Running Pegasus Autocode Programs on Mercury

By A. Gibbons

A new general method of translating autocodes has been used to translate Pegasus Autocode programs into Mercury machine code. In addition to describing this translation scheme, the problems connected with simulating Pegasus Autocode on Mercury are discussed. The specification of the program is appended to the paper.

Introduction

Pegasus Autocode, a simplified method of preparing programs for the Ferranti Pegasus computer, has been described in some detail by its authors (see Clarke and Felton, 1959).

The autocode deals with two kinds of numbers, namely indices and variables, which are denoted by nN and vN respectively. There are 1,380 variables, which represent floating-point numbers. These are recorded to a precision of 29 bits and have binary exponents in the range ± 256. They can be modified by the 28 indices, which are fixed-point numbers in the range $\pm 8{,}192$.

Arithmetic can be done with both kinds of numbers, and typical instructions are:

$$v1 = v2 + v3$$
$$n1 = n2 + n3$$
$$vn1 = v(-3 + n2) - v(2 + n3).$$

Simple functions of the variables can also be calculated, as in the instruction

$$v1 = \log v2.$$

An autocode program consists of a series of such instructions together with input, output and jump instructions, and is followed by one or more instructions enclosed in brackets. When the program tape is fed into the machine, the instructions are decoded and stored, the first of the bracketed instructions being obeyed when the terminating bracket has been read. If one of the bracketed instructions transfers control to the main program, the calculation proper is started, otherwise more instructions are read.

This scheme of coding is well suited to Pegasus, since the small high-speed store is partly occupied by floating-point arithmetic routines, and information can be transferred between the stores either in blocks of eight words or in single words of 39 bits.

The Ferranti Mercury computer is both larger and faster than Pegasus. It is a floating-point machine with a word length of 40 bits, and the numbers conveniently correspond, in both range and precision, to those of Pegasus. The high-speed store consists of 1,024 words and is supplemented by a drum store of some 16,384 words. Information is transferred between the two in blocks of 32 words.

The instruction code is of the single-address type, and ten-bit B-lines are provided to modify the addresses. Each instruction occupies a half-word, and, although instructions can only be obeyed in the first half of the store, the partitioning between instructions and numbers is arbitrary.

Programs for Mercury are usually divided into *chapters*. These are stored on the drum and brought into the high-speed store, one at a time, as required. A chapter change takes about 180 msec, compared to the addition time of 180 μsec. Consequently it is desirable that each chapter corresponds to some major part of the calculation.

Problems of Representation and their Solution

Although it is possible to simulate Pegasus Autocode exactly on Mercury, the resulting system would be inefficient, the prime reason being that it would not be possible to have immediate access to all the 1,380 variables. Normally there are three variables in each instruction. The variables must be stored on the drum and it would be necessary to test if each variable required by the instruction were present in the store. The time taken to do this would far outweigh the actual instruction time. Another disadvantage is that direct modification by a ten-bit B-line would be impossible.

Both these difficulties can be overcome by restricting the number of variables to, say, 600. They would then occupy only a little over half the store, leaving plenty of room for instructions, and could be modified in the normal way. This restriction is not serious and most programs can be adapted to use only these variables.

The range of values which the indices can take must be preserved, as they can be used for ordinary arithmetic. It is also necessary to have them readily available for use as modifiers. To achieve this, the indices are stored as unstandardized floating-point numbers. Unrounded arithmetic operations are quite straightforward, and the least significant ten bits of the index, the part used for modification, can be picked up immediately into a B-line.

The next problem is that of partitioning the instructions into blocks which will fit into the store of Mercury. Each translated autocode instruction, which consists of from two to twelve Mercury machine instructions, is packed in the store. No record of the individual instruction lengths is kept and, as a consequence, certain instructions cannot be translated. One of these is

$$\text{tape } N_1, N_2$$

which reads more instructions into the program, beginning N_2 instructions after the one labelled N_1.

It would be unrealistic to restrict the program to a

length that could be wholly contained in the high-speed store, since this only holds about 80 autocode instructions. Yet care must be taken in dividing up a long series of instructions since, if a chapter change occurred in an "inner" loop, it could be very time-consuming. Several methods have been considered for solving this well-known problem, and the one finally adopted was suggested by Dr. D. Morris of Manchester University.

