuted. This can be cone by using flag bits as identification, at the same time insuring that the lead element of every cycle has the smallest (or alternatively largest) address possible. The method is essentially that of M. F. Berman, Journal of the Association for Computing Machinery, 5, 383 (1958). For similar techniques see P. F. Windley, Computer Journal, 2, 47-48 (1959); G. Pall and E. Seiden, Mathematics of Computation, 14, 189-192(1960).

For square matrices each of the cycles have only one (diagonal) or two (off-diagonal) elements, and there exist methods much more efficient than the present one. Rectangular matrices offer few direct hints about the nature of the cycles, except that the first and last elements are unaltered by the transposition process.

Method 1. Use V-flags for permuted elements. Assume the matrix elements do not contain V-flags originally. Advantage is taken also of $VF interruption.

```
TRANSP   BD, $+0.32                    'special interruption scheme
         LV, $1, $IA                   '$IA assumed to have meaningful value
         V+I, $1, 37.0                 'find $VF interrupt table address
         SVA, $1, SWAP2                'insert address in exit instruction
         SWAPI, 1, 0($1), INST         'swap old and new entries of interrupt table
         TI, 1, $IND+1.0,INST+1.0      'save old $MASK
         CM1111(BU,1), $IND+1.37       'force $VF mask to be 1
         BE, $+.32                     'end of interrupt measure
         LVI, $1, 0
         LI(BU, 18), M
         *I(BU,18), N
         -I(BU,18), 1, 20              'M*N-1
         ST(BU, 25), 20.0, 20-7        'at full-word position of $4 value field
         LC, $1, 20.0                  'copy into $1 count field
         CB+, $1, BZBN
         B, BEW; CNOP; NOP             'to insure CYCLE will start at full word
NUCYCL   LX, $2, 17.0
         TI, 1, MATRX($2), TEMP        'file away leading element of cycle
CYCLE    L(U), 18.0                    'location of old element
         *I(BU,18),N,128-18            'answer is at 20 offset
         /(BU,18),20.0,20              'divide by M*N-1
         L(BU,18),$RM+.60-.18,128-18   'location of new element
         LX, $3, $L
         LWF(U), MATRX($3)             'if operand has V-flag, interruption ensues
         CM1111(BU,1), $SB+0.7         'create V-flag
         ST(U), MATRX($2)              'store into vacated location
         LX, $2, 19.0                  'new address modifier
         B, CYCLE                      'endless loop dependent on $VF exit
ENDCYC   TI, 1, TEMP, MATRX ($2)       'transmit lead element of cycle. It has V-flag
BZBN     BZBN, MATRX+0.63($1),NUCYCL
         CB+, $1, BZBN
SWAP2    SWAPI, 1, 0, INST         'left-address to be inserted by SVA instruction
         TI, 1, INST+1.0, $IND+1.0
BEW      BEW, $;CNOP
INST     B, ENDCYC; NOP              'must begin at full word boundary, and does
TEMP     DRZ(N), 1
MATRX    SYN(BU,24), 1000.0            'user specified starting address
M        SYN, 20                       'user specified, no. of rows
N        SYN, 5                        'user specified, no. of columns
```

Comments: (a) To avoid conflicts, all but the leading members of each permutation cycle are given a V-flag during the permutation, and the end of cycle is sensed by the fetching of an element already with a V-flag. The BZBN instruction tests elements of the entire matrix proceeding from the lowest addresses. If an element has a V-flag, it must have been an element of some previous permutation cycle. The flag is removed and test is made on the next element. If an element is encountered without a V-flag, it has not been in any permutation cycle before, and it must be the leading element of a new permutation cycle. The first and last elements of any rectangular matrix are not affected by permutations. (b) The judicious use of interruption to exit from an otherwise endless loop can lead to much saving of programming and execution time.

Usually, however, interruption should be done with the help of the master-control or other supervisory programs to insure that other interruptions are also handled properly. Here one entry of the interrupt table has been changed at the beginning and restored at the end. (c) There exist numerous ways to improve the present program. In particular the replacement of VFL operations by proper floating-point counterparts may be recommended.

3 FLOATING-POINT ARITHMETIC

PROBLEM 3.1 SEPARATION INTO INTEGER AND FRACTION PARTS

The floating-point number N in location DOG has a small (≤ 48) exponent magnitude. Create two normalized floating-point numbers I, F in CAT, CAT+1 respectively such that:

I = an integer;
$|F| < 1.0$, sign of F = sign of N;
and I + F = N.

```
Method 1.   DL(U), DOG
            D+(U), X48
            ST(N), CAT
            SLO(N), CAT + 1.0
            BEW, $
X48         DD(N), 0.0X48                    'binary exponent of 48
```

Comments: (a) The number X48 forces the fraction of N to shift right the proper amount. (b) For better understanding, the reader should illustrate the program for himself using, for example, N = 2.5. (c) In dealing with normalized numbers, the (N) modifier is needed only for arithmetical operations which may otherwise generate an unnormalized result. The (U) modifier means "do not perform normalization," not "denormalize." L(U) and ST(U), when applied to an operand which has already been normalized will leave the number still normalized.

PROBLEM 3.2 INTEGER PART OF FLOATING-POINT WORD

The floating-point number N in location DOG is defined as in the previous problem. Put the lowest 18-bits of the VFL integer corresponding to I into the first 18-bits of the count field of $1.

```
Method 1.   DL(U), DOG
            D+(U), X48
            ST(BU, 18), 17.28, 68            '$1.28 is also acceptable
            BEW, $
X48         DD(N), 0.0X48
```

PROBLEM 3.3 POLYNOMIAL EVALUATION

Evaluate the polynomial

$$P(x) = \sum_{k=0}^{20} a_k x^k$$

where x is located in X, a_k is located in A + K, K = 0.0(1.0)20.0. Store the result (single precision) in POLY.

Method 1. Term-by-term evaluation.

```
POLYN   L(U), A
        ST(N), POLY
        L(U), X
        LX, $2, XW2
        B, STOR
LOAD    L(U), XK
        *(N), X
STOR    ST(U), XK                    'new power of x
        *(N), A($2)
        +(N), POLY
        ST(U), POLY                  'new partial sum
        CB+, $2, LOAD
        BEW, $
XW2     XW, 1.0, 20, $
XK      DR(N), (1)
```

Comments: (a) This is a relatively inefficient way to evaluate a polynomial but the technique applies to any finite series.

Method 2. Use the nesting technique.

$p(x) = (\ldots((a_{20}\,x + a_{19})x+\ldots)x+a_0.$

```
        LX, $2, XWORD2
        L(N), A+20.0
MULTI   *(N), X
        +(N), A($2)
        CB-, $2, MULTI
        ST(U), POLY
        BEW, $
XWORD2  XW, 19.0, 20, $
```

Comments: (a) The nesting technique for polynomials is twice as fast, more accurate, and requires fewer instructions than the term-by-term method.

Method 3. Use nesting technique and double operations for extra accuracy.

```
        LX, $2, XW2
        L(N), A+20.0
DMULT   D*(N), X
        D+(N), A($2)
        SRD(N), 8.0
        CB-, $2, DMULT
        ST(U), POLY
        BEW, $
XW2     XW, 19.0, 20, $
```

Comments: (a) The double operations are not much slower than the corresponding regular operation.

PROBLEM 3.4 MODIFIED TRAPEZOIDAL RULE

Evaluate the integral

$$I = \int_0^2 x^4 \, dx$$

by the modified trapezoidal rule

$$\int_a^b f(x)dx \cong h \left[f(a+h/2) + f(a+3h/2)+\ldots+ f(a+nh-h/2) \right]$$

where h = (b-a)/n. Use n = 20 for this purpose.

Method 1. Create a summing loop with the $f(x_k)$ evaluation inside the loop.

```
MTZR    LX, $1, XW1
        L(U), B
        -(N), A
        /(N), N
        ST(U), H
        E-I(U), 1                    'or 128.0
        +(N), A
        B, STOR
LOOP    L(U), TEMP
        +(N), H
STOR    ST(U), TEMP                  'update temp
        *(N), 8.0                    'or $L
        *(N), 8.0                    'new integrand value
        M+(N), ANS
        CB, $1, LOOP
        *(N), H
        BEW, $
XW1     XW, 0.0, 20, $
A       DD(N), 0.0                   'lower limit
B       DD(N), 2.0                   'upper limit
N       DD(N), 20.0                  'no. of strips
ANS     DD(N), 0
TEMP    DR(N), (1)
H       DR(N), (1)
```

Comments: (a) The E ± I instructions may be used for multiplying the floating-point number in the accumulator by powers of 2. They are more efficient than multiplications or divisions. (b) For a floating-point instruction the address 8.0 or $L means the leading 60 bits of the accumulator plus the lowest four-bits of $SB.

Method 2. Separate the function evaluation from the summing action in the loop.

```
MTZR2   LX, $1, XW1
        L(U), B
        -(N), A
        /(N), N
```

```
            ST(U), H
            E -I(U), 1
            +(N), A
            B, STOR
LOOP        L(U), TEMP
            +(N), H
STOR        ST(U), TEMP                 'new x
            B, FUNCT                    'branch to f(x) evaluation
RTURN       M+(N), ANS                  'new partial sum
            CB, $1, LOOP
            *(N), H
            BEW, $
ANS         DD(N), 0
TEMP        DR(N), (1)
H           DR(N), (1)
FUNCT       *(N), 8.0                   'function evaluator
            *(N), 8.0
            B, RTURN
XW1         XW, 0.0, 20, $
A           DD(N), 0.0                  'lower limit
B           DD(N), 2.0                  'upper limit
N           DD(N), 20.0                 'no. of strips
```

Comments: (a) The present program requires two additional branch instructions per loop, and is slower than that of Method 1. What it loses in speed is offset by the gain in clarity, however, and if a new integral is to be evaluated, only the lower portion of the program needs to be replaced.

PROBLEM 3.5 CONTINUED FRACTION EVALUATION

Evaluate the continued fraction

$$F = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \cfrac{x^2}{7 - \ddots - \cfrac{x^2}{39}}}}}, \quad \text{with } x = \pi/4.$$

Method 1.

```
CONF        L(U), X
            *, X
            ST(U), TEMP                 'x**2
            LX, $2, XW2
            L(U), NUM                   '39
LOOP        R/N, TEMP                   'X*X/39
            ST, TEMP1
            L(U), NUM
            -, TWO
            ST(U), NUM
            +, TEMP1                    '37-X*X/39
```

```
            CB, $2, LOOP
            R/, X
            ST(U), TEMP2
            BEW, $
X           DD(N), $PI/4
NUM         DD(N), 39.0
TWO         DD(N), 2.0
XW2         XW, 0.0, 19, XW2
TEMP        DR(N), (3)
TEMP1       SYN(N), TEMP+1.0
TEMP2       SYN(N), TEMP+2.0
```

Comments: (a) The most efficient way to evaluate a continued fraction is to start from below. (b) The R/N instruction should not be confused with R/(N). The reverse divide feature in the 7030 is convenient for continued fractions. (c) Where the dds is not explicitly given in an instruction, STRAP will insert the dds of the rightmost symbolic address. If the latter has no meaningful dds, the next-to-the-rightmost symbolic address will be used, etc. If the collection of symbolic addresses for the instruction is exhausted without a proper dds having been found, STRAP will use the (N) modifier for instructions which are unambiguously floating-point in nature. The exception being E+I and variants which are assigned the (U) modifier. An operation which could be either VFL or floating-point is assumed VFL.

PROBLEM 3.6 SCALAR PRODUCT OF VECTORS

Find the following vector scalar product

$$(a,b) = \sum_{k=0}^{16} a_k b_k$$

where a_k is in A+K, b_k in B+K, K=0.0(1.0)16.0. Put the result in C.

Method 1. Use LFT, *+.

```
            LX, $3, SXTEEN
            L(U), A
            D*(N), B
LOFT        LFT(N), A($3)
            *+(N), B($3)
            CB+, $3, LOFT
            SRD(N), C
            BEW, $
SXTEEN      XW, 1.0, 16, $
```

Comments: (a) The *+ operation yields a double-precision result. (b) The LFT operation is a "memory to memory" operation, since $FT is a bona fide memory location. Since it does not involve the execution arithmetic unit and since the temporary indicator $MOP is turned on only for execution arithmetic-to-memory operations, $MOP is turned off by LFT. (c) While the LFT operand is on its way to $FT (location 14.0 in memory) it is also made available in the lookahead to facilitate the *+ operation. This "forwarding" operation allows the *+ operation to proceed before $FT is actually loaded, freeing the program from memory access delays due to the store and

a subsequent fetch (for the *+). Forwarding is always done when information needed for the execution arithmetic unit is known to be available in the Lookahead.

PROBLEM 3.7 CUBE ROOT

Program to compute the cube root of a normalized floating-point number N by the following iteration formula:

$$X_{k+1} = X_k \cdot \frac{X^3_k + 2N}{2X^3_k + N} = X_k \left[1/2 + \frac{3N/2}{2X^3 + N} \right]$$

Use it to compute the cube root of 8, with $X_0 = 2.5$. Ten iterations will give full-length accuracy except for the round-off error in the last iteration.

Method 1

```
CBRT    L(U), EN
        E -I, 1
        +(N), EN
        ST(U), TEMP          '3N/2 stored in TEMP
        LX, $2, XW2
        L, GUESS
LOOP    ST, XK
        *, XK
        *, XK
        E + I, 1
        +, EN                '2X ** 3 + N
        R/, TEMP
        +, HALF
        *, XK                'new XK created
        CB, $2, LOOP
        ST, ANS
        BEW, $
XW2     XW, 0.0, 10, XW2
HALF    DD(N), 0.5
ANS     DR(N), (1)
TEMP    DR(N), (2)
XK      SYN (N), TEMP + 1.0
EN      DD(N), 8.0
GUESS   DD(N), 2.5
```

Comments: (a) This is a third order process: if x_k has a relative error ε, one iteration later x_{k+1} has a relative error of $C\varepsilon^3$, Here C=2/3. See E.G. Kogbetliantz, IBM Journal of Research and Development, 3, 147-152 (1959).

PROBLEM 3.8 NORMALIZED FLOATING-POINT VECTORS FROM VFL DATA

Given a string of 25 fields beginning at STRNG. Each field contains an integer with the description (D, 48, 6). Write a program to: (a) Change each number N_k into a normalized floating-point number F_k. (b) Create the sum of squares of F_k, then take the square root. (c) Divide each F_k by the square root, and store in FLOAT through FLOAT+24.0. (d) The sum of the squares of the resultant set of floating-point numbers should now be unity (barring a small round-off error). The vector composed of the

set is said to be normalized. Note vector normalization is not related to the machine hardware normalized floating-point arithmetic.

