CONVERTING SCHOONSHIP TO IBM

(Part of this text to be presented at the
Winter Project Meeting of SEAS in Lyngby, Denmark, 17-18 January 1977)
INTRODUCTION

Since 1973 the use of SCHOONSCHIP has been steadily increasing, mainly because of the fact that an appropriate documentation, support and distribution system was set up. At the moment, SCHOONSCHIP is installed in 50 computer centres all over the world. As its users are mainly researchers who often move from one institute to another, it was felt a limitation that Schoonschip is available only on CDC computers. Rather than starting from scratch and producing a new (but similar) algebra system which could run on many kinds of computers, it was decided to analyze the existing CDC-assembler code (25000 instructions), make proper documentation for it, and translate it into IBM assembler. It is hoped that in this way the debugging time of the new version will be rather short, in particular as there should not be major design errors, neither should the handling of special exceptions be overlooked. Moreover, this seemed to be the only way to guarantee that the new Schoonschip will be fully compatible with the old one.

ANALYSIS OF THE CDC-ASSEMBLER CODE

This is the hardest part of the project. The code is analyzed -- line by line -- in order to find out the exact action of the program. At the same time comments are added to the source code and a list of all variables, their uses and bit structures are made (approximately 2000 lines). The result of the analysis is written down in a PL/I-like language.

ANALYSIS OF THE PL/I AND FORTRAN CODE

SCHOONSCHIP has only 1000 FORTRAN statements in it. However, as these perform all the I/O and various conversions (e.g., from floating point to display code), they are completely machine dependent and have to be largely rewritten, taking into account the different word length, display code, number representation, etc. On the other hand, FORTRAN I/O is, in principle, independent of the operating system.

It is then tried to convert the PL/I code into a readable and machine-independent program. This is again far from trivial as can be judged from the following simple examples:

IF K = 0 THEN GO TO L1
FLAG = K

Is it important that FLAG has the value of K, or is it only relevant that FLAG ≠ 0? In which case FLAG = 1 would be a clearer statement. How many bits does one need for FLAG?
Simple statements like $K = K + 1$ are not necessarily machine-independent. If $K$ is a pointer in an array where all kinds of different quantities are stored, it might have to be rewritten as:

$$K = K + \text{length of floating point number in words } (= 1 \text{ for CDC}; = 2 \text{ for IBM}),$$

$$K = K + \text{address of next word } - \text{address of current word } (= 1 \text{ for CDC}; = 4 \text{ for IBM}).$$

After all this, a new list of all variables and their bit structures can be made, taking into account important differences:

<table>
<thead>
<tr>
<th></th>
<th>IBM</th>
<th>CDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 character in display code</td>
<td>8 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td>1 absolute address</td>
<td>24 bits</td>
<td>18 bits</td>
</tr>
<tr>
<td>1 integer</td>
<td>16 or 32 bits</td>
<td>60 bits</td>
</tr>
</tbody>
</table>

**THE CHOICE OF PL/I**

The following statements are representative for the PL/I language in which SCHOONSCHIP is rewritten.

```
LABEL:    A=B+C;
          GO TO LABEL2;
          IF EXPR1 < EXP2 THEN DO; K=K+1;
              L=L+7;
          END;
          ELSE K=K-1;
          DO J=1 TO J2 BY 3;
              IBUF(J)=IBUF(J+1); /* SHIFT */
          END;
```

The major power of PL/I for our application are statements of the following type:

```
FILES(K).NAME='NEWFL';
FILES(K).LOC=MBU+1;
IF FILES(K).KEEP=0 THEN GO TO DELETE1;
```

This goes together with the knowledge that "FILES" is an array of 60 bit words, each of them having the following bit use:
<table>
<thead>
<tr>
<th>name in</th>
<th>no. of args</th>
<th>properties</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>display code</td>
<td>6 bits</td>
<td>3 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td>5 characters</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bit subfields are defined in a "DECLARE" statement and can then be used as if they were normal variables. Clearly, only the "DECLARE" statements are machine-dependent. All the other statements look invariant.

CONSIDERATIONS ABOUT FIXED BIT PATTERN

We could convert the above array "FILES" to IBM by defining:

<table>
<thead>
<tr>
<th>name</th>
<th>properties</th>
<th>location</th>
<th>no. of args</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 + 8</td>
<td>8 bits</td>
<td>16 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

(5 chars)

As 1 array element is no longer an integral number of words (or even of bytes), the addressing of FILES(K) gets very complicated. (But PL/I has been set up so general that it can cope with this!)

