1959 ~~18~~

7030 PROGRAMMING EXAMPLES

FORWARD

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 nor 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.

T. C. C.

STRETCH PROGRAMMING EXAMPLES

STRETCH PROBLEMS

~~1~~. Instruction Arithmetic Unit Instructions

Problem 1.1. Transmittal of 2 full words.

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

Method 1.~~1~~ Use the immediate transmit instructions.

```
TI, 2, DØG, CAT
        or
TBI, 2, DØG + 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 ensure that all the source words are transmitted properly, use TBI if CAT>DØG; use TI if CAT<DØG. In the following we shall assume no overlap.

Method 2.~~1~~ Use an index register to control the number of words transmitted.

```
LCI, $1, 2.0
T, $1, DØG, 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 the J field referring to an index register with a zero count field means the maximum count possible.

~~Method~~ 3.~~1~~ Use index instructions,

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

Comments,

a. Although data transmission is not the primary function of index registers, the timing here is not too different from that of the more concise transmit instructions.

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.

c. The two "unused" bits (bits 27 and 28) of the index register are available for data transmittal. They serve no specific purpose otherwise.

Problem 1.2, Interchange of two word-pairs.

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

Method 1.~~1~~ Use immediate swap instructions.

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

Comments.

a. The swapping of each word-pair involves two memory fetches followed by two stores into the "fetch" locations. Since the same memory unit must wait 2.2 microseconds between requests, the execution time of swap instructions generally takes more time than transmit.

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

Comments.

a. The execution time of this sequence is not heavily dependent on memory delays, and Method 2 is expected to be faster than Method 1. 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 13, *Cyclic permutation of a group of full words.*
Given quantities A, B, C, D, E, F, G, H, I, in full words DØG through DØG + 8.0. Cyclically permute the information such that the new contents will be in the sequence DEFGHIABC.

Method 1.

```
TI, 3, DØG, 17.0                "store A, B, C, in $1, $2, $3, respectively
TI, 6, DØG + 3.0, DØG           "DEFGHI GHI
TI, 3, 17.0, DØG + 6.0          "DEFGHIABC
```

Method 2.

```
SWAPI, 8, DØG, DØG + 1.0        "cyclic left shift one unit
SWAPI, 8, DØG, DØG +1.0         "shift another unit
SWAPI, 8, DØG, DØG + 1.0        "complete the 3 unit cyclic left shift
```

Method 3.

```
SWAPI, 3, DØG, DØG + 6.0        "~~place ABC~~ GHIDEFABC
SWAPI, 3, DØG, DØG + 3.0        "complete the permutation
```

Method 4.

```
SWAPI, 6, DØG, DØG + 3.0
```

Comments.

a. In order to permute N consecutive full words (so DØG thru DØG + N-1) cyclically left K places, if K is a divisor of N the single instruction

SWAPI, N-K, DØG, DØG + 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 immediat
swap instructions should be replaced by equivalent direct swap instructions.

<u>Problem14, *Replacement of full words by zeros.*</u>
Replace the contents of full words DØG through DØG + 24.0 with zeros.
<u>Method 1</u> Set up a small loop using CB+ instructions.

```
            LX, $3, XW3
A[illegible] Z, DØG($3)
            CB+, $3, A[illegible]
B[illegible] BEW, B[illegible]
XW3) XW, 0.0, 25, XW3
```

<u>Comments,</u>

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.

<u>Method 2,</u>

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

<u>Comments,</u>

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 be done with some care. For instance, the insertion of a half word instruction between the Z and CB- instructions <u>without</u> corresponding change in the CB- instruction will cause branches to this new instruction rather than to the Z instruction.

<u>Method 3,</u> 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, DØG
            TI, 12, DØG, DØG + 1.0
            TI, 12, DØG, DØG + 13.0
```

<u>Comments,</u>

a. The execution sequence is:
zeros $\longrightarrow$ C(DØG) $\longrightarrow$ C(DØG + 1.0),
C(DØG + 1.0) $\longrightarrow$ C(DØG + 2.0), etc.   C(Q) means the contents of the location Q.

<u>Problem15, *Replacement of isolated full-word groups by zeros,*</u>
Replace the following full words by zeros: DØG through DØG + 24.0, CAT through CAT + 15.0, CHICK through CHICK + 34.0.

Method 1. Use chain indexing.