Method 1

```
NORMV    Z, SUM
         LX, $1, XW1
         LX, $2, XW2
LOOP     LCV(V+I)(D, 48,6), 0.48($1), 68
EPLUS    E+I, 48                          'number is now unnormalized FP integer
         ST(N), 0($2)
         *(N), $L
         +, SUM
         ST(U), SUM
         CBR+, $2, LOOP                   '$2 will be refilled on exit
         SRT, ROOT
LOOP2    L(N), 0($2)
         /, ROOT
         ST(U), 0($2)
         CB+, $2, LOOP2
         BEW, $
XW1      XW, STRNG, 25, $
XW2      XW, FLOAT, 25, $
SUM      DRZ(N), (1)
ROOT     DRZ(N), (1)
```

Comments: (a) A word full of zero bits is being used as the "zeroth partial sum." Note that a sequence of zero bits is only an "order of magnitude" zero, not a "true zero." A true zero can be approximated by a number with what looks like a very large negative exponent. An order of magnitude zero has a meaningful exponent, and can be interpreted as a number with no significant fraction digits. In addition-type operations, an order of magnitude zero, by virtue of its exponent, may force the fraction of a nonzero number to shift towards the right before the addition. In the present case the nonzeros all have larger exponents and the use of order of magnitude zero to start a sum will not lead to difficulties. (b) The EPLUS instruction could be removed from the loop without causing any damage; the errors introduced would exactly cancel in the normalization process. (c) The leading instruction is not really needed unless the program is to be reused in the machine. (d) The DRZ pseudo-operation leads to the reservation of strings of zero bits.

PROBLEM 3.9 DOUBLE-PRECISION COMPARE

The accumulator contains a double precision floating-point quantity. Another double precision floating-point quantity is stored in two full words, with the more significant part in M1, less significant part in M1+1. Compare the two double precision quantities and set the appropriate indicators $AE, $AL and $AH.

Method 1. Full-scale double-precision subtract followed by a test on the result.

```
MKOMP    ST(U), A1                        'save accumulator
         SLO (U), A1+1.
         DL(U), A1+1.                     'double-precision subtract
         D-(N), M1+1.
         D+(N), A1
```

```
            D-(N), M1
            L(BU, 3), $RLZ          '$RLZ, $RZ, $RGZ fetched
            ST(BU, 3), $AL          '$AL, $AE, $AH stored
            DL(U), A1+1.            'restore accumulator
            D+(U), A1
            BEW, $
A1          DR(U), (2)
```

Comments: (a) The temptation is strong to compare the high order parts first, and accept the indicator settings unless equality is indicated, and in the latter compare the lower order parts. This is not correct because the compare instruction is based on a floating subtract operation rather than a bit-by-bit comparison. For example: if (A1, A1+1) and (M1, M1+1) have

E	111--------1	+

E-48	1--------0	+

and

E+1	100-------0	+

E+1-48	000-----0	+

respectively

then a comparison between A1 and M1 leads to $AE=1 (the first 48 fraction bits of the subtraction result being zero). A straightforward compare of the second order parts will lead to the erroneous conclusion that (A1, A1+1.) is larger than (M1, M1+1.), whereas in reality (A1, A1+1.) represents

$(1-2^{-49}) * 2^{E}$ but (M1, M1+1.) represents the larger quantity

$(1/2) * 2^{E+1} = 1*2^{E}$ the difference being noticeable at the fiftieth bit.

(b) Aside from the above considerations the program presented does not use conditional branches, eliminating the need for wrong branch recovery. (c) The present program is applicable even if the lower order parts are slightly off standard (say with an exponent only 46 units lower than the higher order counterparts).

Method 2. Compare high order parts. If they compare "equal," perform the double precision subtraction to ascertain the result.

```
DKOMP2      ST(U), A1               'save accumulator
            SLO(U), A1+1.
            K(U), M                 'single precision compare
            BAE, DPSUB              'usually unsuccessful
END         BEW, $                  'end of program
DPSUB       DL(U), A1+1.            'full-scale double precision subtract
            D-(U), M1+1.
            D+(N), A1
            D-(N), M1
            L(BU,3), $RLZ           '$RLZ, $RZ, $RGZ fetched
            ST(BU,3), $AL           $AL, $AE, $AH stored
            DL(U), A1+1.            'restore accumulator
            D+(U), A1
            B, END
A1          DR(U), (2)
```

Comments: (a) The present program is free of the objections outlined in Method 1. It is fast if the higher order parts decide the outcome (as is usually the case). Very effective for normalized double-precision numbers, it may yield erroneous answers if

one of the high order parts has a zero fraction, as seen in the following case.

(A1, A1+1.)	E+1+48	0	+		E+1	0	+
(M1, M1+1.)	E	111-----1	+		E-48	1------1	+

Present program would lead to (A1, A1+1.) larger, because of the 48 unit difference in the exponent. Correct answer should lead to (M1, M1+1.) larger since the exponent difference is not 96 units, and

$(1-\varepsilon)*2^{E}$ is clearly larger than $0*2^{E+48}$

(b) Even for a program with many different branches, it is convenient to end at the same place as a debugging aid. Any other instruction counter setting at the termination of computation will then be an error signal.

PROBLEM 3.10 INTEGER PART OF Log_2N

N is a positive floating number in DOG, and $\log_2 N$ can be written as an integer plus a positive fraction. Find the integer and put its magnitude in the first 18-bits of the value field of $1, and the sign in the sign position of the value field of $1. Assume no exponent flag.

Analysis: If $N = 2^{\alpha}\beta$, $\quad 1/2 \le \beta < 1$

Then $\log_2 N = \alpha + \log_2 \beta \qquad -1 \le \log_2 \beta < 0$

$= \alpha - 1 + (1 + \log_2 \beta)$

evidently α -1 expressed as a 18-bit VFL integer, is the desired quantity.

Method 1.

```
L(N), DOG
E-I, 1
L(B, 12, 1), 8.0, 6
ST(B, 25, 1), $1
BEW, $
```

4 SPECIAL PROBLEMS

PROBLEM 4.1 VFL FRACTION SQUARE-ROOT

Given a 64-bit binary unsigned VFL fraction in FRAC, extract the square root and put it in the 64-bit field beginning at ROOT.

Analysis: By the Newtonian process of extracting the square root x of the number N,

$$x_{k+1} = x_k + N/x_k)/2$$

If x_k has a relative error of ε, namely

$$x_k = x_t(1+\varepsilon); \quad x_t = \text{true } x$$

then $$x_{k+1} = x_t\,(1 + \varepsilon^2 / 2 + 0\,(\varepsilon^3))$$

Thus if we are able to find a guess which has a relative error of 2^{-32}, one iteration later the relative error would be reduced to 2^{-65}.

The 64-bit fraction is equivalent to a floating-point number with zero exponent. If this latter is manufactured and normalized, the SRT instruction can be used to give a relative error less than 2^{-47}, which is more than adequate for our initial guess. The subsequent iteration is done in double precision, with the second order part of the initial guess understood to be zero.

Method 1

```
SQRT    L(BU, 64), FRAC, 52        'looks like FP number
        BRZ, STOR
        D+(N), 0                   'normalized long fraction
        SRT(U), GUESS              'first guess
        D/(N), GUESS
        ST(U), QUOT                'first order quotient
        DL(U), $RM
        /(N), GUESS                'obtain second order quotient
        D+(U), QUOT                'double length quotient
        D+(N), GUESS
        E-I(U), 1                  'divide by two
        D+(U), 0                   'shift until exponent zero
STOR    ST(BU, 64), ROOT, 52
        BEW, $
GUESS   DR(U), (1)
QUOT    DR(U), (1)
FRAC    DR(U), (1)                 'to be supplied
```

Comments: (a) Had the original fraction not been prenormalized, it may contain a number of leading zeros. The relative error of the square root of the first 48 bits may no longer be the guaranteed 2^{-47}, but may be as large as 1 (when the first 48 bits are all zeros). (b) The result is not rounded, as rounding will create an overflow in the exceptional case when FRAC is almost 1.0.

PROBLEM 4.2 DOUBLE-PRECISION BINARY TO DECIMAL CONVERSION

Given a 96-bit binary fraction beginning at BFRAC, transform it into a 112-bit decimal fraction beginning at DFRAC.

Analysis: The binary fraction F can be recoded in terms of any integer radix R:

$$F = \sum_{k=1}^{\infty} a_{-k} R^{-k} \quad ; \quad R = \text{integer}, \quad 0 \leq a_{-k} \leq R-1.$$

The problem is to find the a_{-k}'s up to, say, k = m. Now

$$RF = a_{-1} + \sum_{k=2}^{\infty} a_{-k} R^{-1-k} = a_{-1} + F_1 ,$$

$$RF_1 = a_{-2} + \sum_{k=3}^{\infty} a_{-k} R^{-2-k} = a_{-2} + F_2 ,$$

$$\cdots\cdots$$

$$RF_{\ell-1} = a_{-\ell} + \sum_{k=\ell+1}^{\infty} a_{-k} R^{-\ell-k} = a_{-\ell} + F_\ell ,$$

$$\cdots\cdots$$

$$RF_{m-1} = a_{-m} + F_m .$$

The integers a_{-k} can be extracted after each binary multiplication. They are binary quantities still, but can be recoded in terms of known conventions. F_m can be used to create a rounded result, but is more often ignored.

For our problem let $R=10^{14}$. This is the largest power of 10 expressible by 48 bits, and will contribute to the speed of conversion. The binary multiplcation will be that between a single precision number R and a multiple precision quantity F_L.

The a_{-k}'s will have no more than 48 bits, and can be converted into decimal by the CONVERT type instructions. The recoded a_{-k} will each have no more than 56 bits. Since 2*56=112, we need only the first two "super digits."

Method 1

```
DFCONV  L(BU, 48), BFRAC +0.48,68       'second order part
        *(U), RADIX
        L(BU, 48), $L +0.12, 20         'third order result ignored
        LFT(BU, 48), BFRAC
        *+(U), RADIX                    'there will be forwarding
        ST(BU,48),BUFFER +0.12,20       'save second order part
        CV(BU, 48)                      'convert first super digit, zero offset
        ST(DU, 56), DFRAC
        L(U), BUFFER
        *(U), RADIX
        CV(BU, 48)
        ST(DU, 56), DFRAC +0.56
        BEW, $; CNOP                    'next item begins at full word
RADIX   DD(BU,12), 0
        DD(BU,48),(8)2657142036440000   '10**14
```

```
            DD(BU,4), 0
BUFFER      DD(BU, 64), 0
BFRAC       DR(BU,48), (2)          'data to be supplied
DFRAC       DR(DU,56), (2)
```

Comments: (a) a 96-bit binary number contains information actually equivalent to 116.25 bits of a decimal number. Only 112 bits are needed for the problem as stated. (b) In an n-fold precision calculation, (n+1)st order quantities frequently (though not always) have little effect, and can be ignored. Here the neglected third order quantity is nowhere larger than 2^{-95}. (c) The 64 bits beginning at RADIX is being used as an unnormalized floating-point number with plus zero exponent.

PROBLEM 4.3 BIT IMAGE OF A SEQUENCE OF NUMBERS

Given 64 numbers in successive full words beginning at NUMB. Many of these are floating-point zeros, but some are not. Create a full word beginning at BIMAGE in which successive bits reflect the condition of the successive words, such that a zero number will be represented by a zero bit image and a nonzero will have a 1 bit as image.

```
Method 1.   LX, $1, XW1
            LX, $2, XW2
            Z, BIMAGE               'assume most are zeros
LU          L(U), NUMB($1)
            BZRZ, FIX               'usually unsuccessful
            V+, $2, BIT             'increase by one bit
CAB         CB+, $1, LU
            BEW, $
FIX         CM1111(BU, 1) (V+I), 0.1($2)
            B, CAB
BIT         VF, 0.1
XW1         XW, 0.0, 64, $
XW2         XW, BIMAGE, 0, $
```

Comments: (a) The bit image is very useful in, say, sparse matrix multiplication. The bit image of each vector involved can be created, and the nontrivial multiplications needed between any two such vectors can be tested via the logical connective "and," and the subsequent querying of $AOC and $LZC.

PROBLEM 4.4 COMPRESSION OF SPARSE VECTOR

Given a sparse vector of N components stored in consecutive floating-point words beginning at SVEC. It has a bit image stored in consecutive bits beginning at the full word beginning at BIMAGE. Compress the vector into the smallest possible storage space on the basis of this bit image, and put the result in consecutive words beginning at SVEC also.

```
Method 1.   LX, $1, XW1; LX, $3, XW3
            LVNI, $2, 1.0
            B, CONN
LOWF        LWF(U), SVEC ($2)
            ST(U), SVEC($3)
```

```
        V+IC, $3, 1.0
CONN    C0011(BU, 1) (V+IC), 0.01($1)
        BXCZ, END
        V+I, $2, 1.0
        BZRZ, LOWF
        B, CONN
END     Z, SVEC($3)                          'zero unused region
        CB+, $3, END
        BEW, $
XW1     XW, BIMAGE, N+1, $
XW3     XW, 0.0, N, $
```

PROBLEM 4.5 SCALAR PRODUCT OF COMPRESSED SPARSE VECTORS

X and Y are two N-dimensional sparse vectors, $N \leq 64$, with the non-zero components stored in consecutive floating-point words beginning at XVEC and YVEC respectively, and bit images stored at XBMAGE and YBMAGE respectively. Find the scalar product of these two vectors.

Analysis: In the scalar product

$$(x, y) = \sum_{k=1}^{N} x_k y_k.$$

the multiplication need be performed only when x_k and y_k are both non-zero. This information may be obtained with a connect operation on the bit images of the two vectors. The $AOC will yield the number of multiplications to be performed and the $LZC will give information about the subscript K for a needed multiplication.

Method 1.

```
        TI, 3, 17.0, SAVEX                   'save $1, $2, $3
        L(BU, N), XBMAGE
        C0001(BU, N), YBMAGE                 '1 bit if both items non-zero
        ST(BU,N), KEYVEC
        L(BU,7), $AOC, 64+18
        LX, $3, 8.0                          "$AOC in $3 count field
        DL(U), ZERO;ST(U),PRODT
        BXCZ, FIN
        B, LOF; CNOP
LOOP    CM0000(BU,0),KEYVEC+0.1,0($1)        'field length indexing
        CT0011(BU,N),KEYVEC                  'test left zeros
LOF     LF(BU,25),$LZC-0.2,128-25            'low order part untouched
        LV, $1, 8.0                          'C(LZC)at field length position
        CT0011(BU,0),XBMAGE,0($1)            'field length indexing
        LV,$2,7.32
        V+, $2, 18.0                         'XVEC modifier properly positioned
        CT0011(BU,0),YBMAGE,0($1)            'field length indexing
        LV, $3, 7.32
        V+C, $3, 19.0                        'YVEC modifier
        L(U), PRODT                          'restore high order part
        LFT(U), XVEC($2)                     'computation part
        *+(N), YVEC($3)
        ST(U),PRODT
        BZXCZ, LOOP
```

```
FIN     TI, 3, SAVEX, 17.0          'restore $1, $2, $3
        BEW, $                      'answer in acc. as well as PRODT
ZERO    DD(N), 0
PRODT   DRZ(N), (5)
KEYVEC  SYN(N), PRODT+1.0
SAVEX   SYN(N), PRODT+2.0
```

Comments: (a) If half of the elements of each vector are zero, then statistically speaking only one quarter of the multiplications need to be performed. Thus the loop in the present program can take four times as long as the corresponding loop in the straightforward multiplication method, and still be efficient for sparse vectors and sparse matrices. (b) The second I field in a VFL instruction can be used to index the field length and byte size besides the offset. Bits in the half-word position in the index value field influence the offset directly, bits in 2^6 times full word position influence the byte size directly, and bits in 2^9 times full word position influence the field length directly. Note that $LZC is given at the bit level and $AOC is given at the half-word level, necessitating a small amount of adjustment.

