THE UNIVERSITY OF MICHIGAN
COMPUTER AND COMMUNICATION SCIENCES

Final Report

A STUDY OF PROGRAMMING LANGUAGE EFFECTIVENESS

Bruce W. Arden
Principal Investigator

James Hamilton

ORA Project 03222

under contract with:

U. S. ARMY SAFEGUARD SYSTEMS COMMAND
CONTRACT NO. DAHC60-70-C-0036
CONTRACTS OFFICE SSC-C
HUNTSVILLE, ALABAMA

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

May 1970

A. INTRODUCTION

The goal of this study is to compare the effectiveness of three programming languages, in terms of object code efficiency. There has been very little effort to evaluate the ease of coding in the three languages, although number of source lines is some measure of this.

The study consists of three separate problems, each coded in three languages: Assembler, FORTRAN IV, and PL/1. Each problem was tested for several different cases (data sets), in order to test each of the paths in the logic of the program.

The object machine is the IBM 360 Model 67, running under the Michigan Terminal System (MTS). The language processors used were:

1. MTS G Assembler.

2. FORTRAN H Compiler.

3 PL/1 F Compiler, version .

No statistics are given on the relation efficiency of these processors.

B. PROBLEM SPECIFICATION AND CONSTRAINTS

The problems were given by complete flowcharts, which were abstracted from the Safeguard ABM system. The data structures were also specified. Very little variation was allowed in program structure; the flowcharts were followed quite closely. Greater variation was allowed in the implementation of the data structures, and in an effort to evaluate the effects of data structure constraints, two of the problems were coded in two versions—one a strict interpretation of the data structure specification, and the other, a fairly free interpretation.

A brief verbal description of the three problems follows:

1. GET TASK—A task scheduler for a multiprocessor system. This essentially implements a pert-like network of tasks and routines, in which each task is conditionally enabled by each of its predecessors, and absolutely enabled when it has been conditionally enabled by all of its predecessors. The task is then started and passed the address of its data set. This is intended to be a list processing problem, but it has bit manipulation aspects also.

2. EXLIGEN—The execution list generator. This is part of a collection of tasks which ultimately determines the sequency of pulses to be transmitted by the radar. It processes a radar template and pulse pattern table, and produces an execution list of radar "events," ordered by priority and timing constraints, etc. This is intended to be a bit manipulation problem, but bit manipulation is not inherently part of the problem, and the free interpretation of the data structure requires no bit manipulation at all. Instead it involves a fairly complex flow of control.

3. FILTER--A Kalman filter which is a part of the radar tracking processing. This is a very simple program using little more than floating point arithmetic. It is intended as a numerical problem.

C. DRIVER PROGRAMS

For each problem a driver program was written to test the programs and to collecting timing statistics. There were separate driver programs for each problem, but, in general, they all went through the following steps:

1. Read input specifying:
    a. Whether or not to provide trace output verifying the correctness of the program.
    b. The number of iterations for which each case is to be run, thus specifying also the number of cases.
2. Set up the data structure, and execute each case for the indicated number of iterations, printing the average time for each case.

The driver for problems 1 and 2 was written in assembly language, and was identical for the three subject languages, except that the driver for the PL/1 version had to be modified to interface properly with the PL/1 library. The driver for problem 3 was written in PL/1.

The timer used was essentially the 360-interval timer, which has an increment of 13 µsec. The MTS supervisor attempts to simulate this timer for each task, in such a way that it indicates only the CPU time used by that task, but inevitably some of the supervisor overhead for other MTS tasks will be included in this time supply because, at the time of an interrupt, the interrupted task state must be preserved before the timer is sampled.

Part of the time, measured by the timer, actually includes the time required to execute the driver program and timer processing routines. In an attempt to exclude this component, a run was made for each problem, in which the subject program was replaced by a null program (one which just returns). The timing statistics resulting from these runs were subtracted from those measured with the subject program. This yields the "adjusted" time listed in the table of results.

D. RESULTS

	ASSEMBLER	FORTRAN	PL/1
GET TASK (strict data structure)			
Source lines[a]	79	50	48
Object instructions	75	251	327
Object size[j]	632	1254	1786[b]
Time - Case 1[c]	2.37	4.51	8.41
Time - Case 2	2.35	4.10	7.04
Adjusted time Case 1[d]	0.70	2.84	6.74
No. of subroutines[e]			
Explicit	0	8	5
Implicit	0	0	7[f]
GET TASK (free data structure)			
Source lines	67	39	48
Object instructions	65	170	292
Object size	332	1036	1650
Time - Case 1	2.16	3.42	4.81
Time - Case 2	2.15	3.45	4.86
Adjusted time Case 1[d]	0.49	1.78	3.14
EXLIGEN (strict data structure)[i]			
Source lines	259	122	133
Object instructions	228	477	1100
Object size	1220	2618	4918
Time - Case 1	1.83	4.98	19.2
Time - Case 2	1.75	4.90	18.46
External references	0	1	13
EXLIGEN (free data structure)[i]			
Source lines	228	126	130
Object instructions	200	405	783
Object size	1104	2228	4295
External data[h]			
Structure size	1036	1214	952
Time - Case 1	1.42	2.86	7.03
Time - Case 2	1.40	2.89	6.84
External references	0	0	8

	ASSEMBLER	FORTRAN	PL/1
FILTER			
Source lines[a]	88	27	28
Object instructions	89	134	351
Object size	364	774	1786[b]
Time	1.86	2.51	3.44
Adjusted[g]	0.74	1.39	2.32
External references	0	0	5
AVERAGE RATIOS			
Source lines		0.50	0.54
Object instructions		2.2	4.3
Object size		2.2	4.0
Time		2.7	7.4

NOTES:

a. Excluding comments.

b. Includes static storage csect and program csect.

c. All times are in milliseconds.

d. Adjusted time equals total time minus fixed time due to driver and timer routines (1.67 msec).

e. Does not include calls to RTEXQ AND EOT, the two driver entries.

f. Actually, the number of external references. Not all of these are actually called by the program. (In fact none are called by FILTER.) All must be loaded, however, and the PL/1 library required turns out to be about 40K bytes, only a small amount of which is actually used.

g. Fixed time for FILTER driver - 1.12 msec.

h. For each of the problems that part of the data structure which was specified by the problem was external to the procedure itself. In problems I and III this quantity was essentially constant, even in the review GET TASK.

i. The fixed time for the Problem II driver was negligible (0.016 msec), so no adjusted time is given.

j. Object size is bytes, decimal.