```
            PRNID, JØE BLØWE, DEPT. 333
            PUNID, J. BLØWE
            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, DØG, 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, DØG, 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 settnng of the index flag indicator is done prior to the refill.

Method 3. Use transmit instructions.

```
            Z, DØG
            TI, 12, DØG, DØG + 1.0
            TI, 12, DØG, DØG + 13.0
            TI, 16, DØG, CAT
            TI, 12, DØG, CHICK
            TI, 12, DØG, CHICK + 12.0
            TI, 11, DØG, 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, DØG
            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 16, Subtraction of value fields,

Subtract the value field of $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)
XVGZ (11.51)
XVZ (11.50)
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 to use the connective instruction CM1100(B4, 1), 17.24 in place of the BBN instruction.

Problem 17. 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 interruption 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, DØG($3).
~~Do not simulate the indicator settings.~~
Method 1.

```
RNAME   SX, $2, X2                  "save $2
        SR, $0, 18.0
STØX    SX, $1, 0($2)
        LX, $2, X2                  "restore $2
        LVE, $1, LØX
        LR, $0, 17.0
LØX     LX,$1, DØG($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 STØX, and the use of $2 would be avoided. This is not possible because in the SR operation the refill field concerned is right appended by zero bits to create a 25 bit value field. The latter is then stored. The STØX 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 LØC. 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.

```
TPØSE   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, LØC + 1., N-1, XW22
XW3     XW, LØC + N, N-1, XW33
XW22    XW, 0
XW33    XW, 0
```

```
N         SYN, 100.0                     "if 100 x 100 matrix
LØC       SYN, 32768.0                   "if matrix starts at 32768.0
```

Comments.

d. 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.

Method 2, Start from the upper and lower co-diagonals of the matrix and proceed through the exchange of the north-east-most and the south-west-most elements.

```
TPØSE2    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, LØC+1., N-1, XW22
XW3       XW, LØC+N, N-1, XW33
XW22      XW, 0
XW33      XW, 0
N         SYN, 100.                      "size of matrix
LØC       SYN, 32768.0                   "starting location
```

2. Variable Field Length Instructions.

Problem 2.1. Cyclic bit shifting.

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

Method 1.

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

Problem 2.2. Length of an unknown file.

Information of unknown length is written in consecutive 7-bit bytes beginning at INFØ. 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
LØAD    L(BU,7,8), INFØ($1)
        K(BU,7,8), ENDB
        V+, $1, SEVN
```

```
            BZAL, $+1.0

            B, LØAD

            BEW, $

ENDB        (2)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

CØMP        K(BU,7)(V+I), 0.07($1)

            BAE, $+1.0

            B, CØMP

            L(BU,7)(V-I),INFØ($1)

            BEW, $

VFIELD      VF, INFØ
```

Comments.

a. The last VFL instruction serves mainly to perform the (V-I) operation, for an alternative technique see Method 3. ~~there being no other simple way of doing the same thing.~~

b. In binary unsigned operations the machine uses a byte size of 8 regardless of the data description, except for logical connectives.

STRAP inserts byte size 8 if unspecified.

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 for 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

$$0.(127*2^{-7}) = 0.(127*2^{17})$$
$$= (127*2^{11}).0 = 260096.0$$

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

Method 3. Use connective and branch on $RZ.

```
        LVE, $1, VF1
        LI(BU,7), (2)1111111
CØNT    CT0110(V+I)(BU,7), 0.07($1)
```

```
        CNØP
        BRZ, $+1.0
        B, CØNT
        V+, $1, VF1
        BEW, $
VF1     SIC, INFØ
```

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, WØRD 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).

e. The CNØP forces the next two half-word branch instructions to be packed in the same full word. This has a beneficial effect on Instruction Unit timing.

Problem 13. Deletion of every 5th bit in a field.

Given a string of 60 bits starting at FIELD, delete every 5th 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
LØAD    L(B,5,1)(V+I), 0.05($2)
        ST(BU,4)(V+IC), 0.04($3)
        BZXCZ, LØAD
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
LØADA   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 LØADA and LØADA + 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 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; LV1, $2, 56
            L(BU, 60), FIELD
STØRE       ST(BU, 4) (V + I), 0.04($1), 0($2)
            V - I, $2, 5
            BZXVLZ, STØRE
            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.

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?"

$AØC "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 WØRD contains a binary message which would be easily interpretable when every bit in the word is reversed (WØRD + 0.63 becomes WØRD + 0.0, etc.). Perform the bit reversal and put the result in DRØW.

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

