กลไกของการรีโลเคทฟอร์แทรนคอมไพล์เล่อร์ ในระบบเครื่องคอมพิวเตอร์นีแอค ๒๒๐๐/๒๐๐ โดยใช้ฟอร์แทรนคอมไพล์เล่อร์-ดี

นายยรรยง เต็งอำนวย

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
แผนกวิชาวิศวกรรมคอมพิวเตอร์
บัณฑิตวิทยาลัย จุฬาลงกรณ์มหาวิทยาลัย
พ.ศ.๒๕๒๒

RELOCATION MECHANISM IN FORTRAN COMPILER BASED ON FORTRAN COMPILER-D OF NEAC 2200/200 SYSTEM

Mr. Yunyong Teng-amnuay

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science

Department of Computer Engineering

Graduate School

Chulalongkorn University

1979

Thesis Title	Relocation Mechanism in Fortran Compiler Based on Fortran Compiler-D of Neac 2200/200 System.
By	Mr. Yunyong Teng-amnuay
Department	Computer Engineering
Thesis Advisor	Assistant Professor Somchai Thayarnyong

Accepted by the Graduate School, Chulalongkorn University in partial fulfillment of the requirements for the Master's degree.

S. Bunnag
..... Dean of Graduate School

(Associate Professor Supradit Bunnag, Ph.D.)

Thesis Committee

..... Chairman

(Assistant Professor Sawat Seangbangpla, Ph.D.)

..... Member

(Assistant Professor Somchai Thayarnyong, M.Sc.)

..... Member

(Assistant Professor Suyut Satayaprakorb, M.S. in E.E.)

..... Member

(Vinai Varanyanond, M.S.E.E.)

Thesis Title	Relocation Mechanism in Fortran Compiler Based on Fortran Compiler-D of NEAC 2200/200 System
By	Mr. Yunyong Teng-amnuay
Thesis Advisor	Assistant Professor Somchai Thayarnyong
Department	Computer Engineering
Academic Year	1978

ABSTRACT

The format and algorithm of the relocation mechanism of object program based on the computer NEAC 2200/200 at the Computer Service Center, Chulalongkorn University, is studied. The important aspect is discussed in terms of the theoretical classification of several schemes of relocation. Some linkage problems are also studied and discussed. Guidelines are made to aid further studies in this field.

หัวข้อวิทยานิพนธ์	กลไกของการรีโลเคทฟอร์แทรนคอมไพล์เล่อร์ในระบบเครื่องคอมพิวเตอร์นีแอค ๒๒๐๐/๒๐๐ โดยใช้ฟอร์แทรนคอมไพล์เล่อร์-ดี
ชื่อนิสิต	นายยรรยง เต็งอำนวย
อาจารย์ที่ปรึกษา	ผู้ช่วยศาสตราจารย์ สมชาย ทยานยง
แผนกวิชา	วิศวกรรมคอมพิวเตอร์
ปีการศึกษา	๒๕๒๑

บทคัดย่อ

วิทยานิพนธ์นี้ เป็นการศึกษาแผนแบบ และขั้นตอน ในการรีโลเคท ออปเจคโปรแกรมของเครื่องคอมพิวเตอร์นีแอค ๒๒๐๐/๒๐๐ ซึ่งติดตั้งอยู่ที่สถาบันบริการคอมพิวเตอร์ จุฬาลงกรณมหาวิทยาลัย จุดที่สำคัญของผลลัพธ์ที่ได้ ถูกนำมาเปรียบเทียบกับ ลักษณะการรีโลเคท ที่จัดแบ่งประเภทไว้ตามทฤษฎี รวมถึงปัญหาบางประการ เกี่ยวกับการเชื่อมโยงโปรแกรมเข้าด้วยกัน นอกจากนั้นยังได้แนะนำแนวทางเพื่อการศึกษาค้นคว้าต่อไปในแขนงวิชาการด้านนี้

ACKNOWLEDGEMENTS

I would like to pay gratitude to my advisor Mr. Somchai Thayanyong for his insight and understanding both technically and personnally, Mr. Chaisiri Pandhitanond, my senior colleague, who, due to his in depth understanding of the hardware and software concepts of the machine and good theoretical background, highlights many difficult obstacles, and lastly Mr. Lersak Varadul whose thesis is closely related to mine and be of great help at desperate moments.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

CONCEPTS OF RELOCATION

Relocation is the mean by which programs can be moved or assigned to any area in main storage.[1] There are two main considerations governing the use of this mechanism. One primitive use is the mapping into available storages of object modules prepared in a generalized main memory image or in other words relocating of object modules. The other governing idea bases primarily on the utilization of the expensive main storage in multiprogramming environments. Shifting, segmenting and packing of programs in main storage are some of the many attempts in ensuring fully active and utilized main storage elements.

Usually, programs submitted for compilation consist of often used standard subroutines such as scientific packages. If these routines are kept in source statements or high level languages, they would have to be recompiled each time the need arises. This redundancy of resource usage in a computer installation can not be tolerated. Object code is a convenient mean to catalogue these subroutines in order to avoid recompilation. Still, the whens and wheres to put these routines are unforseeable. The only way is to compile them into relocatable modules, that is, the starting location is arbitrarily usually at zero and information concerning the modules would be incorporated to ensure correct relocation into a contiguous load module and the linking of parameters, constants and working areas of these modules together to

form one logically working program.

In contrast to static relocation described above which would be invoked once for every user's program compiled, multiprogramming environments need constant adjustments of the many programs competing for their shares on main storage. Sometimes swapping is forced to dislodge some unused areas of inactive programs and even more dramatic policies are continuously in use in memory management. This leads to the unavoidable certainty that programs must always be prepared for unexpectedly shifting around in main memory or even swapping out to auxillary storage. These involved dynamically relocating of programs kept in relocatable form and the understanding and creating of well-behaved relocating routines would somehow ensure efficient and practical control programs of computer system in one way or the other.

In general, modules or program units used in execution of each user's job, especially the standard scientific functions such as sine, cosine, absolute, etc, are separately compiled. The resulting object modules are always in relocatable format, that is, they anticipate shifting and combining with other modules to form a single program unit. Usually each relocatable module starts at location zero. When needed the modules are shift in memory space to form one contiguous program unit, still in relocatable format starting at address zero. Then this program unit would be mapped or relocated into the available memory area at the time prior to execution, after that it is ready for execution by the central processor.

There are two main approach in relocating the combined modules into real storage.[2] One is static relocation or relocation-at-load-time where the address references in each instruction are changed according to where the program unit is situated. Fig. 1.1 shows two relocatable modules after compilation which combined to form a single relocatable module. Next, this module is relocated into available storage, say, 2000, as in Fig. 1.2 where address references are adjusted accordingly.

As can be seen, there is inherent relocation in combining modules in Fig. 1.1 and usually this is bypassed via incorporating combine and relocate steps into one step which is the feature of the system under study.

The second approach, dynamic relocation or relocation-at-execution-time, called for base register as hardware support. In this scheme, each address reference is formed by adding address offset value in the instruction to the value in base register at execution time. That is, there is no adjusting of address references in instruction during load time, only the base register is set to the starting location that the module is to be loaded. The module still essentially starts at location zero as illustrated in Fig. 1.3. The combine step in Fig. 1.1 is still needed to create one contiguous relocatable module.

If the mechanism is as simple as described, dynamic relocation would seem a waste of hardware and ingenuity. But that is not the case. In their machine code representation, instructions cannot be separated from data. Creating relocatable object module, therefore, involved tagging information along with the module to specify which is instruction and which is data, because data is data and not to be relocated.

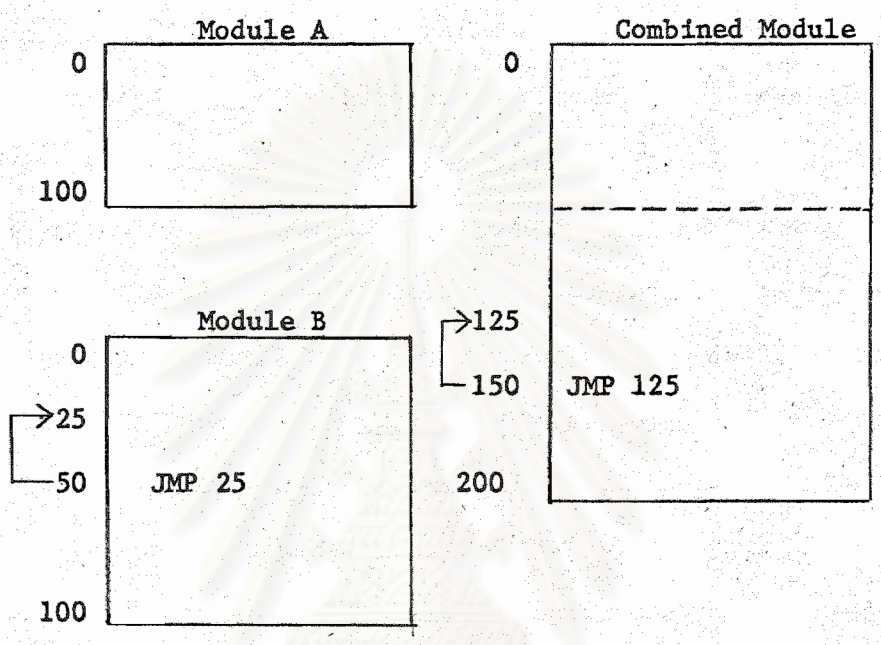


Fig. 1.1 Combining relocatable object modules.

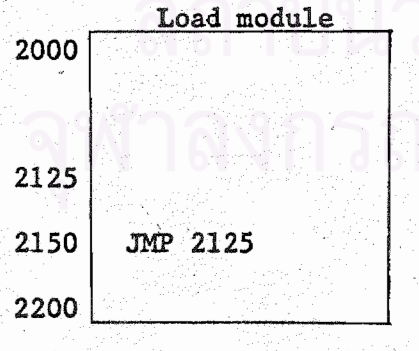


Fig. 1.2 Static relocation.

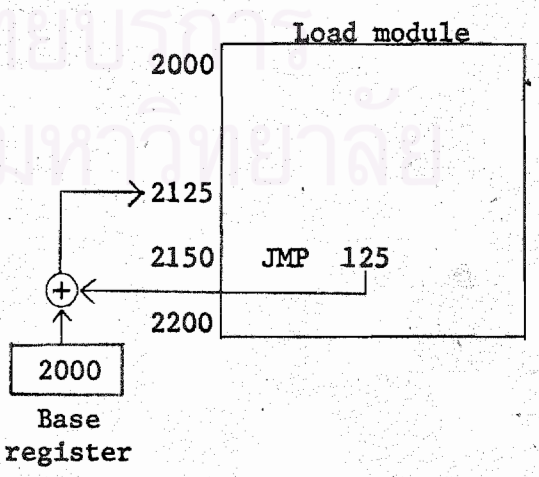


Fig. 1.3 Dynamic relocation.

Once mapped into real storage these information are lost for good and any attempt in relocating the module again is hazardous, only if it is relocated statically.[2] But in dynamic relocation the whole module is just placed where it belongs and base register set accordingly. So when relocation is needed again, moving and setting are all that is needed to achieve the requirement. This kind of mechanism is extremely useful in multiprogramming environment where modules are continuously shifted around to make the best of available main storage.

Relocation plays a major role in multiprogramming computer system. The basic idea may be simplified using Fig. 1.4. When program B and D finish execution and leave main storage, The areas left are together large enough to accommodate program E. The point is that they are not contiguous areas of main memory and if program E attempts to seize storage it would have to be splitted into two segments and separately relocated. Table would be required to keep track of where each address reference is expected to refer to. This type of splitting is called program fragmentation and as time goes on, this tends to get worse,[1] hence, the task of keeping track of program fragments will become intorerable.

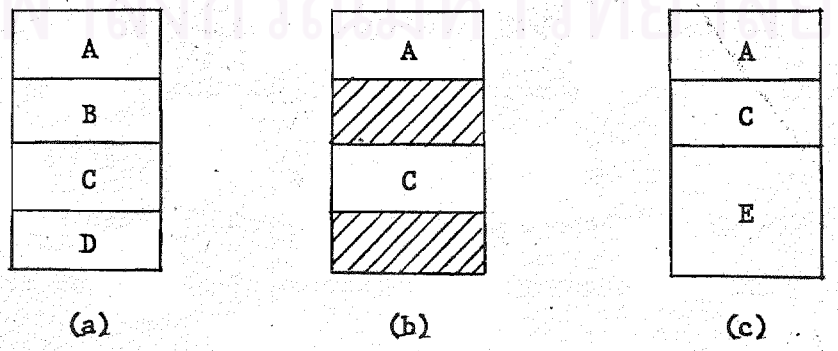


Fig. 1.4. Fragmentation and compaction.

Compacting operation is one of the solutions to this problem. In this case program C is moved until the free areas left are contiguous and can allow program E to take hold of the available spaces as in Fig. 1.4c.

But as can be seen, this compaction requires relocation of program module that has been relocated before and this rules out static relocation due to the one way translation of address references during load time. Therefore dynamic relocation which postpones relocation until execution time has to be employed and there are two important implementations worth noting here. They are segmentation and paging for the logical and physical aspects of dynamic program relocation.

The basic idea concerning these two approaches of implementing dynamic relocation is virtual addressing. Virtual address concept originates from the need to relocate program in memory more than once. In order to avoid adjusting address references in the program, mapping functions are employed to translate addresses in the original relocatable format called virtual address space into physical address of real storage or physical address space or, in mathematical notation,

$$f(a_i) = b_i$$

where f denotes mapping function and a_i and b_i are addresses in virtual and physical address space respectively. This concept is exemplified in fig. 1.5.

