A COMPUTER PROGRAM TO TRANSLATE
MACHINE LANGUAGE INTO FORTRAN

William A. Sassaman
TRW Systems, Inc.
Redondo Beach, California

This paper describes a computer program which translates machine language into FORTRAN. The program was developed at TRW, Inc., to aid in the conversion process from our existing equipment to a third generation computer. The translator was written to be a real help to people personally involved in conversion, and is intended to be an operational program rather than a pure research project.

As the title of this paper indicates, the output language is FORTRAN. Since the translator design is not very dependent upon output language, this appears to be arbitrary. FORTRAN was chosen since it is a standard. It is well defined and runs on both our second and third generation computers. It has deficiencies but they are known. It was desired to go to a problem-oriented language to increase future human productivity and therefore machine language output was not encouraged. In addition, compiler writers expand considerable effort to obtain efficient machine language codes and duplicating this effort appears a waste. Anyone who really wants machine language can get it as a by-product of the FORTRAN compilation and hand-massage it to any degree of perfection he desires. If the problem is so complex that no FORTRAN translation is possible, then a completely human effort appears in order.

The input is symbolic assembly language for the IBM 7000 series computer operating under a privately developed execution supervisor. The symbolic media was chosen over the binary machine language since the symbolic cards make it easier to distinguish between the various types of data, allow macro identification and contain otherwise useful information. Since continuity of usage is expected between the original and translated versions of the program, it appears highly desirable to maintain much of the symbolic notation.

In my original thinking of the translational process, I was impressed with the concept that actual translation was, in general, a clerical process rather than an inventive one. That is, the programmer following the assembly listing figures out from it (and any documentation) what the original programmer was doing and codes this in the appropriate language for the new machine. Although often it is necessary to have a knowledge of the problem being solved, much of the time the translating programmer operates as a clerical symbol manipulator. It is true that the rules for the symbol manipulation are complex, but the task is basically clerical and therefore subject to automation. In the translator I have tried to assign these simple clerical tasks to the machine and allow the human more time to perform in the areas where he can contribute the most.

From a technical viewpoint, it is probably impossible to write a program which will translate all of one computer program into a similarly efficient program for a second computer. However, as with
many mathematical processes, it is feasible to approach the solution as a limit, such that a maximum automated transfer of source programs may be effected with minimal cost and human intervention. The task of writing a translation program has as its major obstacles the definition of the rules for translation. It would seem unlikely that anyone could a priori define all the rules. Therefore a learning approach has been defined to allow the development of the model as experience is gained in translation.

I did not try to design for a 100% translation. Input/output, functions, subroutines, standardized routines, etc., need not be translated. Further the conversion effort is not by nature one that may be completely automated. During the process of converting a program, decisions are made as to the plan of attack during conversion, i.e., the human programmer, who has cognizance of the physical problem being solved and the capabilities and shortcomings of the program, decides which areas are to be rewritten, which areas are to be deleted, which areas will be replaced by system subroutines or standardized routines, and finally the remainder is to be translated (or transliterated if you wish). It is rather apparent that these decisions are probably not the sort to be made by the computer. Further, the rewriting or regrouping of computations must also be performed by humans. The remaining area, translation, is a potential area for automation. Of course, I try to do a good job in that which is translated; however, the law of diminishing returns dictates that the translational rules limit one to about 90% of the code.

Further, I concluded that the source machine language program contains a great deal of information and the translator, retrieving and organizing this information could perform a very valuable service in documentation as well as aiding the conversion effort.

The translator during translation attempts to operate as the human does. The programmer in translation recognizes in coding not only the individual instructions but also, and more specifically, problem-oriented functions, which may be one or more machine language instructions. It is the purpose of the translator then to recognize these functions with their terminals as well as to gather and organize the program information pertaining to the translational rules. The functions are gathered into statements as appropriate before output.