```
        TI, 1, WØRD, $R
        LX, $1, XW1
        LX, $2, XW2
STØR    ST(BU,1)(V+I), 0.1($1), 0($2)
        CBH, $2, STØR
        BEW, $
XW1     XW, DRØW, 0, $
XW2     XW, 0, 64, $
```

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

```
        LX, $1, XW1
        LX, $2, XW2
LØ      L(BU,1)(V+I), 0.1($1), 0($2)
        CBH, $2, LØ
        ST(BU,64), DRØW
        BEW, $
XW1     XW, WØRD, 0, $
XW2     XW, 0, 64, $
```

Problem 2.5. Removal of key words.

Given a string of 100 six-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
LØDE    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, LØDE
```

```
        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, LØDE
XW1     XW, DATA, 100-3, $
XW2     XW, ANSW, 0, $
X18     VF, 0.18
```

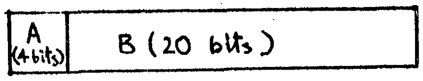
Comments.

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

b. Relatively error-free instructions can be packed together in the same line to enable the programmer to focus his attention on the rest of the program in the debugging stage.

Problem 2.6. Sorting on the basis of subfields.

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



The 4-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.

```
ASØRT     LX, $2, XW2
LØØP      L(BU, 4) (V + I), 0.04($2)
          * (BU, 24),VF20                  "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, LØØP
          BEW, $
XW2       XW, DATA, 16, $
VF20      VF, 20
ANS       DRZ(BU, 20), (16)
```

Comments.

a. If the A fields are not all different mis-stores will be made.

Method 2. A slight modification of Method 1.

```
ASØRT2    LX, $2, XW2A
LØØP      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, LØØP
          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)	($2 has X in value field)	($2 has X + 0.4 in value field)
L(BU,4) (V + I), 0.04($2)	L(BU,4)(V + I), 0.24($2)	L(BU,4), -0.04($2)
.	.	.
.	.	.
.	.	.
L(BU,20)(V + I), 0.20($)	L(BU,20), - 0.20($2)	L(BU,20)(V + IC), 0.24($2)
ST(BU,20), ANS($1)	ST(BU,20), ANS($1)	ST(BU,20), ANS($1)
CB, $2, LØØP	CB, $2, LØØP	BZXCZ, LØØP

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

Method 2, *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
KØMP      K(BU, 4)(V+ICR),0.24($1), 20       "inner loop, test against assumed min
          BAH, FIXMIN                        "usually unsuccessful
AGAIN     BZXCZ, KØMP
          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 quess 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 undesir~~e~~able, 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 3, *Repeated compares for both maximum and minimum.*

```
          LX, $1, XW1
          LX, $2, XW2
          LX, $3, XW3; SX, $3, XW33
LØDE      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, LØDE-1
PACK        LV, $1, XW11;LX, $2, XW22
LØAD2       L(BU, 24)(V+I), 0.24($1)
            ST(BU,20)(V+IC), 0.20($2)
            BZXCZ, LØAD2
            BEW, $
SWICH       SWAP, $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 areas 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

LØAD        L(BU, 4)(V+I), 0.24($2), -18

            LVE, $3, MTABL($1)

            L(BU,20), -0.20($2)

            ST(BU,20)(V+I), 0.20($3)

            SVA,$3, MTABL($1)