In this case f_D acts as relocating function and problem of repetitive relocation is solved, but only theoretically, because

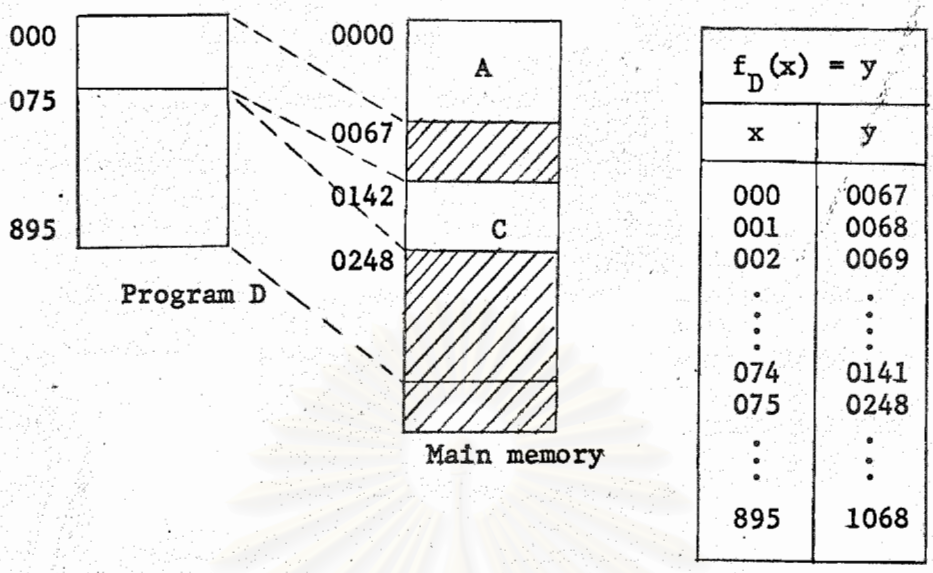


Fig. 1.5 Virtual storage concept.

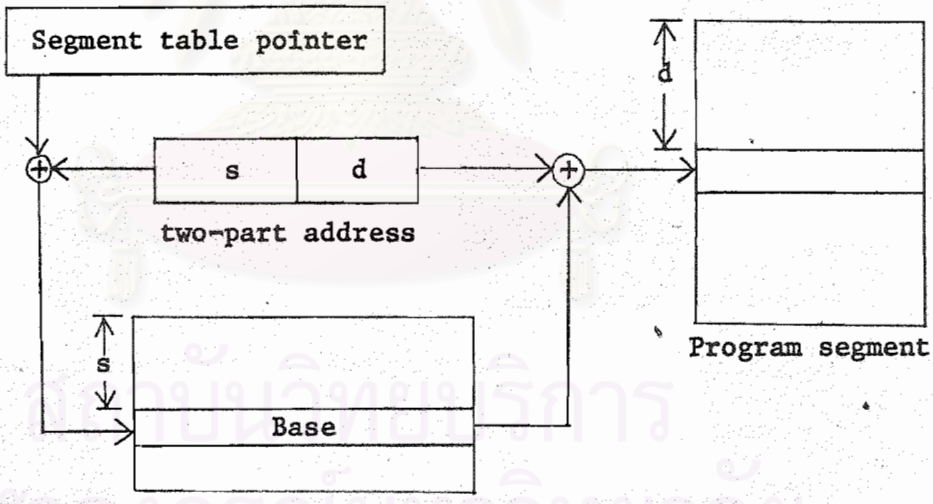


Fig. 1.6 Address reference in segmentation.

the map shown in Fig. 1.5 is lengthly, occupying much storage. Segmentation is one attempt in implementation using logical architecture of program modules. In this scheme program's address space is splitted into variable-sized blocks, called segments, which reflect logical divisions of the program.[3] A segment may represent a procedure or a data area. Any location within the program is identified by a two-part address which specifies the "name" of the segment and the "displacement" of that location in the segment. When relocated, a base register, pointed to by the "name" implemented by identifier in the two-part address, holds starting location of each segment. The "displacement" part of the address is then added to this base register when needed to form physical address. This structure is represented graphically in Fig. 1.6 for clarity.

As opposed to segmentaion, in paging implementation memory is divided physically into fix-length blocks of equal size called page frames. Programs and data are divided into logical units, called pages, of the same size as the memory page frames. Although there are differences in fundamental ideas, the implementations of both techniques are the same, except that in paging linking of program units and data areas is somewhat more complicate than segmentation for the programs are divided according to physical layout of main storage. But in paging memory allocation is simplified considerably. Since all pages are of equal size, bringing a new page into main storage is easier than bringing in a new segment. No pages need be rearranged or shifted to make room for the new page because the page may reside in an empty page frame or replace

another unused page.

To overcome the disadvantages of both approaches, the techniques are combined.[3] The program is divided into pages to fit the memory page frames and divided into segments due to logical division of program units. In this way, programs are easily linked together logically and relocated in main storage physically.

Still, there is overhead involved in bridging the virtual space and the physical auxillary storage used to hold the program and data modules. The solution in use now is to permanently map the physical devices into virtual space that means user would see his program and data as segments in virtual storage and the concept of main memory, secondary storage would suddenly fade away or storage is being smoothed into "one-level" which requires very large virtual space to accommodate the information physically stored in auxillary devices. This ultimate concept has been demonstrated in MULTICS implemented on a Honeywell 645 at MIT. But there are many problems that inherents in the design of such machines and most commercial firm such as IBM takes a cautious approach. The VSAM (Virtual Storage Access Method) in IBM 370 is a substitution to one-level storage idea.[2] It accommodates the mapping of conventional file system into a virtual space at the time of request. Once the file is mapped it can be directly referenced by programs in virtual space environment.

No matter how advance today's technology is striving, all myriad of concepts still stem from the primitive approach used in earlier machine. This study attempts to understand the underlining ideas

of the more elementary relocation mechanism and to explore the lee-ways and many drawbacks inherent in the design, the hardware and software aspects that are required for the evolution of the system under study.

In this study, the computer NEAC 2200/200 at the Computer Service Center, Chulalongkorn University is chosen. The study concentrates on the relocating routine incorporated as part of the FORTRAN compiler-D of the computer and is hoped to highlight some fundamental concepts which would be of valuable use for further study on this subject.

CHAPTER II

SYSTEM FORMATS

The architecture of the computer system under study is character or byte oriented.[4] But unlike other modern business machines such as the IBM system 370 and 360, only six of the eight bits in each character are for information storage with Hollerith code representation. The other two bits are for punctuations and this sacrifice in hardware accounts for the flexibility of variable length instruction and data. Furthermore, these punctuation bits also specify parameters in some controlling functions including the mechanism of relocation.[5] This inherent capability can be implemented to accommodate a more advanced relocating concept as will be discussed in later section.

Routines used in conjunction with the user program are known after the compilation phase. Then the relocating routine, which is part of the compiler, would initiate the loader to bring in routines one by one from the system relocatable object program library, check and relocate only the ones that are needed.

Due to the low main storage capacity of the computer, the relocation on the routines is done separately, that is, the routines are relocated and stored on secondary storage as physically separate modules. But their address references reflect the result of relocating process, that is, they are logically mapped into one contiguous

memory space.

Using information in manual[5] and tape dumps of object codes, the relocatable object module format can be constructed. The module is subdivided into two main components: the text or normal processing function and the relocating information. The format is generally of the form shown in Fig. 2.1.

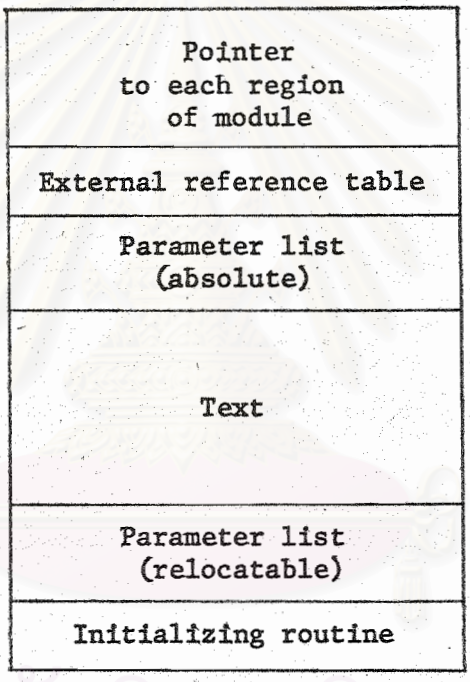


Fig. 2.1 Components of relocatable object program.

Initialization routine is used to modify parameter length such as integer variable as specified by the programmer, and also many other house keeping functions which vary according to the processing logic of the module.

One important component of the format not shown in Fig. 2.1 is the used of punctuation (item mark) bit. This bit is integrated as part of the address references and specify the type of addresses. Table 2-1 and 2-2 tabulate rules governing the interpretation of addresses based on punctuation and address values.

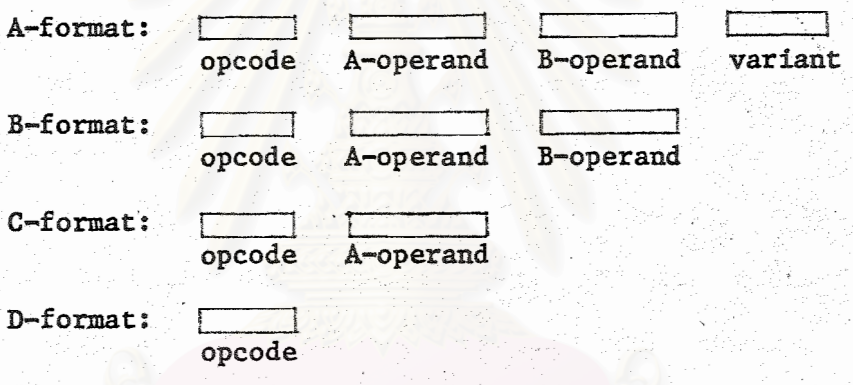
Item mark on first character of address reference*	Absolute address do not relocate
Item mark on middle character	Unlabeled common address
Item mark on first and middle character	Labeled common address
No item mark on first and middle character	Use rules in table 2-2

*Each address reference is three characters long.

Table 2-1 Shows the interpretation of addresses based on item mark and address values.

The characteristic that compounds relocation in this kind of machine is the variable form of words in handling data and instructions.[4] On one extreme side of the scale, an operand can have its field that occupies the whole main storage if punctuation is not detected on the field during execution of the instruction. This behavior is quite useful and is employed in clearing the whole main memory with a single instruction and even the instruction that initiates the sequence is also cleared. On the other hand, length of instruction varies not

only upon the type of operation code but also on the instruction format which is shown in Fig. 2.2. Due to this complication the relocating algorithm is somewhat lengthly in determining the type and length of each instruction and locating the character field that represents address reference of that instruction. The decision taken when encountering an instruction is tabulated in table B-2 of the appendix B.



Note: In case of missing operands or variant character, previous value in instruction register is assumed.

Fig. 2.2 shows different type of instruction format.

Most instruction can have implicit address references using previous values stored in address register and therefore instruction may have nothing but opcode or length of a single character and in the case of DSA (define symbolic address) instruction which is a pseudo instruction the opcode is not required and the relocating

routine must be capable of discriminating this type of character field from ordinary instruction.

Furthermore, there is the problem of linking which is inherent in combining object modules. Though this is not the topic of the study it can, by no means, be avoided and the subject is treated only in passing.

Each program unit to be linked is considered one complete logical module and is referenced as a whole by only one name (six characters as in naming ordinary program) or in a more general term, each logical program unit has only one "entry" point which is referred to by the unit name and the entry point is at the beginning or "turn-on-point" of that unit.

As the compiler is subdivided into physically separate modules then it can be traced during compiling process what module is in present control of compilation. The module which governs relocation of object program can thus be located when address references in object modules compiled are changed from relocatable to absolute ones.

From the above procedure the routine named ACARTG is found to be involved in the mechanism of relocation. Using the changing pattern of the program under compilation together with the behavior of the compiler itself, then, of course, long hours of studying and tracing the ACARTG routine and other related codes, the mechanism and flow of the relocating process can be deduced. The mechanism of relocation worked out here is presented in detailed in the appendix C.

In the next chapter, important aspects of the mechanism and

formats are discussed and attempts made to highlight their advantages and drawbacks together with considerations of further implementation in the system thus studied.

Range of Addresses	Relocation
$0 \leq ADDR < 4096_{10}$	do not relocate
$4096_{10} \leq ADDR <$ end of module	relocate accordingly
end of module $\leq ADDR < 32,767_{10}$	do not relocate
ADDR points to entry in external reference table	replace by entry point of module being referenced

Table 2-2 shows rule for relocation according to range of addresses.

CHAPTER III

THE RELOCATION MECHANISM

Due to the batch-oriented nature of the system,[5] there is only one set of physical modules connected logically into one program that occupies main memory at any one time and that program would be in main storage until its execution is concluded. Hence, the relocating scheme does not support dynamic or repetitive relocation which is never required in such an environment. What it does, therefore, is just relocating the assembled modules into the available main storage. At first glance, the relocation is quite unnecessary if the program starts at location zero and has the whole main storage to its own. But it is not the case for two main reasons.