Many of these functions are easily recognizable as functions are arithmetic codes. These may easily be built up into larger statements. One of these arithmetic statements consists of a string of functions appropriately connected. The translator inserts a right parenthesis prior to each multiplication or division, and a left parenthesis following each square root or other function. A similar left or right parenthesis must be entered at the start of the statement. The statement is normally terminated by a store instruction. The address of this store instruction is obviously the left side of the FORTRAN statement and followed by an equal sign. The method of translating addresses illustrates the buildup of rules for translation. Consider the coding:

1.) CLA A
2.) AXT 10, 4
3.) D FAD A + 1
4.) FDP B, 4
5.) STQ C + 10, 4
6.) TIX D, 4, 1

The first A is obviously translated as A. Since A = A + 1 must be translated as A (2). Instructions three through six are translated as a DO loop using the dummy variable ND X 4. Since it is subject to index register modification and has no additive address, the B is assumed to be a vector running backwards in storage and is translated as a forward running FORTRAN array B (ND X 4). The C is assumed to be forward running and is translated as C (ND X 4 + 10 - 10), where the tens cancel out leaving C (10). Obviously the D is translated as a FORTRAN numeric statement number. Functions may be more complex and require more complex rules for translation. A good example of these are the programmer tricks of using instructions for something the manufacturer never intended; for the IBM 7000 series, a PXD 0, 0 will clear the accumulator; a LRS of zero will impose the sign of the accumulator into the MQ, etc. Such translation is analogous to the handling of idioms and slang in human language outside of a word for word grammatical translation.

The last bit of philosophy in the design of the translator is the target. The programs to be converted are engineering applications involving algebraic algorithms. These algorithms are easily defined and form the basis of the translator rule set.

As a result of these thoughts, the translator was designed to intimately interact with and operate under the supervision of the human user. The human describes the rules for the particular pro-
gram involved via control cards, defines areas to be
translated and criteria for recognition of areas of
coding to be translated as FORTRAN subroutines.
Operating with these rules and the basic set, the
computer then performs any initial translation.
This initial attempt normally tells the user what
tasks cannot be handled in FORTRAN, indicates
the need for additional rules such that the translator
will give a better translation. The deck is then re­
submitted to the computer. The human examines
the computer output and either edits it to achieve
the desired code or redefines the rules or control
cards and translates over again. This learning proc­
ess and human interface dictates the need for a
system to afford maximum convenience and ease of
communication to the user. Although this would
appear to be an ideal on-line application, the sched­
ule, hardware and manpower available dictated the
utilization of a typical centralized large-scale com­
puter.

OPERATION

The translator’s functions are to retrieve informa­
tion from the source deck, organize this informa­
tion, merge it with other data, apply the rules for
translation and provide interfaces with the human
during the process. This is not done on-line, al­
though the nature of the problem indicates an on­
line solution might enhance the process. In order
to accomplish these functions, the translator is de­
signed in six separate (and recoverable) phases. The main task of each of these phases is:

Phase I Separate the program into log­
ical groups.
Phase II Handle parameters — data
types, dimensions, COM­
MON, initial values.
Phase III Core map of symbol alloca­
tion and overlay.
Phase IV Translation of macros.
Phase V Translation into FORTRAN,
routine by routine.
Phase VI Editing and merging.

Although these are the main functions, the phases
have additional tasks because of convenience of
execution. In order to explain the process, I will go
through it phase by phase, explaining what is done
and where the information comes from.

Phase I is the initial phase whose primary task is
to divide the program into logical groups of man­
geable size. The input to the translator is the
source symbolic card deck, or a tape containing the
card images. The input is read, basically a card at a
time, and broken into the following categories.

- Areas to be treated as FORTRAN sub­
routines (tape)
- Data and parameters (tape)
- Symbolic equivalence (core/cards)
- Macro skeletons (tape)

In order to make these distinctions, the translator
must know the algorithms for separation. The
BEGIN pseudo operation is recognized as the start
of a routine and the terminus of a previous routine.
Origin and transfer cards are assumed to signal the
end of a routine. Decimal, octal and Hollerith data
are presumed to be in the data domain unless they
appear to be in a routine and do not have a sym­
bollic location assigned to them. The programmer
is allowed to enter control cards in the data stream
to allow the programmer to label name COMMON
in the output stream. Each card included in a sub­
routine area is assigned a FORTRAN location
number.