The instruction space is divided into four equal blocks numbered 0 to 3. On input, the instructions are translated and arranged on the drum in blocks of the same length. The number of the block on the drum, modulo 4, indicates where the block will go in the store, and the internal addresses of the block are adjusted accordingly. Consequently, any four consecutive blocks can be in the store at one time. In practice, this means that if a loop in the program does not span more than about 60 autocode orders, it can be obeyed with the minimum of drum transfers.

A label list is kept in which there are two entries for each label, the line and block numbers corresponding to the beginning of the translated instruction. All jump instructions transfer control to a set of orders which test if the block required is already in the store, and bring it down if it is not.

No attempt has been made to translate Pegasus machine orders. However, Pegasus Autocode permits the insertion of some machine orders at the beginning of a tape in order to alter the storage space allocated to instructions, labels, indices and variables. When running the program on Mercury these are unnecessary and are omitted, as the number of variables is restricted to 600, for the reasons stated above, and the drum can accommodate about 2,000 autocode instructions, which is rather more than Pegasus itself holds. Varying numbers of labels and indices are allowed for by storing them end to end, with the indices numbered backwards. If L is the highest numbered label and N the highest numbered index, then

$$2N + L \leqslant 184.$$

Functions of variables are calculated by standard routines known as *quickies*. In a conventional Mercury program, the quickies required are included, as blocks of instructions, in each chapter. This is not practicable in this scheme and, instead, the quickies are stored on the drum and brought down to a particular part of the store as required. To save time, a quicky is not brought down a second time if it is already present. A reciprocal routine is always present in the store.

The routines for reading data and printing results are adapted from standard Mercury subroutines. They are stored on the drum and when required are transferred to the space normally occupied by the autocode instructions in the store.

These subroutines, together with others for changing blocks of instructions, selecting labels and functions, form the "background" for the translated Pegasus Autocode program.

Translation

Having decided upon a method of representing Pegasus Autocode programs on Mercury, it is necessary to find a suitable method of translation. The method described below is general in its application and was suggested by Mr. R. A. Brooker, who is developing a similar scheme for the Atlas computer (Brooker and Morris, 1960).

The input routine recognizes two forms of input: firstly, a description of the autocode instructions and how they are to be translated, and secondly the instructions themselves.

The instructions are described by defining the various classes of symbol strings which can appear. Each class is identified by a letter and, to distinguish class identifiers from basic symbols, the former will be written as upper-case letters. No confusion can arise in the text, since only letters of one case appear in a Pegasus Autocode program. However, when actually punching the tape, special devices, described below, have to be sought, because the Creed teleprinter used in preparing data for Mercury has itself only a single case of letters. Each class is defined by giving a dictionary of its elements, which may be strings of basic symbols or class identifiers.

Corresponding to a class definition is a list of procedures to be followed in the event of an element of the class being encountered. A procedure usually states the order in which the procedures corresponding to the subclasses of the element are dealt with. At a basic level, the procedure is given in terms of *Z-routines*. These are subroutines which do all the actual writing of the compiled program and correspond, in a sense, to the basic teleprinter symbols in the class definitions. In practice, the class Z is the class of all possible instructions and is defined in terms of the other 25 subsidiary classes. When the description of the instructions and the method of translation is complete, the machine is ready to accept the autocode instructions. The *analysis routine* reads an instruction and tests if it can belong to class Z. A mistake which results in an illegal instruction can easily be detected, and when this occurs the offending instruction is copied to the output punch and the next instruction is read.

To fix the idea of a class definition, suppose that a class A has, as elements, the strings of symbols

bcd	(1)
b$\dot{g}$	(2)
c[*d*]*g*	(3)
bcde	(4)

The symbols "$\dot{}$" and "[ ]" are used to indicate that the item so marked can in the first case appear an arbitrary number of times and in the second need not appear at all. These concepts are frequently useful although the same results can be achieved in other ways.