E. CONCLUSIONS

FORTRAN H appears to produce roughly twice as much object code as the Assembler, for about half as many source lines, and PL/1 produces about four times as much, for, again, about half the source code. The ratios in execution time are even worse. In an attempt to provide general explanations for these differences, the following five categories are proposed:

1. Problem specification contraints.

2. Language restrictions.

3. Language generality.

4. Inefficient compilation—difficulty of compiling for 360; inherent and compiler induced object code inefficiencies.

5. Programmer bias and unfamiliarity with language features.

In more detail:

1. When designing systems, one takes language abilities into account. Both program and data structure are heavily dependent upon language considerations. The programming language dictates, to a large extent, the style of programming. A LISP program, for example, will typically be a collection of recursive "functionals," PL/1 programs may have complex data structures, and several levels of DO group nesting, procedures, and block structure, and FORTRAN offers only a primitive DO loop. But the program structure for these problems was specified by flowcharts taken from assembly language programs. This structure did not allow the use of PL/1 block structure, and in many cases did not even allow simple DO loops in places where DO loops would be useful. This can only add to the differences due to the other four possibilities.

Data structure constraints are probably even more responsible for differences of this type. FORTRAN, for example, deals only with arrays and simple variables, and it is unfair to ask more of it than this. In an attempt to determine the effects of data structure variation, the problem I and problem II programs were revised, to allow a completely free data structure. All three languages appear to benefit about equally in terms of object instructions, and object size, but in execution time, the higher level languages benefit somewhat more than the assembler versions. In the problems at hand, the only important data structure variation involved removing the necessity for some bit manipulation. Therefore, the effects of this variation on the three languages depends on the ability of the language to handle bit manipulation. PL/1 has this ability in the language, but its implementation tends to be extremely inefficient. FORTRAN does not have it at all, so it was necessary to write small assembler language programs to perform the bit string functions. This tended to benefit the FORTRAN versions, since the functions written were more specialized than the very general bit string routines which implement the PL/1 functions.

Thus one concludes that problem specification constraints tend to exaggerate the differences between languages. This should not be interpreted, however, as meaning that the problems themselves are inherently difficult to code in higher level languages, but only that the specifications of this study prevented the most efficient use of such languages.

2. Language restrictions. One might think that a perfect compiler could compile a program equal in efficiency to an assembly language program. But language restrictions, and, to some extent, language generality, prevent this. Most languages restrict the parameters of certain underlying functions in such a way that additional statements are required to preprocess data into the required form. Examples are the restriction of FORTRAN DO-loop parameters to simple integer variables or constants, and the restrictions of pointers, used as bases or in pointer qualification in PL/1, to be simple, nonbased, pointer variables. These restrictions result in extra processing which is irrelevant to the task at hand.

3. Language generality. One answer to the problem of language restrictions is to make the functions more general. PL/1, for example, has removed the restriction on the DO loop. But this requires either that the resulting object code be more complex to handle the more general inputs, or that the compiler recognize the simple cases when they occur. Unfortunately, the former case usually occurs, because it is very difficult, and sometimes impossible for the compiler to recognize possible simplifications.

Many modern languages have added many powerful functions, and much more varied data types, in an attempt to make the languages more generally useful. It is only fair in evaluating the language, to use these facilities wherever they are useful, even though it might be more efficient to do things differently. This has been done in PL/1, where such features are abundant. It would have been possible to translate the FORTRAN programs directly to PL/1, resulting in more efficient PL/1 programs, but using none of the power of PL/1; but it was felt that this would not be a true evaluation of PL/1.

It turns out that the most natural way to generalize language primitives is to add levels of "indirection" in data references. In PL/1 this means that the more complex data structures are referenced via dope vectors. This turns out to be unnecessary in many cases, but it is done just the same. This is perhaps the most common form of inefficiency due to language generality.

4. Still, it is abundantly clear that the available compilers do not recognize many of the possible simplifications, even where it would be simple to do so. FORTRAN H, for example, does not implement multiplication and division by powers of two as shifts, and PL/1 puts out many instructions which might as well be No-ops. Sloppy compilation is still the greatest contributor to the differences observed here.

It has been suggested that the IBM 360, and machines like it are difficult to compile for, either because of the base-displacement addressing, which requires checking of addressibility, or because the 360 instruction set is not well matched to the underlying procedure oriented language primitives. There seems to be evidence for this in the fact that FORTRAN compilers for other machines seem to produce code which rivals assembly language efficiency. However, a study of the object code from these problems does not show any clear instances of language-machine mismatch.

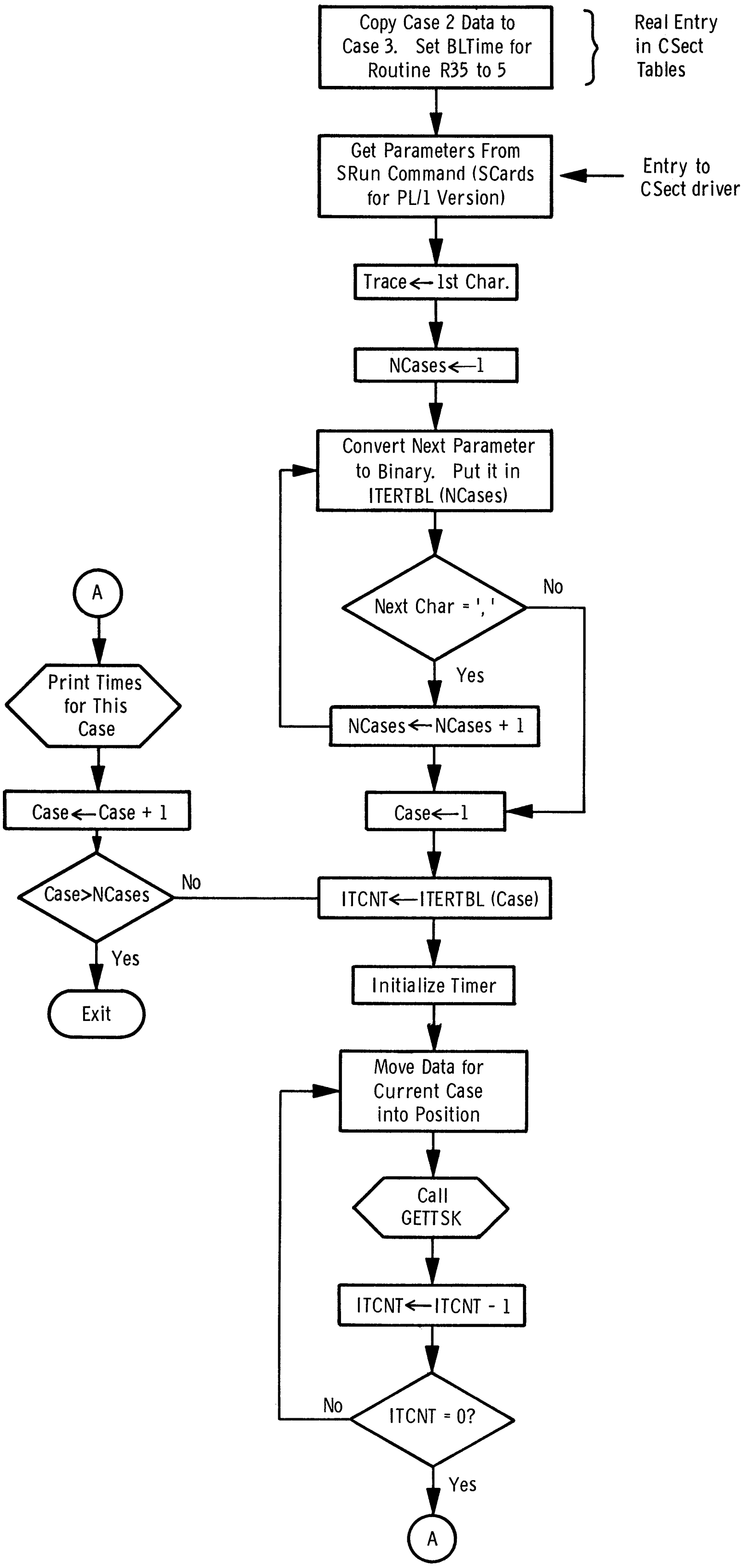
It is true, beyond a doubt, however, that the 360 has many instructions which do not correspond to any language primitives. Examples include translate and test, and branch on count. These instructions are very useful to the assembly language programmer, but useless to FORTRAN or PL/1. It generally requires several FORTRAN statements to perform the equivalent function. This is not so much an incompatibility, as the inability of the compiler to use the full 360 instruction set.

