ADVANCED PL/1 PROGRAMMER'S MANUAL

March, 1977

Data and Instruction Formats

Introduction

The microprocessor used in the Q1/LMC has such a low-level instruction set that it is impractical to use it for compiler output. The resulting program would use too much memory if it were. Instead, Q1/PL/1 generates a pseudo-machine code which is much more powerful and compact. This code is interpreted by the operating system in much the same manner that a microprogram executes macroinstructions.

The pseudo-machine is a stack machine. All addresses and intermediate results appear in a First-in/First-out storage area. The instruction set is primarily reverse polish in nature: the operand address appears or is computed before the operators are executed at which time the operands are in the stack.

Instruction Format

The following table gives the properties of the instructions, which may be one, two or three bytes. All stack data which is used is removed from the stack. In arithmetic and logical operations, the result is placed on top of the stack.

Op-codes could range up to $3F_{16}$. The operating system will interpret higher values as the most significant byte of an address which will be put on the stack.

PSEUDO-MACHINE INSTRUCTION SET

Opcode	Description
00	Stack a binary number from the address on top of the stack.
01	Stack a floating decimal number from the address on top of the stack.[1]
02	Stack a fixed decimal number from the address on top of the stack.[1,2]
03	Store the binary number which is on top of the stack at the location pointed to by the address next to the top.
04	Store a floating decimal number.[1]
05	Store a decimal number as a fixed point.[1,2]
06	Store a string of bytes. Uses 4 binary numbers from stack. Top is number of bytes of source string, followed by address of source string followed by size and address of destination string. String padded with binary zero or truncated as necessary.
07	Compare binary numbers. Stack binary zero if false; -1 if true.[3]
08	Compare decimal.[3]
09	Compare character strings.[3]
0A	Adds the two binary numbers on top of the stack.
0B	Add decimal.
0C	Replace the binary number on top of the stack with its negation.

[1] Number of bytes.

[2] Scale is 40_{16} plus number of bytes before decimal point plus 80_{16} if least significant digit is not used.

[3] Relation Type: Relation Type is 4 if true when next to top operand is greater than top operand plus 2 if true when equal plus 1 if true when less than.

0D	Negate decimal.
0E	Multiply binary.
0F	Multiply decimal.
10	Divide binary.
11	Divide decimal.
12	And binary.
13	Or binary.
14	Ones complement binary.
15	The PUT statement will direct data to the specified device driver address.
16	Convert from character to binary.
17	Convert from character to decimal.
18	Convert from binary to decimal. For operand next to the top of stack.
19	Convert from binary to character.
lA	Convert from decimal to character.
1B	Return to operating system for next program.
1C	Go to address on top of stack.
1D	Go to address on top of stack if binary number next to top is not zero.
1E	Call to address on top of the stack. Parameter addresses will be removed from the stack until a zero is encountered.
1F	Return from subroutine.
20	Do loop control for binary indexes. Go to address on top of stack if signs of operands third and fourth from top differ or the latter is zero. Otherwise, go to address next to top.
21	Same for decimal.
22	Open file. File description address is on top of stack.
23	Get new line of keyboard input.
24	Get decimal number from input.
25	Get character string from input. Length is on top of stack and address is next to top.

26	Put character string.
27	Put decimal number.
28	Put Edit character string. Field size is on top of stack with string description next.
29	Put Edit decimal number. Description of picture string is on top of stack with number next.
2A	Output character next to top of stack the number of times indicated by the number next to the top.
2B	Read from disk. Stack from top is: file description address, memory space available for each record, and target address.[1]
2C	Write on disk. Same stack format as Read.[1]
2D	Rewrite on disk. Same stack format as Read.[1]
2E	Read key. Stack from top is: key offset, target address, memory space per record, file description address, key address, key length.[1]
2F	File description address is on top of stack.
30	Character string description on top of stack is replaced by binary number indicating its length.
31	Same as store string operator (06) except destination is not padded and the description of the unused portion of the destination string is left on the stack.
32	The second byte of the instruction is put on the stack as binary number.[2]
33	The second byte and third bytes form a binary number which is put on the stack.[3]
34	VERIFY function. ' Two character string descriptions are on the stack, second parameter first.
35	INDEX function. Same format as VERIFY.