PROBLEM 4.6 TRANSPOSITION OF AN 8 x 8 BIT MATRIX

Given an 8x8 matrix whose elements are bits stored consecutively and row-wise starting at BMATX8. Create the transpose and store the latter in the same area.

Method 1. Bit-by-bit operation

```
BMX8T   LX, $2, XW2; SX, $2, XW22
        LX, $3, XW3; SX, $3, XW33
LOF     LF(BU,1),0.0($3),64
        LF(BU,1), 0.0($2)
        SF(BU,1)(V+ICR),0.1($2),64
        SF(BU,1)(V+ICR),N($3)
        BZXCZ,LOF
        V+C,$2, VF;SX,$2,XW22
        V+C, $3, VF; SX, $3,XW33
        BZXCZ, LOF
        BEW,$
XW2     XW, LOC+0.1,N-1,XW22
XW3     XW,LOC+0.N,N-1, XW33
XW22    XW,0
XW33    XW,0
VF      VF,0.1+0.N
LOC     SYN, BMATX8
N       SYN,8
```

Comments: (a) The program is written to accommodate an NxN bit matrix beginning at LOC. The SYN pseudo-instructions define LOC as BMATX8 and N to be 8. BMATX8 is assumed to be defined elsewhere in the symbolic program. (b) 0.N is equivalent to 0.8, since N is 8.

Method 2. Take advantage of the special properties of connective operations.

```
BMX8T2    LX, $1,XW1
          LVI,$2,8-1                          '7 half-words
          LI(BU,1),0                          'zero accumulator
          B,CNNECT
VMI       V-I, $2, 1                          'reduce offset by 1
CNNECT    C0 111(BU,8,1)(V+IC),0.8($1),0($2)
          BZXCZ,VMI
          ST(BU,64),BMATX8
          BEW,$
XW1       XW,BMATX8,8,$
```

Comments: (a) This is a much more efficient program. Instead of transporting 2*64 bits one at a time, 8-bits are loaded with each connect instruction and the entire transposed matrix is stored in one instruction. The indexing here is less involved also. The price one pays is the lack of generality--for a square matrix of size greater than 8x8 the coding would have to be considerably different.

Method 3. Same technique as above, but coded to accommodate all NxN matrices with $N \leq 8$.

```
          LX,$1,XW1
          LVI, $2,N-1
          LI(BU,1),0
          B,CNNECT
VMI       V-I, $2,1                           'reduce offset by 1
CNNECT    C0111(BU,N,1)(V+IC),0.N($1),0($2)
          BZXCZ,VMI
          SF(BU,N*N,N),LOC
          BEW,$
XW1       XW,LOC,N,$
LOC       SYN,BMATX8                          'or any location desired
N         SYN,8                               'or any integer not exceeding 8
```

Comments: (a) The store field instruction will not be assembled correctly by STRAP I, because of the multiplication in the data description field. STRAP II will do it properly.

PROBLEM 4.7 TRANSPOSITION OF A 64 x 64 BIT MATRIX

Given a 64x64 matrix whose elements are bits stored consecutively and row-wise starting at BMX64. Create the transpose and store it in the same area.

Method 1. Bit-by-bit operation. Same as Method 1 of previous program with LOC and N redefined to be BMX64 and 64 respectively.

Method 2. Use logical connectives. The matrix is partitioned into 8x8 submatrices or blocks and each is transposed separately.

```
BMX64T    LX,$1,XW1;SX,$1,XW11;SX,$1,XW111  'row block index
          LX,$2,XW2;SX,$2,XW22,SX,$2,XW222  'column block index
          LX,$3,XW3                         'offset index
          LX,$4,XW4;SX,$4,XW44              'block counter
```

```
DIAG     LI(BU,1),0                                  'clear accumulator
DIAG1    C0111(BU,8,1)(V+ICR),0.64($1),7($3)         'loop for diagonal block
         V-ICR,$3,1                                  'lower offset by 1
         BZXCZ,DIAG1                                 'until block completed
DIAG2    ST(BU,8,8)(V+ICR),0.64($1),64-8($3)         'store diagonal block row-wise
         V-ICR,$3,8
         BZXCZ,DIAG2                                 'until block stored
         CBZR,$4,BEW                                 'branch if last diagonal block complete
OFDIAG   V+,$1,VFP8;SX,$1,XW111                      'loop for off diagonal block pair
         V+,$2,VF8P;SX,$2,XW222
         LI(BU,1),0
OFDIA1   C0111(BU,8,1)(V+ICR),0.64($1),7($3)         'row block treatment
         C0111(BU,8,1)(V+ICR),0.64($2),64+7($3)      'column block treatment
         V-ICR,$3,1                                  'lower offset
         BZXCZ,OFDIA1                                'until block pair complete
OFDIA2   ST(BU,8,8)(V+ICR),0.64($2),64-8($3)         'store into column block area
         ST(BU,8,8)(V+ICR),0.64($1),128-8($3)        'store into row block area
         V-ICR,$3,8
         BZXCZ,OFDIA2                                'until block pair stored
         CBR,$4,OFDIAG                               'until one row, one column complete
NEWROW   LX,$1,XW11                                  'procedure for new row
         V+,$1,VF8P8
         SX,$1,XW11;SX,$1,XW111
         LX,$2,XW22
         V+, $2, VF8P8
         SX,$2,XW22;SX,$2,XW222
         C-I,$4,1;SX,$4,XW44
         B,DIAG
BEW      BEW,$
VFP8     VF,0.8
VF8P     VF,8.0
VF8P8    VF,8.8
XW1      XW,LOC,8,XW111
XW2      XW,LOC,7,XW222
XW3      XW,0,8,$
XW4      XW,0,8,XW44
XW11     XW,0                                        'to contain row information
XW22     XW,0                                        'to contain column information
XW44     XW,0                                        'to contain block counter
XW111    XW,0                                        'to contain row block information
XW222    XW,0                                        'to contain column block information
LOC      SYN,BMX64
```

Comments: (a) The matrix is (mentally) partitioned into 64 square submatrices, or blocks, each of size 8x8. The (I, J)-block of the transposed matrix is the transpose of the (J,I)-block of the original matrix. (b) XW1, XW2, XW3, and XW4 are not destroyed in the program. XW11, XW22, and XW44 are changed upon the completion of permutation of a row of blocks with a column of blocks. XW111 and XW222 are changed upon the completion of permutation of each pair of blocks, or that of a diagonal block.

PROBLEM 4.8 PRODUCT OF SQUARE MATRICES

NxN full-word floating-point matrices L, R are stored row-wise beginning at LMTRIX and RMTRIX respectively. Create P=L*R and store it row-wise beginning at PMTRIX.

Method 1. Use $2 for left matrix elements, $3 for right matrix elements and $4 for product matrix elements. Program generates successive rows of the product matrix.

```
          TI,3,XW2,$2              'load three index registers
SIX       SX,$2,XW22
LU        DL(U),ZERO
LIFT      LFT(U),0($2)             'main loop
          *+(N),0($3)
VPI       V+I,$3,N                 'advance $3 to next row
          CBR+,$2,LIFT             'advance $2 to next element
          SRD(N),0($4)             'new product matrix element
          V+I,$4,1.0
          V-ICR,$3,N*N-1.0
          BZXCZ,LU                 'towards new product element of same row
          V+I,$2,N                 'procedure for new row
          CB,$4,SIX;BEW,$
XW2       XW,LMTRIX,N,XW22
XW3       XW,RMTRIX,N,$
XW4       XW,PMTRIX,N,$
XW22      XW,0
ZERO      DD(N),0
```

Comments: (a) STRAP I does not perform multiplication of addresses, but STRAP II will do it properly. (b) XW2, XW3, and XW4 are not destroyed and the program can be used repeatedly without re-assembly or reloading into the machine.

PROBLEM 4.9 COSINE OF $2\pi x$

Given a number $-1/8 \leq x \leq 1/8$ in the accumulator. Create $2\pi x$ in the accumulator.

Analysis: Since $-\pi/4 \leq 2\pi x \leq \pi/4$, the series

$$\cos 2\pi x = 1 - \frac{(2\pi x)^2}{2!} + \frac{(2\pi x)^4}{4!} - \cdots\cdots$$

$$= \sum_{k=0}^{\infty} \frac{(-1)^k (2\pi x)^{2k}}{2K!}$$

is rapidly convergent. If the series is truncated at some point, the absolute error δ is estimated by the magnitude of the first omitted term. Further, since $\cos 2\pi x \geq \cos \pi/4 > 0.7$, the relative error defined by $\varepsilon_r = \frac{\text{absolute error}}{\text{true answer}}$ is less than or equal to 1.43δ.

If the last term included has 2K=16, the relative error estimate is less than 0.3×10^{-15}, well within the round-off error due to arithmetical operations using a 48-bit fraction field length.

Method 1

```
COSF    *(N), TPI                                          '2*$PI
        D*(N),$L                                           'square
        SRD(N), TEMP
        LX, $2, XW2
DMULT   D*N(N), CONST($2)
        D+(N), WON
        D*(N),TEMP
        CB+, $2, DMULT
EMI     E-I, 1
        D-(N), WON
        SRDN,$L
        BEW, $
TPI     DD(N), 2*$PI
XW2     XW, 0, 7, $
CONST   DD(N), 1/16*1/15, 1/14*1/13, 1/12*1/11,1/10*1/9
        DD(N), 1/8*1/7, 1/6*1/5, 1/4*1/3
WON     DD(N), 1, 0
TEMP    DR(N), (1)
```

Comments: (a) Instruction EMI is used in lieu of a multiplication by 1/2*1/1 to gain a little speed. (b) By a redefinition of the constants the multiplication by 2*$PI could be eliminated, but then instruction EMI would have to be replaced by a full-scale multiply operation. (c) The nesting technique used tends to keep the round off error to a minimum. (d) The number (2) of multiplication operations in the loop can be halved by using 1/2n! as the constants.

Method 2. Since cos 2A = $2\cos^2 A - 1$, it is possible to reduce the number of terms in the series by evaluating cos π x first. Examination shows that terms up to K = 12 would be adequate.

```
COSF2   D*(N),$L
        SRD(N),TEMP
        LX, $2, XW22
DMULT   D*N(N), KONST($2)
        D+(N), WON
        D*(N), TEMP
        CB+, $2, DMULT
        D*N(N), KONST($2)
        D+(N), WON
        D*(N), $L                                          'create cos 2A
        E+I, (U), 1
        D-(N), WON
        SRD(N), $L
        BEW, $
XW22    XW, 0, 5, $
KONST   DD(N), $PI/12*$PI/11,$PI/10*$PI/9
        DD(N), $PI/8*$PI/7, $PI/6*$PI/5
        DD(N),$PI/4*$PI/3,$PI/2*$PI
WON     DD(N), 1.0
TEMP    DR(N), (1)
```

Comments: (a) The error situation is somewhat worsened in the present method. Suppose cos A has been evaluated with absolute error δ_1; then

$$\cos A = (\cos A)_{true} + \delta_1 ,$$

$$2\cos^2 A - 1 = 2(\cos^2 A)_{true} - 1 + 4\delta_1, \cos A$$

The total absolute error is therefore

$$\delta = 4\,\delta_1 \cos A \text{ or } 4\delta_1 \text{ roughly.}$$

The relative error can be examined in the same light.

PROBLEM 4.10 NATURAL LOGARITHM

A positive single-precision normalized floating-point number x is in the accumulator. Replace it by lnx. Assume zero exponent flag for x.

Analysis: $x = F*2^E = \sqrt{2}\ F*2^{E-1/2}$

$$\ln x = (E-1/2)\ln 2 + \ln(\sqrt{2}\ F)$$

$$\ln \sqrt{2}F = 2\sum_{k=0}^{\infty}\left(\frac{\sqrt{2}F-1}{\sqrt{2}F+1}\right)^{2k+1}/(2k+1) = 2Z\sum_{k=0}^{\infty}(Z^2)^k/2K+1$$

Since $Z^2 = \left(\frac{F - 1/\sqrt{2}}{F + 1/\sqrt{2}}\right)^2$ lies approximately in (0, 1/36), the series is rapidly convergent.

Replacing the upper limit by $K_{max}=8$ the absolute truncation error in the determination of $\ln \sqrt{2}\,F$ would be much less than 2^{-48}. If the (E-1/2)ln2 term dominates in lnx, the relative truncation error would also be much less than 2^{-48}, and further improvement in this direction cannot be seen in the single precision fraction.

If on the other hand, (E-1/2) ln2 does not dominate the result, $|E-1/2|$ itself must be small. But it can be no smaller than 1/2, since E is an integer. Therefore, the worst that can happen is when E=0, F~1. In this case one can show the error cannot be improved without knowledge of the fraction part of x beyond 48 bits.

Method 1.

```
LNX     ST(U), TEMP                 'F+1/RT2
        F+(N), Q
        ST(U), TEMP+1
        F-(N), QQ                   'F-1/RT2
        /(N), TEMP+1                'Z created
        ST(U), TEMP+1
        *(N), $L                    'Z**2
        ST(U), TEMP+2
        LX, $1, XW1
        D*(N), CONST($1)
ADD     D+(N), CONST+1.0($1)
        D*(N), TEMP+2
        CB+, $1, ADD
        D+(N), CONST+1.0($1)
        *(N), TEMP+1
        E+I, 1
        ST(U), TEMP+2
        L(B,12,1), TEMP, 69         "exponent treatment
```

```
            -I(BU, 1), 1, 68
            D*(N), FLN2                                  '2E-1 times ln2
            D+(N), TEMP+2
            BEW, $
Q           DD(N), 1/1.41421356205080                    '1/RT2
QQ          DD(N), 2/1.41421356205080                    '2/RT2
XW1         XW, 0, 7, $
CONST       DD(N), 1/17, 1/15, 1/13, 1/11, 1/9,1/7,1/5,1/3,1
FLN2        DD(N), $NX47                                 '$N*2**47
TEMP        DRZ(N), (3)
```

Comments: (a) In function evaluation an understanding of the properties of the function and the format of the numbers used frequently leads to great improvement in speed and accuracy, as shown by this example. (b) The truncated Taylor series in Z can be replaced by a polynomial with fewer terms but comparable accuracy. The coefficients of the optimal polynomial(s) for the evaluation of functions can be computed by an iterative process, or can be excellently approximated by appealing to the properties of the orthogonal Chebyshev polynomials. See, for example, C. Lanczos, Applied Analysis (Prentice-Hall, 1956) Ch. VII; F.D. Murnahan and J.W. Wrench Jr., Mathematical Tables and Other Aids to Computation, 8, 185(1959). (c) Instead of divisions by (2k+1), multiplication by the inverse is used for speed. (d) In FLN2,X47 means replace the exponent field by +47.' In the present case, $N, having the magnitude of 0.7, normally would have an exponent of zero, and $NX47 is the same as $N*2**47. This would not be true has $N a magnitude of, say, 1.5.