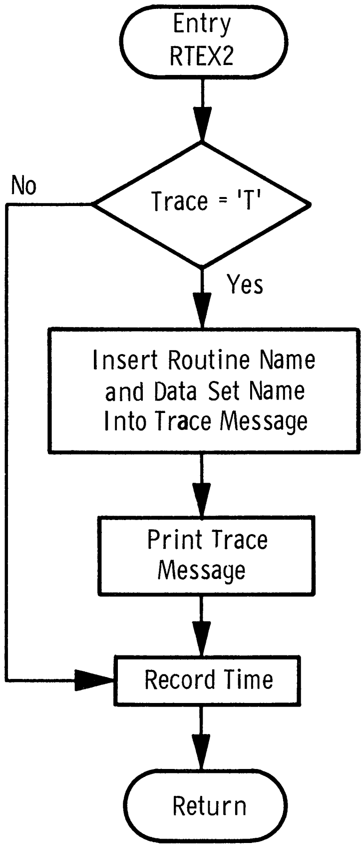
5. It is only fair to point out that the programmer is biased toward assembly language programming, and is certainly most conversant in it. The programmer is quite familiar with FORTRAN, however, and it is felt that the FORTRAN programs are reasonably effective for this reason. PL/1, however, is a different matter. While he is reasonably familiar with the facilities of that rather complex language, he is by no means an expert. To cite a specific example, one of the biggest inefficiencies in the PL/1 object code is its handling of bit strings with implicit calls on external functions, which requires creation of dope-vectors, etc. It turns out that if the bit strings are declared ALIGNED, in-line code is generated. This reduced the execution time in problem I from 11.99 msec to 8.41 mxec. It is not known whether ther are any more changes of this nature which would so significantly increase the efficiency of the PL/1 programs. One might argue, however, that if intimacy with the details of a language are required in order to use it effectively, then that makes the language that much less desirable.

