Logical Organization of the PACT I Compiler*

Owen R. Mock

North American Aviation, Los Angeles, Calif.

Introduction

The question with which this paper is concerned is the programming and coding of the PACT I Compiling Program. The original purpose in writing the PACT I Compiler was to gain and communicate experience in compiling. The purpose of this paper is to try to describe the organization of the PACT I Compiler in a form useful to future programmers.

The object of the PACT I Compiling Program is to translate an abstract language (PACT I) into a target language (IBM type 701 instruction language) using as a compiling tool the IBM 701. The fact that our compiling tool is the same machine as that for which we wish to create our target language is in a sense merely coincidental.

Certain characteristics of the IBM 701 are pertinent. With respect to the target language the 701 is a two arithmetic register, fixed point, 36 binary digit, single address machine with a large high speed storage capacity (compared to early machines) and no extra registers available for automatic address incrementing. The 701 can do full word or half word arithmetic with equal facility and each 701 instruction occupies a half word. The most readily accessible input to the 701 is from binary cards.

The large capacity of the 701 dictated that a generated program would be the most efficient. The lack of automatic floating point insured that our initial abstract language would be a fixed point language. The lack of index registers has undoubtedly influenced our subscript manipulation rules, perhaps here imposing a positive influence in making them more flexible than they might otherwise have been. On the other hand, based on our own experience and on an apparent trend towards exclusively full word arithmetic, we decided to forego use of half word data storage and to make all storage full word except for instructions and instructional constants.

The compiling program we wish to code in addition to utilizing the characteristics of the 701 pertinent to the target language may also make use of the 4 magnetic tapes and 4 magnetic drums on the standard 701. The 701 tape units have for the purposes of the compiler an effectively infinite capacity but have a transcription rate much slower than the access time to high speed store and are very awkward whenever random access is desired. The 701 drums, on the other hand, have both a limited capacity and are slower still than the tapes.

The tape units then are readily adaptable for sequential examination or storage of large volumes of information whereas the drums might better be utilized to preserve limited infrequently used information.

An obvious solution to the compiler and the solution utilized is to translate our

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

problem into a tape processing problem performing as many operations in high speed storage as possible, and to avoid if possible any random search of tape.

PACT I Abstract Language

We shall begin with a brief review of the PACT I abstract language. The fundamental unit of the PACT I abstract language is the step. Each PACT I step contains information in part or all of the following fields:

a. *Region*—PACT I steps are associated into logical blocks each of which is called a region.
b. *Step number*—The step numbers order the steps within a region.
c. *Operation*—The following types of PACT I operations are allowable.
   i. *Arithmetic*—PACT I arithmetic operations are fixed point and employ a single abstract arithmetic register.
   ii. *Executive*—The PACT I *DO* or *LIB* instructions cause PACT I to execute the Region or Library Program (respectively) specified by the factor and to return after completion.
   iii. *Transfer*—Permit conditional or unconditional transfer of control to another step within a region. To facilitate generation of PACT I arithmetic operations a transfer is not allowed to interrupt a sequence of arithmetic operations.
   iv. *Subscript Manipulation*—The subscript manipulation operations *USE*, *SET*, *TEST* permit automatic sequential operations on arrays of numbers. In PACT I subscript manipulation is restricted to those subscripts which are contained in the same region and are either bracketed by *SET* and *TEST* or preceded by *USE*.
   v. *Definitive*—The *ID* and *FOR* instructions permit redefinition of information being transmitted between regions. Definitive instructions must be in parallel following the originating *DO* instruction and the initiating *CALL* instruction of the Region specified by the factor of the *DO* instruction.
   vi. *Duplication*—The operation *DUP* is a purely clerical one, it causes the region specified to be actually physically inserted at the point of the originating instruction and to be considered henceforth as a portion of the corresponding region.
d. *Factor*—The following types of factors are included as possible objects of PACT I operations.
   i. *Variable*—A variable designates the name of a scalar or array. The variable designations are universal in a program. Each variable must have its binary characteristic specified at some point in the program.
   ii. *Results of a Step*—Any arithmetic step in a program may designate as its factor the results of another step within the same region provided only that execution of another region does not intervene.
   iii. *A Step*—Only control transfer steps may specify a step as a factor. This implies that the ability to modify a step has been specifically precluded.
   iv. *A Number*—Numbers may be specified directly as factors.
   v. *Regions*—Cross references between regions are only permissible with the executive instructions *DO* and *LIB*. Whence, one may only refer to another region *in toto*.
e. *Subscripts*—Each variable may have one or two subscripts. One use for these subscripts can be to expand the name of a variable. This serves only as a mnemonic extension of the name and this use will not be considered here. The other use is to specify manipulable elements of arrays.
f. *Q*—Specifies the binary characteristic of the results of an arithmetic step (this will be later referred to as Q_R).