To simplify the addressing, we have defined suitable subarrays:

Consider the arrays F$NAME(K) : 5 bytes per element
F$PROP(K) : 1 " " "
F$LOC(K) : 2 " " "
F$ARGNR(K) : 1 " " "

So now a file name and its properties are no longer adjacent, but they are related to the same K. As there are exactly 8 properties, these can be tested by TM (test under mask) instructions and set by 0I or NI (Or immediate, And immediate) instructions.
The other array elements can be loaded as bytes, half words or character strings.

The PL/I compiler knows the bit structures of all SCHOONSCHIP variables and does the described array transformation.

CONSIDERATIONS ABOUT MOVING BIT PATTERN

Trouble starts when the same array can have different bit structures which are determined dynamically. Consider as an example the case of input cards representing nested do loops (SCHOONSCHIP notation).

```
DO J = 1,5
  statement 1
DO K = 3,J,2
  statement 2
ENDDO K,J
```

This is translated for CDC (inside the array IT) to

```
0000 pointer  J  1  0  1
| name in      | value | value |
| display code |

0  5
| value |

0000 pointer  K  3  0  3
| name in      | value | value |
| display code |

J  1
| name in      | value |
| display code |

0077 pointer  0077 pointer  7777...76
|  | pointer | terminator |

The rule is the following: (knowing that 00 and 77 are impossible display characters).
A word starting with

0000 is a DO statement. It is followed by 4 words: J, J1, J2, J3 referring to
the general form DO J=J1, J2, J3.
Each word has 30 bits for the display code of the name (if present, Else = 0) and 18 bits for its value.

0077 is an ENDDO statement, containing a pointer to the beginning of the loop.

7777 is the terminator of the entire loop.

other: is a statement inside the loop. It can have any length. As blank suppression takes place, 00 is needed as the separator of the cards.

To represent such structures in a machine-independent way is far from trivial. We chose the following approach which is an extension of the PL/I features DEFINED, BASED and POINTER.

First we define several possible "structures" for the array IT.

<table>
<thead>
<tr>
<th></th>
<th>is bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT(K).DODIR</td>
<td>1-12</td>
</tr>
<tr>
<td>IT(K).DOJMP</td>
<td>43-60</td>
</tr>
<tr>
<td>IT(K).JNAM</td>
<td>61-90</td>
</tr>
<tr>
<td>IT(K).JVAL</td>
<td>103-120</td>
</tr>
<tr>
<td>IT(K).J1NAM</td>
<td>121-150</td>
</tr>
<tr>
<td>IT(K).J1VAL</td>
<td>163-180</td>
</tr>
<tr>
<td>IT(K).J2NAM</td>
<td>181-210</td>
</tr>
<tr>
<td>IT(K).J2VAL</td>
<td>223-240</td>
</tr>
<tr>
<td>IT(K).J3NAM</td>
<td>241-270</td>
</tr>
<tr>
<td>IT(K).J3VAL</td>
<td>283-300</td>
</tr>
<tr>
<td>IT(K).LOOP</td>
<td>301-...</td>
</tr>
</tbody>
</table>

or

<table>
<thead>
<tr>
<th></th>
<th>bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT(K).CARD</td>
<td>1-54</td>
</tr>
<tr>
<td>IT(K).CHARR</td>
<td>55-60</td>
</tr>
<tr>
<td>IT(K).NEXTC</td>
<td>61-...</td>
</tr>
</tbody>
</table>

and more such structures. This leads to a high degree of machine independence, as we can now refer to IT(MBE).J3NAME rather than to IT(MBE +4).NAME, since this last quantity could well become IT(MBE+7).NAME in IBM with shorter words.

Of course, when we come to the end of a particular structure, the starting point of the next structure has to be given. This is why the variables LOOP and NEXTC were introduced.
Here we have to use the address function:

\[ \text{MBE} = \text{ADDR(IT(MBE)).LOOP}; \]

which brings us to the beginning of the first card of the loop. (N.B. the addressing in the IT array is absolute.) In practice, the PL/I compiler knows all the names of these structures and determines which sequence they belong to. It then generates unique names, e.g.,

\[ \text{IT(MBE).JZVAL} \text{ becomes T$1JZVAL(MBE)} , \text{ indicating the first sequence;} \]
\[ \text{IT(K).CHARR} \text{ becomes T$2CHARR(K)} , \text{ indicating the second sequence.} \]

These names can now easily be looked up in a table (which is the only difference from one machine to the other) so as to find the required word and bit position.

ADVANTAGES AND DISADVANTAGES OF A HIGH-LEVEL LANGUAGE

From the point of view of support for a large program, only the assembler is a stable and reliable language: FORTRAN libraries are modified every so often; all PL/I compilers do not produce the same code (depending on how much the host installation is willing to pay for them); the PASCAL compiler might not exist anymore under the next IBM operating system, etc.