PROBLEM 4.11 EXPONENTIAL OF x

Given a normalized floating-point number x in the accumulator. Find e^x., put it in the accumulator and branch to 1.0($15). If e^x cannot be found or stored, branch to 0.0 ($15). Alteration of $L, $R, $SB, $LCZ, $AOC and $14 is permitted.

Analysis: If $|x| > 1024 \ln 2$, $e^x = 2^{x/\ln 2}$ cannot be stored as a regular floating-point number. A 0.0($15) return with the exponent flag on is sufficient. Otherwise the following algorithm can be used:

$$e^x = 2^{x/\ln 2} = 2^{I+F} = 2^I \cdot 2^F$$

$$2^F = e^{F \ln 2} = \sum_{k=0}^{\infty} \frac{(F \ln 2)^k}{k!}$$

terms beyond k = 15 can be safely neglected.

It is also possible to reduce the range of the argument in the series to improve convergence. For instance:

$$2^F = (2^{F/2})^2 = (2^G)^2$$

$$2^G = e^{G \ln 2} = \sum_{k=0}^{\infty} \frac{(G \ln 2)}{k!}$$

and terms beyond k = 12 can be neglected. The subsequent squaring lead to a round off error twice as large as before, however,

Method 1.

```
EXP     KMG(N), KOMP
        BAH,EXIT1
        D*(N), RLN2                          '1/LN2
        D+(U), E11
        ST(B,12,1), TEMEX, 128-12-11         'I as exponent
        SHFL,11
        *(N), LN2X                           'LN2X-11
        LX, $14, XW14
        ST(U), TEMPF
        D*(N), CONST($14)
DPLUS   D+(N), CONST+1.0($14)
        D*(N), TEMPF
        CB+, $14, DPLUS
        D+(N), CONST($14)
        E+(N), TEMEX
        B, 1.0($15)                            'normal return
EXIT1   C0011(BU,1),10.4,128-11                'exponent sign
        LA(U), $L                              'remove sign
        C1111(BU,1) $L, 127                    'insert exponent flag
        B, 0.0($15)
KOMP    DD(N), 1024*$N
RLN2    DD(N), 1/$N
E11     DD(N), 0X11
LN2X    DD(N), $NX-11
XW14    XW,0,14,$
CONST   DD(N), 1/13076743680000,1/87178291200
        DD(N), 1/6227020800, 1/479001600, 1/39916800
        DD(N), 1/3628800, 1/362880, 1/40320, 1/5040, 1/720
        DD(N), 1/120, 1/24, 1/6, 1/2, 1
TEMEX   DR(N), (1)
TEMPF   DR(N), (1)
```

Comments: (a) There are numerous ways to improve the speed of the program. The multiplications by 1/2 and 1, for instance, can be replaced by more efficient devices. The creation of F ln2 also would not be needed if $(\ln 2)^k/k!$ are used instead of 1/k! as coefficients. (b) The present program is actually written as a subroutine, assuming the convention of 1.0($15) normal return and 0.0($15) error return. Aside from $L, $R, $SB, $14, and $15, none of the other internal registers is altered during exit. The memory requirement is also modest. Further, the program can be used again and again to evaluate the exponential of whatever floating-point number is given in the accumulator.

PROBLEM 4.12 TRANSCENDENTAL FUNCTION EVALUATION

Assume the existence of the previous exp (x) program. Compute

$$f(x)=2xe^{e^x}/\sqrt{1-e^{-x}} \qquad \text{for } x = \pi$$

and put the answer in the accumulator as a floating-point number.

Method 1.

```
          LN(N), EKS
          LVI, $15, $+1.0; B, EXP
          B, ERR; NOP
RTURN     ST(U), TEMP
          LVI, $15, $+1.0; B, EXP
          B, ERR; NOP
          *(N), EKS
          E+I(U), 1
          ST(U), TEMP+1
          LN(U), TEMP
          +(N), WON
          SRT(N), $L
          R/(N), TEMP+1
          BEW, $                    'normal exit
ERR       BEW, $                    'error exit
EKS       DD(N), $PI
TEMP      DR(N), (2)
WON       DD(N), 1.0
```

Comments: (a) The present program is designed to demonstrate the usefulness of subroutines for repeated usage. (b) The accepted way to enter the subroutine SR (say) is to write

LVI, $15, $+1.0 (or LVI, $15, $+2)

before branching into SR. In STRAP II a pseudo-instruction

LINK (no address needed)

is available for this purpose. (c) It is obvious that the present program can be recast into conventional subroutine form also, if ever needed. (d) The present program requires the EXP subroutine, and therefore is usually assembled together with the latter. Fortunately there is no multiply defined symbol to produce difficulties and no conflict in the use of special registers and $14, $15. A good subroutine should keep the number of symbols small, and the "tailing" feature available in STRAP can be used by the user of the subroutine to avoid memory conflict.

PROBLEM 4.13 NUMERICAL INTEGRATION

Provide a subroutine to handle the numerical integration of any function over any finite interval. Use it to evaluate:

$$I = \int_0^1 2 \times e^{e^{-x}} \Big/ \sqrt{1 - e^{-x}} \; dx$$

Analysis: (a) For standard intervals, say (p,q), an n-point numerical integration quadrature formula is the approximation

$$\int_p^q w(z)F(z)\,dz \sim \sum_{i=1}^{n} W_i \; F(z_i)$$

with prescribed $\{W_i\}$ and $\{z_i\}$. In the well-known Newton-Cotes quadratures the z_i's are evenly spaced over the interval.

In the case of the highly accurate Gaussian quadratures the z_i's are the zeros of the nth degree orthogonal polynomial $P_n(z)$, where

$$\int_p^q w(z)\, P_n(z)\, P_m(z)\, dz = 0, \quad n \neq m.$$

The n-point Gaussian quadrature will yield an exact answer (barring round-off error) if F(z) is a polynomial of degree no higher than 2n-1. For other integrands the approximation is, in general, quite excellent. The most commonly used Gaussian quadrature is the Legendre-Gauss quadrature with

$$(p,q) = (-1, +1) \text{ and } w(z) = 1.$$

For even n the formula becomes

$$\int_{-1}^{+1} F(z)\, dz \sim \sum_{i=1}^{n/2} W_i \left[F(z_i) + F(-z_i) \right]$$

For finite limits (a,b) other than (-1, +1), we have

$$\int_a^b f(x)\, dx = s \int_{-1}^{+1} f(sz+t)\, dz = s \int_{-1}^{+1} F(z)dz$$

$$\sim s \sum_{i=1}^{n/2} W_i \left[f(sz_i + t) + f(-sz_i + t) \right] \quad ;$$

where $s = (b-a) / (q-p) = (b - a) /2$, $t = a - sp = (b + a) /2$.

(b) The integration subroutine has to be able to obtain $f(sz_i + t)$ and $f(-sz_i + t)$ for a number of z_i's. It is thus desirable to have available an integrand evaluation subroutine, written in a standard format. The integration subroutine does not need to know the integrand subroutine in detail, only its address and calling sequence. It is conceivable that the integrand subroutine also requires other subroutines, but this would not be the direct concern of the integration subroutine itself. (c) The following specifications for the 8-point Legendre-Gauss integration subroutine LEGQ8 are therefore reasonable:

1) The main program branches to the integration subroutine by the standard LINK entry, in the following format:
   LVI, $15, $+1.0;B, LEGQ8
2) The leading 19 bits of the ensuing full word must contain the address of the subroutine for the evaluation of the integrand.
3) The next full word (i.e., 1.0($15)) must contain the floating-point lower limit A.
4) The next full word (2.0($15)) must contain the floating-point upper limit B.
5) If an error occurs in the integration program, a return should be made to 3.0($15).
6) If the evaluation is successful, the approximate value of the integral must be in the accumulator during the normal return. The normal return address is 4.0($15).
7) All internal registers except $L, $R, $SB, $RM, $FT, $TR, $LZC, $AOC, and $14 are to be restored during exit, as is desirable for all subroutines. Further, LEGQ8 must allow for the fact that the integrand evaluation subroutine will use these special registers without restoring.

(d) The arrangement of the symbolic program is something like the following:

1) Idenfication for assembly program and "SLC."
2) A main program which makes use of LEGQ8.
3) LEGQ8, which makes use of a subroutine, say SUBR.

4) SUBR, which happens to require the subroutine EXP.
5) EXP, which is self-sufficient.
6) Indication to end assembly and indication of the first instruction to be executed.

All pieces should be made available and assembled together by the STRAP assembler.

Method 1

```
'Main program for numerical integration.  Answer should be in ANS.
Main        LVI, $15, $+1.0; B, LEGQ8
            SIC, SUBR; NOP
            DD(N), 0.0                          'lower limit
            DD(N), 1.0                          'upper limit
            BEW, $; NOP                         'error measure
            ST(U), ANS; BEW, $                  'normal end of program
ANS         DRZ(U), (1)
'8-point Legendre-Gauss integration subroutine
'integrand evaluation subroutine with 1.($15) return must be provided by user, with
   effective address at 0.($15), lower limit must be at 1.($15) and upper limit at
   2.($15), both as floating-point numbers.
'the integration subroutine will return normally at 4.($15), with answer in $L.
'error return is 3.($15).
LEGQ8       SX, $2, LEGQ82;SX,$15,LEGQ8F
            LVE,$2,0.($15)
            SVA, $2, LEGQ8A
            SVA, $2,LEGQ8B
            DL(N), 1.($15)
            D-(N), 2.($15)                      'a-b
            E-I(U),1                            '-(b-a)/2
            SRD(N),LEGQ8P
            D+(N), 2.($15)                      '(b+a)/2
            SRD(N),LEGQ8Q
            LX, $2, LEGQ8I;L(U),LEGQ8Z
            ST(U), LEGQ8S;ST(U), LEGQ8T
LEGQ8L      DL(U), LEGQ8Q
            LFT(U), LEGQ8P
            *N+(N), LEGQ8X($2)                  '(b-a) z/2 + (b+a)/2
            LVI, $15, $+1.0
LEGQ8A      B, $                                'branch address changeable
            B, LEGQ8E; NOP                      'error
            ST(N), LEGQ8R                       'normal return from integrand subroutine
            DL(U), LEGQ8Q
            LFT(U), LEGQ8P
            *+(N), LEGQ8X($2)                   '-(b-a) z/2 + (b+a)/2
            LVI, $15, $+1.0
LEGQ8B      B, $                                'branch address changeable
            B, LEGQ8E; NOP
            +(N), LEGQ8R
            D*(N), LEGQ8W($2)
            D+(N), LEGQ8T
            D+(N), LEGQ8S
            ST(N), LEGQ8S
            SLO(U), LEGQ8T
```

```
            CB+, $2, LEGQ8L
            *N(N), LEGQ8P
            LX, $2, LEGQ82                  'normal return to main program
            LX, $15, LEGQ8F
            B, 4.0($15)
LEGQ8E      LX, $2, LEGQ82
            LX, $15, LEGQ8F
            B, 3.0($15)
LEGQ82      XW, 0                           'changeable
LEGQ8F      XW, 0                           'changeable
LEGQ8Z      DD(N),0.0
LEGQ8I      XW, 0,4,$
LEGQ8R      DR(N), (3)
LEGQ8S      SYN(N), LEGQ8R+1.0
LEGQ8T      SYN(N), LEGQ8R+2.0
LEGQ8P      DR(N), (1)
LEGQ8Q      DR(N), (1)
LEGQ8X      DD(N), .96028 98564 97536, .79666 64774 13627
            DD(N), .52553 24099 16329, .18343 46434 95650
LEGQ8W      DD(N), .10122 85362 90376, .22238 10344 53374
            DD(N), .31370 66458 77887, .36268 37833 78362
'end of LEGQ8 subroutine
'SUBR is a bona fide subroutine with 0($15) error exit and normal return 1.0($15).
SUBR        SX, $15, SAVE15
            ST(N), SAVEX
            LN(N), SAVEX
            LVI, $15, $+1.0;B, EXP          'go to EXP subroutine
            B, ERR; NOP
RTURN       ST(U), TEMP
            LVI, $15, $+1.0;B, EXP          'go to EXP subroutine
            B, ERR; NOP
            *(N), SAVEX
            E+I(U), 1
            ST(U), SAVEX                    '2*X*e**e**-X
            LN(U), TEMP
            +(N), WON
            SRT(N), $L                      'square root of 1-e**-x
            R/(N), SAVEX
            LX, $15, SAVE15; B, 1.0($15)    'normal return
ERR         LX,$15, SAVE15; B,0.0($15)      'error return
WON         DD(N), 1.0
SAVE15      XW, 0
SAVEX       DR(N), (1)
TEMP        DR(N), (1)
'EXP subroutine
(identical with a previous program)
```

Comments: (a) The instruction execution should begin with MAIN, which triggers all other programs. (b) The seemingly elaborate way of doing the problem is actually