In addition to the steps of the abstract language there are also available variable definition sentences which can be used to order the actual storage

locations used for the variables in the target machine. A minimal requirement is that all arrays have their maximum dimensions specified by variable definition sentences.

Translation of PACT I Language into 701 Language

701 storage assignments for target language—One of the most striking things we notice about the abstract language is that it is hierarchical and hence invites a hierarchical form of storage assignment. First, we note that regions may be referred to only *in toto* and hence can be completely addressed in our target language provided only that we know the origin. The variables can be assigned as a universal form of storage for a given program and the numbers can be compacted and assigned as universal storage also. We also note that according to the rules of our abstract language we are permitted to overlap between regions any "Result of Step" storage we desire into a pool we shall call temporary storage. Finally if we establish the rule that all library programs are independent then we know that we can overlap any temporary storage of these programs in a special pool we shall call perishable storage. A schematic diagram of our storage appears on following page.

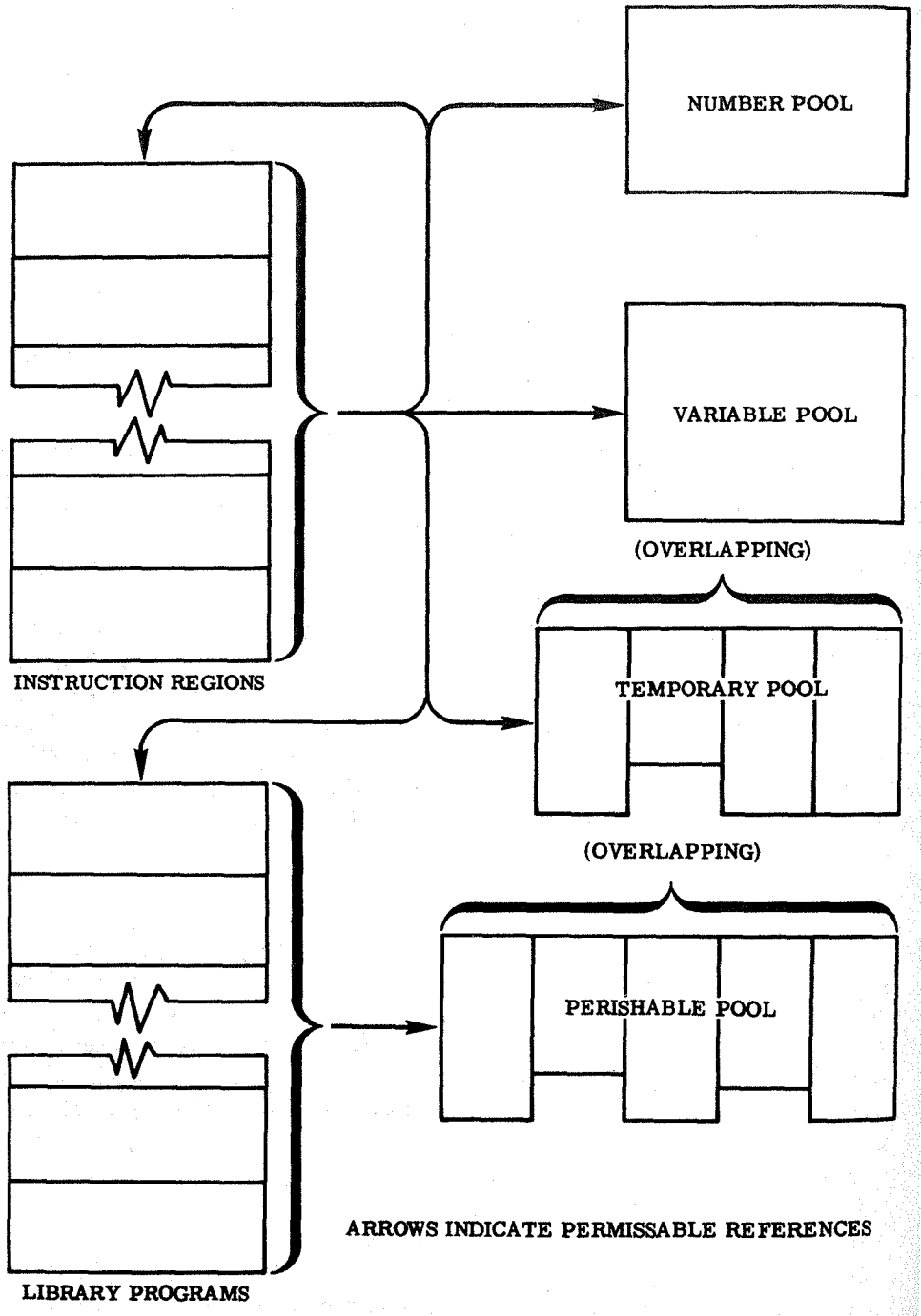
Subscripts, we note on the other hand, are defined only within a specific region and in our compiler we shall associate our temporary storage and constant storage necessary for array storage generation directly with the region involved. This is not the only way in which the subscript storage problem might be solved. We might also associate some of the constants with the arrays or in a special array definition storage. Our decision in this respect is somewhat arbitrary; however, the alternative method of generation may require more intercommunication between the variable definition and loop generation stages of the compiler, something which we wish to avoid in the interest of independence of stages.