APPENDIX

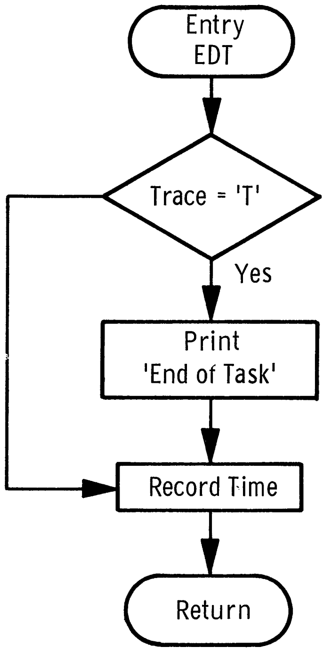
FLOWCHARTS

GET TASK DRIVER



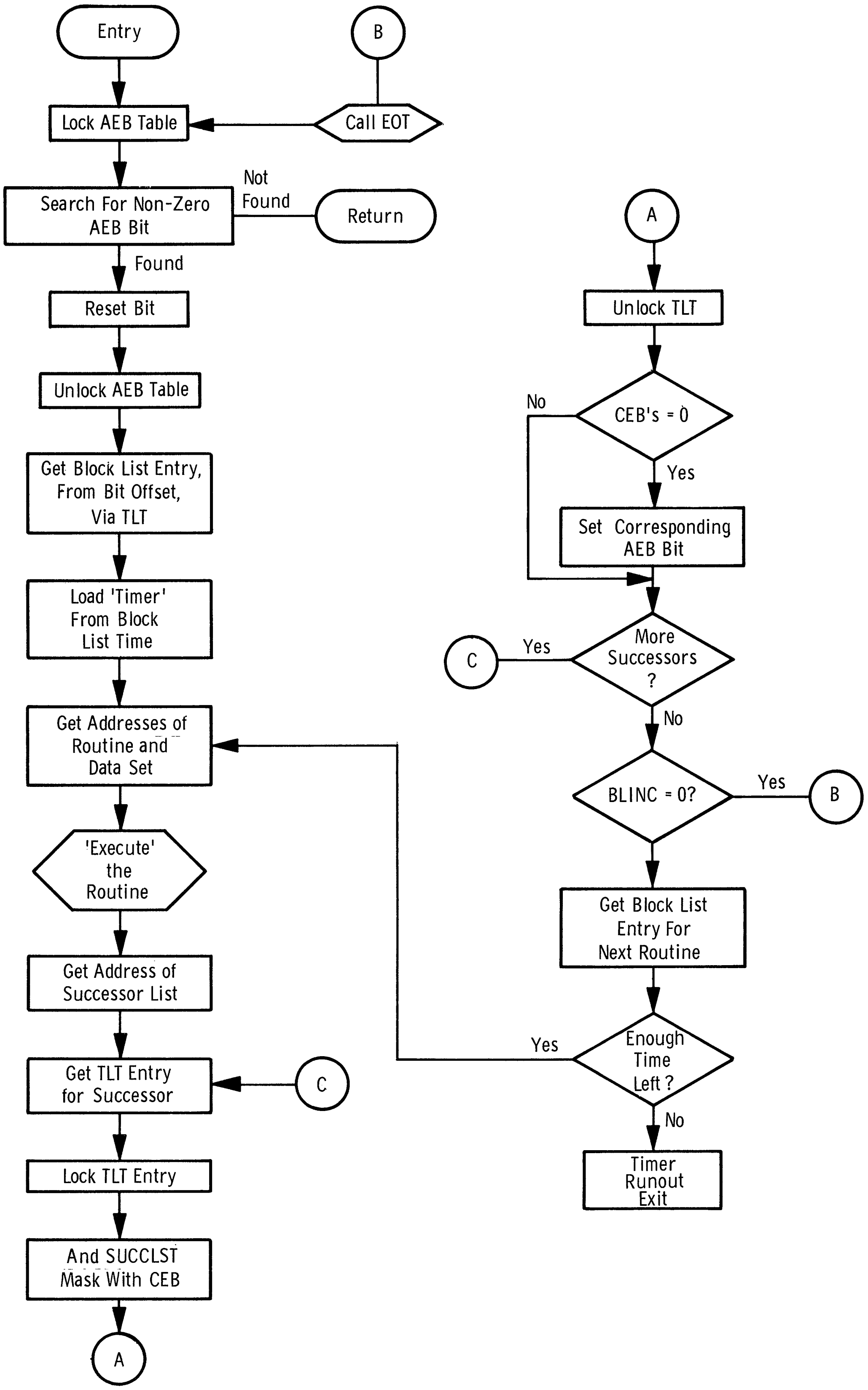


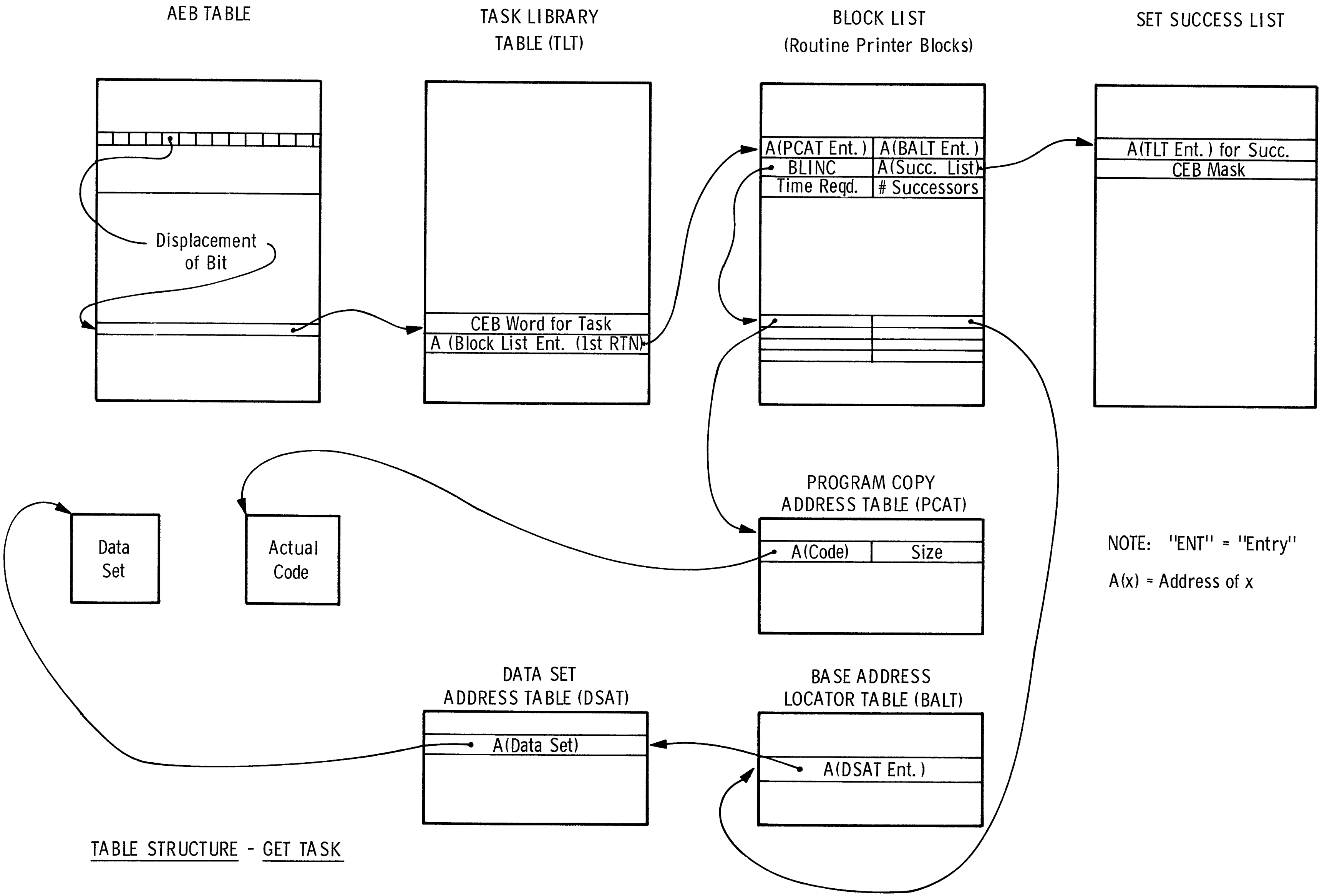
This Subroutine "Executes" a Routine



This Subroutine Records "End of Task"

GET TASK


Entry
B
Lock AEB Table
Call EOT
Search For Non-Zero AEB Bit
Not Found
Return
Found
Reset Bit
Unlock AEB Table
Get Block List Entry, From Bit Offset, Via TLT
Load 'Timer' From Block List Time
Get Addresses of Routine and Data Set
'Execute' the Routine
Get Address of Successor List
Get TLT Entry for Successor
C
Lock TLT Entry
And SUCCLST Mask With CEB
A
A
Unlock TLT
CEB's = 0
No
Yes
Set Corresponding AEB Bit
C
Yes
More Successors ?
No
BLINC = 0?
Yes
B
Get Block List Entry For Next Routine
Yes
Enough Time Left ?
No
Timer Runout Exit