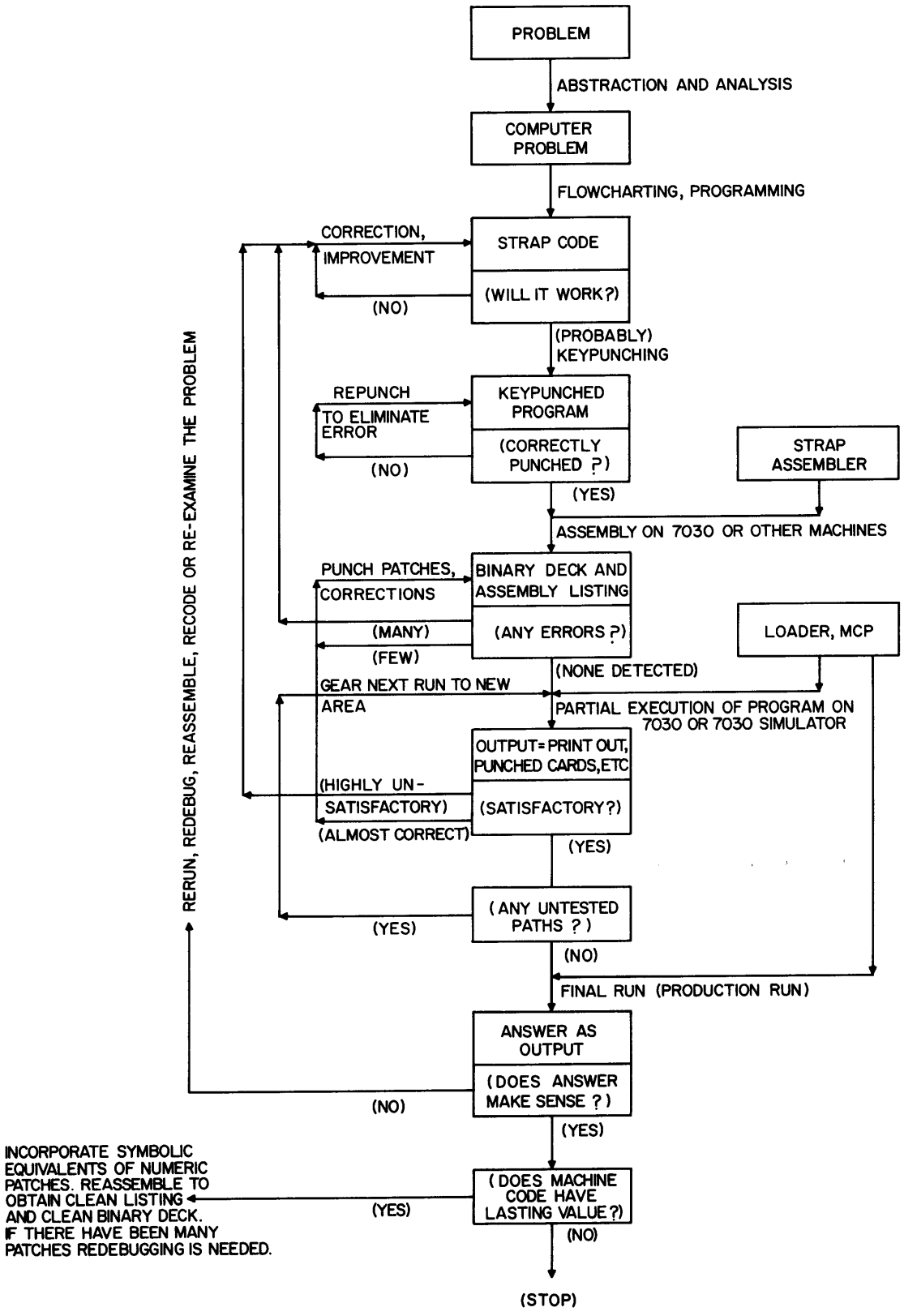
very easy to use, particularly if most of the subroutines are available. (c) For multiple integration the same integration subroutine can be assembled at different locations and one can be made subservient to the other. For example:

$$\int_A^B \int_C^D \phi(x,y)dydx = \int_A^B \left[\int_C^D \phi(x,y)\,dy\right] dx$$

$$= \int_A^B f(x)dx$$

and one of the integration subroutines is used to provide f(x). (d) Barring round-off errors, the 8-point Legendre-Gauss integration subroutine will yield exact results if f(x) is a polynomial in x of 15th degree or less. Otherwise the approximation amounts to an exact integration of a finite expansion of f(x) in terms of the orthogonal Legendre polynomials $P_k(x)$ up to and including k=7. (e) A discussion of errors in numerical integration is outside the scope of this work. It suffices to say that in case of suspicion of inaccuracy, the domain can be subdivided, and the numerical quadrature can be used for each subinterval to improve accuracy. This necessitates only a trivial change in the main program.

A1 PROBLEM SOLVING BY 7030 STRAP PROGRAMMING



A2. CHECK LIST FOR PROGRAM BEFORE ASSEMBLY

A2.1 General Format

Check for presence of PRNID, PUNID, SLC, and END. Make sure that the address of SLC is a true bit address with a decimal point.

A2.2 Symbol Definition

Are there undefined symbols? Circularly defined symbols? Multiply defined symbols?

A2.3 Instruction Format

Every operation field should be separated from the address field by a comma.
Look for missing right parentheses.
Look for missing quotation mark at the beginning of comment field.

A2.4 Nature of Instructions

Check integers to make sure they are not bit addresses with missing decimal point.

Half-word instructions cannot be addressed down to the bit level. Check particularly the address fields of V+, V+I, and floating-point operations.

Check VFL instructions for field length>64 or byte size>8.

Check TI, SWAPI, etc., for count exceeding 16.

The address field of immediate index arithmetic instructions cannot be indexed; the address field of CB, Bind and BB can only be indexed by $1. VFL immediate instructions cannot use progressive indexing.

Make sure that J fields are supplied in the following operations: CB, V+, and V+I.

A2.5 Loops and Paths

Visually trace through all the possible paths in the program.

Trace the entry into, and exit from loops.

If a loop is closed by a CB, make sure the index register "J" has a valid (non-zero) count field at the beginning.

Termination of a loop by BAE or BZAE after a floating-point compare is a dangerous practice, because of unforeseen roundoffs.

A2.6 Proofreading

After the program has been keypunched, produce a 407 listing and check the over-all alignment, particularly the location of the NAME fields. Proofread carefully, look for missing cards, mispunches, and off-punches.

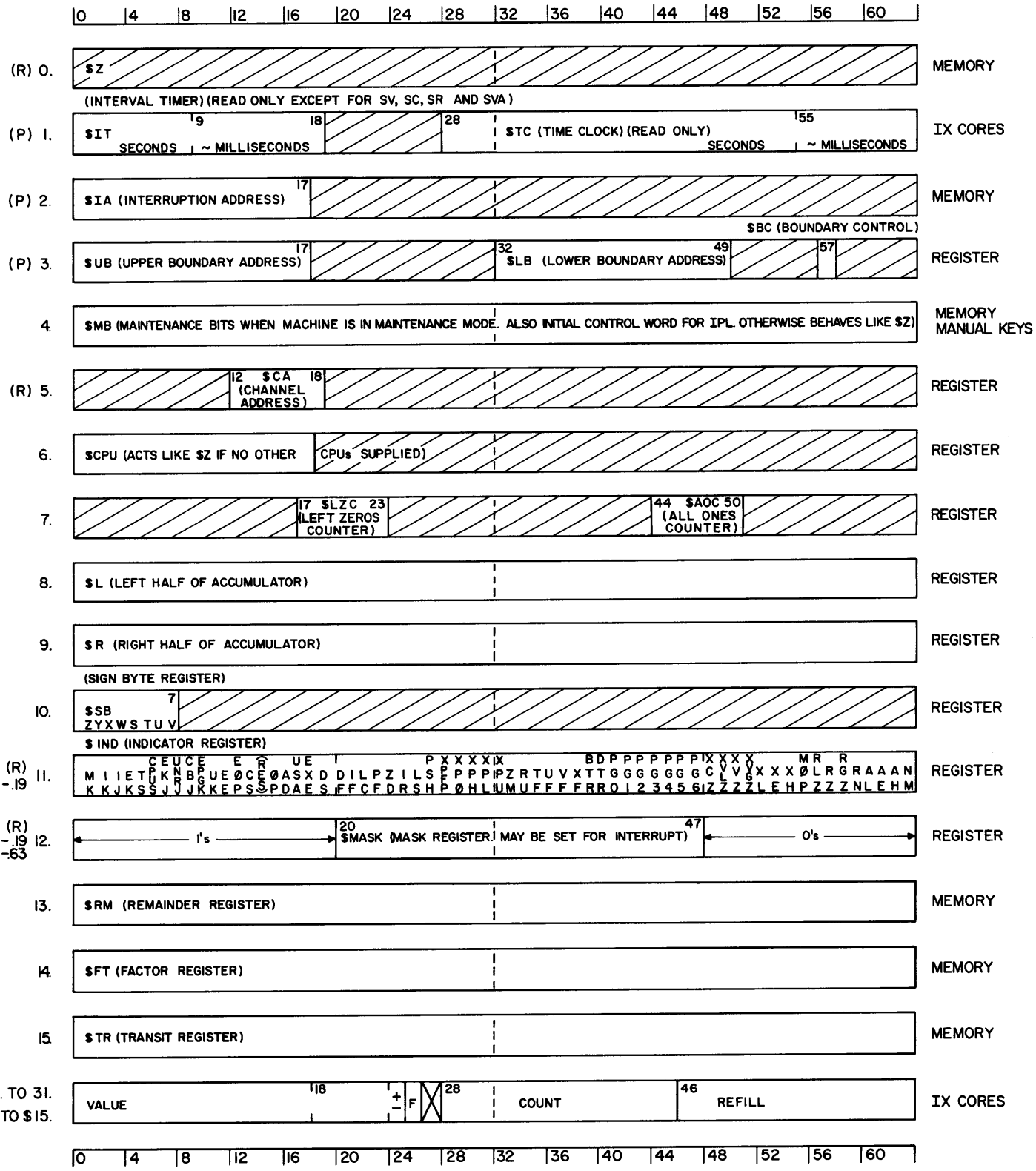
Character Code for Symbolic Decks

No Digit	(N) No Zone (Blank)	(Y) 12 Zone +	(X) 11 Zone -	0 Zone 0
1	1	A	J	/
2	2	B	K	S
3	3	C	L	T
4	4	D	M	U
5	5	E	N	V
6	6	F	O	W
7	7	G	P	X
8	8	H	Q	Y
9	9	I	R	Z
3,8		.	$	,
4,8	",@,-	) , ◊	*	(,%

Also; is defined to be equivalent to an (11, 0) double punch. On 407 listings this double punch is usually considered to be 0. On assembly listings the semicolon is replaced by a skip of the printer to the next line. On the keypunched card it looks like the Greek letter 0.

A3 7030 SPECIAL REGISTERS (LOCATIONS 0. THROUGH 31.)

(SHADED AREAS CONTAIN ZERO BITS)



(P): PROTECTED REGARDLESS OF BOUNDARY SPECIFICATIONS. (R): READ ONLY:

A4. OPERAND ADDRESSING IN 7030 PROGRAMMING

A4.1 Addressing Down to the Bit Level

The 7030 computer owes much of its power and programming convenience to the fact that it permits addressing down to the bit level. This facility is particularly evident in the variable field length instructions, but exists throughout the instruction set to a lesser degree.

It is therefore advantageous for the new 7030 programmer to familiarize himself with a general scheme for addressing fields and subfields. This scheme will also lend understanding to the more conventional types of operand addressing.

It is expedient to visualize consecutive full words in the addressable 7030 memory as lying end to end, starting with the full word bearing the lowest (word-) address on the extreme left. Further, if each bit is imagined to have a unit width, then any operand in a 7030 instruction can be represented in this continuous array by a string of consecutive bits characterizable by the extent of the interval (the field length) and the location of the leading bit.

The field length is usually understood in the instruction, but is explicitly given in variable field length instructions. In the latter case the manner of subdivision (the byte size) is also explicitly given.

A4.2 The Bit Address

The concepts of bit distance and bit position will be useful in the ensuing discussion.

A given bit is said to be at bit distance k relative to a reference bit if it is k units to the right of the reference bit. If the reference bit is the leading (i.e., leftmost) bit of a string, and if the bit in question forms part of the string, it is said to occupy the bit position k within the string. It is evident that the leading bit of a string is at bit distance zero relative to itself, and occupies bit position zero within the string. See Figure A4-1.

With the concept of bit distance established, it is now possible to locate any bit in the memory by its distance relative to, say, the leading bit of a reference full word. In other words, a bit address can be defined as a pair of integers, the one on the left giving the location of a full word, and the one on the right providing a bit distance from the leading bit of the full word.

Frequently when bit addresses are presented for human scrutiny, a "point" is used to separate the integers to enhance legibility, and leading zero bits are often suppressed.

Unless the choice of the reference full word is standardized, several different bit addresses may refer to the same bit. This is actually desirable in programming, the extra degree of freedom often can serve as a mnemonic aid.

If the reference full word is so chosen that it contains the bit in question, the bit distance becomes the bit position within the full word. The resultant unambiguous notation shall be called the standard bit address. Since the 7030 full word has a 64 addressable bits, the standard bit address is characterized by a bit distance part no higher than 63.

The 7030 is designed to accommodate a maximum of 2^{18} full words each of 2^6 programmable bits, and 18+6 = 24 bits suffice for the standard binary bit address. Its octal equivalent has 6+2 = 8 octal digits, each of which corresponds to 3 consecutive bits. The bit configuration is less explicit when other number radices are employed.

The standard bit address bears a strong relationship to machine function. For instance, within the machine, references to the main memory and the index memory are always in terms of full words. The bit position part of the standard bit address is used to select a field contained in one or two consecutive full words.

The following are examples of bit addresses and standard bit addresses in decimal and octal radices. The last two entries in each line below are standard bit addresses. Note the "point" is not used as the conventional separator between integer and fraction, and $3.4 = 3.04 \neq 3.40$. See Figure A4-2.

$(0.64)_{10} = (0.100)_8 = (1.00)_8 = (1.00)_{10}$ (leading bit of word 1).

$(4.123)_{10} = (4.173)_8 = (5.73)_8 = (5.59)_{10}$ (bit 59 of word 5).

$(32657.400)_{10} = (77621.620)_8 = (77627.20)_8 = (32663.16)_{10}$ (bit 16 of word 32663).

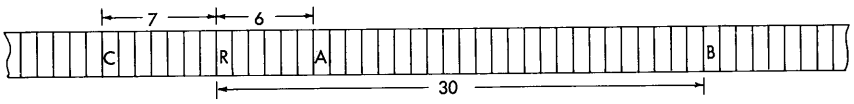
$(1.8160)_{10} = (1.17740)_8 = (200.40)_8 = (128.32)_{10}$ (leading bit in second half of word 128).

A4.3 The Numeric Address Field in an Instruction

In the 7030 an operand is located by the standard bit address of the leading bit. The nature of the instructions, however, imposes certain restrictions on the operands such that their standard binary bit addresses frequently contain at least a predictable number of trailing zeros. For example, floating-point operands must begin at a full word boundary, and the last 6 bits in the standard binary bit address therefore must be zero. Similarly a branch address must refer to the beginning of a full word or a half word, and the last 5 bits in the standard bit address are zeros. The number of trailing zero bits in variable field length instructions is, however, not predictable.

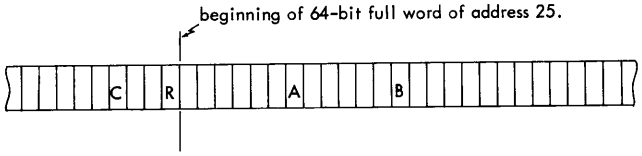
To enhance the information content of 7030 instructions, the predictable trailing zero bits are omitted from the instruction numeric address field, leaving more room for the operation code.

The numeric address field in an instruction is therefore an abbreviation of the full-scale (24 bit) numeric address. The size of the numeric address field is dependent on the instruction class, as seen from bits 24 through 27 of the half-word containing the address in question : (X stands for any value).