Firstly, the main storage is not completely devoid when the program is ready to be loaded, even in strictly batch processing there is the problem of loading the program unit. Furthermore, some supervisor programs such as logical input and output modules or standard calculation-handling routines are absolutely necessary to the execution of such programs compiled from the FORTRAN compiler.[5] These system modules occupy, besides the loader, additional space which varies from program to program. So, unavoidingly, the memory is not available from the same location as the starting location of the program compiled but starts from elsewhere and the program compiled which starts at location 1024 decimal[5] has to be relocated into the available storage

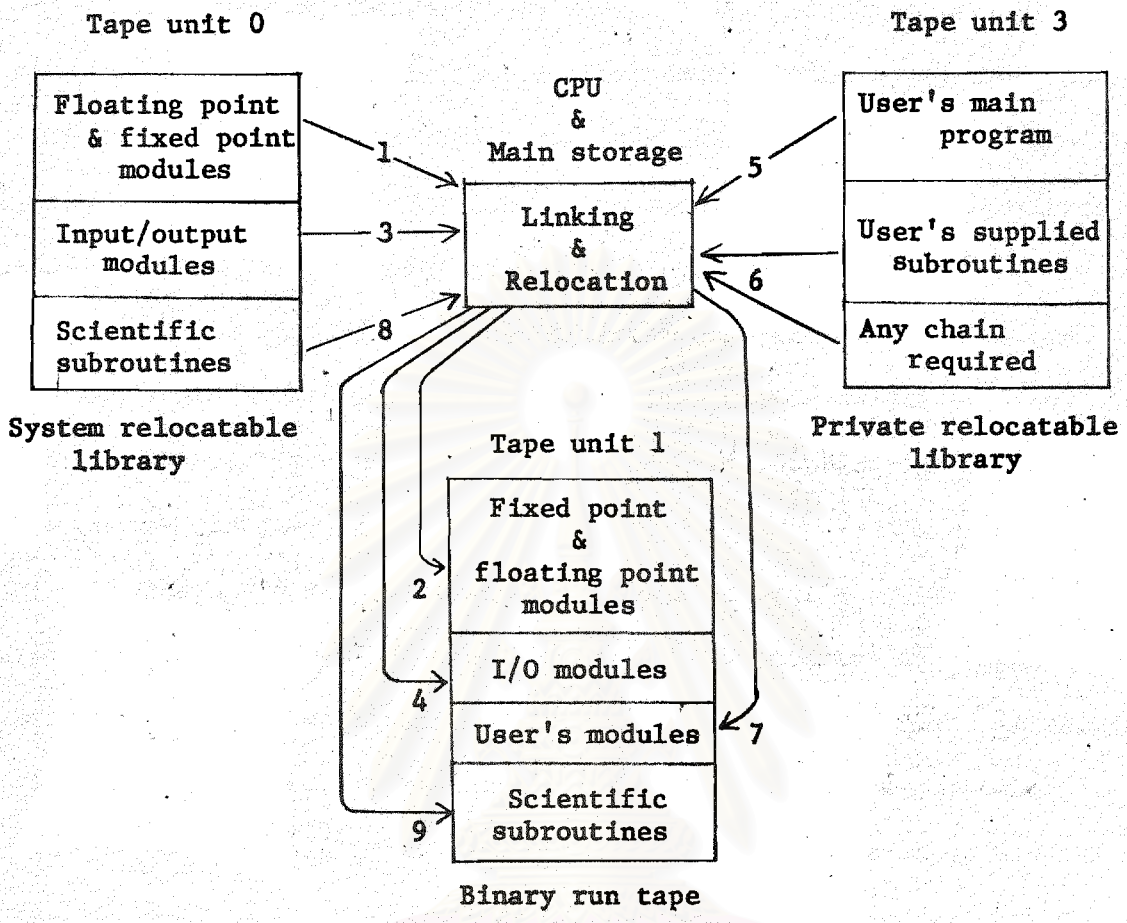
left from the occupation of the system control program.

The next reason for the use of relocation is the problem of linking several modules such as scientific subroutines to the user's object program unit. This time each module has to be relocated in order that its normal starting location is modified to reflect one contiguous logical module in the form of main memory image.

But as the main storage space available is always fixed for each individual program execution because no other user's programs are to exert their shares on the main memory, there is no need to perform the relocation process twice, firstly to combined several logical modules into one contiguous program unit and secondly, to put the linked modules into the available space. In the compiler under study, the separate modules are relocated into the image of available area one by one in the manner which will reflect one contiguous module, thus save relocation time at the cost of dynamic relocation.

This drawback is not to be blamed, though, the system is basically design for simple manipulation of software and the technique suffices for all purposes and intentions.

As for the modules involved and their relationships, fig. 3.1 shows the movements of program units and subroutines under relocation. The system relocatable library which is actually, in the system, part of the compiler is located on logical tape unit 0. The library is composed of three major group of program units. The first is the fixed and floating point arithmatic handling routines, next is the input and output program units which perform the necessary linkage between user's program and the more complicated input and output operations. Following



Note:The binary run tape (BRT) is constructed on tape unit 1 from system and private library. Fixed and floating point modules are selected (1), relocated and put on BRT (2) followed by the selection and relocation of input and output modules (3&4). Next, user's programs, subroutines or any chains presented on tape unit 3 is loaded (5&6), relocated and saved on BRT (7). The last to be relocated are the scientific subroutines (8&9) and together with some control modules, the BRT is then ready to be loaded.

Fig. 3.1 shows movements of modules during relocation.

these two groups of routines is a lengthly list of scientific program units such as sine, cosine, exponential function and so on. Interspersed with these units are the control program units which supervise the logical flow of user's program execution such as job-to-job transition, chain linkage, error checking and handling and the likes.

On the logical tape unit 3, after compilation phase is completed, there exists a group of relocatable user's object program units which consists fo main program and user supplied subroutines or any chains necessary there of.

These two sets of system and compiled user's program units are selectively merged together and relocated into the available main storage image called a load module or binary run tape (BRT). Actually these program units are separately relocated and put on tape unit 1. But though physically they are separate modules on the tape, nevertheless their starting locations and address references reflect a single contiguous load midule and once loaded into main storage they would be physically as well as logically merged into one continuous instruction stream.

Logically the program units involved would be modified according to the normal static relocation scheme as in fig. 3.2.

As for the mechanism itself, the compiler module named ACARTG which performs relocation is logically divided into several parts. The details of each part is presented in appendix C together with the logical flow diagram of the mechanism.

The first part, from location 31174 to 32313 octal, handles housekeeping and initialization, that is calculation of base or starting

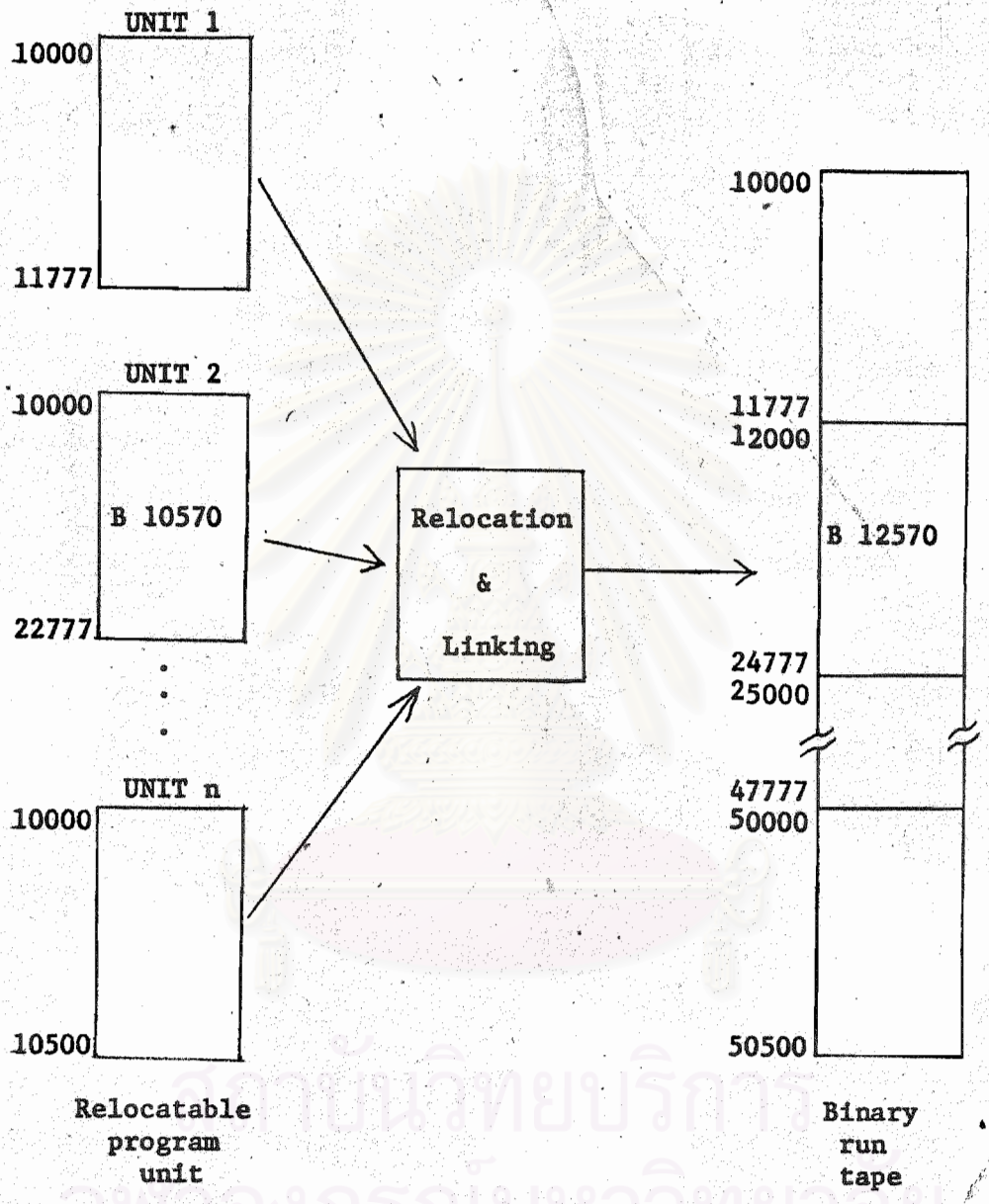


Fig. 3.2 shows logical mapping of relocatable modules into the BRT.

location of the program together with data area under relocation, setting of header label for first record (Q-record) of the relocated object codes or, in subsequent record, the label for J-record, and initialization of pointers to be used in the relocation process such as pointers to the current location in the tape buffer and in the object codes. In this part there exists first attempt at resolving external references in region 2 of the program unit to be relocated and inclusion of this unit's name into the resolved external reference table (address 31663 through 32054).

Another point worth noting is the use of general input and output routine (32133 through 32160) which composed of two routines at address 3442 through 4252 and 7440 through 7777. These two routines, via parameters, handle input and output operation of every devices employed in the compilation process including the physical controls such as rewinding or forwarding tapes, checking error or availability of channels and devices.

The next part (32317 through 33076) locates field in the instruction stream using word and item marks (instructions at addresses 33042 and 33052), relocates and transfers fields to the tape buffer (33026 through 33076) and handles field defined via DS (Define Storage) instruction (32360 through 32631).

Codes at address 33115 through 33527 involve classification of instructions into groups by using the length attribute, then locate the address part of the instruction and determine the relocation value to be augmentes on each address according to the type of instruction (subroutine at 36020 through 36266). Only the branch and DSA (Define

Symbolic Address) instructions are passed to the next part for further processing.

Due to their complicated relationships to the overall structure of the BRT, the address parts of the branch and DSA instructions are treated separately by the codes from 33533 through 34571 which comprises the lengthly process fo linking logically the current module to the other parts of the BRT because the branch or DSA instructions can refer to the chaining of user's program (handled by instructions at addresses 33533 through 33706), normal branches inside the current unit (instructions at 33761 and 33770), references to external modules (34021 through 34155 and 34546 through 34556), DSA instructions with variant character (34562 and 34571) or references to the standard fixed and floating point arithmatic handling routines (34161 through 34542).

After the relocation of the current module is completed, the unresolved external reference table is scanned for any reference to the current module (34772 through 35033). If there is any, then information is placed at the end of the module to aid the second-pass resolving process at load time of the BRT, this process will be discuss shortly in this chapter.

Although the compiler has its own input and output routines (at 3442 and 7440), they are only used for outputting to tapes and input or output with other peripherals. As for the loading of program unit into main storage which involved mapping the BRT string into main memory format the compiler turns to the aid of the loader. This interaction with the loader is reflect in this relocating module through the codes from 36272 through 36414 which is an exit point to

the loader when the next program unit to be relocated is required.

The rest of the module is devoted to determining which of the program units is to be loaded and relocated next and at the end of the module are the data areas and parameters used in the relocation process.

One important point worth noting is the mechanism employed in resolving external references, the process of which is illustrated in fig. 3.3.

The starting address of each program unit is calculated before the unit is loaded. After loading the turn-on-point is calculated and entered into the resolved external reference table together with the unit's name (1), then external references in the form of program unit names in region 2 of the current unit are replaced by their corresponding relocated turn-on-point if the program names being referenced appear in the resolved table (2). Any external reference is affected through branch instruction to the external program name in region 2 (3), and if the corresponding name is resolved then the branch instruction is modified, during relocation process of the unit, to branch directly to the turn-on-point of that external name (4). When there is reference to the unresolved name, then the name is placed in the unresolved external reference table (5) along with the relocated address of the branch instruction (6,7 & 8). When the unresolved program unit is encountered later in the process (9) and its turn-on-point relocated and put in the resolved table (10) then, after this unit is relocated, the unresolved table is scanned for any reference to this unit. If there is, codes are entered at the end of this unit (11) which would direct the loader to substitute the unresolved branch instruction at