The elements of the class with common stems are

arranged together, or made into a dictionary, to facilitate the work of the analysis routine. Thus class A is written

$$\begin{array}{lll} b\{ & c\,d\{ & e \quad (4) \\ & & - \quad (1) \\ & \dot{g} & (2) \\ c[d]g & & (3) \end{array}$$

A more particular item must be written before a general one, otherwise the particular one will be ignored (cf. items 1 and 4).

This array is made one-dimensional by using the symbols α, a branching symbol, and β, a terminal symbol, which are actually integers confined to certain ranges:

$$b\ \alpha\ c\ \alpha\ d\ e\ \alpha\ \beta_4\ \beta_1\ g\ \alpha\ \beta_2\ c\ d\ \alpha\ g\ \beta_3$$

An α usually indicates an alternative path to be followed if the preceding comparison is unsuccessful. However, as may be seen,

$\dot{g}$ becomes $\dot{g}\ \alpha$

$[d]$ becomes $d\ \alpha$

and the combination $[\dot{d}]$ becomes $\dot{d}\ \alpha\ \alpha$

The number attached to a β indicates which element of the class has been recognized. To distinguish between α's and β's when the definitions are punched on tape, 100 is added to each β. Thus the definition of class A would be punched on tape as:

$$b\ 12\ c\ 9\ d\ e\ 8\ 104\ 101\ g\ 9\ 102\ c\ d\ 14\ g\ 103$$

and would be stored inside the machine in this form.

These concepts can be illustrated by defining the classes D, N, and K—a decimal digit, an integer, and a fixed point number, respectively. The class D has members

0, 1, 2, 3, 4, 5, 6, 7, 8 and 9

and would be defined as

$$0\ \alpha\ \beta_0\ 1\ \alpha\ \beta_1\ 2\ \alpha\ \beta_2\ 3\ \alpha\ \beta_3\ 4\ \alpha\ \beta_4\ 5\ \alpha\ \beta_5\ 6$$
$$\alpha\ \beta_6\ 7\ \alpha\ \beta_7\ 8\ \alpha\ \beta_8\ 9\ \beta_9$$

The class N, an arbitrary number of decimal digits, could be defined as

$$\dot{D}\ \alpha\ \beta_1 \quad \text{or} \quad D\ 0\ 101$$

and the class K which can have the different elements

$$N[.], \quad .N \text{ and } N.N$$

could be defined as

$$N\ \alpha\ .\ \alpha\ N\ \alpha\ \beta_3\ \beta_1\ .\ N\ \beta_2$$

or $\quad N\ 8\ .\ 7\ N\ 7\ 103\ 101\ .\ N\ 102$

This process is arbitrary and depends upon how the definition is to be used.

Just as classes are defined in terms of other classes, they can also be defined recursively, i.e. in terms of themselves. Thus the class N can be usefully defined as

$$D[N], \text{ i.e. } DN\ \alpha\ \beta_1 \quad \text{or} \quad D\ N\ 3\ 101$$

Generally the analysis routine asks the question: "Can the following symbols belong to the class under consideration?" To answer it the class definition is selected and the first character examined. If it is a basic symbol then it is compared with the first of the symbols under test. If, however, it is a class identifier, then the definition of this class is selected and examined by the analysis routine using itself at a lower level. If a β symbol is successfully reached, then the symbols under test do belong to the class and the analysis routine returns to a higher level with a positive answer. When a negative answer is arrived at, any alternative path is explored and, if this also is unsuccessful, then the symbols under test do not belong to the class.

As the analysis routine tests the validity of an instruction it builds up a detailed record of what classes of symbols are present. This analysis record has to be consulted by the synthesis routine before the appropriate procedures can be selected.

The analysis record of an element of a class consists of the records of its component classes or, in the case of basic symbols, the symbols themselves. The records are linked by α symbols which give the address of the next α, and are terminated by a β which specifies the element of the class recorded. The α symbols here do not correspond at all to those in the class definitions.