Distance relative to reference bit labelled R
for bit labelled A = 6 bits
for bit labelled B = 30 bits
for bit labelled C = -7 bits
for bit R itself = 0 bits

FIGURE A4-1. BIT DISTANCE



bit address for bit labelled A is $25.7)_{10} = 24.71)_{10}$, etc.
bit address for bit labelled B is $25.13)_{10} = 24.77)_{10}$, etc.
bit address for bit labelled R is $25.0)_{10} = 24.64)_{10}$, etc.
bit address for bit labelled C is (in decimal)
25.0 - .3 = 24.0 + (.64 - .3) = 24.61 = 23.125, etc.

FIGURE A4-2. BIT ADDRESS

(Bits 24 through 27)

1000	24 bits	(left address of full word instruction)
XX10	18 bits	(floating-point address; right address of T, SWAP)
Others	19 bits	(mainly instruction arithmetic unit instructions)

The full-scale numeric address can be formed by right-appending a sufficient number of zero bits to the numeric address field. Being an unsigned quantity, it is considered positive whenever a sign is called for.

A4.4 Indexing

Most numeric addresses referred to by 7030 instructions could be modified through indexing to produce effective addresses. Further, in variable field length (VFL) instructions the field length, byte size and offset could also be indexed producing the corresponding effective quantities. A discussion of VFL indexing will be deferred to the next section.

A (non-VFL) instruction may occupy a half word or two consecutive half words. Each half word is indexed separately. The most common type of indexing is the I-type, in which the last four bits (the I-field) of the half word provides indexing information. In K-type indexing, as in CB, Bind and second half of BB instructions, the last bit (the K-field) provides indexing information. A zero I- or K-field specifies no indexing. Otherwise the numeric contents of the fields indicate the index register to be used in indexing. The index register $13, for instance, is designated by an I-field contents of 13. K-type indexing therefore implies either no indexing at all, or indexing by means of $1. No indexing is permitted for immediate index arithmetic instructions.

If no indexing is called for, the 24 bit unabbreviated numeric address, affixed with a trailing zero sign bit, becomes the (25-bit) effective address. In the case of indexing, the effective address is created by adding the (positive) numeric address to the proper index value field. Overflow beyond the leading bit position is ignored.

The actual address meaningful to the execution of the instruction is never the full effective address. The sign bit is invariably ignored, and only the leading 18, 19 or 24 bits are meaningful. The number of meaningful bits is usually the same as the size of the numeric address field, although for LX, SX, Z, R, RCZ, T and SWAP only the leading 18 bits in each half word are meaningful.

In the machine there is no execution address per se; the trailing bits of the effective address are simply ignored. From a programming point of view it may be convenient to define an execution address as the 24-bit quantity formed from the effective address by first ignoring the sign bit then replacing the low order bits (if any) by zeros.

The entire process from numeric address field to execution address is summarized in the accompanying diagram.

It is noteworthy that the low order non-zero bits in a specified index field may influence the higher order bits of the effective address and hence the execution address. This occurs only when the index value field is negative; the indexing action is then a subtraction.

A4.5 Indexing VFL Instructions

In variable field length instructions (including logical connective instructions), every instruction occupies two half-words, each with its own I-field. The use of the first (left) I-field is dependent on the P-field (bits 32-34) in the second half-word, and the second I-field is used to create effective offset, effective byte size and effective field length.

The numeric address field, as revealed by bits 24-27 of the VFL instruction, has 24 bits. Normal indexing by the first I-field is specified if P contains $(000)_2$ (direct addressing) or $(100)_2$ (immediate addressing). Otherwise progressive indexing is specfied:

000	direct addressing (standard indexing)
001	V+I (progressive indexing)
010	V+IC (progressive indexing)
011	V+ICR (progressive indexing)
100	immediate addressing (standard indexing)
101	V-I (progressive indexing)
110	V-IC (progressive indexing)
111	V-ICR (progressive indexing)

In progressive indexing, the effective address, hence the execution address is completely furnished by the specified index value field. The numeric address field plays no role for operand designation, but instead is used to increment (+) or decrement (-) the designated index value field algebraically, in anticipation of the next application. The counting and refilling (on zero count) processes can be induced by proper specification. Progressive indexing with a zero I-field is not admissible.

An instruction with progressive indexing thus behaves like two consecutive instructions. The first is an arithmetical-logical instruction with effective address created with a zero numeric address. The second is not unlike an immediate index arithmetic instruction, with the numeric address modifying the specified value field. Note, however, that in regular immediate index arithmetic instructions, the numeric address field is at the half word level, i.e., has only 19 bits. In progressive indexing the index value field is modified down to the bit level. Further, the index register needs only to be specified once.

Example: The instruction

```
CT1101(BU,25,3)(V+IC), 10.57($2)
```

behaves like the collection

```
{ CT1101(BU,25,3),0($2)
{ (V+IC),$2,10.57
```

Another unusual feature in VFL instructions is the indexing of the second half-word. In STRAP coding the second I-field is placed immediately after the offset, although in general the field length and byte-size can be modified besides the offset.

The procedure is extemely similar to conventional address modification. The bits 24-27 of the second half-word specifies a 19-bit numeric "address" field. The latter is extracted, assumed positive, appended by zeros, and added to the prescribed index

value field to produce effective field length, byte-size and offset. There are two simple departures from conventional indexing:

a. The P-field is pre-copied into a special register, and is therefore unaffected by indexing. The "original" P-field, however, does participate in the indexing action and can affect the field length, byte-size and offset if the specified index register is negative and large in magnitude.

b. The indexing of the second half-word precedes any index modification by progressive indexing. Thus if both the first and second I-fields refer to the same index register in an instruction with progressive indexing, the unmodified index value field will be used not only as the effective address, but also as the modifier of the second half-word.

A4.6 Immediate Addressing, and Direct Addressing

The most common mode of operand specification by an instruction is the direct addressing scheme, in which the execution address refers to the leading bit of the operand. In branch type instructions the branch address refers to the leading bit of the leading half-word of a new instruction sequence.

In the 7030, information transmittal to and from the main memory or index register memory is in terms of full words, and the leading 18 bits of the execution address have particular significance. Bit position information is used for field selection in the arithmetical details. For VFL instructions the operand may overstep the memory word boundary. This is automatically detected from field length specification during the initial decoding, and the transmittal of two full words will be effected.

If the instruction specifies immediate addressing, the execution address is used as the operand itself, and no memory data fetch is required. This feature is available in many index arithmetic and VFL instructions not involving storing into memory. In the 7030, immediate index arithmetic instructions do not possess indexable addresses; and while the VFL immediate address is indexable in the normal manner, progressive indexing with immediate addresses is not prescribed. This last is due to the fact that immediate VFL instruction require one $((100)_2)$ configuration of the P-field, and a progressive indexing instruction requires another.

A4.7 Indirect Addressing, LVE, EX and EXIC

Indirect addressing is characterized by the fact that the execution address of an instruction may refer not to an operand, but to another address which enables the ultimate location of the operand. An instruction with this property is a link in the operand locationing chain.

It is possible to systematize the three modes of addressing in the 7030 in the following manner:

a. An instruction or operand will be collectively designated as levels.

b. The levels are given a level count. The original instruction could be called the zeroth level.

c. The level may contain the operand, and be the terminal level in the chain; or it may contain the execution address of the (k+1)st level and be a propagation level.

d. In immediate addressing, the zeroth level is the terminal level: the execution address serves as the operand, the address of this operand is the address of this zeroth level.

e. In the case of direct addressing the 1st level is the terminal level. The execution address of the original instruction refers to the operand.

f. In the case of indirect addressing, there may be any number of propagation levels each of which is identifiable by the computer. The terminal level is characterized by the fact that it is a non-propagational member of the sequence of levels.

In the 7030, three instructions are capable of indirect addressing:

a. Load value effective (LVE). The operand is the terminal level effective address, to be loaded into the value field of the index register (specified by the J-field) of the original instruction possibly for future address modification. Any LVE instruction, regardless of J-field, may serve as propagation level.

b. Execute (EX) and execute indirect and count (EXIC). The operand at the terminal level is an instruction to be executed in its entirety without changing of the instruction counter. The control of the machine is thus lent to the terminal level. The level immediately referred to by an EXIC level is automatically a propagation level, and in addition the numeric address field is increased by an amount equal to the length of the next level (treated as an instruction), whether the latter is a propagation level or not. Any EX or EXIC instruction may serve as propagation level.

To avoid unending chaining of propagation levels, the following provision is made. Whenever the instruction counter is found to have been unchanged for longer than one millisecond (corresponding to several hundred propagation levels), the remaining part of the LVE, EX and EXIC instructions will not be performed, and the $USA indicator (unending sequence of addresses) will be turned on.

For EX and EXIC there is the additional danger of loss of program control. This is resolved by the following devices:

a. If the execution of the instruction at the terminal level tends to alter the instruction counter contents (as in a successful branch), the $EXE indicator (execution exception) will be turned on.

b. Except for the fetching of the first pseudo-instruction counter level in EXIC, which is subject to the usual restrictions, the interruption system will be active for the duration of the EX and EXIC chain, whether the disabling of the interruption system has been prescribed or not. All interruptible conditions will cause an

interruption when the chain is completed (i.e., when the terminal level instruction is executed), terminated or suppressed. The temporary enabling of the interruption system means that, among other things, address monitoring is in full force, and interruptions can be caused by $EXE and $USA indicators.

A4.8 Addressing in STRAP Coding

Any of the following may appear in the address field of a STRAP-coded instruction:

a. A standard bit address (A.B , $B \leq 63$)
b. any bit address (C.D , D need not be less than 64)
c. an integer
d. symbolization of any of the above
e. simple functions of any of the above

During the assembly process whatever is in the address field is converted into standard form suitable for the numeric address field of the (binary) machine instruction. The existence of a variety of formats is clearly a burden for the assembler. It is also a great convenience for the programmer.

The STRAP assembler normally assumes a decimal radix. Any radices up to 16 are permitted by specification in STRAP II. For STRAP I the allowable radices are 2 through 10, and 16.

A4.8.1 Integer Addresses

The bit address equivalent of an integer address is decided by the environment, which defines a subfield. The integer address is treated as an integer of the subfield (e.g., the non-zero bit for the integer 1 would occupy the rightmost position of the subfield). Then the left margin of the subfield is placed in juxtaposition with the leading bit of the address field, leading to a bit address identification.

Where the environment seems to suggest more than one subfield, as is frequently so in immediate type instructions, the smallest one is to be adopted.

Instruction	Subfield Size
Floating point	at most 18 bits
VFL	at most 24 bits
most others	at most 19 bits
VFL immediate	at most the field length
E ± I, SHF	11 bits + sign
immediate index arith.	at most the size of the field in question in the index register

Examples

SRT(N), 215	is equivalent to	SRT(N),215.0
M + 1 (BU,63,8),17	is equivalent to	M+1 (BU,63,8),0.17
CBRH, $4,28	is equivalent to	CBRH, $4,14.0
C+, $3,13	is equivalent to	C+,$3,6.32
E -(N),17	is equivalent to	E -(N),17.0
*I (BU,7,8), 127	is equivalent to	*I(BU,7,8),260096.0 ((127.0).2^{11}=260096.0)

SHFR,2 is equivalent to SHF, 320.0 ($(5.0)2^6 = 320.0$).
V+I,$3,13 is equivalent to V+I,$3,6.32 (25 > 19)
C-I,$3,13 is equivalent to C-I,$3,13.0 (18 < 19)

It is readily seen that integer addresses are extremely convenient for immediate instructions, where the bit address equivalents are not only hard to obtain, but has no mnemonic value whatever. In most floating-point instructions integer addresses and bit addresses are almost identical in appearance, and the former could be considered as an abbreviation of the latter. For other instructions integer addresses may be confused with bit addresses with a missing "point," and their use may complicate the debugging process.

Note that all integers in STRAP coding are considered to be integers of some subfield. Integer addresses are distinguished only by the fact that more than one subfield is usually present. The use of integers in the specification of count fields, offset fields, etc., is usually straighforward. In "parenthetical integer entry" the subfield size is explicitly given, and the information "OR"ed in.

A4.8.2 Address Symbolization and Functions of Addresses

Either a bit address, or an integer, can be symbolized, i.e., represented by a collection of alphanumeric characters.

Simple functions of bit addresses, integers, symbolized bit addresses and symbolized integers can represent an address. The latter can also be symbolized. This process may be repeated a number of times in STRAP coding.

Some restrictions are given below:

a. In STRAP I no multiplication or division can be performed on any address or symbolization thereof. This restriction is lifted in STRAP II.

b. Any symbol must ultimately be definable in terms of bit address equivalents.

c. A "point" placed in front of or behind a symbolized integer converts it into a bit address; a "point" placed in front or behind a symbolized bit address, however, has no effect on the latter.

E.g., If M has been defined as 5, and JOE has been defined as 817.35, then
.M = 0.5
M. = 5.0
.JOE = JOE. = 817.35
M.M = 5.5
JOE.JOE = 817.35 + 817.35 = 1634.70

d. During the assembly if the evaluation of the bit address equivalent of an address yields a negative answer, it is replaced by its two's complement, which amounts to replacing the "bit address" -N by 262144.0 -N.

A4.8.3 Associated Properties of a Symbolized Address

Aside from being a valuable mnemonic aid, symbolic addresses can supply missing data. During STRAP assembly if the information regarding data description of the

operation code of an instruction is inadequate, the assembler may examine the defining statement (s) for the symbolic address to fill in the details. If several symbolic addresses are present, the rightmost is examined first.

E.g.

```
        +, DOG + CAT
CAT  DD(D,27,6),32
DOG  DD (BU,12,7), 738
```

Leads to +(D,27,6), DOG + CAT

If the rightmost symbolic address possesses no meaningful data description, the next-to-the-rightmost symbolic address will be used, and so on. If all symbolic addresses are exhausted without a proper data description, STRAP will make the following plausible guesses to produce a reasonable program.

a. E ± I, E ± , SHFR, SHFL: assumed unnormalized.
b. Other unambiguously floating-point instructions: assumed normalized.
c. If instruction can be interpreted as either VFL or floating-point, it will be assumed to be VFL.
d. All VFL immediate instructions: assumed (BU,24,8).
e. All other VFL instructions: assumed (BU,64,8).

A5. MACHINE HANDLING OF FLOATING POINT EXPONENT FLAGS IN THE 7030

A5.1 Exceptional Floating-Point Quantities

Exponent overflow and underflow occur only infrequently in most floating-point computations. In machines of earlier design, the "overflowed" and "underflowed" numbers have the appearance of normal quantities, and further operations tend to lead to untraceable contamination of the results. The conventional way of circumventing this difficulty is to test for the exceptional events from time to time.

Some machines now have a "floating trap mode" feature which automatically interrupts the normal instruction sequencing immediately after an exceptional event, without the need for test instructions. A wide choice of interrupt conditions (XPFP, XPO, XPH, XPL, XPU) is available on the 7030, enabling a firm control on the quantities used in floating-point instructions. Interruption feature, however, tends to treat exceptional events equally and is not capable of knowing the consequences of these events without elaborate programming.