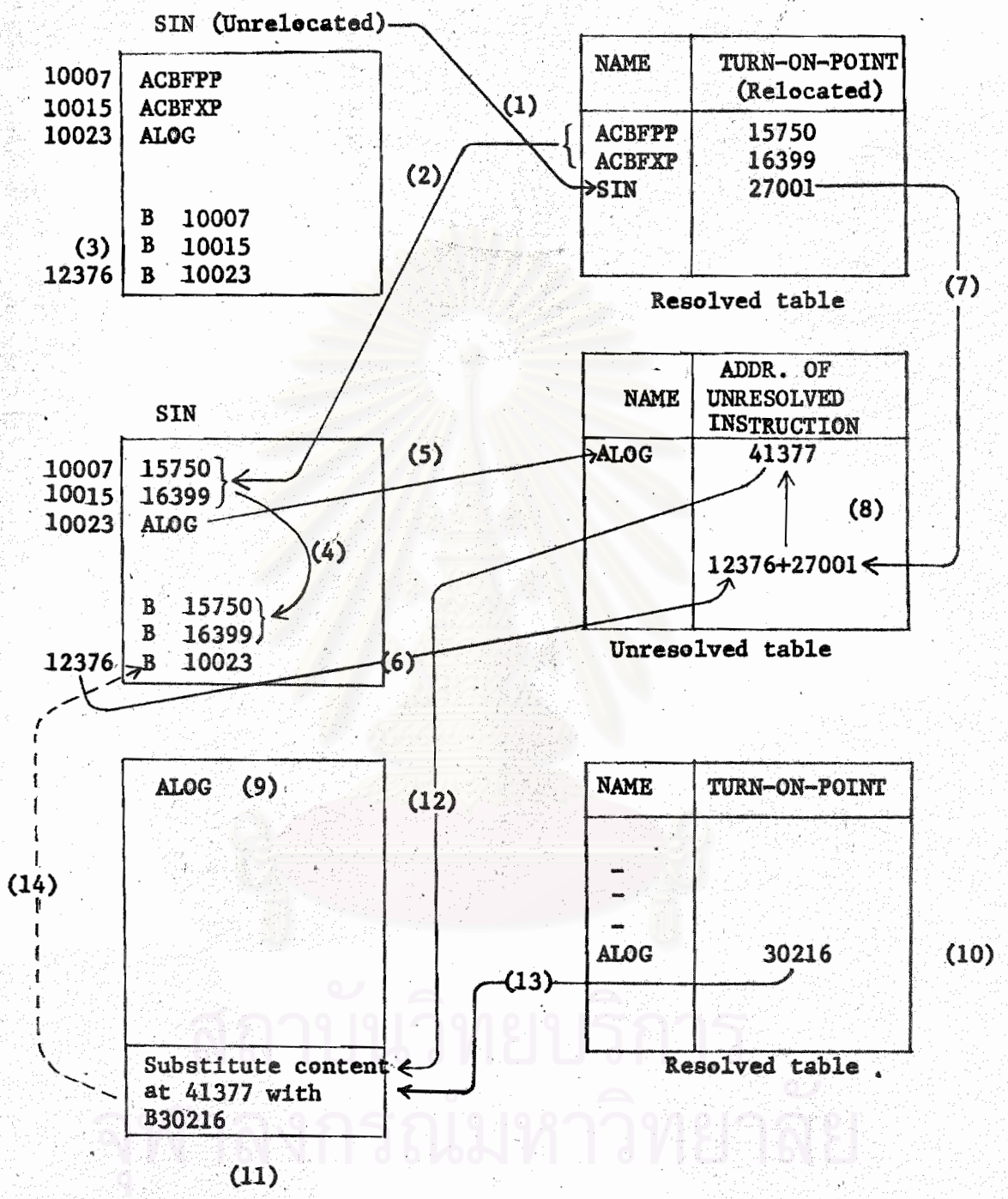


Fig. 3.3 illustrates the steps in resolving external references.

address specified in the unresolved table (12) with the turn-on-point of this unit (13) and when the whole BRT is loaded further by the loader, any unresolved branches would be resolved by this substitution scheme (14).

In the next chapter, discussion on this and previous chapters are forwarded, involving advantages and drawbacks of the system and prospect on further implementation of dynamic relocation.

CHAPTER IV

DISCUSSION OF THE MECHANISM

Although the relocation mechanism of this compiler can handle usual situations in the software environment, there are some drawbacks due to the primitive design.

As can be seen from the system format described in the previous chapter and in the appendix, the relocatable object module is divided into regions and region 2 is deleted when the external references are completely resolved after the load module is created. This means that region 2 is not part of the load module and it amounts to the separation of relocatable module into two groups, one group contains only region 1 and the other region 3, 4 and 5 which then have to be relocated with different starting location. This results in complications both in the format of the module and the mechanism related. Nevertheless, the whole concept of programmer bearing the weight of developing the relocatable modules according to the system format all by himself if he needs such modules in working with the FORTRAN program is somewhat cumbersome. Though the present state of software technology still does not support such luxury as obtaining relocatable modules from ordinary user written assembly program, it is hoped that a more sophisticate system software would aid the programmer greatly especially in the development of the system relocatable library and many service modules.

In modern system, most program units written are composed of more than one logical subunit which amounts to simplification in the software development. This results in the multiple calling points when utilizing these modules. This multiple entry points of the program unit complicates the system format somwhat, but the flexibility both in program writing and logical structure of the program is greater than the single entry or turn-on-point in the NEAC system.

As the system relocatable mechanism is strictly static, resolving external references is somewhat simplified as this normally requires two passes of scanning the references like resolving symbol table in the normal assembler. In this system the first pass is done in the relocation phase and any unresolved references are entered in table to be resolved after the modules being references are encountered. Then the second pass is implicitly performed during loading the binary run tape which logically comprises a pass through all the modules involved. This saves time but amounts in the inflexibility of further expansion into dynamic relocation scheme.

The bright side of the system under study lies in hardware construction. The implementation of dynamic relocation technigue requires inherent information in the program all the time it is kept active in the system. But using additional software pointers for the addressing structure is the overhead that cannot be tolerated both in space and supervisor execution time. In modern machines there are special hardware and system software that support the need. NEAC-2200 system, on the other hand, uses variable length concept of data and instructions and so requires hardware punctuations, which in

this case the word-mark and item-mark bits, in each character.[4] These bits are normally used in a number of ways to handle length and type of the data. They are, in some instance, used as flags and condition bits. So this kind of hardware overhead is a necessity for this type of machine. But as the bits are under-utilized, they can be used to indicate information which aids dynamic relocation. In the FORTRAN system, addressing structure uses three consecutive characters to refer to any location in main storage which means item marks and word marks of three bits each. The word marks inform of word boundaries and so cannot be of any use, on the other hand the item marks normally indicate types of addressing but there are combinations of the three bits left to accommodate more information. This type of bit utilization is employed in the Burrough machine, B5500 and 6700 for example,[6] which uses two special bits associate with each word of which the four combinations, 00, 01, 10 and 11 specify whether the words are integer, floating point number, character or address pointer respectively.

Eventhough the hardware architecture of this machine can be of use to the dynamic relocation implementation, the overall architecture is still far from adequate. Only in the environment of multiprogramming can the dynamic relocation be of any use, and this kind of environment requires and enormous amount of support both in software and hardware. Furthermore, the restructuring of information concerning the address references means rewriting all the relocatable programs in the system library.

Nevertheless, this kind of approach is quite useful if theoretically created multiprogramming environment is employed instead

of a real and practical one. Though the system would be of little or no direct use to the installation, the model developed can be an important tool to the underdeveloped computer field in Thailand. Due to the simplicity of the machine hardware and software, the work involved in the developmentof a pseudo multiprogramming environment would not be overwhelmming and the observed behavior of the model would be less clustered by unimportant details commonly found in more modern machines.

This simplicity also lends itself in every direction especially in the studies of so complex a machine as a computer, so much complexities of the system format tends to obscur important points.

Although much more work remains to be done in this field of study, this project so far is inadequate due to many limitations of the machines and documents. But it would nevertheless highlights some fundamental concepts and rouses any further practical or theoretical studies in this or related fields, especially in Thailand.

CHAPTER V

CONCLUSION AND SUGGESTION

As has been stated the study is based on the FORTRAN compiler of the NEAC 2200/200 computer system and the details of relocation mechanism are somewhat machine dependent, so the study of the mechanism had to be done on this machine. Due to the lack of documentation and source listing, object program has to be traced and ideas constructed by testing the effect of individual processing flow on the machine. This tends to be akward and it turns out that the machine time spent doing this kind of work is very excessive. The total time required in tracing the coding off-line is about four months while that of tracing on-line is about 50 hours in 2 to 3 hours session.

Apart from this there is the problem of acquiring basic knowledge of the system, for example hardware architecture, data structure, instruction repertoire. The general concepts of relocation and linking takes three months of study and conceptualization. Utilities and many operating tricks such as memory and tape dump while a minor concern needs practice.

The work done while far from adequate is what can be achieved under the circumstance in view of limited time, experience and resource. But approach to further study on this subject is worth the effort and would one way or the other enable a finer understanding of the complicate concept of relocation. The implementation of dy-

namic relocation as a work study on the machine is presented as followed to ensure a continuity of the study somewhat.

Implementation of Dynamic Relocation

Although the system under study does not support dynamic relocation scheme, the available hardware features can be combined with the modified software to give the effect of dynamic relocation. Eventhough the system implemented is of little or no practical use, it would be imvaluable in the light of academic study and, with special attention and effort, can even be useful in real application in a limited way.

The first thing in concern is the base or relocation register which keeps track of the starting location of the program unit and when the operand addresses are derived in the form of displacement relative to this base address the dynamic movement of the program can be achieved.

In the IBM 370 series, for example, the base register is user selected and can be any of the 16 general purpose register and when the starting location of the program is selected any other operand addresses are translated into the base-displacement reference pair. Similarly, the NEAC machine also has six index registers specified by three bits which is part of the operand address and any one can be selected for this purpose. But eventhough the compiler can be modified to accomplish this feature, it would mean that the other five index registers together with

indirect addressing mode cannot be used by the programmer. In contrast with the IBM machine, the latter has extra field in the operand address to denote indexing. But the compiler may be further tailored to retain the indexing feature which is very useful in program development. Fig. 5.1 depicts the use of implicit indexing. Ordinary operand is translated as operand using index register number 1. If other index register is required the other index register must be added to the base register (index register number 1) before reference to the operand is performed. Of course, means must be included to retain the previous information in the base register. Indirect addressing may be performed via this method but would give much more complication and the result may not justify the effort.

In the assembler level the modification is quite simpler but would increase programmer's burden, on the other hand in FORTRAN programming, the language structure is not changed but the compiler must be tailored to reflect this kind of implementation and although the compiler may still be in the static relocation mode, the standard scientific routines, which is combined with the user's program, must be changed to suite the dynamic relocation mechanism implemented.

As for the use of this kind of implementation besides academic studies, if the system is carried on to some extent with faster auxilary storage devices such as disks, multiprogramming in such a system is not infeasible and though it may seem less useful than

buying newer and better machines, the knowledge gain is somewhat more than can be found in any text or manufacturer's manual.

Source statement	Object code in symbolic form
ADD DATA,CONST, RESULT	ADD DATA+X1,CONST+X1,RESULT+X1
ADD DATA+X3,CONST,RESULT	ADD X3,X1,X2
	ADD DATA+X2,CONST+X1,RESULT+X1

Note: ADD A,B,C means C=A+B.

Fig. 5.1. Translation to effect the use of base register.

REFERENCES

1. Gear, C. William. Computer Organization and Programming. 2nd ed. New York: McGraw-Hill Book Co., 1974.

2. Hsiao, David K. Systems Programming: Concepts of Operating and Data Base Systems. Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1975.

3. Tsichritzis, Dionysios C.; Bernstein, Phillip A. Operating Systems. New York: Academic Press, Inc., 1974.

4. NEAC-Series 2200 Operating System Mod 1: Easycoder Assembly Language. Tokyo: Nippon Electric Co.

5. NEAC-Series 2200 Operating System Mod 1: FORTRAN Compiler D. Tokyo: Nippon Electric Co.

6. Madnick, Stuart E.; Donovan, John J. Operating Systems. New York: McGraw-Hill Book Co., 1974.

7. Gries, David. Compiler Construction for Digital Computers. New York: John Wiley & Sons, Inc., 1971.

APPENDIX A

FORMAT OF THE BINARY RUN TAPE

General

In NEAC 2200/200 the portion of object code which is load by the loader is referred to as a program unit. The tape containing these units is called Binary Run Tape or BRT. As recorded on a BRT, a unit appears as one or more consecutive records. Each record starts with a banner character which specifies the type of the BRT record and is classified as unblocks, variable-length data record and may vary in length up to a maximum of 250 characters. The first record of each program unit is called a segment header record and contains identification and control information pertaining to the unit it represents. Subsequent records or non-header records within the unit contain a minimum of control information immediately followed by object codes to be loaded.

Figure A-1 shows the BRT format and table A-1 and A-2 contain a brief description of the various types of field presented in a BRT records.

Record Formats of Program Units

1. Types of information.

   A program unit record contains two categories of information:

   1.1 Identification and control; and

   1.2 Data.

2. Identification and control field.

For a segment header record, the identification and control field occupies the first 24 characters and 7 character locations for a non-header record. The first character of each record is a banner character which identifies the BRT record type as explained in table C-1. Character at location two through four contain an octal number which indicates the number of characters in the record. Character location five and six contain the record sequence number which is used by loader in performing efficient backward searching over a BRT.

The seventh character designates the number of characters in the identification and control fields, which is 24 for segment header records and 7 for non-header records.

The format for the first seven character locations is the same for both types of record. However, the segment header record contains additional identification and control specified by the programmer in his symbolic program. The identification and control fields are tabulated by record type and character locations in table A-1.

3. Data fields.

Immediately following the control information in both types of record is the data portion which is variable in length. The data portion consists of strings of data characters to be loaded interspersed with control characters. The control characters are needed for two purposes:

a) To specify the loading location;

b) As the tape does not have extra bits for punctuations as in main storage, the control characters provide information about punctuation.

The data of each record has the following characteristics:

a) It begins with a control character;

b) Every record except the last record of a unit terminates with the control character octal 77;

c) The last record of a unit terminates with control octal 61, followed by a three-character address.

The formats and description of the nine control characters are provided in table A-2.

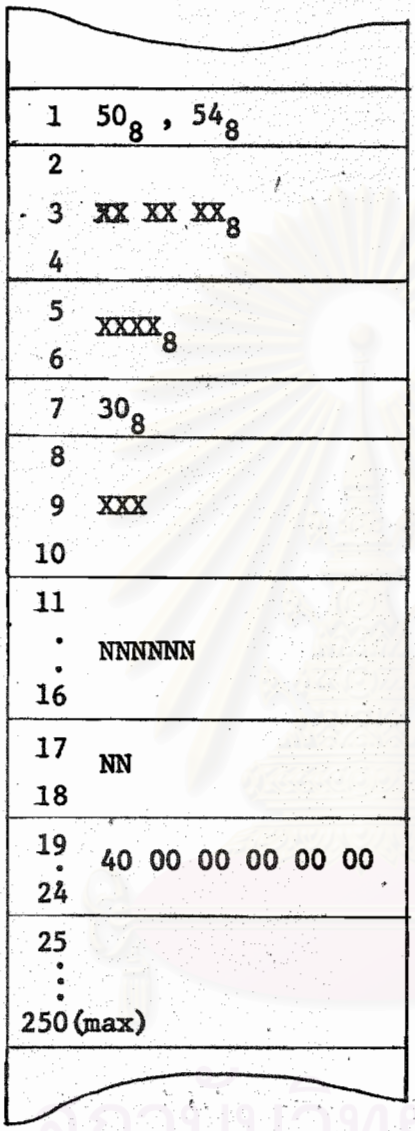


Fig. A-1a. Segment header record.