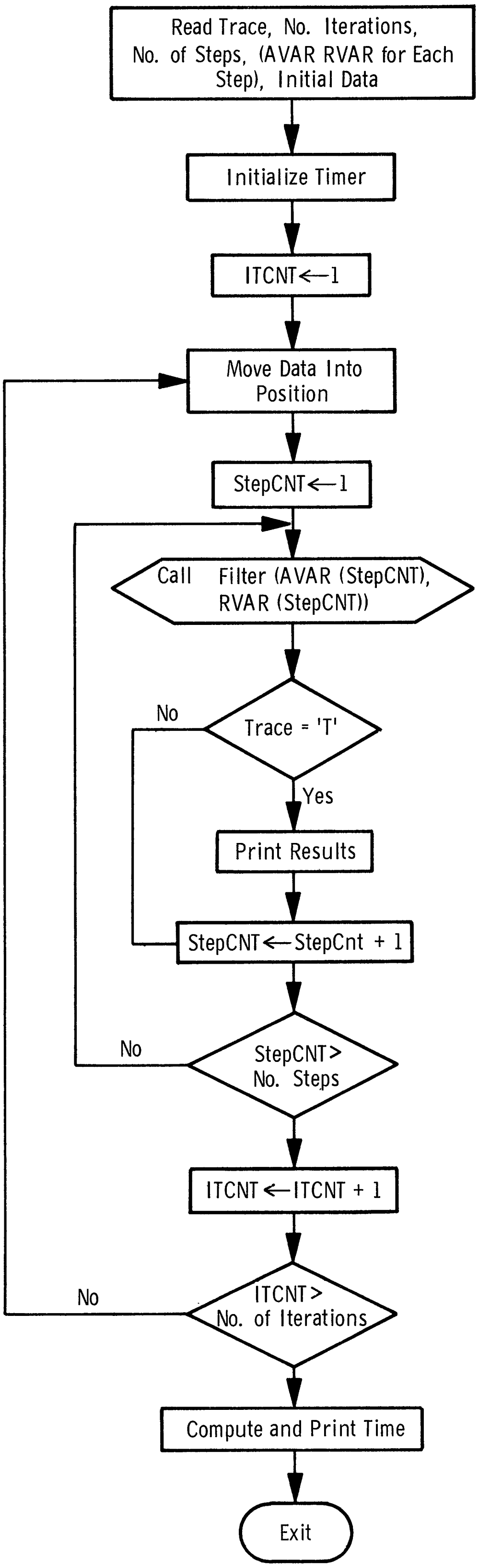


NOTE: "ENT" = "Entry"