[1] Number of records.

[2] Constant.

[3] 2 Byte Constant.

Character Data Format
Character data is stored in ASCII using the codes described in the Q1 Processor Manual. If the string does not have its maximum length, it is padded with binary zeros.

Floating Decimal Number Format
The most significant bit of the first byte is the sign (one, if negative). The rest of this byte is the exponent (base 100) plus 40_{16}. The remaining bytes are digits. Thus, 3.14 is represented as $.0314 * 100^{1}$ or 41 03 14.

Fixed Decimal Format
The most significant half of the first byte is 8 if negative and zero if positive, the digits follow.

Binary Format
The least significant byte is first, except for addresses within the program.

Stack Format
FIXED numbers are converted to FLOAT when they are stacked, and converted back to FIXED when stored. FLOAT numbers are e represented the same in the stack as regular memory except that the exponent is on the top of the stack making the order of the bytes opposite.

Arrays
In multidimensional arrays, elements with the first subscript the same are stored together. The address of an array element is computed by multiplying each subscript times a predetermined constant and adding the result to a base address which may not be inside the array. This base address and the length of a single element are printed for the array when a printout of variable attributes is requested.

Procedures
An eight byte buffer for function values is allocated before each procedure in case it is a function. At the entry point there is a list of addresses where the argument addresses are to be stored, terminated by a 3 for a PL/1 subroutine or a zero for assembly language. The argument addresses are fetched from their locations.

Files
Refer to the documentation of the disk section of the operating system for the format of the file description and the files.

In order to obtain a listing of the addresses and lengths (in bytes) for each variable and constant, as well as the memory usage for the entire program, enter PL1,L FILE. To obtain memory usage only, enter PL1,S FILEA.

FLOAT XXXX
A 6 byte floating point number will be displayed from XXXX.

CHAR XXXX
A character string will be displayed from XXXX.

PATCH XXXX XX XX XX.....
A string bytes starting at XXXX will be changed to XX XX XX..... for however many bytes are entered.

Display Data
A typical instruction might be displayed as follows:

P=431F I=OB M=F5 STACK: 1.37 250 17162(430A)

The instruction is at 430F. It is a OB. Looking this up in the table above shows this to be a decimal add. It will add the two numbers at the top of the stack, 1.37 and 250. Advancing one instruction shows the result about to be stored at address 430A, a 6 byte floating point number.

P=4320 I=0406 M=F5 STACK: 251.37 17162(430A))

Binary numbers are shown in both decimal and hex since they may be used as addresses. Character strings are displayed as two binary numbers: a length and an address. 'M=' shows the byte referenced by the MEM command. It will not appear if this command has not been given.

USING THE DEBUG ROUTINE

Preparation

In order to use the DEBUG routine, it is necessary to know the addresses of labels and variables. To request a printout of these, enter the following:

```
PL1,L FILEA
```

It is often useful to put labels on the statements that are suspected of being faulty so that their addresses will be known. Also, it is advisable to put a label on the last END statment so the last address of the program will be known. This address should not be above 6800 where the regular DEBUG routine resides.

Now enter:

```
DEBUG *
```

The display will read: ENTER DEBUG COMMAND:

Commands

XXXX is used to indicate a four digit hexadecimal number. These commands may be entered any time the program being debugged is not in execution.

STEP

Each instruction will be displayed as the program is executed. The return key or the space bar may be used to advance to the next command.

T1 XXXX
T2 XXXX
T3 XXXX
T4 XXXX

After a RUN command is issued, the program will stop at XXXX and display the instruction. Using any one of these commands overrides the previous use of that command but does not affect the other T's.