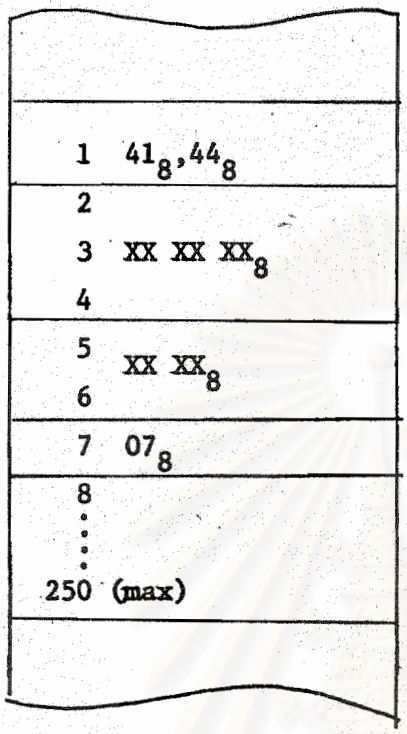


Fig. A-1b. Non-header record.

Record type	Character		Function
	Location	Name	
Segment-header and non-header records (50_8,54_8, 41_8,44_8)	1	Banner	Identifies record type (octal designations below): 50^{Q}_{8} - segment header rec., not last record of a unit; 54^{*}_{8} - segment header rec., last rec. of a unit; 41^{J}_{8} - not segment header rec., not last record of a unit; 44^{M}_{8} - not segment header record, last record of a unit.
	2-4	Record length	Designates number of characters in rec. in octal.
	5-6	Record sequence number	Specifies no. of backspaces (in octal) to position BRT for reading previous segment header record.
	7	Length of ID and control	Designates the no. of char. in the ID & control fields; octal 30 for segment-header records, 7 for non-header rec.
Segment-header (50_8,54_8)	8-10	Revision number	Three-char. no. assigned by programmer in his symbolic program. If unassigned, the assembly program assigns zero.
	11-16	Program name	Six-character program name assigned by programmer.
	17-18	Segment name	Two-character segment name assigned by programmer.
	19-24	Visibility key (octal)	Six-char. loading key assigned by programmer and used by loader-monitor when searching for a unit. (It may be use to correlate two or more units as a system subset to be run together or to distinguish between different versions of the same program.)

Table A-1. Identification and Control Fields of BRT Program Unit.

Control character			Meaning
No.	Octal	Binary	
1	01 to 17	00nnnn	Interpret the nnnn digits as a binary number. Move following nnnn chars. to successive locations, placing leftmost char. in location specified by current setting of distribution counter (in X6), clear punctuation in locations into which the chars. are moved. Advance distribution counter by nnnn.
2	21 to 37	01nnnn	Perform same function as control char. no. 1, and set word mark in leftmost char. location loaded.
3	41 to 57	10nnnn	Perform same functions as control char. no. 1, and set an item mark in leftmost char. location loaded.
4	60	110000	Place following 3 chars. into distribution counter. (The next string will be loaded with its leftmost char. at this address.)
5	61	110001	Terminate loading. Interpret the following 3 chars. as normal starting location for unit just loaded.
6	62	110010	Clear area of memory, using following 7 chars. to identify area to be cleared and char. with which to clear it. (Chars. 1 thru 3 are interpreted as lowest address of the area to be cleared; chars. 4 thru 6 are interpreted as highest address; and char. 7 is transferred to every location in the cleared areas with punctuation marks cleared.)
7	63	110011	Set an item mark in location whose address is one less than current setting of dis. counter.
8	64	110100	Set an item mark in the location whose address is 1 less than current value of dis. counter.
9	77	111111	Read the next record.

Table A-2. Data Field Control Characters.

APPENDIX B

RELOCATABLE OBJECT MODULE FORMAT

In FORTRAN D compiler the compiled program units are kept in relocatable format before linked, and relocated with other units to form a single load module with fixed address. The relocatable module is organized into code regions and certain addressing conventions is used to allow proper relocation and intercommunication between modules. There are three considerations concerning these conventions:

1. Regionalization;
2. Address Interpretation;
3. Relocation of address fields.

Regionalization

Each program unit begins at 4096_{10} and five DSA statements (Define Symbolic Address) at this origin define six coding region boundaries as specified in table B-1. Each region must be contiguous and consecutive in memory, and each contains specific data type to which a corresponding relocation factor applies. These regions are shown diagramatically in fig. B-1 and are described as followed.

Region 1 contains constants, data storage areas, and argument DSA statements. This region may be empty. All information and punctuation is reproduces when this region is relocated.

Region 2 contains the names of all other program units which are "called" by the present unit. Each name appears left justified in a six-character field. A word mark on the leftmost character is the only punctuation. Region 2 may be empty; it is deleted during relocation.

Region 3 is the instruction string or "text" and must contain at least one instruction. The first character in this region is the "turn-on-point" to which control will be transferred when the unit is called. There must be a word mark on the leftmost character of each instruction; no other word marks are permitted. All address fields in this region are relocated. Calls to other program units are indicated by branches to the addresses of the left-justified names in region 2. These addresses will be replaced by the relocated turn-on-points of the called unit when this region is processed.

Region 4 contains DSA statements that are relocated in the same manner as region 3. This region may be empty.

Region 5 contains constants, data storage areas, or argument DSA. Thus it serves as an auxillary storage region which can be conveniently adjusted by initialization code in region 6. This region may be empty and reproduced when relocated.

Region 6 contains initialization coding. This code is executed before the program unit is relocated. After loading of the unit, control is passed on to this region. Job parameters such as integer or mantissa precision, memory size, etc., can be interrogated by this region and other regions, usually region 5, are adjusted accordingly. If the region dimensions are altered as the result of initialization, the corresponding region-boundary pointer is adjusted accordingly. This region is

deleted when the program unit is relocated.

Address Interpretation

Each instruction in region 3 begins with a word-marked character and continues thru to the location preceeding the next word mark. Address fields within instructions are determined by an instruction length algorithm summarized in table B-2.

Relocation of Address Fields

Punctuation of a direct or indirect address field indicates whether it is to be relocated and which relocation delta (offset) should be used. The item marks to control relocation can be set by initialization coding.

Punctuation	Type of Address
Item mark on first character	Absolute addr.: no relocation
Item mark on middle char.	Unlabeled common address
Item mark on first and middle characters	Labeled common address
No item marks on first and middle characters	Not a common address; use appropriate regional delta as followed.

If there are no item marks on the first and middle characters of the address, the relocation delta appropriate to the region is chosen by the following table

$0 = ADDR - 4096_{10}$	do not relocate
$4096 = ADDR - DSA_1$	add delta of region 1
$DSA_1 = ADDR - DSA_2$	replace by address of turn-on-point in unit called
$DSA_2 = ADDR - DSA_5$	add delta of region 3
$DSA_5 = ADDR - 32,767_{10}$	do not relocate

For indexed address, the delta for region 1 is used in relocation.

Location	DSA	Contents	Notes
4096-4098	1	Points to 1st location not in region 1.	Normally the first location in region 2.
4099-4101	2	Turn-on-point.	When region 2 is empty DSA1 is same as DSA2.
4102-4104	3	Points to last opcode in region 3.	If region 4 empty, DSA3 should be repeated as DSA4.
4105-4107	4	Points to last location in region 4.	
4108-4110	5	Points to 1st char. in region 5.	Normally 1st location in region 6.

Table B-1 specifies region boundaries using DSA entries.

Delete	Region pointer DSA	
	Region 1	Unrelocated data area
Detete	Region 2	External reference
	Region 3	Text
	Region 4	DSA argument
	Region 5	Auxillary storage area
Delete	Region 6	Initialization code

Fig. B-1 shows regionalization of relocatable module.

Instruction length (character)	Secondary characteristics	No. of address fields	Character position addresses	Example
1	--	0	--	NOP
2	--	0	--	CAM 00
3	--	1	123	DSA addr
4	No item mark on last char.	1	234	BS addr
	Item mark on last char. Branch instruction		234	B addr
	Item mark on last char. Not a branch instruction		123	DSA addr,V
5 or 6	--	1	234	SCR addr,V
7 or more	PDT,PCB	1	234	PDT addr,V_1,...
7 or more	not a PDT,PCB	2	234,567	MCW addr,addr

Table B-2 shows algorithm used in determining types of instruction.

APPENDIX C

RELOCATING ROUTINE

The relocating routine is the compiler program unit named ACARTG and is list on the following pages together with general descriptions of the working of the codes. The next is the flowchart as derived from the object program.

```
31174   W   15400000100000
31203   W   14037135
31207   W   65032036
31213   W   22036124
31217   W   65031420
31223   W   15006700000030
31232   W   22000022000026
31241   W   5403130103130010
31251   W   33006511006512
31260   W   6503134745
31265   W   140371560007715
31274   W   6500651700
31301   W   14037232400007
31310   W   65007502
31314   W   20
31315   W   14037237400007
31324   W   65007502
31330   W   20
31331   W   65007420
31335   W   22
31336   W   65007420
31342   W   22
31343   W   23031300
31347   W   14037157007715
31356   W   14000020004407
31365   W   14000175000151
31374   W   5500620601000054
31404   W   5403121300523004
31414   W   23036124
31420   W   35006671
31424   W   14006727006671
31433   W   14006727005323
31442   W   34010002006727
31451   W   35037225006727
31460   W   14000030006700
31467   W   14003000000030
31476   W   14600205600204
31505   W   65005231
31511   W   14005330600102
31520   W   14006727005323
31527   W   65005231
31533   W   14005330600124
31542   W   14000111600054
31551   W   65007502
```

Replace entry (program name) in External Reference Table(Region 2) with turn-on-point of that program unit in resolved ESD Table.

Calculate base location of program and data areas; convert to octal then print out as part of Object Memory Map.

```
31556   W   140000030003000
31565   W   15006700000030
31574   W   65005570
31600   W   33010002037225
31607   W   6503164542
31614   W   140371630331 40
31623   W   140372250372 76
31632   W   350372250066 71
31641   W   65031663
31645   W   140100050372 76
31654   W   350100050066 71
```

Initialize pointer to tape buffer area; check starting address of Program unit to be loaded.

```
31663   W   140373130372 42
31672   W   150001117067 04
31701   W   15006727
31705   W   2400670670
31712   W   2400000470
31717   W   330067060371 41
31726   W   6500546544
```

Include the name of program unit just loaded (being relocated) and its starting address in resolved ESD Table.

```
31733   W   330100020100 05
31742   W   6503206142
31747   W   147100030000 04
```

Check for any external reference in Region 2 of this program unit.

```
31756   W   2400000467
31763   W   140067030000 20
31772   W   331000004000 00
32001   W   2400002070
32006   W   6503117442
32013   W   350371450000 20
32022   W   330067060000 20
32031   W   6503177245
32036   W   350371510000 04
32045   W   330100020000 04
32054   W   6503176344
```

Replace each external reference in Region 2 with its starting address if that reference is resolved.

```
32061   W   140067436000 32
32070   W   14000113
32074   W   14
32076   W   14037157
32102   W   14006734
32106   W   140373106000 03
32115   W   140371440067 34
32124   W   65032216
```

Set header label for the segment head record of program unit (Q-record)

```
32130   W   000372
```

Contain number of char. in buffer.

```
32133   W   140371767000 22
32142   W   140321326000 06
32151   W   200305540311 52
32160   W   65007502
```

Set end of record character ($), record mark and no. of char. in the record; then branch to write this record to tape.

```
32165   W   14037264600003
32174   W   14037153600011
32203   W   14006734
32207   W   34037136006734
32216   W   15037135000024
32225   W   14600011037351
32234   W   14037351000024
32243   W   34037145000024
32252   W   34000030000024
32261   W   14000030037245
32270   W   34037132037245
32277   W   5503273360000341
32307   W   22710014
32313   W   20710014
32317   W   14037276500003
32326   W   22500001
32332   W   34006671500003
32341   W   23500001
32345   W   34037147000024
32354   W   65033000
32360   W   14037276037124
32367   W   35006653037124
32376   W   14037124006712
32405   W   1073727400671601
32415   W   14006716032447
32424   W   5503244003725200
32434   W   65032714
32440   W   55032454737122 00
32450   W   65032714
32454   W   34037136037124
32463   W   33037276037124
32472   W   6503244045
32477   W   14006716032540
32506   W   54032522737122 40
32516   W   65032541
32522   W   34037136037124
32531   W   55032506737122 15
32541   W   14037124037276
32550   W   35037136037124
32557   W   14037124006715
32566   W   22006707
32572   W   34006671006712
32601   W   34006671006715
32610   W   23006707
32614   W   1000670770665031
32624   W   2400002470
32631   W   65032317
```

Set label for non-header record (J-record).

Initialize X5 for the relocation process (X5 signifies address of current character in program string being relocated).

Set pointer to the end of buffer area.

Branch if J-record.

Transfer control character 60 and relocate starting address of this program string to tape buffer area; if Q-record 37276 would contain 10017.

Routine to handle data area defined by DA-statement; this routine scans the area until difference in character of punctuation mark is detected then use control character 62 to specify this area in the tape buffer area.

```
32635   W   5503310200665300
32645   W   33037276010016
32654   W   6503267442
32661   W   14037163032733
32670   W   65036674
32674   W   5503457500665300
32704   W   5503311500665300
32714   W   3200665303725217
32724   W   14037252706650
32733   W   65033000
32737   W   33037245000024
32746   W   6503213342
32753   W   14033062032733
32762   W   14037246700022
32771   W   34037136000024
33000   W   35006653
33004   W   14000024006652
33013   W   34037136000024
33022   W   35037252
33026   W   33037245000024
33035   W   6503267446
33042   W   5403270473727410
33052   W   5403263573727420
33062   W   65036674
33066   W   5503236000665317
33076   W   65033026
33102   W   14037163037252
33111   W   65033062
33115   W   14037156037252
33124   W   33037276010002
33133   W   6503342244
33140   W   65033223
33144   W   14033062033140
33153   W   14006671006662
33162   W   22037274
33166   W   34037276006671
33175   W   23037274
33201   W   35010005006671
33210   W   14010005037276
33217   W   65033022
```

Handle character with item mark: if item mark at the begining of the field then normal processing (control character 40); if not, required special handling eg. check end of program, or check buffer-filled condition.

