The PACT I Coding System for the IBM Type 701*

CHARLES L. BAKER

Douglas Aircraft Company, Santa Monica, Calif.

Introduction

In the development of an automatic coding system, two major problems arise. The first is to develop a coding language which permits a programmer to specify the computation he wants the machine to perform. Once this has been done, there remains the task of coding a compiler for a particular high speed calculator which will translate the language into actual machine instructions. Although initially the language may not be biased towards a particular machine, the process of coding the compiler results in feedback which alters the original specifications for the language. The final system is a compromise between ease of coding and ease of compiling.

The language described here is the one translated by the PACT I Compiling Routine into instructions for the IBM Type 701. In several instances the effect of the feedback mentioned above will be apparent.

Objectives in Coding PACT I

Before a coding language was specified, the PACT committee spent several weeks studying coding systems then in use. The desirable features and inherent deficiencies of each were noted. At the conclusion of this study, the feature which seemed most desirable was the master-routine sub-routine concept. This means that small, logical pieces of the problem are coded in separate blocks known as regions. These regions are tied together by another region which is a control region. These, in turn may be controlled by still higher level regions, and so on, until the entire problem has been coded.

A second very desirable feature agreed upon was that of a language of short, single operation instructions, each on one line of a worksheet. It was felt that even with an ideal coding system coding would not be perfect the first time. Also, few problems survive many runs on the machine before a change in the problem requires a change in the program. The use of short instructions on a vertical format permits changes, insertions and deletions to be made easily.

The two features mentioned above became the starting point for the development of the PACT I coding system. Other features deemed desirable were:

1. Use of mnemonic symbols where possible.
2. Denoting of variables by name rather than through reference to a location.
3. Direct logic description.
4. Provisions for handling elements of one or two dimensional arrays.
5. Fixed point arithmetic such that scaling could be specified or changed without adding instructions to the code.

* Presented at the meeting of the Association, Sept. 14–16, 1955.

6. Reduction of bookkeeping tasks to a minimum
7. A resulting system easy to learn and relatively easy to debug.

Arithmetic Instructions of the PACT I Code

In PACT I, the coding consists of a sequence of single operand, single operation orders, called steps. The result of a step is available as a factor in succeeding steps of the sequence. A special operation, "TAKE", is used to start a sequence. The arithmetic operations available are "ADD" (written +) "SUBTRACT" (−), "MULTIPLY BY" (X), "DIVIDE BY" (/), "ADD THE ABSOLUTE VALUE OF" (+ABS), "SUBTRACT THE ABSOLUTE VALUE OF" (−ABS), "TAKE THE ABSOLUTE VALUE OF" (ABS), "CLEAR TO PLUS ZERO" (CL), and "TAKE THE RESIDUAL OF THE PREVIOUS MULTIPLICATION OR DIVISION" (i.e., the low order part of the product or the remainder from the division) (RES). The functions available are sine (SIN), cosine (COS), square root (SQRT), logarithm (LOG), exponential (EXP), and arctangent (ARCT).

Figure 1 shows the PACT I coding for the solution of a quadratic. The quadratic formula has been rearranged slightly to save steps. The region designation written at the left is "SQ", a mnemonic designation for "solve quadratic". The required steps are numbered sequentially, but not necessarily consecutively. This allows the addition or deletion of instructions as required.

On the first step, we "TAKE" (the operation is left blank for this instruction) the factor "B". On the next step, we "DIVIDE BY" the factor "2". Here we eliminate some of the bookkeeping by having the compiler store the number "2" somewhere and referring to it by its location. (Any three-digit integer may be

PACT I CODING

$$X = \frac{-(B/2) \pm \sqrt{(B/2)^2 - AC}}{A}$$

REG. 9	STEP 12	OP. 16	FACTOR 20	S_1 25	S_2 28	Q 31	± 33	NUMBER 34–44
SQ	0		B					
	1	/	2					
	2	X						
	3		A					
	4	X	− C					
	5	+	R 2					
	6	SQRT						
	7	−	R 1					
	7 5	/	A					
	8	EQ	X	1				
	9		N 6					
	10	−	R 1					
	11	/	A					
SQ	12	EQ	X	2				

FIG. 1

written on the factor field of our coding sheet; the "NUMBER" field to the right of the coding form provides space for decimal numbers in the range of .0000000001 to 99999999999. Position of the decimal point is indicated by writing the point in one of the spaces.) The next step is to square the result, B/2. We accomplish this with the operation "MULTIPLY BY", and leave the factor field blank. This indicates that both factors are the result of the previous operation. The first series of equations is now complete, and the compiler will take over the job of storing our answer for later use.