            CB, $2, LØAD
```

```
XW2       XW, DATA, 0,8
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
```

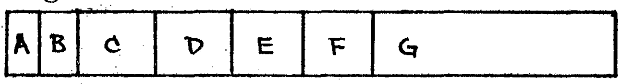
<u>Comments.</u>

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. <u>Purchasing List Arithmetic.</u>

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



Subfield A is an 8-bit byte consisting of 1's.

Subfield B has 2 8-bit bytes (item no.).

Subfield C has 6 8-bit bytes (coded name of product).

Subfield D has 3 8-bit bytes, and contains the no. of units of the product desired in decimal (DU, 24, 8).

Subfield E has 6 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 3 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
LCØN        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
            ST(DU, 32, 8)(V+I), 0.32($2)
            M+(DU, 64, 8), $FT              "update grand total
            L(BU, 32), TESTW
            KF(BU, 24), 0($2)                "test for end of string
            BAE, LAST
KØMF        KF(BU, 24), 0.8($2)         "test for beginning of new field
            BAE, KØMF
MØRE        V+I, $2, 1.0                 "bypass 64 more bits to new D field
            B, LCØN
```

```
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
              (BU,8,8),
            DD^(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. The 4-bit bytes are expanded into 8-bit bytes in the final store of the grand total.

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 4 bit index field
         BAE, NØX                   "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
NØX      Z, 17.0
TEST     KFI(BU,2), (2) 10,4        "test if floating point
         BZAE, NØTFP
```

```
FP        -(BU, 32-18), $R. 32+0. 18       "floating point measure
MPLUS     M+(B, 25, 1), 17. 0, 32-24       "25 bit add
          BEW, $
NØTFP     KFI(BU, 4), (2)1000, 4           "test if VFL left address
          BZAE, NØTVFL
VFL       -(BU, 32-24), $R. 32+0. 24       "VFL measure
          B, MPLUS
NØTVFL    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
```

<u>Comments.</u>

a. The effective address ~~is~~ as 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 4-bit I-type indexing specification.

If bits 23-27 have10000: 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.

<u>Problem 2.10.</u> <u>Fetch (p,q)the element of rectangular matrix.</u>

Given a matrix A of size M×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.

<u>Method 1.</u>

```
LØCATE  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($3)
        BEW,m$
ENN     DD(BU,18), N               "N is assumed defined elsewhere
VF      VF, MTRIX-1.0
```

<u>Comments</u>.

a. The element A_{p1} is in MTRIX+(p-1)N. The element A_{pq} is therefore 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 2-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), R                    "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 M×N 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

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

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

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

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

$$K'=(j-1)*N+(i-1)$$
$$= \text{integer remainder of } (K*N)/(M*N-1)$$

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 done by using flag bits as identification, at the same time ensuring that the lead element of every cycle has the smallest (or alternatively largest) address possible. The method is essentially that of M.F. Berman, J.A.C.M. 5, 383(1958). For similar techniques see P.F. Windley, Computer J., 2, 47-48(1959); G.Pall and E.Seiden, Math. of Computation, 14, 189-192(1960).

For square matrices each of the cycles have ~~the~~ 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, ~~though~~ except that the first and last elements are unaltered by the transposition process.

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

```
TRANSP    BD,$+0.32
          LV,$1,$IA                     "$IA assumed to have meaningful value
          V+H,$1,37.0
          SVA,$1,SWAP2
          SWAPI,1,0($1), INST
          TI,1,$IND+1.0,INST+1.0
          CM1111(BU,1),$IND+1.37
          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,BZBZ
          B,BEW;CNOP;NOP                "to ensure 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 a Vflag
BZBN        BZBN,MATRX+0.63($1),NUCYCL
            CB&,$1,BZBZ
SWAP2       SWAPI,1,0,INST
            TI,1,INST+1.0, $IND+1.0
BEW         BEW,$;CNOP
INST        B,ENDCYC;NOP
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 ensure 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 DØG 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), DØG
              D+(U), X48
              ST(N), CAT
              SLØ(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 DØG 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), DØG
            D+(U), X48
            ST(BU, 18), 17.28, 68         "$1.28 is also acceptable
            BEW, $
X48         DD(N), 0.0X48                 "binary exponent of 48
```

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 PØLY.

Method 1. Term-by-term evaluation.

```
PØLYN       L(U), A
            ST(N), PØLY
            L(U), X
            LX, $2, XW2
            B, STØR
LØAD        L(U), XK
```

```
            *(N), X
STØR        ST(U), XK                 "new power of x
            *(N), A($2)
            +(N), PØLY
            ST(U), PØLY               "new partial sum
            CB+, $2, LØAD
            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, XWØRD2
        L(N), A+20.0
MULTI   *(N), X
        +(N), A($2)
        CB-, $2, MULTI
        ST(U), POLY
        BEW, $
XWØRD2  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), PØLY
            BEW, $
XW2         XW, 19.0, 20, $
```

Comments.

a. The double operations are essentially no slower than the corresponding regular operations.