Initialization before processing of the next field.

This loop transfers character to buffer until buffer is filled or punctuation mark is detected or number of character transferred exceeds 17_8 (data area defined by DA-statement).

Process field with item mark (control character 40)

Start processing of field begins with word mark; check if Region 1 is being processed.

For field in Region 1, shift Pointer if turn-on-point is not normal (not 10017).

```
33223   W   14037276000020
33232   W   21033712
33236   W   33037276010013
33245   W   6503342241
33252   W   10400001477777730
33262   W   2403712467
33267   W   2403712770
```

Set pointers to appropriate positions in the current instruction (field begins with word mark)

```
33274   W   33037276010010
33303   W   6503341641
33310   W   35000020037124
33317   W   55033712037124O4
33327   W   34037136000020
33336   W   33037124037153
33345   W   6503347441
33352   W   33037124037150
33361   W   6503342244
33366   W   55033716737274 65
33376   W   54034562737125 20
33406   W   55033436737274 00
33416   W   65036020
```

Determine length of instruction and type of opcode.

```
33422   W   54032645737274 20
```

Check IM at beginning of field.

```
33432   W   65033062
```

Branch to transfer to buffer.

```
33436   W   65036020
33442   W   10737274037342 10
33452   W   2403346567
33457   W   32037003033457 70
33467   W   32
33470   W   65033422
```

If NOP instruction (00) branch to check type of address and relocate then change opcode to 40.

```
33474   W   55033416737274 64
33504   W   55033416737274 66
```

PDT/PCB checking (single operand)

```
33514   W   65036020
33520   W   34037145000020
33527   W   65033416
```

2 operands instruction

.For instruction of length 7 or more.

```
33533   W   35037351
33537   W   14400003033573
33546   W   14005351037127
33555   W   34037136037351
33564   W   55033641737125 71
33574   W   34037155037127
33603   W   33005354037127
33612   W   6503355543
33617   W   14037014036752
33626   W   15000111737311
33635   W   65006052
33641   W   14037351400003
33650   W   34006671000020
33657   W   15037272737305
33666   W   15000020
33672   W   2403730770
33677   W   35006671000020
33706   W   65005500
```

Routine for exit to call-chain pro unit (ACBCCH).

```
33712   W   20033712
33716   W   5403341640000120
33726   W   33400002037017
33735   W   6503353342
33742   W   144000020000004
33751   W   3203713300000270
33761   W   330000040100005
33770   W   6503341643
33775   W   55034161000000200
34005   W   330000040100002
34014   W   6503341644
34021   W   55034546100000000
34031   W   340066710000020
34040   W   151000057373305
34047   W   15000020
34053   W   2403730770
34060   W   350066710000020
34067   W   54034136000000420
34077   W   5403411303371220
34107   W   65005534
34113   W   140373070000004
34122   W   3203717610000140
34132   W   65005534
34136   W   140373070000004
34145   W   3203717610000120
34155   W   65034077
34161   W   330000040370022
34170   W   6503432042
34175   W   330000040370025
34204   W   6503433342
34211   W   330000040370030
34220   W   6503434642
34225   W   330000040370033
34234   W   6503436142
34241   W   330000040370036
34250   W   6503437442
34255   W   330000040370041
34264   W   6503440742
```

Handle branch or DSA instruction; This part classifies normal branch (operand point to address in Region 3), branch to special program unit and call to external program unit.

Check external reference to program unit, if external symbolic name not resolved incluse into unresolved ESD Table, if resolved branch to 34546 to resolve reference point.

Set flag in unresolved external reference entry to designate branch or DSA instruction.

Check dummy branch instruction which standard program unit is being referred to.
Standard program units are;
ACBFPR
ACBFXR
ACBFPP
ACBFXP
DAOIO
ACBOIO

```
34271   W   5403430540000020
34301   W   65033416
34305   W   21400000000002
34314   W   65034556
34320   W   14037063037011
34327   W   65034416
34333   W   14037071037011
34342   W   65034416
34346   W   14037077037011
34355   W   65034416
34361   W   14037105037011
34370   W   65034416
34374   W   14037113037011
34403   W   65034416
34407   W   14037121037011
34416   W   14006703000014
34425   W   33300000037011
34434   W   2400001467
34441   W   6503453342
34446   W   35037145000014
34455   W   33006706000014
34464   W   6503442545
34471   W   34006671000020
34500   W   15037011737305
34507   W   15000020
34513   W   2403730770
34520   W   35006671000020
34527   W   65005534
34533   W   14300000400002
34542   W   65034556
34546   W   1010000540000221
34556   W   65033422
34562   W   35037136000020
34571   W   65033416
34575   W   35037136000024
34604   W   33037276010016
34613   W   6503213345
34620   W   14000024037124
34627   W   34037145037124
34636   W   4003526003735220
34646   W   33037245037124
34655   W   6503213344
34662   W   65034756
```

Select standard program unit name to check if resolved.

Check if external reference (standard program unit) is resolved; if yes replace operand of branch instruction with resolved address; if no include symbolic program name and address of reference into unresolved ESD Table.

Place turn-on-point of resolved external reference in operand field of branch instruction.

Handle DSA instruction with variar

Check end of program unit.

Check if buffer is filled before placing information to resolve external reference.

```
34666   W   140330620346 62
34675   W   4003472403735220
34705   W   14006727500003
34714   W   14005343
34720   W   65035245
34724   W   140372465000 04
34733   W   14006727
34737   W   14005343
34743   W   340371500000 24
34752   W   65034620
34756   W   330373070373 13
34765   W   6503527342
34772   W   140372420000 04
35001   W   330373070000 04
35010   W   6503527342
35015   W   331000000001 11
35024   W   350371550000 04
35033   W   6503500145
35040   W   140371630346 62
35047   W   141000035000 03
35056   W   14037273
35062   W   32037133500001 60
35072   W   54035115100001 02
35102   W   140371450053 43
35111   W   65035124
35115   W   140371460053 43
35124   W   20737305
35130   W   54035166100001 01
35140   W   140371630346 36
35147   W   140371630346 75
35156   W   21037352
35162   W   65035210
35166   W   140371670346 36
35175   W   140371670346 75
35204   W   20037352
35210   W   10100000100011 41
35220   W   14000004037242
35227   W   340371550372 42
35236   W   340371550373 07
35245   W   340371470000 24
35254   W   65034620
35260   W   340371360371 24
35267   W   65034646
```

Place starting location of program unit just relocated after pointer to branch or DSA instruction that makes the reference.

If there is no unresolved ESD branch to 35273.

Check if there is any reference to the program unit just relocated;
if no branch to 35273
if yes continue through 35040.

Place pointer (address) of branch instruction, that refers this program unit, at the end of this program unit (in buffer area).

Set & modify flag and opcode to indicate type of reference (Branch or DSA instruction).

Delete entry in unresolved ESD table that has just been resolved.

```
35273   W   15005336737305
35302   W   5403535500672420
35312   W   33037135006724
35321   W   6503577142
35326   W   5403534200523005
35336   W   65035355
35342   W   14006727006724
35351   W   20006724
35355   W   6501453140
35362   W   34010016006727
35371   W   35010005006727
35400   W   65035415
35404   W   40
35405   W   5402767400667520
35415   W   32037147600000307
35425   W   14005360500003
35434   W   35000030000024
35443   W   34037136000024
35452   W   14000024600006
35461   W   20030554031152
35470   W   65035517
35474   W   40
35475   W   14003427000114
35504   W   14037136000157
35513   W   65035745
35517   W   34037136000010
35526   W   34037151037055
35535   W   55035675000001006
35545   W   55035571000001007
35555   W   54035711737045 20
35565   W   65035517
35571   W   23035474
35575   W   14037136000157
35604   W   14006706006633
35613   W   14037163035761
35622   W   14033062006262
35631   W   65035745
35635   W   14037163006262
35644   W   14006633006706
35653   W   23036324035404
35662   W   14003427000114
35671   W   65036272
35675   W   33007377037134
35704   W   6503557142
35711   W   14737053000111
35720   W   14004662000004
35727   W   14037273000157
35736   W   14003434000114
35745   W   65007502
```

Compute starting address for next program unit to be relocated.

Set banner character to 54_8 or 44_8 (last record), set 61000202 at the end of buffer (branch to loader after load) and set number of character of the record.

Branch to check if any standard program unit is required.

Set condition to load program unit by visibility & relative position and tape drive#0.

Check if any of the standard program unit is required; if yes, load that unit.

Set conditions to load program unit after standard program unit (by visibility & relative position) and set conditions to test when user's program unit is required (program name: 000000).

Set condition to test if user's program unit (drive # 3) is to be loaded.

Check if ACBOIO is needed.

Prepare to load standard program unit needed (search mode 60, drive # 0 and program name as required).

```
35752   W  140000300067OO
35761   W  65036272
35765   W  65006230
35771   W  5403600500523004
36001   W  65035326
36005   W  14006727006724
36014   W  65035355
36020   W  2403614770
36025   W  144000020371 27
36034   W  35037351
36040   W  3203712503735170
36050   W  3203713303712570
36060   W  5403622703712620
36070   W  5403626103712520
36100   W  5503615003735100
36110   W  5503615003735170
36120   W  65036150
36124   W  40
36125   W  34006662400002
36134   W  3203735140000070
36144   W  65033422
36150   W  33037127006647
36157   W  6503613444
36164   W  330371270100O2
36173   W  6503612544
36200   W  33037127010016
36207   W  6503613443
36214   W  34006671400002
36223   W  65036134
36227   W  5403625203712520
36237   W  350373424000O2
36246   W  65036261
36252   W  35006721400002
36261   W  21400001
36265   W  21
36266   W  65036134
```

Routine to determine type of address & relocate accordingly.

Relocate with delta I; reset index and return.

If address is not a common address or not absolute; check if to relocate with delta I,III or no relocation.

For common address; test labeled or unlabeled, then relocate accordingly.

```
36272   W   35005230
36276   W   14004407000020
36305   W   14036774000151
36314   W   65000202
36320   W   65036405
36324   W   40
36325   W   35037351
36331   W   33001735036776
36340   W   6503640545
36345   W   3200177703735101
36355   W   5503642003735101
36365   W   3200177703735104
36375   W   5500641603735104
36405   W   14000175000151
36414   W   65000172
36420   W   14004407000014
36427   W   14037176300003
36436   W   14300002000014
36445   W   65007420
36451   W   22
36452   W   65007420
36456   W   22
36457   W   65007420
36463   W   20
36464   W   65007420
36470   W   20
36471   W   33300007037232
36500   W   6503646445
36505   W   65007420
36511   W   22
36512   W   65007420
36516   W   22
36517   W   14000020004407
36526   W   22031300
36532   W   65036571
36536   W   5503655500173220
36546   W   14037147001732
36555   W   3203731000173270
36565   W   65036626
36571   W   14037156036644
36600   W   14037156007715
36607   W   14037001000151
36616   W   5403653600173205
36626   W   3200205140012277
36636   W   14
36640   W   65007502
```

Set X4 to point to buffer area; set own code exit; branch to search program unit required then return when founded to execute own code.

Check own code requirement of user's program unit (program name: 000000).

Reset own code exit and load program unit.

```
36645   W  140000020000004
36654   W  20003437
36660   W  65003442
36664   W  21003437
36670   W  65000172
36674   W  2403674170
36701   W  1073727470002201
36711   W  34037136037276
36720   W  34037136000024
36727   W  34037136006653
36736   W  65033066
36742   W  004547
36745   W  050003
36750   W  000000
36753   W  001732
36756   W  001735
36761   W  001777
36764   W  002051
36767   W  002002
36772   W  036320
36775   W  0047
36777   W  036626
37002   W  4000
37004   W  1515151515151
37012   W  050002
37015   W  704435
37020   W  004753  ACBFPR
37023   W  004754  ACBFXR
37026   W  004755  ACBFPP
37031   W  004756  ACBPXP
37034   W  004761  DAOIO
37037   W  004757  ACBOIO
37042   W  050000
37045   W  207361
37050   W  037055
37053   W  037063
37056   W  212322264751
37064   W  212322266751
37072   W  212322264747
37100   W  212322266747
37106   W  242146314615
37114   W  212322463146
```

Transfer one character from progra to the tape buffer and update pointer as necessary.

Constants, parameters and working areas for the relocating routine.