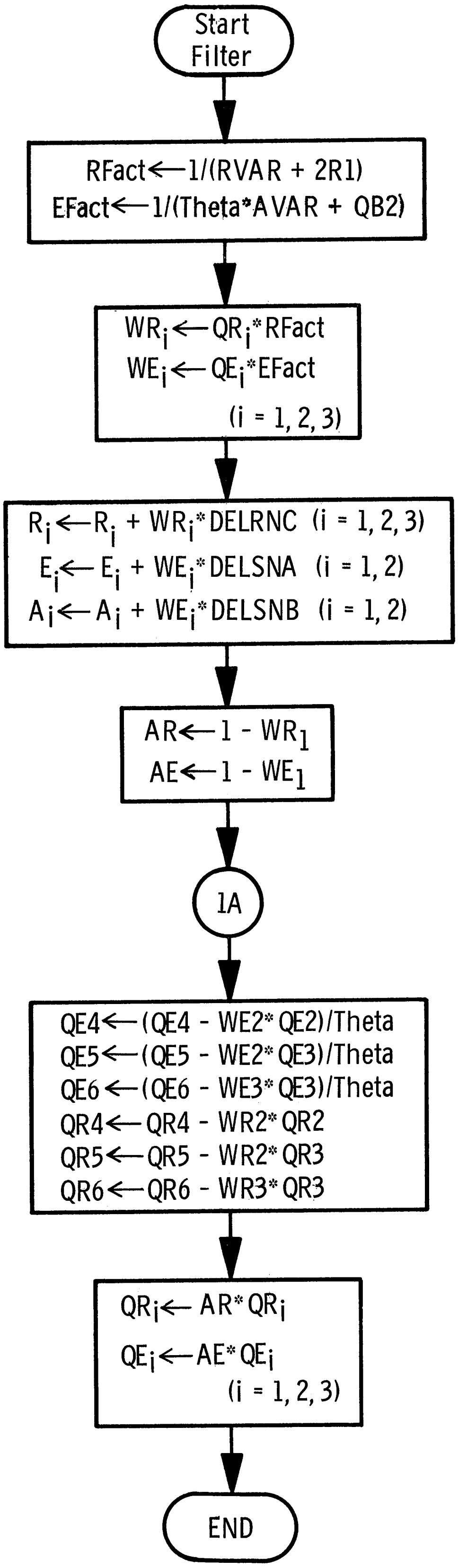
A(x) = Address of x

TABLE STRUCTURE - GET TASK

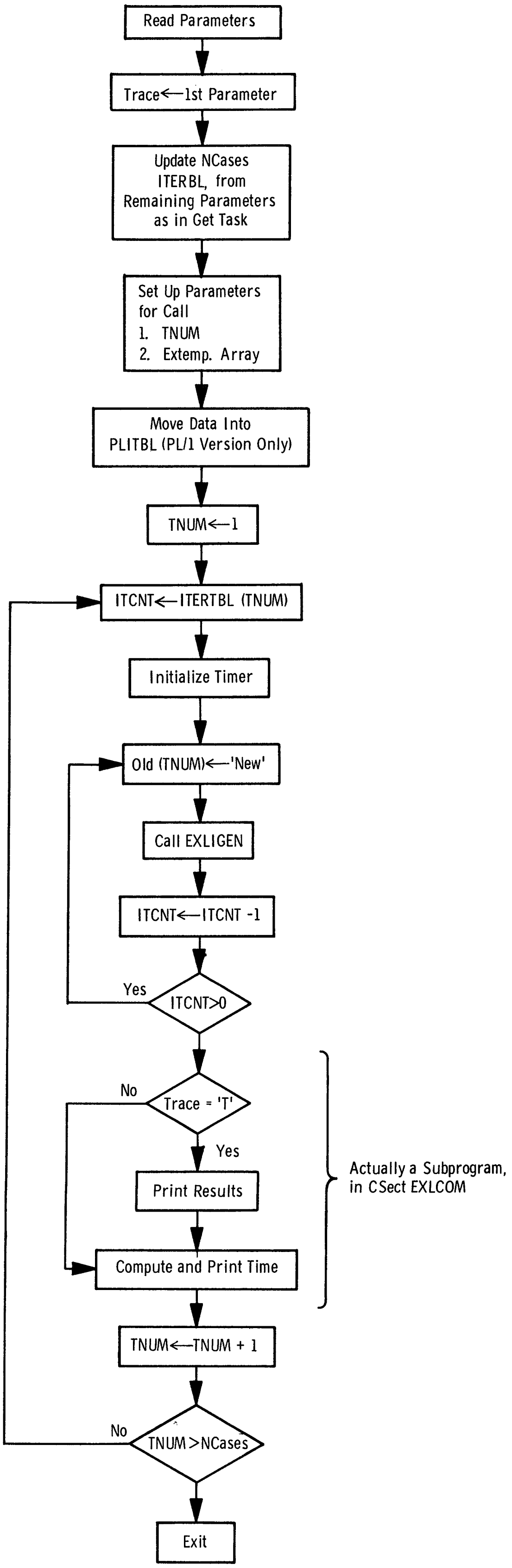
FILTER DRIVER



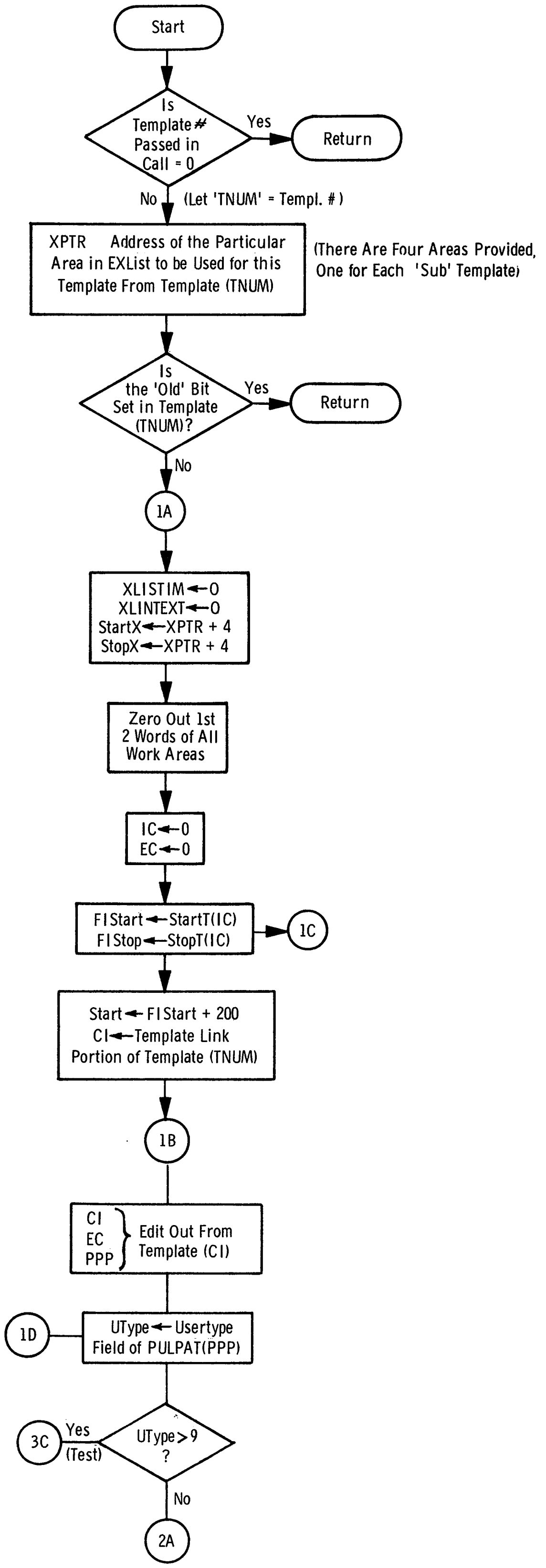
FILTER FLOWCHART

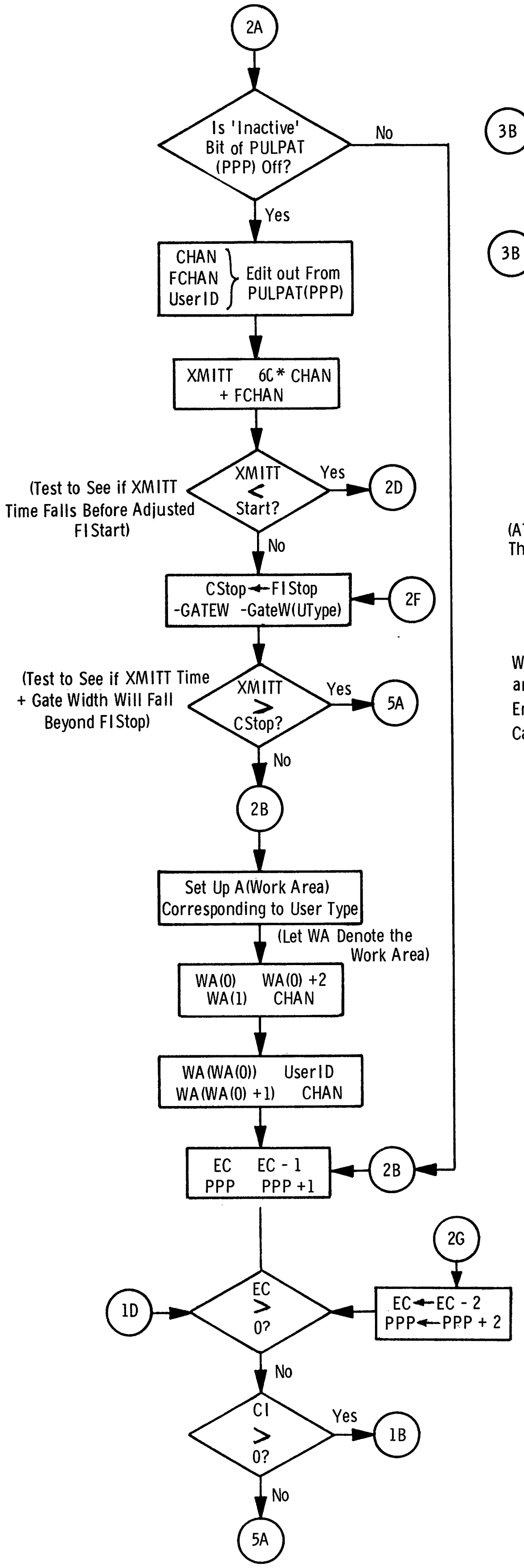


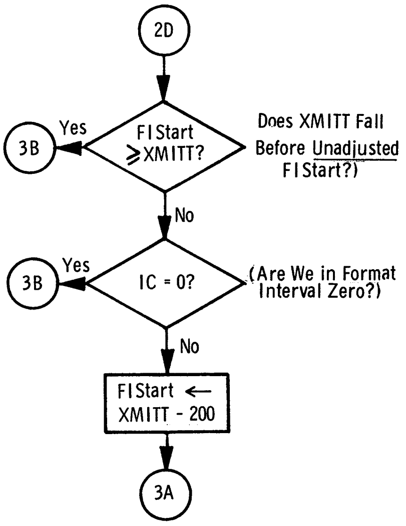
EXLIGEN DRIVER



EXECUTION LIST GENERATOR