On the other hand, if the "overflowed" or "underflowed" quantities, which are responsible for the exceptional events, are themselves clearly labelled, if the numbers contaminated by these labelled numbers are also labelled in a consistent manner, it would be possible to perform an entire computation without any test instruction nor interruption. In this scheme, drastic action would be not needed unless part of the results bear the "exceptional quantity" label.

In the 7030 the exceptional number is labelled by a "1" bit occupying the leftmost (exponent flag) position of the exponent field. An exceptional number therefore appears to be a number with an extremely large exponent magnitude. The consistent rules governing the generation, propagation and disappearance of the exponent flag are reminiscent of algebraic operations involving infinite and infinitesimal quantities.

In the following EF represents the exponent flag, ES the exponent sign.

EF = 1 signifies a very large floating point exponent magnitude. If EF = 1, ES = 0, the magnitude of the floating point number is extremely large ($\geq 2^{1023} \sim 10^{308}$), and may be symbolized by ∞ (XFP case).

If EF = 1, ES = 1, the magnitude of the floating point number is extremely small, and may be symbolized by ϵ (XFN case).

If EF = 0, the number is said to be normal, and will be represented by the symbol N.

The sign bit (bit 60) of the floating point number retains its normal meaning in all cases.

The following scheme is designed to disallow the loss of EF bit due to irretrievable overflows.

A5.2 Generation of Exceptional Quantities

In floating point operations involving normal numbers only, EF behaves like an extension of the regular 10-bit exponent magnitude field, and will be turned on in the result if the expected answer has an exponent either greater than 1024 or less than -1024. An exponent overflow is said to have occurred in the former case, rendering $XPO = 1. In the latter case an exponent underflow is said to have occurred, and $XPU will be set to 1. In D/, $RU may be set to 1. In either case, an exponent flag is said to be generated.

Other operations will proceed normally for all generated EF cases except in the following situations which might otherwise generate exponent overflow beyond EF:

a. Multiplications which lead to generated ϵ results prior to any normalization. The normalization and noisy mode, if stated, will be suppressed. E+, E+I instructions behave like multiplications.

b. Divisions where prenormalization of the two operands yields an N and a generated ϵ. The quotient fraction is developed normally, but the quotient exponent will be either that of ϵ (case of small dividend), or that of 1/ϵ (case of small divisor).

The following table gives the conditions and the apparent range of normal as well as exceptional numbers, when EF is imagined to be an extension of the exponent magnitude field.

Condition of FP Number	Symbol	EF	ES	Fraction Sign	Apparent Range for Normalized Fraction
XFP, +	$+\infty$	1	0	0	$\geq 2^{1023}$
Normal, +	+ N	0	0,1	0	$< 2^{1023}, \geq 2^{-1024}$
XFN, +	+ ϵ	1	1	0	$< 2^{-1024} > 0$
XFN, -	- ϵ	1	1	1	$0 > -2^{-1024}$
Normal, -	- N	0	0,1	1	$< -2^{-1024}, > -2^{1023}$
XFP, -	$-\infty$	1	0	1	$\leq -2^{1023}$

A5.3 Exceptional Number Arithmetic

In floating-point arithmetic involving numbers with EF = 1, the mathematical laws concerning extremely large and extremely small numbers apply where the results are unambiguous. If the outcome is indeterminate in a strict mathematical sense, the ambiguity is resolved in the machine by the choice of ∞, producing the most alarming situation possible:

$\infty + \infty = \infty$;
$\infty \pm N = \infty$;
$\infty \pm \epsilon = \infty$;
$\epsilon \pm N = N$;
$\epsilon \pm \epsilon = (\pm)\epsilon$;

$\infty * (\pm \infty) = \pm \infty$;
$\infty * (\pm N) = \pm \infty$;
$\epsilon * (\pm N) = \pm \epsilon$;
$\epsilon * (\pm \epsilon) = \pm \epsilon$;

$\infty / (\pm N) = \pm \infty$;
$\infty / (\pm \epsilon) = \pm \infty$;
$\epsilon / (\pm \infty) = \pm \epsilon$;
$\epsilon / (\pm N) = \pm \epsilon$;
$\pm N / \infty = \pm \epsilon$;
$\pm N / \epsilon = \pm \infty$;

$\sqrt{\infty} = \infty$;
$\sqrt{\epsilon} = \epsilon$

The following are resolved ambiguous cases:

$\infty - \infty = \infty$; $\infty * (\pm \epsilon) = \pm \infty$; $\infty / (\pm \infty) = \pm \infty$;
$\epsilon / (\pm \epsilon) = \pm \infty$;

For details, see A5.8. Note that normal answers are obtained only by special ϵ + N operations, and exponent overflows beyond the EF position which may yield harmless-looking results are prevented from occurring.

A5.4 Propagation of Exponent Flag

In operations other than K, KMG, KMGR, and KR, if both the result and at least one of the operands are in the ∞ range, an "exponent flag positive" condition is said to have been propagated, and $XPFP is set to 1. The propagation of ϵ conditions does not lead to special indicator settings.

A5.5 Comparison Involving Exceptional Quantities

All ∞ are treated as equal in magnitude in K, KMG, and KR; all ϵ are likewise treated as equal in magnitude.

A5.6 Approximation of the True Floating Point Zero

The true floating point zero is approximated by an ϵ. If a floating point zero is requested of STRAP, what appears to be $0 * 2^{-1024}$ will result from the compiling.

A5.7 The "Zero Multiply" Indicator

$ZM cannot be turned on if the result of the multiplication is ϵ with zero fraction.

A5.8 Summary of Floating Point Arithmetic with Exceptional Operands
(Only Exponents are Shown in Equations Below.)

A5.8.1 Addition, Subtraction, Load, Store, and SLO. (Result may be N)

$\infty_1 + \infty_2 = \infty_1$ or ∞_2#; $\epsilon_1 + \infty = \infty$;
$\infty_1 + N = \infty_1$; $\epsilon_1 + N = N$;
$\infty_1 + \epsilon = \infty_1$; $\epsilon_1 + \epsilon_2 = \epsilon_1$ or ϵ_2#;
$\infty_1 - \epsilon = \infty_1$; $\epsilon_1 - \epsilon_2 = \epsilon_1$ or $-\epsilon_2$#;
$\infty_1 - N = \infty_1$; $\epsilon_1 - N = -N$;
$\infty_1 - \infty_2 = \infty_1$ or $-\infty_2$ #; $\epsilon_1 - \infty = -\infty$.

Fraction arithmetic: suppressed. Normalization and noisy mode: allowed only if pre-normalized answer is normal.

#Whichever has the higher exponent; or if the exponents are equal, whichever is from the accumulator.

F+ behaves like NOP for accumulator being ∞ or ϵ, since the memory fraction is given the accumulator exponent.

A5.8.2 Multiplication, E+ and E+I. (Results Always ∞ or ϵ.)

$\infty_1 * \infty_2 = \infty_1$ or ∞_2#	$\epsilon_1 * \infty = \infty$
$\infty_1 * N = \infty_1$	$\epsilon_1 * N = \epsilon_1$
$\infty_1 * \epsilon = \infty_1$	$\epsilon_1 * \epsilon_2 = \epsilon_1$ or ϵ_2#

Fraction arithmetic: allowed to proceed. Normalization and noisy mode: suppressed.

In *+, where accumulator does not contain operands, whichever is from memory; otherwise whichever is from the accumulator.

A5.8.3 Division (Result Always ∞ or ϵ.)

$\infty_1/\infty_2 = \infty_1$;	$\epsilon_1/\infty_2 = \epsilon_2 (= 1/\infty_2)$;
$\infty_1/N = \infty_1$;	$\epsilon_1/N = \epsilon_1$;
$\infty_1/\epsilon_2 = \infty_2 (= 1/\epsilon_2)$;	$\epsilon_1/\epsilon_2 = \infty_2 (= 1/\epsilon_2)$;
$\infty_1/\infty_2 = \infty_1$;	$\infty_1/\epsilon_2 = \infty_2 (= 1/\epsilon_2)$;
$N/\infty_2 = \epsilon_2 (= 1/\infty_2)$;	$N/\epsilon_2 = \infty_2 (= 1/\epsilon_2)$;
$\epsilon_1/\infty_2 = \epsilon_2 (= 1/\infty_2)$;	$\epsilon_1/\epsilon_2 = \infty_2 (= 1/\epsilon_2)$.

Fraction arithmetic: allowed to proceed. Normalization and noisy mode: suppressed. Operations involving ϵ or ∞ will be treated as unnormalized. Remainder: Exponent same as that of dividend, no normalization allowed.

A5.8.4 Square Root. (Result Always ∞ or ϵ.)

$\sqrt{\infty_1} = \infty_1$
$\sqrt{\epsilon_1} = \epsilon_1$

Fraction arithmetic: allowed to proceed. Normalization and noisy mode: suppressed.

A5.8.5 Shift Fraction

ϵ and ∞ behave normally, since the exponent is unaltered.

A6. NOISY MODE IN 7030 PROGRAMMING

A6.1 Purpose of Noisy Mode

The purpose of the noisy mode is to allow the 7030 to perform its own error analysis in the crucial area of significance loss in normalized floating-point arithmetic.

Essentially the same computing algorithm for the solution of a problem can be pursued twice on the machine, once in "normal" mode and once in noisy mode. During the computation the low order fraction bits are affected differently in each case, the difference being particularly noticeable on normalizing left shifts. When the results are contrasted with each other, if the relative discrepancy is 2^{-n}, then probably the "normal" result has a relative error of 2^{k-n}, the odds being something like 2^k to 1 in favor of this interpretation (and against fortuitous agreement).

In the 7030 the noisy mode is activated only when the indicator bit $NM equals 1, and only for normalized floating-point operations. When normalization is suppressed due to exponent flag conditions (see A6.6), noisy mode will be inoperative. For convenience, we shall speak of the influence due to noisy mode as noise.

A6.2 First Order Noise

An operand may be right-appended by 48 identical bits at the beginning of an operation, to produce a double-length fraction. We may call these "d" bits.

d = 1 if and only if

a. normalized operation is specified (and not suppressed).
b. $NM = 1;
c. the operand is one of the following:

    1) an operand in (single) LOAD type instruction: L, LWF, LFT;
    2) an operand in ST instruction (NOT SRD nor SLO);
    3) the divisor in /, R/, and D/;
    4) the dividend in / and R/;
    5) the unshifted operand prior to arithmetic action in the following single operations: +, M+, +MG, M+MG; K, KMG, KMGR, KR.

d = 0 otherwise.

The unshifted operand in operations described in (5) is the operand with the higher exponent, or if the exponents are equal, the operand from the accumulator.

The d bits, being second order quantities, may influence the first order part (first 48 bits) of the result fraction through post-normalization and/or arithmetic action. The minimum noticeable relative error due to d bits is 2^{-48}; the maximum is just below 1/2.

We shall speak of first order noise as one which can create a minimum noticeable relative error in the first order part (the first 48 bits) of the result fraction, and define second order noise as one which creates a minimum noticeable relative error in the second order part (the second 48 bits) of the (double-length) result fraction. In the 7030 computer the d bits produce only first order noise.

A6.3 Second Order Noise

When a double-length fraction undergoes left shift (in, for example, post-normalization), the positions left vacant are filled in by another kind of identical bits. We shall call them "d_2" bits.

$d_2 = 1$ if and only if

a. normalized operation is specified (and not suppressed);
b. $NM = 1.

$d_2 = 0$ otherwise.

In all operations save one, the d_2 bits produce only second order noise. In the cases where d and d_2 are both present, the result fraction is invariably truncated to 48 bits, revealing only the effect due to d bits.

It must be noted that second order noise is not necessarily small. The largest possible relative error caused by it is the same as that for first order noise, namely just below 1/2. This occurs when a 96-bit fraction before post-normalization has all bits equal to zero except the last bit. Ninety-five d_2 bits will be shifted in.

A6.4 Machine Instruction and Noisy Mode

A6.7 shows the pertinent noisy mode features of floating-point operations.

It is noteworthy that all but one double operations possess second order noise. The exception is D/, which has first order noise through divisor preshifting. On the other hand, the "single" operation * possess only second order noise. The operation *+ has a second order noise if the preceding LFT operation did not introduce first order noise.

SRD and SRT are noiseless operations.

In SLO the low order fraction is left-appended by 48 high order zero bits to produce a 96 bit fraction. This latter is then shifted left at least 48 places, shifting in d_2 bits. Second order noise on the second order fraction thus behaves like first order noise on an ordinary (single) fraction.

Noise in /, R/, and D/ is introduced in both the divisor (always by d bits) and the dividend (d bits for /, R/; d_2 bits for D/). The quotient never needs further normalizing left shifts and the normalization of the remainder is noiseless. First order noise in D/ is desirable if the quotient is to be single precision (say after a rounding operation), but not if truly double precision quotient is required.

It is possible to produce noisy results without any normalizing left-shifts not only from divide-type operations, but in ADD-type operations as well. The 48 d bits may simply create a carry into bit 47 of the fraction during the addition process.

A6.5 Programming Significance

All digital computers have a finite word-length. In normalized floating-point operations the post-normalizing left shifts introduce bits through the right-boundary of the fraction. With few exceptions (some to be mentioned below), the programmer has no idea what these bits ought to be, and he is unwilling to or has no way to find out.

Shifting in all 1's as in noisy operations, very probably introduces errors. It is almost equally probable that errors of a similar magnitude are introduced by the alternative strategy of shifting in zeros. In either case bias is introduced.

The purpose of the noisy mode is to bias the results in a manner as opposite to "normal" as possible for the digits known to have no numerical significance, yet without destroying the digits valid for the particular machine instruction.

In computations involving integers and simple numbers, extremely frequently the result fraction is known to be exact, to be followed by an infinite number of zero bits. It should be evident that such exact answers can be corrupted by noisy mode. $NM should be off, or unnormalized operations should be prescribed.

In programmed double-and multiple-precision arithmetic, the addressed operand may have one or more well-defined lower order part. The use of noisy mode amounts to a redefinition of the lower order part, and extreme caution has to be applied, except perhaps in dealing with the lowest-order fraction.

In programmed double-precision arithmetic second order noise is always permissible, but first order noise should affect only the less significant part of the fraction. The use of LFT(N) as a prelude to *+, and D/(N) for unnormalized first order operands thus should be discouraged; it is much safer to employ the unnormalized counterparts to these operations. It is easy to introduce second order noise through other operations in the instruction sequence.

Under special circumstances, normal and noisy compare type operations may yield different indicator settings (sometimes even for the same two numbers). The user of floating point compare operations should know always that, except for the "exact" operations he is comparing numbers affected by errors, and due allowance must be made for this, whether noisy mode is used or not.

A6.6 Suppression of Normalization

In the great majority of cases normalization, if specified in an instruction, will proceed. The exceptions occur only because of the appearance of exponent flag.

Normalization (and therefore noisy mode) will be suppressed in the following cases:

a. For instructions involving only one operand, if the operand prior to the normalizing shift is either an ∞ (XFP case) or an ϵ (XFN case).

b. For instruction with two operands, neither of them are ∞ or ϵ:

1) instructions of * type, if the product before normalization is an ϵ.
2) instructions of / type, if the operands after prenormalization contain one ϵ and one N (i.e., no exponent flag). (This case does not influence noisy mode in any way.)