Problem 3.4. **Modified Trapezoidal rule.**

Evaluate the integral

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

by the modified trapezoidal rule

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

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,STØR
LØØP    L(u),TEMP
        +(N),H
STØR    ST(u),TEMP         "update Temp
        *(N), 8.0          "or $L
        *(N), 8.0          " new integrand value
        M+(N) ANS
        CB,$1,LØØP
        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     DR(N),(1)
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 4 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), M
         E - I(U), 1
         + (N), A
         B, STØR
LØØP     L(U), TEMP
         +(N), H
STØR     ST(U), TEMP              "new x
         B, FUNCT                 "branch to f(x) evaluation
RTURN    M+(N), ANS               "new partial sum
         CB, $1, LØØP
         BEW, $
```

```
ANS        DR(N), (1)
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.

```
CØNF      L(U), X
          * , X
          ST(U), TEMP                       "x ** 2
          LX, $2, XW2
          L(U), NUM                         "39
LØØP      R/N, TEMP                                   "X*X/39
          ST, TEMP1
          L(U), NUM
          -, TWØ
          ST(U), NUM
          +, TEMP1                                    "37-X*X/39
          CB, $2, LØØP
          R/, X
          ST(U), TEMP2
          BEW, $
X         DD(N), $PI/4
NUM       DD(N), 39.0
TWØ       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
```

Method 1.

```
CØNF        L(U), X
            * , X
            ST(U), TEMP                 "x ** 2
            LX, $2, XW2
            L(U), NUM                   "39
LØØP        R/N, TEMP                               "X*X/39
            ST, TEMP1
            L(U), NUM
            -, TWØ
            ST(U), NUM
            +, TEMP1                                "37-X*X/39
            CB, $2, LØØP
            R/, X
            ST(U), TEMP2
            BEW, $
X           DD(N), $PI/4
NUM         DD(N), 39.0
TWØ         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 right most symbolic address. If the latter has no meaningful dds, the next-to-the-right most 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. 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
LØFT        LFT(N), A($3)
```

```
          *+(N), B($3)
          CB+, $3, LØFT
          SRD(N), C
          BEW, $
SXTEEN    XW, 1.0, 16, $
```

<u>Comments</u>

a. The *+ operation is ideal for vector and matrix products.

b. The LFT operation is a "memory to memory" operation, since $ FT is a bonafide memory location. Since it does not involve the execution arithmetic unit (the E-box) and since the temporary indicator $MØP is turned on only for E-box-to-memory operations, $MØP is turned <u>off</u> 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 look-ahead 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 formua:

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

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
LØØP    ST, XK
        *, XK
        *, XK
        E + I, 1
        +, EN                    "2 X ** 3 + N
        R/, TEMP
        +, HALF
        *, XK                    "new XK created
```

```
          CB, $2, LØØP
          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 $C\epsilon^3$. Here C=2/3. See E.G. Kogbetliantz, IBM Journal of R. and D., 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 the squares of F_k, then take the square root.

c. Divide each F_k by the square root, and store in FLØAT through FLØAT+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.

```
NØRMV    Z, SUM
```

```
            LX, $1, XW1

            LX, $2, XW2

LØØP        LCV(V+I)(D,48,6), 0.48($1), 68

EPLUS       E+I, 48                          "number is now unnorm,      FP integer

            ST(N), 0($2)

            *(N), $L

            +, SUM

            ST(U), SUM

            CBR+, $2, LØØP

            SRT, RØØT
```

```
LØØP2     L(N), 0($2)
          /, RØØT
          ST(U), 0($2)
          CB+, $2, LØØP2
          BEW, $
XW1       XW, STRNG, 25, $
XW2       XW, FLØAT, 25, $
SUM       DRZ(N), (1)
RØØT      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 is approximable 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 re-used 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.

```
MKØMP     ST(U), A1                  "save accumulator
          SLØ (U), A1 + 1.
          DL(U), A1 + 1.             "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
          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)
and 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
$(1/2) * 2^{E+1} = 1 \times 2^{E}$
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 counter parts).

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  | + |
```

result by present program =(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

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

a. 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 $\text{Log}_2 N$.

N is a positive floating number in DØG, 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 $N = 2^{\alpha} \cdot \beta$, $\quad 1/2 \leq \beta < 1$

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

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

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

Method 1.

```
L(N), DØG

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 RØØT.