If an element of a class may appear an arbitrary number of times, a record is made of each appearance, and if an optional element is absent, its record is simply omitted. Thus, if an element is defined as $Ab[C]d\dot{E}$ and the symbols corresponding to $AbdEE$ are present, the analysis record would be

$$\alpha \text{ record of } A\ \alpha\ b\ \alpha\ \alpha\ d\ \alpha \text{ record of } E_1, \text{ record of } E_2\ \beta$$

When the analysis record of an instruction is complete, control is transferred to the *synthesis routine* which skips through the α's of the complete record to determine which element of class Z is present. The corresponding procedure is selected and carried out, and then the next instruction read.

A procedure must be given for each element of each class and is a list of the operations to be done when that element occurs. Thus if an element consists of

$$ABC$$

the procedure definition could be

$$B2,\ A1,\ C3$$

the numbers indicating which item of the element is referred to. This procedure definition would merely call in the definition corresponding to the element of class B which is present, and then the definitions corresponding to the elements of classes A and C.

The Z routines, which are used for performing basic operations, are referred to by number and usually have parameters associated with them. For instance, $Z6, m, n$ forms a floating-point number from the analysis record of an N, item m, in the element under consideration. If $n = 0$ the number is treated as an integer and if $n = 1$ as a decimal. Thus the procedure for the second element of class K could be

$$Z6, 2, 1$$

When punching the definitions on tape, class identifiers, α's and β's, have to be distinguished from basic teleprinter symbols. This is achieved by introducing the concept of double letter and figure shift. If a set of characters is already on letter shift and another letter shift is punched, then the next character is said to be on double letter shift. Double figure shift is defined in a similar way. Class identifiers are punched on double letter shift and α's and β's on double figure shift. It is true that care has to be taken when punching these definitions since the shift characters do not appear on the print-out, but once inside the machine they can easily be checked. These devices are undesirable, but are necessary if all the available characters on the teleprinter keyboard are to be made available for the autocode instructions.

Conclusion

The primary virtue of this input routine is the fact that the calculations in Pegasus Autocode programs can be done about 25 times faster on Mercury than they can on Pegasus. Thus, provided the times for input and output do not constitute too large a part of the total running time, this is very much to the customers' advantage. It must be emphasized, however, that this advantage can easily be lost by having excessive output.

The translation scheme, although perhaps rather elaborate for this purpose, is very flexible, and any mistakes can be quickly rectified and new types of instructions added. In particular, algebraic expressions are easily defined. This can be illustrated by defining a possible general arithmetic instruction for Pegasus Autocode.

The following classes are required:

V: vN, vnN, $v([\pm]N + nN)$
S: $+$, $-$
U: nN

V is the class of all variables and U the class of indices.

F: mod, int, frac
G: sin, cos, log, etc.

F and G are the two different classes of functions.

Q: U, V, K, (E), FQ, GQ

Q is a class of items, K being the class of fixed-point numbers. The class E is defined below.

A: $\times Q$
B: $/Q$
T: $Q[A][B]$
W: ST

T is a term and W a signed term. An expression E is defined as

E: $[S]T[\dot{W}]$

The complete instruction is

$$V = E$$

Not only would this replace all the existing "variable" instructions, but it would also allow the user to write brackets within brackets.

The procedures corresponding to these classes would be simply concerned with building up the terms in the accumulator. The procedure for the class E would be

$$T2, S1, W3$$

In other words, it would form the first term, sign it, and then add on any subsequent signed terms. A slight complication arises when (E) is encountered. The contents of the accumulator have to be stored in a nest while the expression E is evaluated. These difficulties can be easily overcome and, if the instruction were to be incorporated, the user could write such things as

$$v1 = v2 + v3 \times v4 + \sin\cos v5$$

and

$$v7 = (v1 + v2) \times (v3 + v4)/(v5 + v6).$$

Acknowledgement

The author would like to record his appreciation of the assistance which Mr. R. A. Brooker and Dr. D. Morris have so freely given to him.

Appendix: Mercury Library Specification R 3130

Title: Pegasus Autocode Input Routine.

Purpose: To run Pegasus Autocode programs on Mercury.

Programming Notes: The following restrictions must be observed:

1. Only the variables $v0$–$v599$ may be used.
2. If N is the highest numbered index and L the highest numbered label, then
$$2N + L \leqslant 184.$$
3. There must be less than 2,000 instructions.
4. Instructions which refer to a second tape reader will be taken to refer to the single tape reader.