A new series is started on the next two steps, "TAKE" "A" and "MULTIPLY BY" "C". Here, however, a negative multiplication is desired. The "CLUE" column of the factor field allows us to do this by writing a minus sign. It is now desired to add the result of step 2. We use the clue field and write the letter "R", denoting "THE RESULTS OF STEP". In the factor field we specify the number of the step. The square root operation on the next step signals the compiler to write a linkage to a square root subroutine. At the same time the compiler checks that our compiled code contains a square root subroutine. In the next three steps we "SUBTRACT" B/2, which was computed on step 1, "DIVIDE BY" "A", and set the results "EQUAL TO" (EQ) "X". There are two values of X to be computed, however, and some means is needed to distinguish between these. Two more columns of the code sheet, "S_1" and "S_2", are subscript fields and are used to designate the various elements in a one or two dimensional array. Hence, on step, 8 the factor would be "X sub 1". The computation of the second root is done in the same manner. On step 9, "N", a new clue, is used. This says to take "THE NEGATIVE OF THE RESULTS OF STEP", and the step number is again written in the factor field. On the final step, the value of the second root is set "EQUAL" to "X sub 2".

An interesting feature of the code should be noted. If the operation and factor columns are read aloud, using the meanings as given above, we have a description of how our problem has been solved. For this example we would read, "Take B, divide by 2 and multiply the result by itself. Take A, multiply by $-$C, add the results of step 2, take the square root, subtract the results of step 1, divide by A, and set the result equal to X sub 1. Take the negative of the results of step 6, subtract the results of step 1, divide by A, and set the results equal to X sub 2." This permits us to check the correctness of our coding very rapidly, without regard for any actual knowledge of the operation of our machine.

The problem of scaling was not covered in the above example, and indeed need not concern the programmer during the first stage of coding. However, the resulting program is to be fixed point, and some means of specifying the scaling is required. If, in the above example, all numbers and results can be carried as fractions less than 1, no information about scaling need be supplied. Non-standard scaling procedures may be called for, however, by writing the desired scaling factor (the power of 2 by which the machine result must be multiplied to get the true result) in the column headed "Q". The compiler will then supply the necessary shifts in the machine code it produces. In doing this, it will check for any inconsistencies in the programmer's scaling.

Automatic Loop Writing

An important feature of a stored program computer lies in its ability to modify its own instructions. To take advantage of this, two logical instructions, "SET" and "TEST", are provided for instructing the compiler to write a loop.

An example of the coding for a loop is given in Figure 2. Here it is desired to add the two vectors, A and B, to produce the vector C. Since PACT I only allows for operations on individual elements of arrays, we code the addition of the respective elements of the two vectors A and B, and then write the instructions to repeat this for each of the elements of the answer. The initial instruction, "SET", is used to set the value of the subscript written in "S_1" to the numerical value written in "S_2". The first time through the loop we will add "A_1" to "B_1" and obtain "C_1". The instruction "TEST" performs two functions. It increases the value of the subscript written in "S_1" by 1. It then tests this new value against the numerical value written in "S_2". If the subscript value is now greater than "S_2", control is transferred to the next step. If this is not the case, control is returned to the first instruction following the "SET" instruction. (It may happen that we wish to return to a step other than that immediately following the "SET". If so, the desired return point may be written in the factor field of the "TEST" instruction.) The coding in Figure 2 may therefore be read "Take A sub I, add B sub I, and set the results equal to C sub I. Do this for I = 1, 2·····10."

The instruction "USE" is equivalent to the order "SET", except that a "TEST" instruction for the same subscript need not (in fact, must not) follow. It is normally used to pick a certain non-changing value for a subscript on the basis of pre-computed arithmetic results.

There is no restriction on the number of "SET" or "TEST" instructions which may be written. Figure 3 shows the multiplication of the matrix P by the scalar π and the addition of the matrix Q to obtain the matrix R. Note that the sizes of the matrices, M rows by N columns, are variable, but have been computed earlier in the coding. Although the return addresses of the "TEST" instructions have been written, they could have been left blank and supplied by the compiler.

PACT I CODING
VECTOR ADDITION

$$\begin{bmatrix} A \end{bmatrix}_{1\times 10} + \begin{bmatrix} B \end{bmatrix}_{1\times 10} = \begin{bmatrix} C \end{bmatrix}_{1\times 10}$$

REG. 9	STEP 12	OP. 16	FACTOR 20	S_1 25	S_2 28	Q 31
V A	0	S E T		1	1	
↑	1		A	1		
‖	2	+	B	1		
↓	3	E Q	C	1		
V A	4	T E S T		1	1 0	

FIG. 2

PACT I CODING
MATRIX ADDITION

$$\pi \left[P\right]_{M\times N} + \left[Q\right]_{M\times N} = \left[R\right]_{M\times N}$$