Compilation

Having outlined a form of storage assignment in our target language we are in a position to outline the general compilation procedure. The magnetic tapes provide us with the most convenient means for processing our abstract program and the device we shall use is to make successive passes through tapes, at each pass performing only as many operations as can be performed in parallel and for which there is sufficient space in high speed storage. The passes may be divided into six stages:

(a) Conversion
(b) Initial Abstract Assembly
(c) Variable and Number Storage Assignment
(d) Arithmetic Operation Expansion
(e) Subscript Manipulation and Calling Sequence Expansion
(f) Final Assembly

a. *Conversion*—As the instruction cards are read they are converted and each


NUMBER POOL
VARIABLE POOL
(OVERLAPPING)
INSTRUCTION REGIONS
TEMPORARY POOL
(OVERLAPPING)
PERISHABLE POOL
ARROWS INDICATE PERMISSABLE REFERENCES
LIBRARY PROGRAMS

step is written as a separate record on tape. A table look-up is performed and a new compact code is used to replace the initial operation symbols. The new *compact* code occupies less space and facilitates later tests because of its logical arrangement. The conversion pass writes on tape for each step a 5 word record of the following form:

Region	Step
Operation	Factor
S_1	S_2
Q_R	Q_F
N - - - - - - - -	- - - - - - - - - -

where

Q_R = Binary characteristic of result as specified on code sheet or -0 if blank

Q_F = Binary characteristic of the factor. Defined as yet only for numbers.

At the end of each region are also written two tape records, one containing all the step numbers referred to by transfers and the other all of the step numbers referred to by factors calling for results of steps.