FLOWCHART



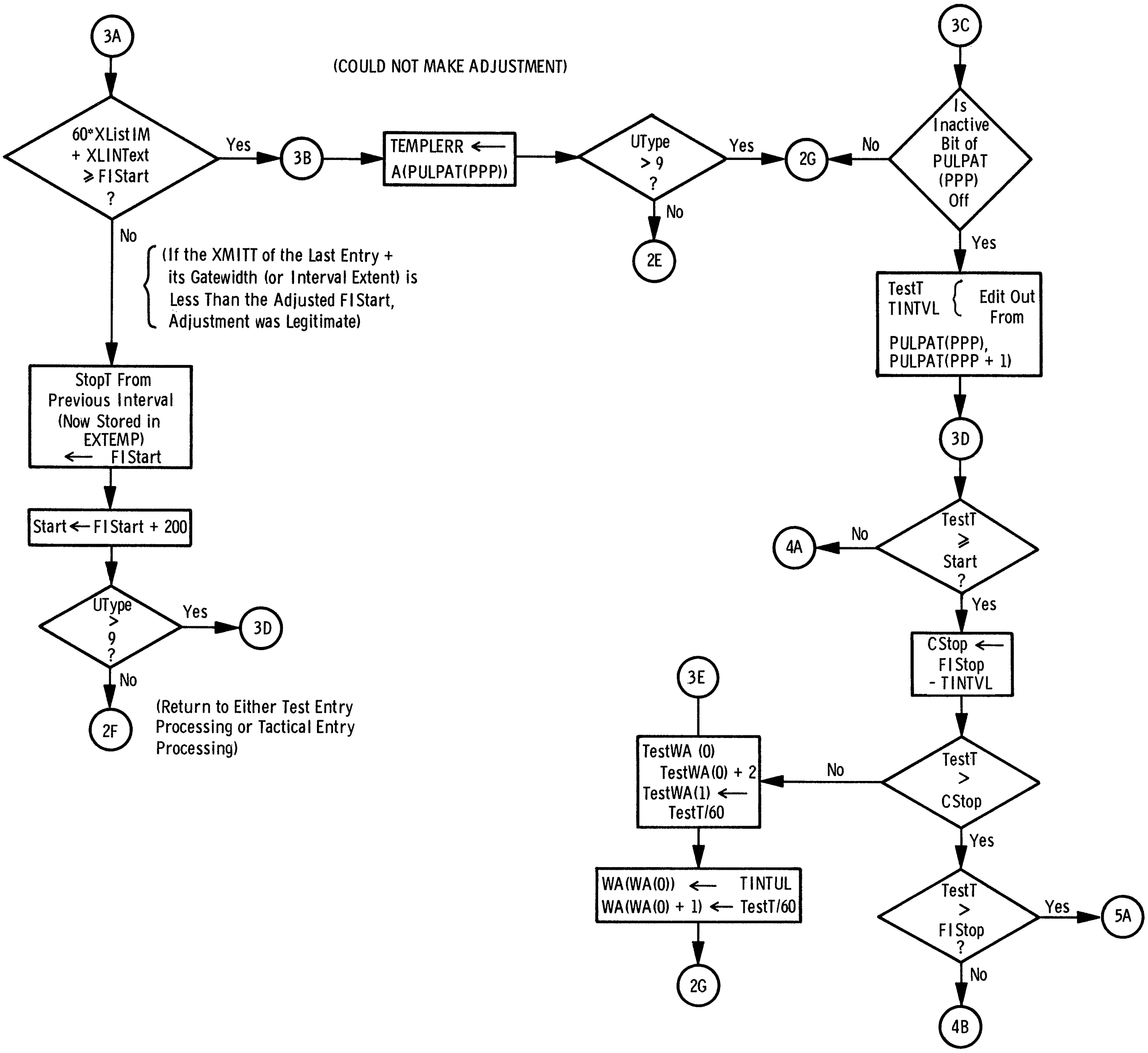




(AT This Point We Have Established That:

(1) XMITT < Start
(2) XMITT > FIStart
(3) IC > 0

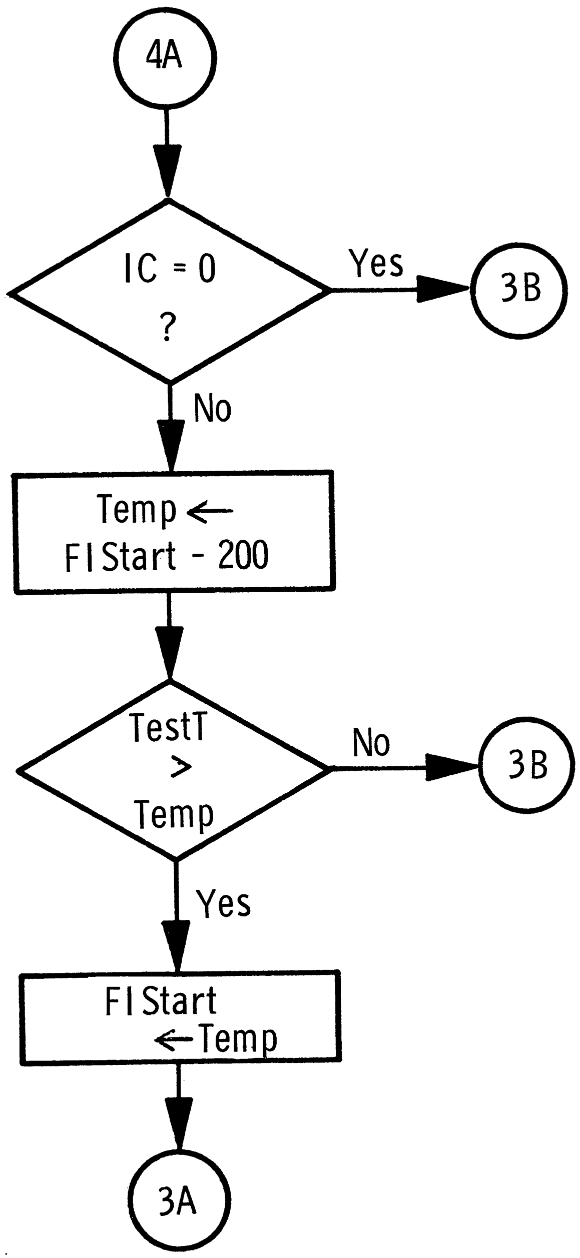
We Have "Moved Back" FIStart, and Now We Must Check the Last Entry Placed in EXList to See if We Can "Get Away" With This Adjustment.