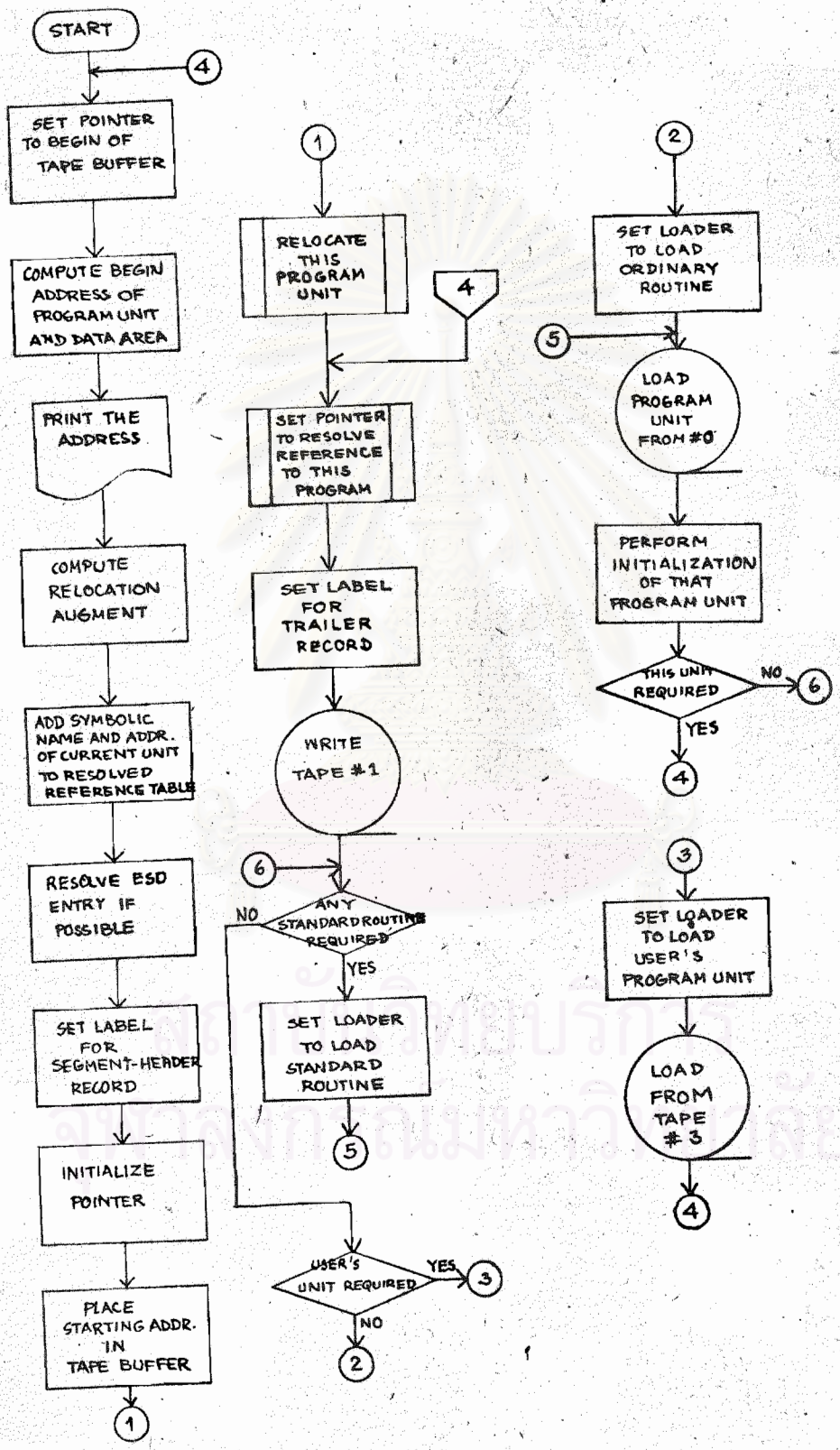
Working area for prog. name of std. prog

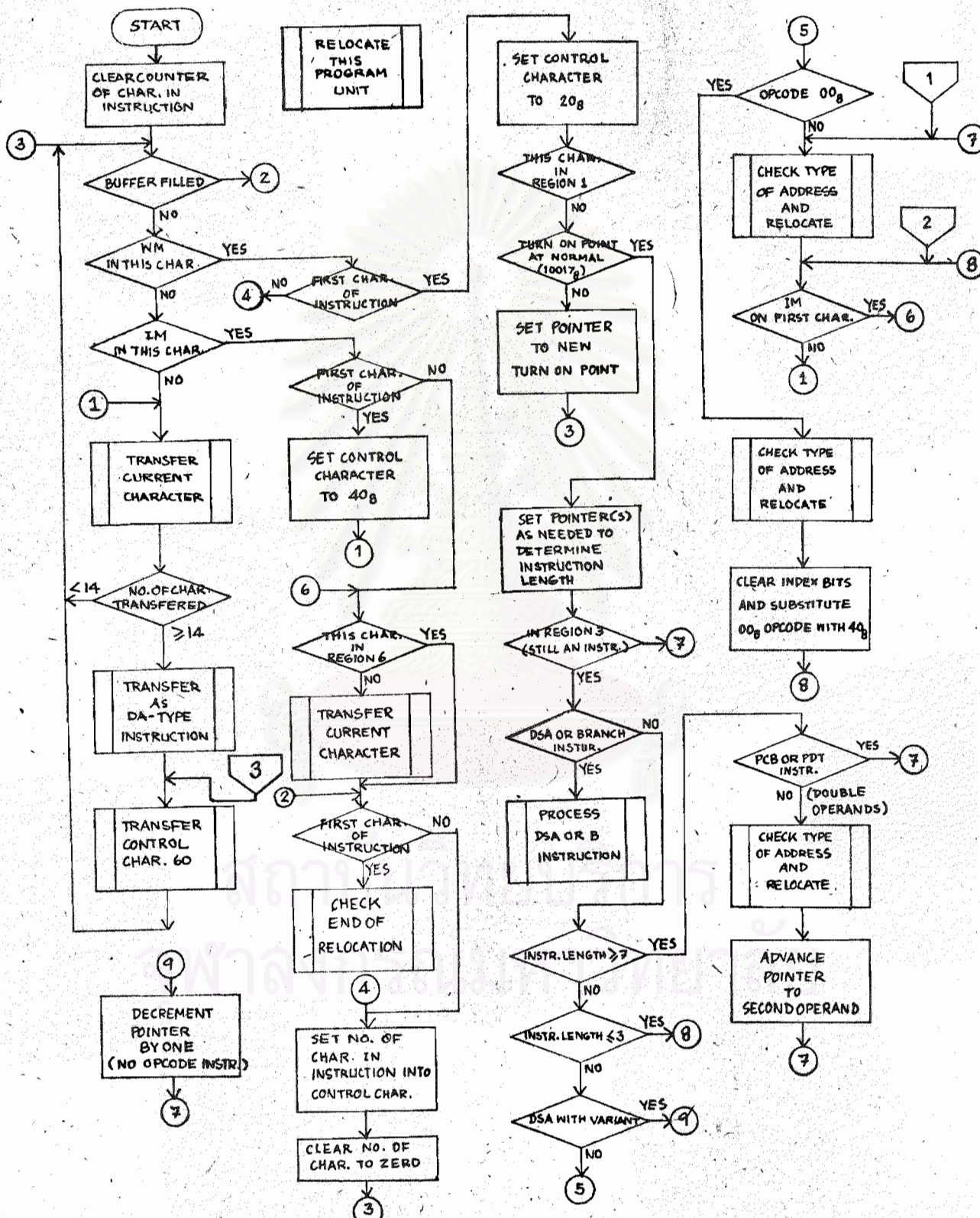
Operand of branch instructions that refer to standard routines.

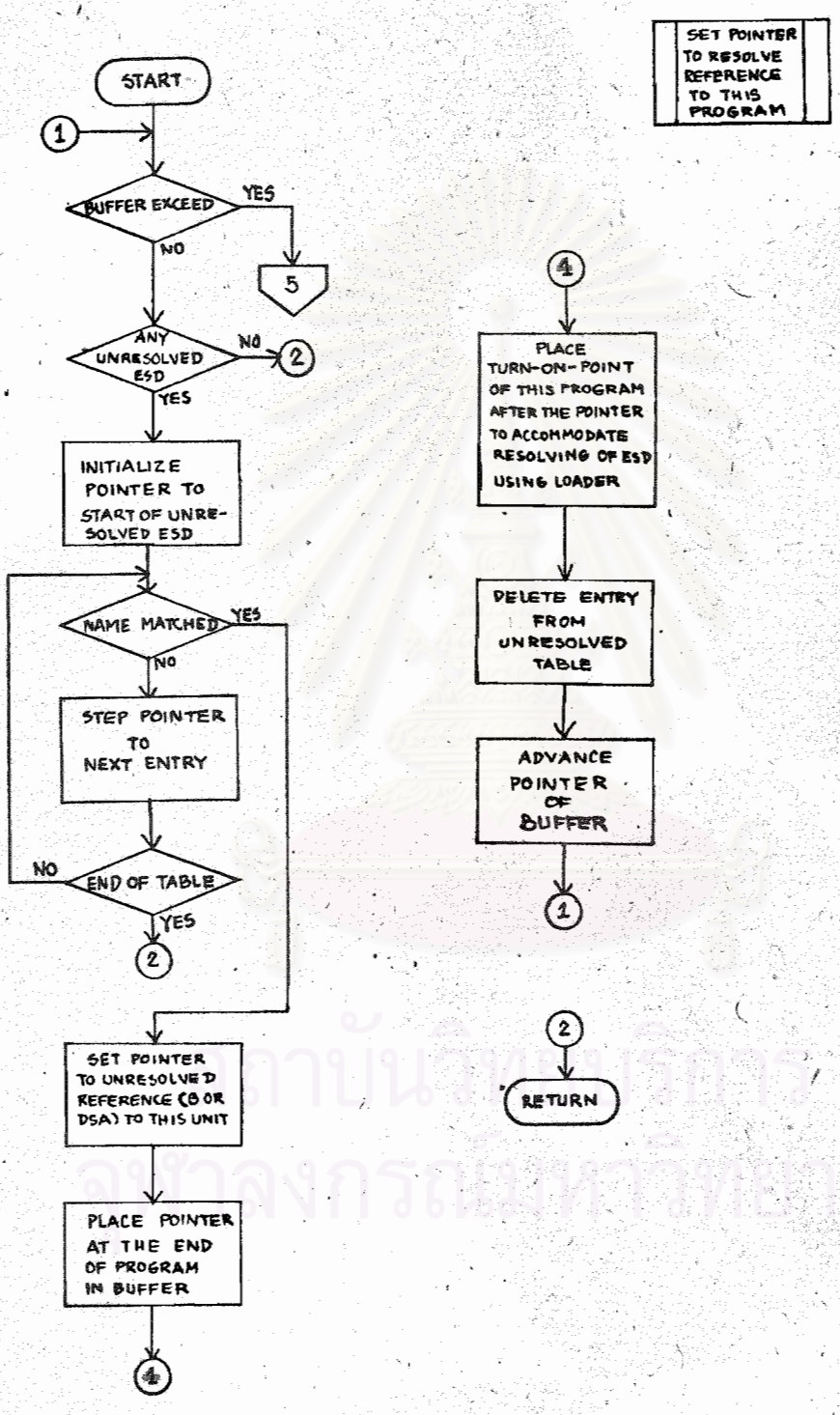
Table of the six standard program unit used.

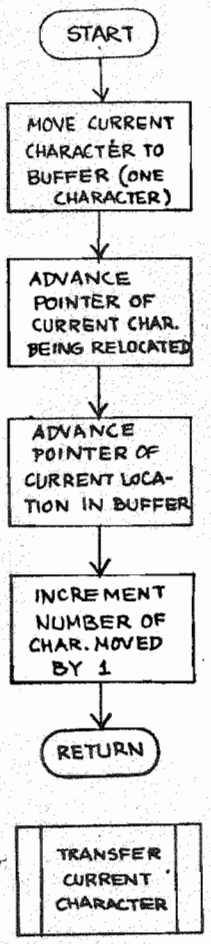
```
37122   W   030657
37125   W   000000
37130   W   000372   (250₁₀); record length of program record in tape
37133   W   000000
37136   W   01
37137   W   037353
37142   W   21
37143   W   0002
37145   W   03
37146   W   23
37147   W   04
37150   W   05
37151   W   06
37152   W   26
37153   W   07
37154   W   10
37155   W   11
37156   W   20
37157   W   30
37160   W   000035
37163   W   40
37164   W   000045
37167   W   54       Banner character for *-record.
37170   W   000065
37173   W   000072
37176   W   77       Control character ($ to set end of tape record).
37177   W   002477
37202   W   005645
37205   W   000114
37210   W   031223   Contain exit point to monitor.
37213   W   000157
37216   W   2123242325
37223   W   010017
37226   W   0125462615
37233   W   0125513115
37240   W   151515
37243   W   031150
37246   W   64       Control character.
37247   W   100000
37252   W   20
37253   W   000161
37256   W   212322254322
37264   W   41       Banner character for J-record.
37265   W   212322232330
```

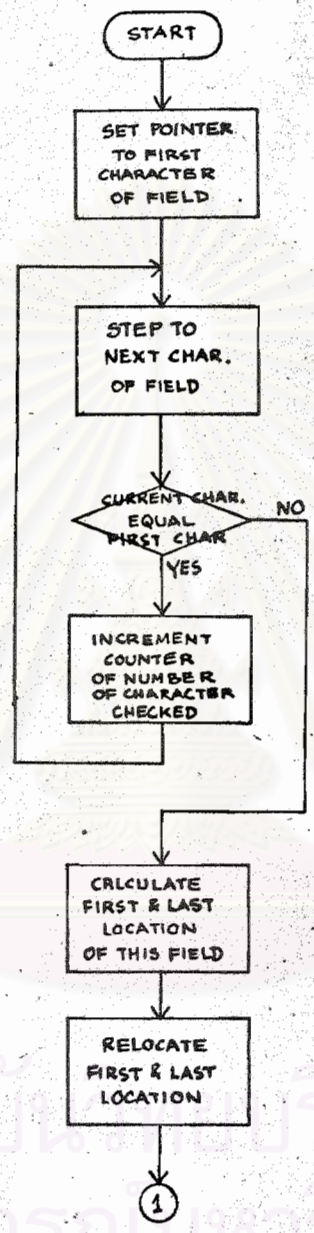
```
37273   W   60004547
37277   W   400000000000
37305   W   030157  Pointer to table of unresolved ESD entry.
37310   W   50
37311   W   030157  Base address of unresolved ESD Table.
37314   W   000202
37317   W   00
37320   W   000202
37326   W   61702474
37332   W   000000000000003231
37343   W   777777777777
37351   W   00
37352   W   156000016126000000000001600001572101600O
                ↓ Tape buffer area.
```