In order to build up an initial table of floating
point (real) symbols, the address of each floating
point instruction is saved as well as the address of a
loading instruction immediately before it or a stor­
ing instruction immediately following it. Each sym­
bollic name is saved, in sequence, with all origin,
intermediate transfer cards and the final end card.
The address of the first intermediate transfer card
(if none, the address of the end card) is saved as the
point at which computation will be initiated.

Phase II is designed to handle data and param­
eters. The list of floating point symbols generated in
Phase I is organized and checked for redundancy.
The translator then reads the data and parameter
tape and compares the parameter symbolic name to
the built up symbol table equivalences. The symbol
is then checked to see whether it is an allowable
FORTRAN symbol (alphanumeric, initiated, by a
letter), what the type is, and what the dimension is.
The translator tries to define a new symbolic refer­
ence for illegal symbols, expands the symbol table of
equivalences and builds up a table of real and
integer symbols not conforming to the FORTRAN
rules. Every attempt is made to keep the original
symbol as it is assumed to be mnemonic. From the
contents of the data card image, a data statement is
generated if initial values are assigned to the pa­
rameter. The name and dimension are included in
the name COMMON block as assigned by the
translator or the programmer if he has such input information. Upon termination of a name COMMON block, the card images are saved on an intermediate tape with the images of the type statements.

Upon processing all data cards, the translator calls the computation, using the address of the end card (or first intermediate transfer card). A listing is then made of all parameters with their original symbolic name, the corrected name, if any, and the comments from the data card.

**Phase III** (storage allocation core map and overlay structure). The program reads the tape containing the above information, compares it to the equivalence table and breaks it into \( n \) strings. These strings are then printed in \( n \) columns with origins matched on the vertical scale.

**Phase IV** (macro translation). Macros are currently translated as functions, if they can be, or are ignored. Details of translation are similar to the translation of subroutines.

**Phase V** (translation into FORTRAN subroutines). This is what most people consider the heart of translation. The card images of the area to be converted are read into core from the tape storage. Each card is assigned a sequential external formula number. Those which are not used will be suppressed at output time. An initial pass is made to find the address of all transfer instructions and to save the concomitant FORTRAN numeric location. Locations which are transfers, or transferred to, are appropriately flagged. New numeric locations are assigned for undefined transfer addresses. Special flags are set for the addresses of TIX instructions, TXI and TXH (under certain circumstances index loading instructions, address modification, etc.).

The actual translation is now begun. The translator is broken into two alternate paths here: the first being a search for instructions or functions that initialize a statement (a statement being merely a string of appropriately connected functions); the second being a search for functions that sustain or terminate a statement. In general, the translator scans the coding until it recognizes the start of a statement; then it switches to the terminal branch where it builds up the function into a statement until some terminating condition is reached. If on the initial scan, a sustaining type instruction were encountered, the translator initiates an appropriate function to start things off and transfers to the terminal branch. Similarly if in the terminal branch the translator finds an initiating statement, it supplies a terminal function to complete the statement being processed, tries to search out particular programmer tricks since something "different" is happening, and then transfers to the initial branch. Loading, storing and transfer instructions, and instructions which are transferred to, are samples of what are considered to start or end a statement. CALL type instruction (TSX) are considered to start or end a previous statement and start a new one unless the translator can determine that they are replaceable by a built-in arithmetic function or other functions, in which case the function is included in the statement being processed and the translator continues on the terminal branch. Arbitrarily terminated or initiated statements are stored in or picked up from "dummy" accumulator, MQ, registers, etc.