GO

The program will execute until stopped by a T or MEM command.

MEM XXXX

The program will stop and the instruction will be displayed after the byte at XXXX is changed.

HEX XXXX

16 bytes starting at XXXX will be displayed in HEX.

Additional Library Subroutines

This chapter describes subroutines in the standard library which are not described in the PL/1 Programming Manual.

Disk Drive Selection

To restrict opening of files to a specific disk drive:

```
CALL CHOOSE(N);
```

where N is a binary variable or an integer.
Files will only be opened on drive N until the restart button is pushed or another call to this subroutine is made. Once a file has been opened on a particular drive READ, WRITE, REWRITE and close will operate on the drive for which the file was opened regardless of any further call to this subroutine.
To return to all drives being allowed:

```
CALL CHOOSE(0);
```

For example, suppose we wish to copy a ten record file named SCUM on drive 1 to a file of the same name on drive 2.

```
DCL SCUM FILE;
DCL BUFFER(10) CHAR(100); /*RECORDS WILL BE
      STORED HERE*/
OPEN SCUM; /*WILL BE ON FIRST DRIVE*/
DO I=1 to 10;
WRITE FILE(SCUM) from(BUFFER(I));
END;
CLOSE SCUM;
CALL CHOOSE(0); /*RETURN TO ALL DRIVES
      AVAILABLE*/
END;
```

Function Keys

Each line of keyboard input may be terminated by the return key or any of the function keys. The KEYFUN subroutine can be used to obtain the code of the key which terminated the line of input:

```
CALL KEYFUN(I);
```

I is a binary variable which receives the code of the function key. If there has not been a line of keyboard input since the last call to KEYFUN, I will receive zero. Refer to the Q1/LMC Reference Manual for the codes for the keys, keeping in mind that the codes are given in hexadecimal, whereas the compiler uses decimal. Also note that once a function key is pushed, the operating system will ignore all further key depressions until the line of input has been disposed of by the proper GET statement.

In the following example, a time consuming calculation proceeds on an array. The operator may at any time inquire about the content of any of the elements of the array by entering the subscript.

```
DCL X(100) INIT ((5)0, (10)2, (15)3, (20)4, (25)5, (25)6);
CALL KEYFUN; /*CLEAR ENTRY CODE*/
DO J=1 TO 100;
DO I=1 TO 99;
X(I)=.5*(X(I)+(I+1));
CALL KEYFUN(N);
IF N THEN DO; /*KEYBOARD ENTRY?*/
      GET SKIP LIST(K);
      PUT SKIP LIST(X(K));
      END;
END; END; END;
```

Simulated Keyboard Entry

The operating system can be made to believe that a character string was typed by the operator as follows:

```
CALL TYPIST(CH,I);
```

CH is a character variable or constant and I is a binary variable or integer giving the number of characters,

In the example below the program calls the SORT routine. A program named ROVER will be loaded after the SORT is finished.

```
CALL TYPIST(SORT FILEA FILEB ROVER↑,23);
END;
```

The ↑ character is the code for the return key. It is entered by three keys: HEX 0 D.

Machine Level Output

CALL OUTPUT (I,J);

This subroutine is the equivalent of an assembly language OUT instruction. Device address I will receive J as output. The Q1/LMC Processor Manual shows that if a 2 (binary 00000010) is output to address 1 (keyboard), a beep will be produced and the keyboard will be in the lower case mode:

```
CALL OUTPUT (1,2);
```

Line Printer

Printer output can be directed to the line printer instead of the standard printer by the statement CALL LPON(1); where the 1 is a dummy parameter, required because a library subroutine must have at least one parameter to be identified as such. Output will continue to be directed to the line printer until the restart button is pushed or a CALL LPOFF(1);.

Variable File Names

A file name may be changed from the keyboard by CALL KFILE(XXX); all references to the file will be to XXX. (XXX must be declared FILE before this statement and opened after it.) No references to the actual file name are made in the program since this is not known when the program is compiled. An example of this usage appears below.