The suppression of normalization in this category is to prevent the loss of EF due to double underflow.

c. For instructions with two operands, at least one of which is either ∞ or ϵ: if the result is not an N before the post-normalization. The result is an N only in the case of ϵ + N, and normalization here, if specified, will proceed.

A6.7 Summary of Behavior of Normalized Floating Point Instructions in Noisy Mode

	Right-Appendage by 48 d Bits (prior to any ±)	Post-Shifting into Bit 95 by d_2 Bits	Order of Noise and Other Comments
Add Type Operations			
+, M+, +MG, M+MG	yes, on unshifted operand	yes (no effect)	1
L, LWF	yes	yes(no effect)	1
ST	yes	yes (no effect)	1
K, KMG, KMGR, KR	yes, on unshifted operand	(no post-shifting)	1
LFT	yes	yes (no effect)	1. Has bearing on *+.
SRD	no	yes (no effect)	Noiseless
SLO	no	yes, before any shifting	1
Multiply, Divide & Root			
*	no	yes	2
/, R/	yes, both divisor and dividend	yes (no effect on operands. No post-left-shift for quotient.)	1
SRT	no	yes (no effect)	Noiseless
Double Operations			
D+, D+MG, F+	no	yes	2
DL, DLWF	no	yes	2
D*, *+	no	yes	2
D/	yes, on divisor preshift	yes, on dividend preshift. No post-left-shift for quotient. Yes (no effect) on divisor preshift.	1. No additional noise introduced in remainer normalization.
Others			
E+, E+I	no	yes	2
SHF	no	no	Noiseless

A7. MAJOR UNITS IN THE 7030

A7.1 Core Storage

Six units of 16K extended words each for the Los Alamos configuration. Each extended word contains a full word with 64 information bits, plus 8 error-check bits.

Controlled by the storage bus control unit (SBC) which aside from initiating all accesses to main memory, also monitors the submitted addresses for "address invalid" condition. The SBC has direct contact with the following units: storage units, exchange, disk synchronizer, Lookahead and I-unit.

A7.2 Instruction Arithmetic Unit (I-unit)

Contains the instruction counter (IC), the 16 index registers ($0-$15), the time clock ($TC) and interval timer ($IT), the "originals" of the index condition indicators ($XF, $XVLZ, $XVZ, $XLGZ, $XCZ, $XL, $XE, $XH), and many registers, circuits needed for efficient decoding and execution of instructions.

Generates all instruction fetch requests on the basis of IC contents.

Develops effective addresses by adding the pertinent index value to the numerical address of an instruction.

Generates arithmetic unit operand requests for Lookahead.

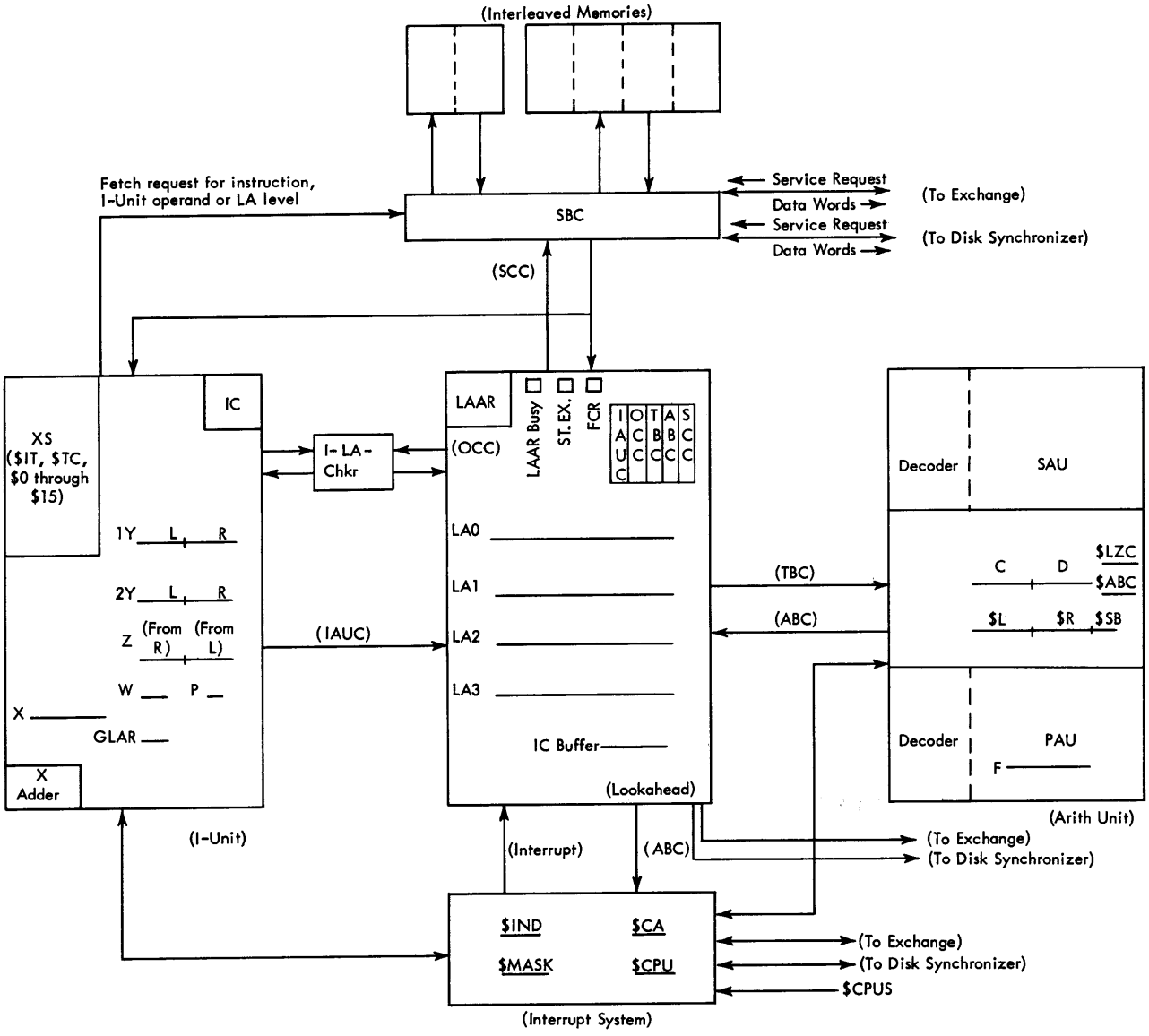
Partially decodes arithmetic unit instructions and converts the latter into information suitable for Lookahead processing. This information is then loaded into Lookahead.

Decodes and executes all index arithmetic instructions as well as the following: Z, R, RCZ, EX, EXIC, T, SWAP, except that stores are performed with the help of Lookahead. If non-I-unit operands are required for I-unit instructions, they will be fetched from the SBC or the Lookahead.

Decodes and executes the following "unconditional" branch instructions: B, BE, BD, BR, BEW, CB, CBR and Bind for the index condition indicators. The complete execution of BB and Bind for non-index conditions, however, requires the assistance of the Lookahead and the arithmetic unit.

Submits indicator conditions for the following indicators to Lookahead for updating of the indicator register: $IT, $OP, $AD, (from SBC), $DS, $DF, $IF + index indicators. These indicators plus a "conditional machine check" may lead to interruption during the updating. Other I-unit generated indicators such as $IK, $TS are gated directly to the indicator register.

Updates $TC and $IT every 1/1024 sec.



THE 7030 CENTRAL PROCESSING UNIT

A7.3 Lookahead (LA)

Contains four buffer levels, an address register (LAAR) and five counters IAUC (I-unit), OCC (operand check), TBC (transfer bus into arithmetic unit), ABC (arithmetic bus and interruption system) and SCC (store check).

When IAUC refers to a level, that level may accept I-unit loading, although the required operand may come later from SBC.

During OCC time the operand from SBC may be checked for error.
During TBC time the level contents may be shipped into arithmetic unit.
During ABC time the interruption system is updated, and if no interruption, signal is given to arithmetic unit for execution of instruction just loaded.
During SCC time the store operand (if any) is checked and sent to SBC on the basis of the contents of LAAR.

Provides interlocks, plus close contact with the interruption system to ensure smooth, autonomous and error-free operations of the various units in the computer.

A7.4 Interrupt System

Contains the indicator register ($IND), the mask register ($MASK), also $CA and $CPU. Has direct connections with I-unit, LA, arithmetic unit and the exchange units to receive updated indicator information.

Interruption occurs if
a. system is enabled
b. a masked indicator bit is a 1

Interruption sequence
a. The leftmost masked indicator bit position (say 0.K) is noted.
b. I-unit is house-cleaned except the index storage.
c. LA house-cleaning is performed. Recovery information is shipped back to the I-unit. This includes all index register recovery information and the "interrupted" IC value.
d. Contents of $IA is fetched and added to K.0.
e. Instruction beginning at the address C($IA) + K.0 is fetched from SBC without disturbing $IF indicator.
f. The "free" instruction is performed with all masked interruption conditions enforced. This instruction may or may not alter IC in I-unit.
g. I-unit fetches new instructions on the basis of IC. Resumption of normal operation.

A7.5 Execution Arithmetic Unit

Contains a parallel arithmetic unit (PAU) for floating-point fraction operations as well as all executed *, /, *+, and binary-decimal conversions.

Contains a serial arithmetic unit (SAU) for variable field length operations (except *, /, *+ and conversions), also for floating-point exponent arithmetic.

Receives instructions and operands from LA during TBC time for decoding and execution.

Submits store operands to LA, arithmetic indicator bits to the interruption system during ABC time.

Contains the following registers:
Accumulator ($L, $R) and sign byte register ($SB)
Buffer registers C, D
PAU buffer register F
Left zero counter ($LZC) and all ones counter ($AOC)

A7.6 Exchange

Contains 32 channels (32-63) for simultaneous I-O processing. Through adapters each channel can be connected to 8 tape units or with one non-tape I-O unit. Each channel is represented by one control word and two data words in the exchange storage. The channels communicate with the exchange storage through a channel scanner.

Contains a main memory address register (MMAR) and a buffer register to communicate with the SBC.

Contains an interruption address register (IAR) which indicates the channel address for the channel which has created an interrupt condition. Contains triggers to indicate the reason for interrupt: EOP, UK, EK, EE and CS. These triggers and IAR contents are set until the interruption system accepts the conditions. When IAR is busy other channels cannot use it to cause other interrupts.

Accepts I-O instructions from LA (two levels per instruction).

Fetches and stores control words and data words directly from SBC subject to $AD restrictions, but not $DF and $DS as these are performed by the I-unit.

Communicates with I-O units (through adapters and channel scanner) in 8-bit bytes.

Has its own clocking circuit (0.1 us/cycles) and ECC check bit generator-comparer. Maximum word rate is 1 extended word/10 us for the entire exchange.

A7.7 Disk Synchronizer

Contains 32 channels (0-31) only one of which can be in operation at any given time. Contains enough storage for one control word and one data word. Each channel can be attached to one disk unit.

Disk word rate is one extended word/8 μs. No direct chaining of I-O control words is permitted. Otherwise the disk synchronizer functions in much the same way as the exchange.

A8. LIST OF IMPORTANT REGISTERS IN THE 7030

A8.1 I-Unit

1Y (64 bits + check bits)	Instruction buffer (even-addressed full words)
2Y (64 bits + check bits)	Instruction buffer (odd-addressed full words)

(Both 1Y and 2Y may be used as I-unit operand buffer)

Z (64 bits + check bits)	Instruction preparation and execution register
XS (17 words, each with 64 bits + check bits)	Index storage (contains locations 1.0, 16.0-31.0)
X (64 bits + check bits)	Index data register (buffer register for XS)
X-adder (32 + check bits)	Index adder (capable of 24-bit additions)
W (18 bits + check bit)	Work register serving miscellaneous functions in I-unit

(LVS address decoding; second operand address in VFL; refill and interruption address; count for T and SWAP)

IC (19 bits + check bits)	Instruction counter
P (3 bits)	(For the storage of P field when decoding VFL instructions)
GLAR	A left zeros counter for LVS instruction execution ("geometric load address register").

Originals of $XF, $XVLZ, $XVZ, $XVGZ, $XCZ, $XL, $XE, $XH.

A8.2 I-Checker (Shared Between I-unit and Lookahead)

A8.3 Lookahead (LA)

LA0, LA1, LA2, LA3 — Lookahead buffer levels, each has
- Op code field (10 bits + check)
- Operand field (64 bits + checks)
- Indicator bit field (15 bits)
- Instruction counter field (19 bits + checks)

Plus these bits:

NOOP	no-op bit
WBC	word boundary crossover bit
LAOP	LA op code bit
IC	Instruction counter bit
INT	Internal fetch bit
LC	Level checked bit
LF	Level filled bit
FF	"Forward from" bit
DISC	Disconnect bit

LAAR Lookahead address register (18 + check)
"LAAR Busy" bit
"Store executed" bit
"Forward cycle required" bit
IC buffer (19 + checks)

Counters:

IAUC	(Instruction-arithmetic unit counter). For LA loading from I-unit.
OCC	(Operand check counter). For check of opnd arrived from SBC.
TBC	(Transfer bus counter). For loading of arithmetic unit.
ABC	(Arithmetic bus counter). For interrupt system updating, internal opnd fetch.
SCC	(Store check counter). For storing into main core storage.

A8.4 Interruption System

$IND	Indicator register (64 bits)
$MASK	Mask register (28 bits)
$CPU	Other CPU (19 bits + check bits)
$CA	Channel address register (7 bits + check)

Left zeros counter to handle interrupts.

A8.5 Arithmetic Unit

$L, $R	Accumulator (each 64 + check bits)
$SB	Sign byte register (8 bits + check)
C,D	Operand buffer register (each 64 + check bits)
$LZC	Left zeros counter (7 bits + check)
$AOC	All ones counter (7 bits + check)
SAU	Serial arithmetic unit

SAU decoder
SAU arithmetic-logical unit

PAU	Parallel arithmetic unit

PAU decoder
PAU arithmetic-logical unit:
PAU adder
PAU multiplier
(PAU) F-register (104 bits + check bits)

A8.6 Exchange

EM	Exchange storage (256 extended words each of 72 bits + 4 check bits)
EMAR	Exchange memory address register (7 bits + check)
	Word register (communicates with EM) (76 bits)
MMAR	Main memory address register (18 + check bits). For dealing with SBC
	Buffer register (72 bits). To handle traffic with SBC
IAR	Interrupt address register (7 + check). Contains address of interrupting channel

Interrupt triggers for 5 exchange interrupt conditions.
"Interrupt wait" bit to indicate if IAR is busy
Channel scanner for dealing with individual channels.
ECC generator and comparing circuits.

(4/61-2M:AG-84)

Printed in U.S.A. C22-6578

IBM®

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York