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

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

If x_k has a relative error of ϵ, namely

$$x_k = x_t (1+\epsilon) \qquad x_t = \text{true } x$$

then

$$x_{k+1} = x_t(1+\epsilon^2/2+O(\epsilon^3))$$

Thus if we are able to find a quess which has a relative error of 2^{-32}, one interation 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, STØR
          D+(N), 0                    "normalized long fraction
          SRT(U), GUESS               "first guess
          D/(N), GUESS
          ST(U), QUØT                 "first order quotient
          DL(U), $RM
```

```
                /(N), GUESS              "obtain second order quotient
                D+(U), QUØT              "double length quotient
                D+(N), GUESS
                E-I(U), 1                "divide by two
                D+(U), 0                 "shift until exponent zero
STØR            ST(BU, 64), RØØT, 52
                BEW, $
GUESS           DR(U), (1)
QUØT            DR(U), (1)
FRAC            DR(U), (1)               "to be supplied
```

Comments

a. Had the original fraction not been pre-normalized, 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:

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

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 interms 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 multiplication 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."
<u>Method 1.</u>

```
DFCØNV  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 superdigit, zero offset
        ST(DU, 56), DFRAC
        L(U), BUFFER
        *(U), RADIX
        CV(BU, 48)
        ST(DU, 56), DFRAC+0.56
        BEW, $; CNØP                     "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}.

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 connectives "and", and the subsequent querying of $AØC 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
LØWF    LWF(U),SVEC($2)
```

```
        ST(U),SVEC($3)
        V+IC,$3,1.0
CØNN    C0011(BU,1)(V+IC),0.01($1)
        BXCZ,END
        V+I,$2,1.0
        BZRZ,LØWF
        B, CONN
END     Z,SVEC($3)
        CB+,$3,END
        BEW,$
XW1     XW,BIMAGE,N+1,$
XW3     XW,0.0,N,$
```

<u>Problem 4.5 Scalar product of compressed sparse vectors.</u>

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.

<u>Analysis:</u> In ~~T~~the scalar product ~~is defined as~~

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

~~The~~ 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 $AØC will ~~give~~ yield the number of multiplications to be performed and the $LZC will give ~~the location in the dummy vector of the first one.~~ information about the subscript K for a needed multiplication.

Method 1.

```
        TI,3,17.0,SAVEX                    "save $1,$2,$3
        L(BU,N),XBMAGE
        C0001(BU,N), YBMAGE                "1 bit if both items non-zero
        ST(BU,N)KEYVEC
        L(BU,7),$AØC,64+18
        LX,$3,8.0                          "$AØC in $3 count field
        DL(U),ZERØ;ST(U),PRØDT
        BXCZ,FIN
        B,LØF;CNØP
LØØP    CM0000(BU,0),KEYVEC+0.1,0($1)      "field length indexing
        CT0011(BU,N),KEYVEC                "test left zeros
LØF     LF(BU,25),$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
        CT0011(BU,0),YBMAGE,0($1)          "field length indexing
        LV,$3,7.32
        V+C,$3,19.0                        "YVEC modifier
        L(U),PRØDT                         "restore high order part
        LFT(U),XVEC($2)                    "computation part
        *+(N),YVEC($3)
        ST(U),PRØDT
        BZXCZ,LØØP

FIN     TI,3,SAVEX,17.0                    "restore $1,$2,$3
        BEW,$                              "answer in acc. as well as PRØDT
```

```
ZERØ      DD(N),(0)
PRØDT     DRZ(N),(5)
KEYVEC    SYN(N),PRØDT+1.0
SAVEX     SYN(N),PRØDT+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 $AØC is given at the half-word level, necessitating a small amount of adjustment.

Problem 4.6; Transposition of an 8x8 bit matrix.

Given an 8x8 matrix whose elements are bits stored consecutively and rowwise 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
```