Multiprogramming

It is possible to execute two programs simultaneously, provided one of them is written with this in mind and does not use the keyboard and display once the dual program mode is entered. The dual program mode is entered by calling MULTI. Instead of ending normally, the program should call UNMULTI when it is finished. The two programs should not interfere with each other's use of memory.

For example, the program below prints one file while another file is being edited:

```
        DCL DUMMY(64) CHAR(128);  /*SKIP FIRST 8K BYTES*/
        DCL WHAT FILE;
        DCL THAT CHAR(80);
        CALL KFILE(WHAT);
        OPEN WHAT;
        CALL MULTI(1);
LOOP:   ON ENDFILE GO TO UN;
        READ FILE(KFILE) INTO (THAT);
        PUT SKIP LIST(THAT);
        GO TO LOOP;
UN:     CALL UNMULTI(1);
        END;
```

Suppose this program were compiled and copied from * (compiler output file) to SPRINT. The sequence of commands required to print FILEB while editing FILEA would be:

```
SPRINT  FILEB
EDIT    FILEA
```

Both of these would be in response to "Q1 LMC AT YOUR SERVICE."

In the preceding example, the allocation of processor time will be taken care of quite efficiently by MULTI: EDIT will receive control of the processor when SPRINT is waiting for the printer, and SPRINT will receive control of the processor when EDIT is waiting for keyboard input.

In cases where one program is never limited by the printer, control may be turned over by additional calls to MULTI. Below is a program which adds a constant to each record of a file while another program can be executed in the first 8K bytes.

```
        DCL DUMMY(64) CHAR(128);
        DCL MY FILE;
        DCL 1 INFI,
              2 X,
              2 NAME CHAR(90);
        OPEN MY;
        PUT FILE(DISPLAY) SKIP LIST('ADD HOW MUCH?');
        GET SKIP LIST(Y);  /*MUST BE BEFORE CALL MULTI*/
        PUT FILE(DISPLAY) SKIP;  /*CLEAR DISPLAY*/
LOOP:   CALL MULTI(1);     /*GIVE OTHER PROGRAM TIME*/
        ON ENDFILE GO TO EOJ;
        READ FILE(MY) INTO (INFI);
        X=X+Y;
        REWRITE FILE(MY) FROM (INFI);
        GO TO LOOP;
EOJ;    CALL UNMULTI(1);
        END;
```

Linkage to Assembly Language Subroutines

Although the linkage is somewhat complex, it is possible to call an assembly subroutine from a PL/1 program. This makes possible high speed manipulation, communications, and auxiliary I/O devices.

Parameter Passing

The first line of the subroutine should be a list of addresses where the parameter addresses are to be stored before control is given to the subroutine. This list must be in high order byte format, so the CON operator must be used instead of the ADCON operator. The addresses from hex 4210 to 42FF should be used:

```
CON 042,010,042,012,042,014,0
```

This list means that there are three parameters, the addresses of which are to be stored at 4210, 4212, and 4214. The zero indicates the end of the list.

Coding the Subroutine

The SET operator should not be used. This is so that the subroutine will be relocatable. In the example below, the first parameter is shifted left the number of bits indicated by the second parameter.

```
START:  CON 042,010,042,012,0    TWO PARAMETERS
        LHL  04212               2ND PARAM. ADDRESS
        L,C                      NUMBER OF SHIFTS
        LHL 04210
        L,E                      LOW BYTE OF OPERAND
        IND,H
        L,D                      HIGH BYTE
        XCH                      OPERAND TO HL
LOOP:   DAD,H                    SHIFT ONE BIT
        DEC,C                    COUNT BITS
        JNZ  LOOP
        XCH
        ST,D                     STORE RESULTS
        DCD,H
        ST,E
        R                        RETURN TO INTERPRETER
        END
```

Putting the Subroutine into the Library

The subroutine must be assembled into the * file. The * file is where the compiler puts the code it generates. Assuming the source code listed above is in a file called SHIFTY, we would enter:

```
ASM SHIFTY *
```

Now use the CAT routine to put it into the library:

```
CAT
ENTER ENTRY POINT NAME SHIFT
ADDRESS? 4300
LAST ENTRY POINT? YES
```

The entry point address was the value given for the nmeumonic START (the entry point name in the source code) given by the assembler.