Moreover, some parts of SCHOONSCHIP require optimized code or do tricky bit handling which is only possible in the assembler. Therefore, intermixing of the high-level language with the assembler is a must (making a call to an assembler subroutine gives too much overhead in IBM).

It was felt that for a program like SCHOONSCHIP, a PL/I (slightly extended) was an appropriate language to formulate algorithms in; however, the PL/I program is not necessarily the real source code! Consequently, a small PL/I compiler was written (1000 lines of FORTRAN on CDC) that converts the PL/I dialect into readable IBM-macro calls. This output is then the basis for the SCHOONSCHIP source code.

The statements that were too complex or ambiguous for the compiler are inserted in by hand. At this stage as well, the original routine is cut into very small pieces so as to fit into the IBM addressing scheme.
MACROS FOR THE IBM ASSEMBLER

The aim was that the output of the PL/I compiler should be a readable program, of which the expansion into assembler is predictable. This can be achieved if a proper set of macros is chosen and if the PL/I compiler transmits the PL/I comments onto the macro calls. The syntax of some macros might look a bit involved, but this is usually done to simplify its expansion. Anyway it does not matter much as the macro calls are generated by the compiler and are not actually punched by hand.

The set of macros chosen is the following (1000 lines all together):

I) Names of 5 letters, character strings, bits or numerical variables can be set or be tested:

\[
\text{if } X(3) \geq A(K)+B-C \text{ THEN GOTO LABEL;}
\]

\[
\text{CMPVAL } X,3,(+A,K,+B,O,-C,O),\text{GE,LABEL}
\]

\[
A(K)='ABC' ; \quad /* \text{NR OF CHARs} = 3 */
\]

\[
\text{SETCAR } A,K,C,'ABC',3 \quad \text{NR OF CHARs} = 3 */
\]

\[
\text{IF EXPR(J).PRINT = 1 THEN GOTO LABEL;}
\]

\[
\text{CMPBIT EXPR,J,PRINT,ON,LABEL}
\]

\[
\text{ANAME(3)=BNAME ;}
\]

\[
\text{SETNAM ANAME,3,BNAME,0}
\]

\[
\text{IF A >= 0 THE GOTO LABEL}
\]

\[
\text{CMPOOO A,O,GE,LABEL}
\]

The use of these macros eliminates the following difficulties (which are typical for IBM):

1. Addressing: look up arrays to see to which COMMON block they belong so as to know with which base register to address them. (A later change of an array from one COMMON block to another is then not a big change in the program.) This will be discussed in more detail later.

2. Even for code, written directly in assembler, the macros

\[
\text{LOAD, STORE, ADD, SUBTR, COMP, LADR}
\]

\[
\text{&REG, &ARRAY, &INDEX}
\]
are extremely useful, e.g.,

\[
\begin{align*}
& \text{LOAD 1,A,5} \\
& \text{SLL 1,2} \\
& \text{STORE 1,B,K}
\end{align*}
\]

Obviously, one needs some firm conventions of register use so as never to modify the registers required for addressing, e.g., LOAD 10,A,5 should lead to an error message during macro expansion if Reg 10 is a base register.

3. The error-prone mnemonic BH, BL, BNH, ... are eliminated in favour of the operators GT, GE, LT, LE, EQ, NE, ON, OFF. Moreover, the operator inversion, required by tests on zero, is done by the macros. Example:

Suppose Reg 0 contains zero throughout the program. Then

\[
\text{IF A >= 0 THEN GOTO LABEL;}
\]

becomes:

\[
\begin{align*}
& \text{compare A with 0 ; jump to LABEL if NOT LOW} \\
& \text{or : compare 0 with A ; jump to LABEL if NOT HIGH}
\end{align*}
\]

This latter translation can be implemented efficiently:

\[
\begin{align*}
& \text{C 0,A}
\end{align*}
\]

\[
\text{BNH LABEL}
\]

II) There is a further set of macros for:

- calling assembler subroutines
- calling FORTRAN subroutines
- prologhs and epiloghs
- doloops

III) Finally, there are the dumping macros: e.g.,

\[
\begin{align*}
& \text{LOOK$ 'CHECK'} \\
& \text{PRINT$ 'A',A,1} \\
& \text{PRINT$ 'ARRAY',ARRAY,50}
\end{align*}
\]

These can be inserted anywhere in the source code and will produce a print-out in the sense of

\[
\begin{align*}
& \text{CONT. OF REGS AT CHECK, CALLED AT ABS.ADDR......, REL.ADDR} \\
& \text{RO = ... till R15 = ... CC = ...} \\
& \text{CONT. OF A, LOCATED AT ABS.ADDR......, REL.ADDR......} \\
& \text{...}
\end{align*}
\]
CONT. OF ARRAY, LOCATED AT ABS.ADDR...REL.ADDR.....

...

During this operation, neither registers nor condition code have been changed.

The same mechanism comes into action when SCHOONSCHIP detects a user or internal error: a dump of the registers and of all relevant parts of memory is printed on request.

ADDRESSING IN IBM ASSEMBLER

For the following, it is important to know that SCHOONSCHIP has a main program (or basic overlay) and 4 first-order overlays. Each of them has its own COMMON and LOCAL variables associated with it.

The following base register use was chosen:

```
BLANK DSECT
    Main COMMON symbols
    Main COMMON arrays
INCOM DSECT
    in COMMON symbols
    in COMMON ARRAYS ) or similar for other overlays
BEGIN CSECT
    BALR 12,0
    USING *,$12
HERE   L 10,=V(BLANK)
    L 11,=V(INCOM)
    USING BLANK,10
    USING INCOM,11
```

When we reference the variable "VAR" in L 1,VAR then we mean in fact:

```
L 1,VAR-HERE(0,12) or
L 1,VAR-BLANK(0,10) or
L 1,VAR-INCOM(0,11), depending on which data block VAR belongs to.
```

This look-up is done by the assembler.

Unfortunately, when we reference arrays, this is no longer possible, as the displacements are likely to be larger than 4096. For loading ARRAY(K) we have
to do:

L 1,K
SLA 1,2
A 1,=A(ARRAY-BLANK-4)
L 3,0(1,10)

and obvious changes in the case ARRAY belonged to another data block. Here the assembler does not help us at all. As the programmer should not have to check for each instruction, which data block his variable belongs to, this task was given to the MACRO-PROCESSOR.

To achieve this, we defined all variables in

BLANK as P type variables, with proper length
INCOM " Z "
LOCAL " F "

Then a basic addressing macro was written that takes into account the data block and the length of the variable (300 lines!).

Example:

If F$ARGNR belongs to INCOM (see higher: half word variable), and we have somewhere the definition

F$ARGNR DS 256ZL2

the statement

LOAD 3,F$ARGNR,K expands automatically into
L 1,K
SLA 1,1
A 1,=A(F$ARGNR-INCOM-2)
LH 3,0(1,11)

These macros can deal with quantities of 1, 2, 3, 4, 16 bytes long. Moreover, if they recognize the array name to start with "T$" then they generate code for "moving bit pattern" as explained before.

For this purpose we first define all the possible bit structures as DSECTS:

T$1 DSECT
T$1DODIR DS 1FL1
T$1DOJMP DS 1FL3
T$1JNAM DS 1FL5
and a statement like: \texttt{MBE = ADDR(IT(MBE)).LOOP} is then compiled to
\begin{verbatim}
LADR 2,T$1LOOP,MBE
STORE 2,MBE,0
\end{verbatim}
which expands into:
\begin{verbatim}
L 1,MBE
 USING T$1,1
 LA 2,T$1LOOP
 DROP 1
 ST 2,MBE
\end{verbatim}
The programmer has never to know the detailed bit use in each "structure".

SEARCH FOR AN UPDATING PROGRAM ON IBM

For programs of the size of \texttt{SCHOONSCHIP}, the use of an updating program is mandatory. The availability of a good, powerful updating program simplifies things enormously. The following properties are felt as essential:
- possibility to add and delete cards;
- a card duplication mechanism to allow \texttt{COMMON} variables, equivalences, etc., to be defined only once in the entire program (once for each language used at the most);
- possibility to modify a few characters of a card.
The following features are very convenient:

- mechanism to convert COMMON declarations in FORTRAN to COMMON declarations in assembler;
- possibility to list part of the new subroutine during the update run;
- possibility to have cross-reference list (indicating variables and line numbers as in the source of the updating program).

The IBM products IEBUPDTE and IEBUPDAT clearly lack most of the above-mentioned facilities.

The PATCHY program (CERN program library) has the card duplication mechanism in it, but is still very primitive. It received a lot of publicity since it is available on several types of computers. The ELECTRIC program (Rutherford High-Energy Laboratory) is considerably better and has the character manipulation facility. Unfortunately, it is only available at the RHEL computer. Moreover, the number of users that can use it at the same time is limited (in batch or interactive!), therefore its use requires a lot of patience and repetitive job submission. If anybody here present knows about a good updating program, please let me know!

PRESENT STATUS OF THE PROJECT

The conversion of the CDC assembler to PL/I is expected to be finished in the very beginning of 1977. The PL/I version will contain approximately 10,000 lines.

Half of this (the main and input overlay) was compiled in the beginning of 1976, leading to 7000 lines of macro calls (+3000 lines of macro definitions, COMMON blocks, etc.). They were then processed by PATCHY for updating. The testing of this part was done on the IBM 360/195 of RHEL, and it was finished around 1.7.1976. We hope that the full program will be available in early 1978.