```
LØF       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,LØF
          V+C,$2,VF;SX,$2,XW22
          V+C,$3,VF;SX,$3,XW33
          BZXCZ,LØF
          BEW,$
XW2       XW,LØC+0.1,N-1,XW22
XW3       XW,LØC+0.N,N-1,XW33
XW22      XW,0
XW33      XW,0
VF        VF,0.1+0.N
LØC       SYN,BMATX8
N         SYN,8
```

<u>Comments.</u>

a. The program is written to accommodate an NxN bit matrix beginning at LØC. The SYN pseudo instructions define LØC 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.

<u>Method 2.</u> 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,CØNNECT
```

```
VMI       V-I,$2,1                          "reduce offset by 1
CNNECT    C0111(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≤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($2)
          BZXCZ,VMI
          SF(BU,N*N,N),LØC
          BEW,$
XW1       XW,LØC,N,$
LØC       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-1, because of the multiplication in the data description field.

Problem 4.7. Transposition of a 64x64 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 LØC 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 rowwise
         V-ICR,$3,8
         BZXCZ,DIAG2                          "until block stored
         CBZR,$4,BEW                          "branch if last diagonal block complete
ØFDIAG   V+,$1,VFP8;SX,$1,XW111               "loop for off diagonal block pair
         V+,$2,VF8P;SX,$2,XW222
         LI(BU,1),0
```

```
ØFDIA1   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,ØFDIA1                          "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,ØFDIA2                          "until block pair stored
         CBR,$4,ØFDIAG                         "until one row,one column complete
NEWRØW   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,LØC,8,XW111
XW2      XW,LØC,8,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
LØC       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.

Method1. 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        L(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
ZERØ        DD(N),0
```

a. STRAP - 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 $\cos 2\pi x$ in the accumulator.

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

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

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

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 $\cos 2\pi x > \cos \pi/4 > 0.7$, the relative error defined by $\epsilon_r = \frac{\text{absolute error}}{\text{true answer}}$ 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.

```
CØSF        *(N), TPI                   "2*$PI
            D*(N), $L                   "square
            SRD(N), TEMP
            LX, $2, XW2
```

```
DMULT       D*N(N), CØNST($2)
            D+(N), WØN
            D*(N), TEMP
            CB+, $2, DMULT
EMI         E-I, 1
            D-(N), WØN
            SRDN, $L
            BEW, $
TPI         DD(N), 2*$PI
XW2         XW, 0, 7, $
CØNST       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
WØN         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 $\cos \pi x$ first. Examination shows that terms up to K = 12 would be adequate.

```
CØSF2      ST(N), TEMP
           LX, $2, XW22
DMULT      D*N(N), KØNST($2)
           D+(N), WØN
           D*(N), TEMP
           CB+, $2, DMULT
           D*N(N), KØNST ($2)
           D+(N), WØN
           D*(N), $L                          "create cos 2A
           E+I (U), 1
           D-(N), WØN
           SRD(N), $L
           BEW, $
XW22       XW, 0, 5, $
KØNST      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
WØN      DD(N), 1.0
TEMP     DR(N), (1)
```

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 $\cos \pi x$ first. Examination shows that terms up to K = 12 would be adequate.

```
CØSF2      ST(N), TEMP
           LX, $2, XW22
DMULT      D*N(N), KØNST($2)
           D+(N), WØN
           D*(N), TEMP
           CB+, $2, DMULT
           D*N(N), KØNST ($2)
           D+(N), WØN
           D*(N), $L                        "create cos 2A
           E+I (U), 1
           D-(N), WØN
           SRD(N), $L
           BEW, $
XW22       XW, 0, 5, $
KØNST      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
WØN     DD(N), 1.0
TEMP    DR(N), (1)
```

Comments,

9. The error situation is somwhat 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 ~~9~~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:

$$x = F*2^E = \sqrt{2}\ F*2^{E-1/2}$$
$$\ln x = (E-1/2)\ln 2 + \ln(\sqrt{2}\,F)$$
$$\ln\sqrt{2}\,F = 2\sum_{k=0}\left(\frac{\sqrt{2}\,F-1}{\sqrt{2}\,F+1}\right)^{2k+1}/(2k+1) = 2Z\sum_{k=0}(Z^2)^k/2k+1$$

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, $F\sim 1$. 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+(N), Q                       "F+1/RT2
         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), CØNST($1)
ADD      D+(N), CØNST+1.0($1)
         D*(N), TEMP+2
         CB+, $1, ADD
         D+(N), CØNST +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, $
CØNST    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)
XX       DD(N), +3, 52
```

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 had $N a magnitude of , say, 1.5.