Calling the Subroutine

The subroutine is called in the same manner as subroutines written in PL/1:

```
I=3; CALL SHIFT(I,2); PUT SKIP LIST(I); END;
```

The result will be 12.

Note that the arguments must be binary since they were assumed to be binary in the coding of the subroutine.

Putting Subroutines into the Library

Using the three catalog handling routines CAT, UNCAT, and CATLIST, additions and changes can be made to the subroutine library file, PL1LIB.

Compiling the Subroutine

The following subroutine puts three binary numbers in ascending order.

```
ORDER: PROC(I,J,K);
REDO:  IF I>J THEN DO; N=I; J=N; END;
       IF J<K THEN RETURN;
       N=K; K=J; J=N;
       GO TO REDO;
       END;                    /*END OF SUBROUTINE*/
       END;                    /*END OF COMPILATION*/
```

In order to compile this subroutine PL1,L FILEA is entered so that the address of the entry point, ORDER, will be printed out by the compiler.

Putting the Subroutine into the Library

Immediately after the compilation is complete enter:

CAT

Answer the system's questions as follows:

ENTER ENTRY POINT NAME ORDER
ADDRESS? 4312
LAST ENTRY POINT? YES

The entry point name need not be the same as the label on the PROC statement as it is here, but it must be the name used when the subroutine is called.

In order to verify that the subroutine is in the library, enter: CATLIST
Do not put subroutines which call other library subroutines into the library.

Calling the Subroutine

Once the subroutine has been put into the library, as described above, it can be called even though it is not in the source of the calling program:

```
CALL ORDER(NA,NB,NC);
```

The type, precision, and scale of the arguments used in the CALL statement must match the parameters in the subroutine, just as if the subroutine were included in the program.

Multiple Entry Points

If a group of subroutines call each other, it is more efficient to compile and catalog them all at once, so that only one copy of a subroutine will appear in the compiler output. For example, add an entry point to the ORDER subroutine which will handle four numbers:

```
ORDER: PROC(I,J,K);
REDO:   IF I>J THEN DO; N=I; I=J; J=N; END;
        IF J<K THEN RETURN;
        N=K; K=J; J=N;
        GO TO REDO;
        END;
ORDER4:PROC(N1,N2,N3,N4);
        CALL ORDER(N1,N2,N3);
        CALL ORDER(N2,N3,N4);
        CALL ORDER(N1,N2,N3);
        END:
        END;
```

Before putting this subroutine in the library, the old copy of ORDER must be taken out:

```
        UNCAT ORDER
```

The subroutines will be entered into the library as follows:

```
        Q1/AT YOUR SERVICE  CAT
ENTER ENTRY POINT NAME ORDER
ADDRESS?431A
        LAST ENTRY POINT?NO
ENTER ENTRY POINT NAME:   ORDER4
        ADDRESS?43A5
        LAST ENTRY POINT?YES
```

Line Printer

Printer output can be directed to the line printer instead of the standard printer by the statement CALL LPON(1); where the 1 is a dummy parameter, required because a library subroutine must have at least one parameter to be identified as such. Output will continue to be directed to the line printer until the restart button is pushed or a CALL LPOFF(1);

Variable File Names

A file name may be changed from the keyboard by CALL KFILE(XXX); all references to the file will be to XXX. (XXX must be declared FILE before this statement and opened after it.) No references to the actual file name are made in the program since this is not known when the program is compiled. An example of this usage appears below.