At the completion of the first pass a list of all *duplicated* regions is preserved on drums for use at the second stage.

b. *Initial Abstract Assembly*—The second stage of the compiler is essentially an ordinary sequential symbolic assembly stage and requires two tape passes. At the beginning of each region on the first pass of this stage the step numbers referred to by transfers are sorted and also the step numbers referred to by steps calling for results of steps. The sorting might equally well have been done at the end of each region on the first pass. However, it would appear that sorting at one stage or the other is preferable because of collating operations that follow.

Corresponding to each distinct step number within a region referred to by a "Result of Step" factor is assigned a relative "Temporary" address 2, 4, 6, ... The storage space which is thus preserved will be called temporary storage, during the first pass then the corresponding relative temporary address will be placed in a field called "Factor*". In addition during the first pass each step number will be matched against the sorted "Result of Step" factor table and when a corresponding step number is encountered an abstract PACT instruction is inserted with an EQ operation and "Result of Step Number" encountered specified as the factor. The Q_F of the step referred to is assigned as both Q_F and Q_R for the EQ operation. During the second pass the assignment of the Q_F's for results of step is completed.

Also, on the first pass the combined steps are numbered consecutively with a "Step*" and this step* is inserted into the transfer table. If there had been no provision for the *duplicated* operation then all that would be necessary to com-

plete the definition of transfers on the second pass would be to make a binary search on step and insert the corresponding step*. However, the presence of a duplicate instruction will have the effect of a set of insertions on step*. The method used to accomodate these insertions is to precompute between pass 1 and pass 2 this adjustment based on the count of each duplicated region and of sub-duplicated regions. The adjustment is computed iteratively and provides a check against a duplication loop such as:

Region A contains Duplicate Region B
Region B contains Duplicate Region C
Region C contains Duplicate Region A

Pass 2 then inserts the corresponding adjusted step* as the factor* for transfers. On pass 1 the tape is rewritten as two tapes, one containing *non-duplicated* regions and the other containing *duplicated* regions. Whenever during the second pass a *duplicated* instruction occurs the duplicate tape is searched for the proper region and this region is collated into the final tape of this stage. The temporary region is extended during this collation so as to avoid overlapping of this storage within a final region.

Beyond this stage a collated region only, together with any regions duplicated or sub-duplicated, is considered to be one region and any duplicated regions lose their identity. The new consecutive step*'s are assigned and these step*'s are retained as designation of the steps for the remainder of the compilation. Each step record written by pass 1 and pass 2 of this stage has the following form:

Region	Step
Operation	Factor
S_1	S_2
Q_R	Q_F
N---------	----------
Step*	Factor*

c. *Variable and Number Storage Assignment*—The function of this stage is to assign binary characteristics and relative addresses to all variables and numbers and to all references thereto. First, the variable definition sentences are loaded from cards and all variables listed on these cards are assigned relative addresses. Then the entire program is searched for additional variables defined during the course of the program by EQ operations. Arrays are considered to be completely defined only when the binary characteristics are specified unambiguously and also, their maximum dimensions are specified. Scalars need only have their binary

characteristics specified. The compiler stops whenever contradictory or incomplete definitions occur.

At the conclusion of this stage each new arithmetic step record will appear as follows:

Region	Step
Operation	Factor
S_1	S_2
Q_R	Q_F
N --------	----------
Step*	Factor*
S_1*	S_2*
Δ_1	Δ_2

where

Δ_1 and Δ_2 represent the amount by which 701 addresses are to be modified for corresponding changes in the value of a subscript.

Q_F will be filled in for all arithmetic steps.

Q_R will be completed for all EQ steps.

Factor* contains the true relative address for all arithmetic steps.

S_1* and S_2*contain information useful to subscript expansion.