Problem ~~10~~ 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, $$B, $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:

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

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

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

<u>Method 1.</u>

```
EXP         KMG(N), KØMP
            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), CØNST($14)
```

```
DPLUS     D+(N), CØNST+1.0($14)
          D*(N), TEMPF
          CB+, $14, DPLUS
          D+(N), CØNST($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($15)
KØMP      DD(N), 1024*$N
RLN2      DD(N), 1/$N
E11       DD(N), 0X11
LN2X      DD(N), $NX-11
XW14      XW, 0, 14, $
CØNST     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 Fln2 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, $\overset{\$SB}{\cancel{\$SB}}$, $14 and $15, none of the other internal registers ~~are~~ 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 given in the accumulator.

Problem 4.12 Transcendental Function Evaluation

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

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

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;NØP
RTURN       ST(U), TEMP
            LVI, $15, $+1.0; B, EXP
            B, ERR; NØP
```

```
             *(N), EKS
             E+I(U), 1
             ST(U), TEMP+1
             LN(U), TEMP
             +(N), WØN
             SRT(N), $L
             R/(N), TEMP+1
             BEW, $                        "normal exit
ERR          BEW, $                        "error exit
EKS          DD(N), $PI
TEMP         DR(N), (2)
WØN          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" ~~technique~~ feature available in STRAP can be used by the user of the subroutine to avoid memory conflict.

<u>Problem 4. 13.</u> <u>Numerical Integration</u>

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

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

<u>Analysis:</u>

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

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

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

$(p, q) = (-1, +1)$ 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 LEG Q8 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, $FT, $TR, $LZC, $AØC, $14 and $15 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 $L, $R, $SB, $FT, $LZC, $AØC, and $14 without restoring.

d. The arrangement of the symbolic program is something like the following.

1) Identification 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 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 at0.($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 to 4.($15).
"  error return is 3.($15), with answer in $L.
LEGQ8       SX,$2,LEGQ82;SX,$15,LEGQ8F
            LVE,$2,0.($15)
            SVA,$2,LEGQ8A
            SVA,$2,LEGO8B
            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)Y/2 + (B+A)/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 toot 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

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

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.

A2. Checklist 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 overall alignment, particularly the location of the NAME fields. Proofread carefully, look for missing cards, mispunches, and off-punches.

Character code for symbolic decks

	(N) No Zone	(Y) 12 Zone	(X) 11 Zone	0 Zone
No Digit	(Blank)	+	-	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	Ø	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, Ø) 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 Θ.

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 reminescent 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 ------)

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 $\in$ (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 $\in$ 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 $\in$. The quotient fraction is developed normally, but the quotient exponent will be either that of $\in$ (case of small dividend), or that of $1/\in$ (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 F.P. 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, +	$+\epsilon$	1	1	0	$<2^{-1024}$, >0
XFN, -	$-\epsilon$	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:

$\infty+\infty = \infty;$	$\infty * (\pm\infty) = \pm\infty;$	$\infty/(\pm N) = \pm\infty;$	$\sqrt{\infty} = \infty;$
$\infty \pm N = \infty;$	$\infty * (\pm N) = \pm\infty;$	$\infty/(\pm\epsilon) = \pm\infty;$	$\sqrt{\epsilon} = \epsilon.$
$\infty \pm \epsilon = \infty;$	$\epsilon * (\pm N) = \pm\epsilon;$	$\epsilon/(\pm\infty) = \pm\epsilon;$	
$\epsilon \pm N = N;$	$\epsilon * (\pm\epsilon) = \pm\epsilon;$	$\epsilon/(\pm N) = \pm\epsilon;$	
$\epsilon \pm \epsilon = (\pm)\epsilon;$		$\pm N/\infty = \pm\epsilon;$	
		$\pm N/\epsilon = \pm\infty;$	

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 condition 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 $\infty_2{}^{\#}$;	$\epsilon_1 + \infty = \infty$;
$\infty_1 + N = \infty_1$;	$\epsilon_1 + N = N$;
$\infty_1 + \epsilon = \infty_1$;	$\epsilon_1 + \epsilon_2 = \epsilon_1$ or $\epsilon_2{}^{\#}$;
$\infty_1 - \epsilon = \infty_1$;	$\epsilon_1 - \epsilon_2 = \epsilon_1$ 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. (Result 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.)

$\infty_1 : \infty_1$

$\epsilon_1 : \epsilon_1$

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 <u>noise</u>.

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 (<u>NOT</u> 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 <u>single</u> 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 <u>first order</u> noise as one which can create a minimum noticeable relative error in the <u>first order</u> 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 <u>second order</u> 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 <u>small</u>. 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 * posses only second order noise. The operation *+ has 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 pre-shift.	1. No additional noise introduced in remainder normalization.
Others			
E+, E+I	no	yes	2
SHF	no	no	Noiseless