5. There must be no instructions of the form
   TAPE *N*, *N*.
6. There must be no ALTER instructions.
7. There must be no machine orders.

Operational Notes:

1. The program is a binary one and occupies sectors 129 onwards, and sector 2.
2. Three modes of starting are available:
   (*a*) H.S. = 0—to read in a new program.
   (*b*) Key 0 up—equivalent to the TAPE instruction.
   (*c*) Key 8 up—re-enters the current interlude.
3. If key 4 is set and followed by I.T.B. then the sector 2 of Pig 2 is restored, i.e. if Pig 2 was previously in the machine then it need not be read in again.
4. To restart after a STOP instruction press the prepulse button, and after a Z (when the machine hoots) press key 9.
5. Spurious instructions are copied and ignored.
6. Errors encountered during execution are printed out and the machine then stops.

Time: The speed of calculation on Mercury (excluding input and output) is about 25 times greater than on Pegasus.

References

CLARKE, B., and FELTON, G. E. (1959). "The Pegasus Autocode," *The Computer Journal*, Vol. 1, No. 4, p. 192.

BROOKER, R. A., and MORRIS, D. (1960). "An Assembly Program for a Phase Structure Language," *The Computer Journal*, Vol. 3, p. 168; see also p. 220 in this issue.

Book Review

Integrated Data Processing and Computers—Report of a Mission to the United States, E.P.A. Project 6/02B. Organisation for European Economic Co-operation, Paris, November 1960. 77 pages. Available from H.M.S.O., price 10s. 6d. (or from O.E.E.C. Sales Agents).

For some time, it had become evident that the progress in exploiting computers and other new equipment had reached a stage in the U.S.A. when a new study of experience would be of interest to Europe. This report has been prepared by the Mission, under the presidency of Mr. Brian A. Maynard (Cooper Bros.), which visited the United States from April to June 1960. Two members of the B.C.S. Council, Mr. John Goldsmith and Mr. Jack Grover, served on the Committee which prepared the report, as also did Mr. Jeremiah Donovan (Aer Lingus): another B.C.S. member was Mr. Michael Wright (N.R.D.C.): the project manager was Mr. Karl Seelmayer of O.E.E.C-E.P.A. The Mission totalled 25 representatives from 10 countries.

The present report will be supplemented early in 1961 by a folder of background material, reports of lectures and reports on individual installations. The report covers a period which is approximately four years after the earlier study by Gregory and Gearing (published in THE COMPUTER JOURNAL, Vol. 1, p. 179: it is unlikely that the second author, now preoccupied with his own business installation, will complete the 1958 survey). Mr. Grover's random reflections were published in *The Computer Bulletin*, Vol. 4, p. 77 (December 1960).

The Mission divided into groups, which reassembled together at intervals, and visited 44 industrial and commercial organizations, 8 government offices, 4 computer service centres and 14 research/educational establishments—in three dozen cities. The division of the party into groups permitted sufficient time to be spent at each establishment for a substantial study and the checking of the findings with the management.

The authors are to be complimented on their brevity, which may, however, have led to the omission of some expected detail. For example, one expected a reference to the introduction of magnetic tapes for greater speed in supplementary storage, when discussing the history of the influence of new equipment (p. 12). The first 29 pages summarize the principal features of A.D.P. with computers, problems of installation, the government role, educational trends, and new developments in equipment engineering and programming already made in Britain. The final chapter summarizes conclusions and recommendations: these recognize the economic differences between Europe and U.S.A., which result in a greater difficulty in justifying an installation, and look to the U.S.A. scene to provide a cross-check on our thinking and stimulus of new ideas: the appendices and folder (to come) will be valuable in this respect.

There are 11 appendices, including 4 illustrations of the effect of integrated data processing on office-routine procedures, government recommendations for co-ordinating departmental work in this field, and a manager's guide to COBOL.

This report should be widely read in office management and accounting circles. It contains nothing of direct interest to the mathematician.

H. W. GEARING.