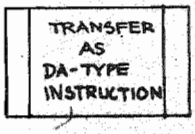

START
4
SET POINTER TO BEGIN OF TAPE BUFFER
COMPUTE BEGIN ADDRESS OF PROGRAM UNIT AND DATA AREA
PRINT THE ADDRESS
COMPUTE RELOCATION AUGMENT
ADD SYMBOLIC NAME AND ADDR. OF CURRENT UNIT TO RESOLVED REFERENCE TABLE
RESOLVE ESD ENTRY IF POSSIBLE
SET LABEL FOR SEGMENT-HEADER RECORD
INITIALIZE POINTER
PLACE STARTING ADDR. IN TAPE BUFFER
1
1
RELOCATE THIS PROGRAM UNIT
4
SET POINTER TO RESOLVE REFERENCE TO THIS PROGRAM
SET LABEL FOR TRAILER RECORD
WRITE TAPE #1
6
ANY STANDARD ROUTINE REQUIRED
NO
YES
SET LOADER TO LOAD STANDARD ROUTINE
5
USER'S UNIT REQUIRED
YES
3
NO
2
2
SET LOADER TO LOAD ORDINARY ROUTINE
5
LOAD PROGRAM UNIT FROM #0
PERFORM INITIALIZATION OF THAT PROGRAM UNIT
THIS UNIT REQUIRED
NO
6
YES
4
3
SET LOADER TO LOAD USER'S PROGRAM UNIT
LOAD FROM TAPE #3
4

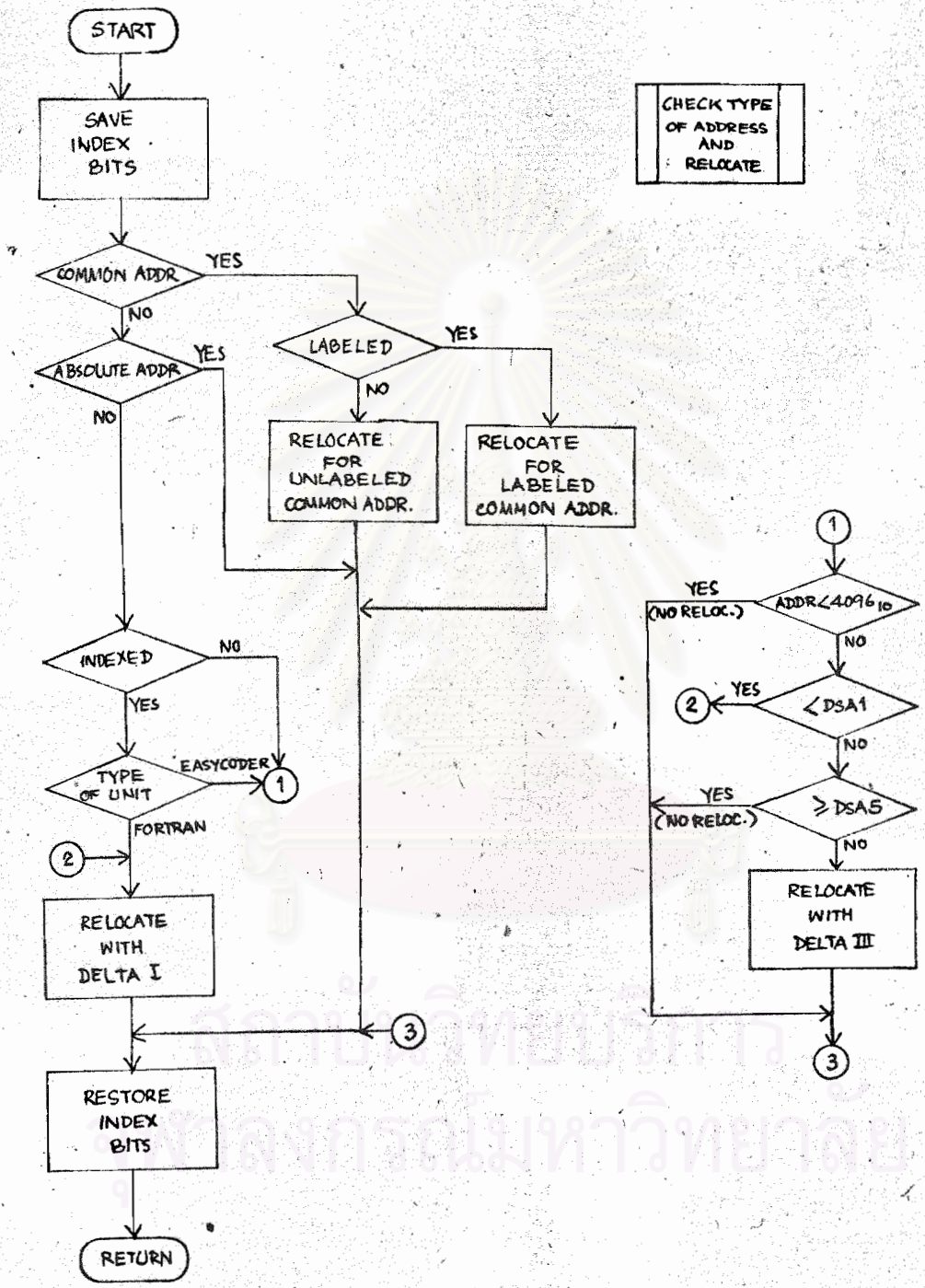

START
CLEAR COUNTER OF CHAR. IN INSTRUCTION
BUFFER FILLED
NO
NM IN THIS CHAR.
YES
NO
FIRST CHAR. OF INSTRUCTION
YES
NO
IM IN THIS CHAR.
YES
NO
TRANSFER CURRENT CHARACTER
NO. OF CHAR. TRANSFERED
≤14
≥14
TRANSFER AS DA-TYPE INSTRUCTION
TRANSFER CONTROL CHAR. 60
DECREMENT POINTER BY ONE (NO OPCODE INSTR.)
RELOCATE THIS PROGRAM UNIT
FIRST CHAR. OF INSTRUCTION
NO
YES
SET CONTROL CHARACTER TO 40₈
THIS CHAR. IN REGION 6
YES
NO
TRANSFER CURRENT CHARACTER
FIRST CHAR. OF INSTRUCTION
NO
YES
CHECK END OF RELOCATION
SET NO. OF CHAR. IN INSTRUCTION INTO CONTROL CHAR.
CLEAR NO. OF CHAR. TO ZERO
SET CONTROL CHARACTER TO 20₈
THIS CHAR. IN REGION 1
NO
TURN ON POINT AT NORMAL (10017₈)
YES
NO
SET POINTER TO NEW TURN ON POINT
SET POINTER(S) AS NEEDED TO DETERMINE INSTRUCTION LENGTH
IN REGION 3 (STILL AN INSTR.)
YES
DSA OR BRANCH INSTR.
NO
YES
PROCESS DSA OR B INSTRUCTION
INSTR. LENGTH ≥7
YES
NO
INSTR. LENGTH ≤3
YES
NO
DSA WITH VARIANT
YES
NO
OPCODE 00₈
YES
NO
CHECK TYPE OF ADDRESS AND RELOCATE
IM ON FIRST CHAR.
YES
NO
CHECK TYPE OF ADDRESS AND RELOCATE
CLEAR INDEX BITS AND SUBSTITUTE 00₈ OPCODE WITH 40₈
PCB OR PDT INSTR.
YES
NO (DOUBLE OPERANDS)
CHECK TYPE OF ADDRESS AND RELOCATE
ADVANCE POINTER TO SECOND OPERAND

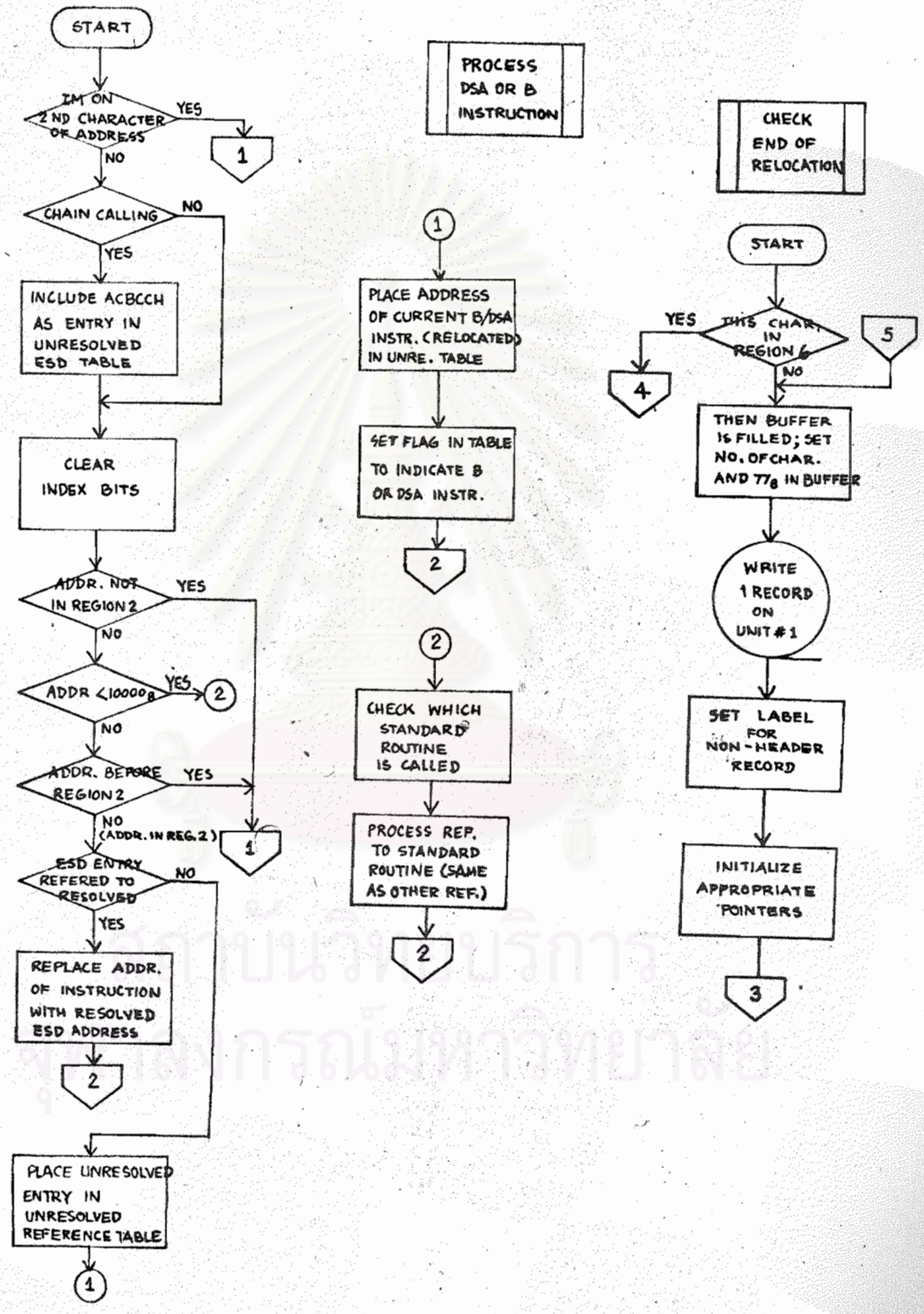

START
1
BUFFER EXCEED
YES
NO
5
ANY UNRESOLVED ESD
NO
2
YES
INITIALIZE POINTER TO START OF UNRESOLVED ESD
NAME MATCHED
YES
NO
STEP POINTER TO NEXT ENTRY
NO
END OF TABLE
YES
2
SET POINTER TO UNRESOLVED REFERENCE (CB OR DSA) TO THIS UNIT
PLACE POINTER AT THE END OF PROGRAM IN BUFFER
4
SET POINTER TO RESOLVE REFERENCE TO THIS PROGRAM
4
PLACE TURN-ON-POINT OF THIS PROGRAM AFTER THE POINTER TO ACCOMMODATE RESOLVING OF ESD USING LOADER
DELETE ENTRY FROM UNRESOLVED TABLE
ADVANCE POINTER OF BUFFER
1
2
RETURN








START
SAVE INDEX BITS
COMMON ADDR
YES
NO
ABSOLUTE ADDR
YES
NO
LABELED
YES
NO
RELOCATE FOR UNLABELED COMMON ADDR.
RELOCATE FOR LABELED COMMON ADDR.
INDEXED
NO
YES
TYPE OF UNIT
EASYCODER
1
FORTRAN
2
RELOCATE WITH DELTA I
3
RESTORE INDEX BITS
RETURN
CHECK TYPE OF ADDRESS AND RELOCATE
1
ADDR<4096 10
YES (NO RELOC.)
NO
2
YES
<DSA1
NO
≥DSA5
YES (NO RELOC.)
NO
RELOCATE WITH DELTA III
3


START
IM ON 2ND CHARACTER OF ADDRESS
YES
NO
1
CHAIN CALLING
NO
YES
INCLUDE ACBCCH AS ENTRY IN UNRESOLVED ESD TABLE
CLEAR INDEX BITS
ADDR. NOT IN REGION 2
YES
NO
ADDR <10000₈
YES
2
NO
ADDR. BEFORE REGION 2
YES
NO (ADDR. IN REG. 2)
1
ESD ENTRY REFERED TO RESOLVED
NO
YES
REPLACE ADDR. OF INSTRUCTION WITH RESOLVED ESD ADDRESS
2
PLACE UNRESOLVED ENTRY IN UNRESOLVED REFERENCE TABLE
1
PROCESS DSA OR B INSTRUCTION
1
PLACE ADDRESS OF CURRENT B/DSA INSTR. (RELOCATED) IN UNRE. TABLE
SET FLAG IN TABLE TO INDICATE B OR DSA INSTR.
2
2
CHECK WHICH STANDARD ROUTINE IS CALLED
PROCESS REF. TO STANDARD ROUTINE (SAME AS OTHER REF.)
2
CHECK END OF RELOCATION
START
THIS CHART IN REGION 6
YES
NO
4
5
THEN BUFFER IS FILLED; SET NO. OF CHAR. AND 77₈ IN BUFFER
WRITE 1 RECORD ON UNIT #1
SET LABEL FOR NON-HEADER RECORD
INITIALIZE APPROPRIATE POINTERS
3

BIOGRAPHY

Mr. Yunyong Teng-amnuay was born on the 20th of August 1954, graduated with the Bachelor Degree in Electrical Engineering with First Class Honour from the Department of Electrical Engineering, Faculty of Engineering, Chulalongkorn University and now works as a lecturer in the Department of Computer Engineering at the same university.