The number pool is compacted as much as possible and is written off on drum to be punched after final assembly.

d. *Arithmetic Operation Expansion*—Having Q_F and a factor* defined for each arithmetic step we are now in a position to define each arithmetic step. The method of generation of the equivalent 701 instruction is to examine two steps (in part) simultaneously and from this to produce a machine language code to accomplish the transition from the state of the previous step to that of the current step. Here our choice of precluding transfers to arithmetic steps which depend on the previous condition of the abstract arithmetic register shows to advantage for we have avoided the necessity to examine for our previous step any step *except* the step actually prior on the instruction tape and have also avoided possible ambiguity.

The generated 701 instructions are added to each arithmetic step record and a new record with 701 equivalent instructions is written on another tape for the next stage. In addition, it also turns out that only one of the generated instructions will have the address of the factor* and this one will be designated by an "increment" which will also be written on the tape record in order that the next stage will know which 701 instruction to refer to in its manipulations.

An arithmetic step record after the arithmetic operation expansion will appear as follows:

Region	Step
Operation	Factor
S_1	S_2
Q_R	Q_F
N- - - - - - - -	- - - - - - - - - -
Step*	Factor*
S_1*	S_2*
Δ_1	Δ_2
Increment	n
Operation and Address	
- - - - - - - - - - - - -	

where

n is the number of machine language instructions generated by the step.

e. *Subscript Manipulation and Calling Sequence Expansion*—As might be expected, the subscript manipulation rules are the most critical with respect to the ease and directness of coding the compiling program. The restriction that subscripts are only valid within a region and in fact within bracketing steps restricts the area that must be scanned. The simple rule is adopted that if a variable has two subscripts then the subscript corresponding to the subscript-set closest to the step where the variable occurs will be the interior subscript; this rule facilitates the expansion of presetting and testing steps. If on the other hand for example, modification were to carry throughout a program then the entire program would have to be scanned and modifications applied to every arithmetic step having the corresponding subscript.

Subscript manipulation may also be performed in two passes. From each subscripted step record the proper address modification "deltas" may be obtained and also the actual 701 instruction, generated by a step, which is to be modified. Instructional constants are stored in two specially flagged steps at the end of each region and again a proper increment is attached to any cross references to ensure the subsequent designation of the proper 701 instruction. An n is attached to subscript manipulation steps in the same manner as to arithmetic generation steps. The calling sequence expansion proceeds in a manner very similar to the subscript manipulation expansion.

At the completion of this stage the PACT instruction tape is in final form ready for assembly.

f. *Final Assembly*—The final assembly is now remarkably simple. Except for the last two steps which are easily flagged, each step in a region has been numbered consecutively by step* and the extent of each step is also available so that a direct table may be computed in one pass and a simple substitution and addition of increment performed in a second pass through the program tape. On the second pass the operation is decoded for listing and a parallel listing is made between the original PACT instructions augmented by editing instructions and of the final 701 relative instructions.

PACT Relative Binary Card Format and Loading Program

There finally comes a place where the definition of absolute addresses can be postponed no longer. In PACT the complete definition is postponed as long as possible—until actual load time. The final 701 instructions are punched 36 to a card and associated with each 701 instruction is a three binary digit index which serves as a regional or even symbolic designation for the address. A table is retained in high speed storage which designates the origin of each region (a library program or a class of storage is here called a region). This table is loaded from one or more control cards which in general may contain the extent of the regions, and the actual absolute origins are computed by the loader. The three binary digit index corresponds to a location in the table:

Index	*Designation on Assembly*	*Region*
0	S	Absolute
1	I	Current Region
2	V	Variable
3	T	Temporary
4	N	Number
5	—	Perishable Storage
6	—	Not Defined
7	S	Region whose location in the table is specified by address of instruction.

An instruction with an index of 7 then has in effect a symbolic address.

To each card is attached a binary equivalent of the region symbol and the current origin is found by determining the value designated by the table entry with the corresponding symbol.

In addition to the instruction regions the number pool library programs and loading program are all punched out so that the resulting binary deck is completely ready to run.