The translator on either branch attempts to search out TIX loops, where a register is counted down from \( n \) to 1 or TXI loops where a register is incremented from 0 to \( n \) and translates as DO loops nested to a level of 7; if all 7 index registers are used and the index registers are not saved internally. A DO statement is inserted just prior to initiation of the loop. The loop is terminated by a dummy CONTINUE. The pseudo symbol NDXA is used to represent index register A. The algorithm for conversion of parameter addresses while in a TIX loop deducts the initial value of the index register from any associated address. Note there is a difference in the assignment of vectors in machine language programs and FORTRAN, each considering the other backward. The algorithm attempts to cover the difference.

As previously mentioned, the easiest instructions to translate are the arithmetic instructions where each operation and address is added to the right-hand end of the statement being generated. For multiplication and division a pair of parentheses must be added, one at each end before the operator and operand are saved. For functions, a similar pair of parentheses must be added except of course, the functions appear on the left, before the initial parenthesis. A storing type instruction adds an equal sign on the left and the address of the operand to the left of that. Translated output statements are built up in a table, word by word, until a terminating condition is reached. The statement is compacted by reducing spurious blanks, continuation numbers are assigned if more than one card image is required; and the images are written on a blocked output tape with an alter number for each card image. If it has been referenced, the FORTRAN numeric location is also written out on the
A COMPUTER PROGRAM TO TRANSLATE MACHINE LANGUAGE INTO FORTRAN

first card image. At this point an almost side by side listing of the original coding and the translated code are printed for the user.

By actually scanning, instruction by instruction, while the whole subroutine is in core, the translator can look for particular sequences of coding which represent a special case. Much of the detail coding to effect these rules is lengthy and tends to be repetitive so that many subroutines are used in the areas of duplication. The coding of the translator is all very straightforward, and often tedious.

Phase VI allows the user to edit the translator output from the previous phases which consisted of an almost side by side listing of the original and translated coding. Also used is the tape which contains all of the previous translation including the alter numbers. It is the purpose of Phase VI to allow the editing of this tape by use of the alter numbers and to produce a new tape and/or a new listing and/or a punched card deck. The editing is performed by a series of control cards which allow the user to add or delete cards from the tape or to juggle large blocks from one place on the tape to another, without actually shuffling through the cards.

To maximize usefulness, the output tape may be fed into the FORTRAN compiler at this time without the submittal of a separate run or punching the cards.

USER INPUTS

The recipe for elephant stew traditionally starts with "clean one freshly killed elephant." Similarly the user’s input starts with the program to be translated. This may be in the form of symbolic source cards or blocked card images on a tape. In general, the areas for which translation is not desired are deleted from the deck. This deck is preceded by a control card that tells the translator whether this is a SMASHT, SCAT or IBMAP deck and whether cards or tape is expected. A number of EQU cards may be input by the user to assign names to illegal symbols, rather than accept the translator’s naming. REAL and FIXED define the type of data these operations refer to when the translator has insufficient information to arrive at the appropriate conclusion. Control cards define the subroutines of function calling sequences for translating the TSX address. The rules for separation of areas into subroutines may be defined either by special coding in the translator or insertion of dummy control cards. Special algorithms for translational rules are coded into the translator at this time. The run is now ready for submittal.

Upon return of the run, the data previously furnished may be modified and the appropriate processes repeated or the user may desire to continue into the editing phase and obtain an output deck.

SAMPLE TRANSLATION

Translation is such a complex function that no all-encompassing sample is feasible in such a short period of time.

CONCLUSION AND SUMMARY

The translator described here is not a perfect tool—it does not translate everything nor is everything it translates perfect. It does not handle dynamic programming, i.e., where coding is actually charged, nor does it handle indirect addressing. Patently it does not translate into FORTRAN those things FORTRAN cannot do. Complex and double precision arithmetic are not attempted. It is designed to relieve the programmer of much of the clerical task of translation and to allow the user input into the translational process and absolute control of the final output. For those applications we have used it for, it has performed rapidly and effectively.