REG. 9	STEP 12	OP. 16	FACTOR 20	S_1 25	S_2 28	Q 31	± 33	NUMBER 34
MA	0	SET		1	1			
	1	SET		J	1			
	2		P	1	J			
	2	5 X						3.14159265 4
	3	+	Q	1	J			
	4	EQ	R	1	J			
	5	TEST	2	J	N			
MA	6	TEST	1	1	M			

FIG. 3

Logical and Bookkeeping Orders

A complete list of PACT I operations is shown in Figure 4. In addition to the arithmetic and loop writing instructions described above, orders to test for zero, the sign of numbers, etc., have been incorporated. They are "TRANSFER ON POSITIVE" (TP), "TRANSFER ON ZERO" (TZ), "TRANSFER ON NEGATIVE" (TN), "TRANSFER ON OVERFLOW" (TF), and "TRANSFER" (T) and operate as do the corresponding 701 instructions. There are also instructions to connect the various regions of a problem. An instruction, "DO REGION AND RETURN" (DO), sets up the machine orders to execute an entire region of PACT I code. A second instruction, "DO LIBRARY REGION AND RETURN" (LIB), directs the compiler to add a library routine to the machine orders it produces. By this means the programmer can use any machine language subroutines he may have. The PACT I compiler now in use contains 60 sub-routines totaling almost 12,000 machine language orders. Two other orders, "IDENTI-

PACT I OPERATIONS

ARITHMETIC	FUNCTIONS	LOGICAL	EXECUTIVE
OP.	OP.	OP.	OP.
	SIN	SET	DO
ABS	COS	TEST	LIB
+	SQRT	USE	CALL
+ABS	LOG	T	EXIT
−	EXP	TF	
−ABS	ARCT	TP	
X		TN	
/		TZ	
RES		HALT	
EQ		SE	
CL			

INPUT – OUTPUT	HOUSEKEEPING
OP.	OP.
READ	ID
LIST	FOR
	DUP

FIG. 4

FICATION" (ID), and "FOR", are used to convey information to library routines via "calling sequence". They are not executed, but their factor parts are used to write information to be extracted by the sub-routine.

The order "CALL" is supplied by the compiler as the first order of each sub-routine to signal a later stage of compilation to add the orders within the routine which will extract information from the subroutine calling sequence. The order "EXIT" may be written if it is desired to leave any region at a point other than after the last order of a region. If it is not written it will automatically be supplied as the last order of a region. To handle the "open" type of sub-routine, the instruction "DUPLICATE" (DUP), signals the compiler to insert the instructions for an entire region of coding at this point in the program.

Input and Output

Two instructions are available for input and output of decimal data. The first, "READ", will read a file of decimal numbers punched four per card, scale them appropriately, and store them in memory. Both the scaling information and the storage locations are punched in the cards, so that the order of the cards is unimportant.

The instruction "LIST", followed by the "IDENTIFICATION" of the factors to be printed, will produce decimal output of up to six numbers per line, each number with decimal point and sign. The order "SENSE" (SE) is used to control spacing and sheet ejection at the printer to produce the desired output format.

Information about Variables

In order for the compiler to produce the correct machine language coding for a problem, we must tell it several things about the variables of the problem. All

PACT I CODING
COMPLETE VECTOR ADDITION PROBLEM

REG.	STEP	OP.	FACTOR	S_1	S_2	Q
CON	0	READ				
	1	DO	VA			
	2	SET		1	1	
	3	LIST				
	4	ID	C	1		
	5	TEST		1	10	
CON	6	HALT	O			
VA	0	SET		1	1	
	1		A	1		
	2	+	B	1		
	3	EQ	C	1		
VA	4	TEST		1	10	

FIG. 5

factors must have their scaling specified. Also, the maximum dimensions of all arrays appearing in the problem must be supplied so that the compiler will allocate the proper amount of storage. If we wish, constraints will be applied to the variable storage allocation. These permit the same location to be used for several different variables, or force variables into certain locations for use with other previously coded instructions. All of this information is supplied to the compiler by means of the variable definition sheet which is loaded along with the arithmetic and logical instructions.

Solving Problems with PACT I

Figure 5 shows the entire PACT I coding for incorporating the vector addition region into a complete problem. Region "CON" (control) reads the data and prints the answers, while region "VA" does the computing.

It should also be mentioned that if the programmer wishes, he may do part of his coding in the PACT I system and part in actual machine language. It is relatively easy to combine the two types of coding into one program.

Conclusion

The members of the PACT committee are in agreement that the PACT I coding language is far from an optimum way of telling a machine how to do a problem. Something much better will be needed in the near future as the problems we face increase in magnitude and difficulty. However, it is a first step in the direction of relegating to the computing machine itself many of the laborious tasks formerly associated with coding problems for a high speed digital computer. PACT I has laid the groundwork for future developments in the field of automatic programming.