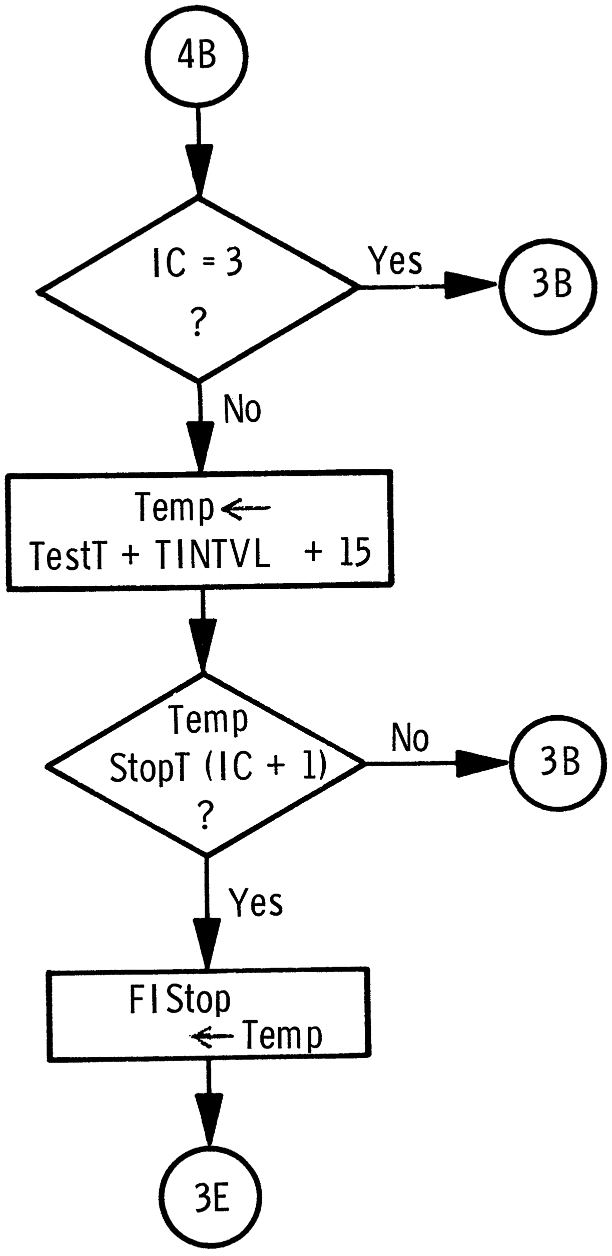


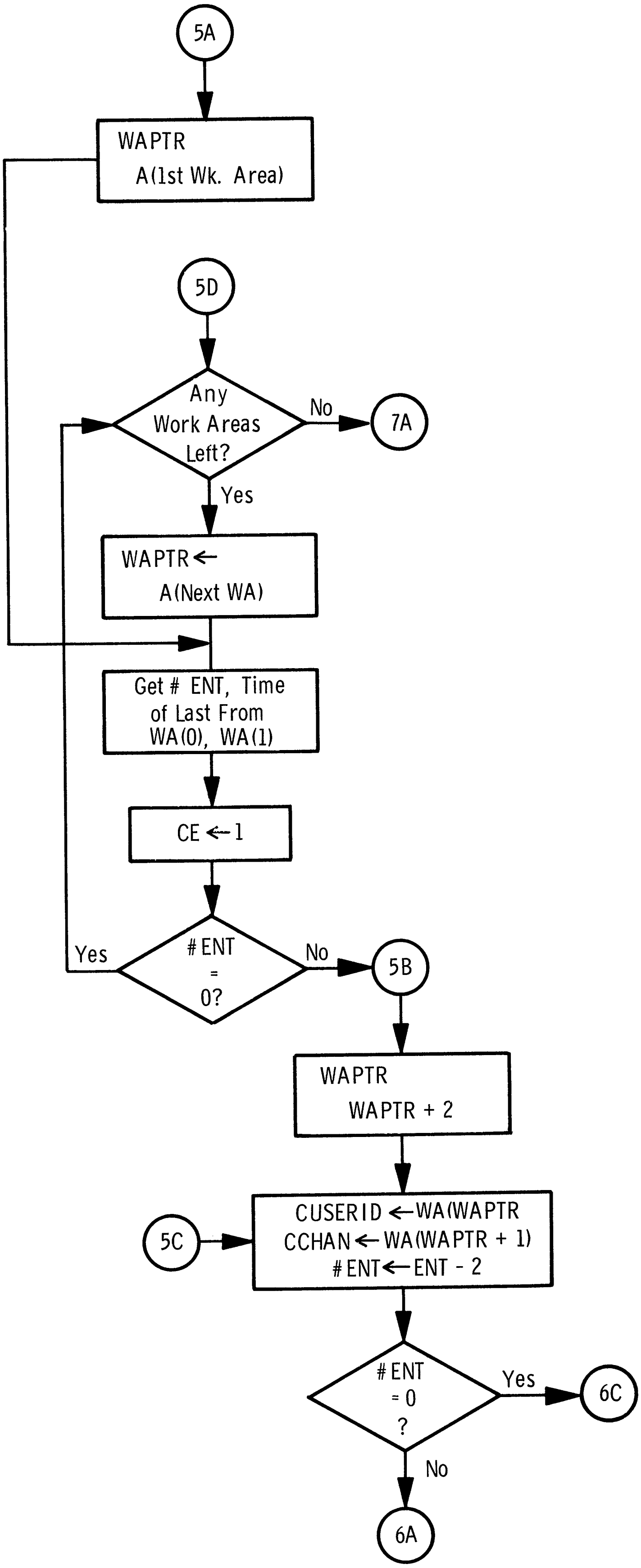
(COULD NOT MAKE ADJUSTMENT)

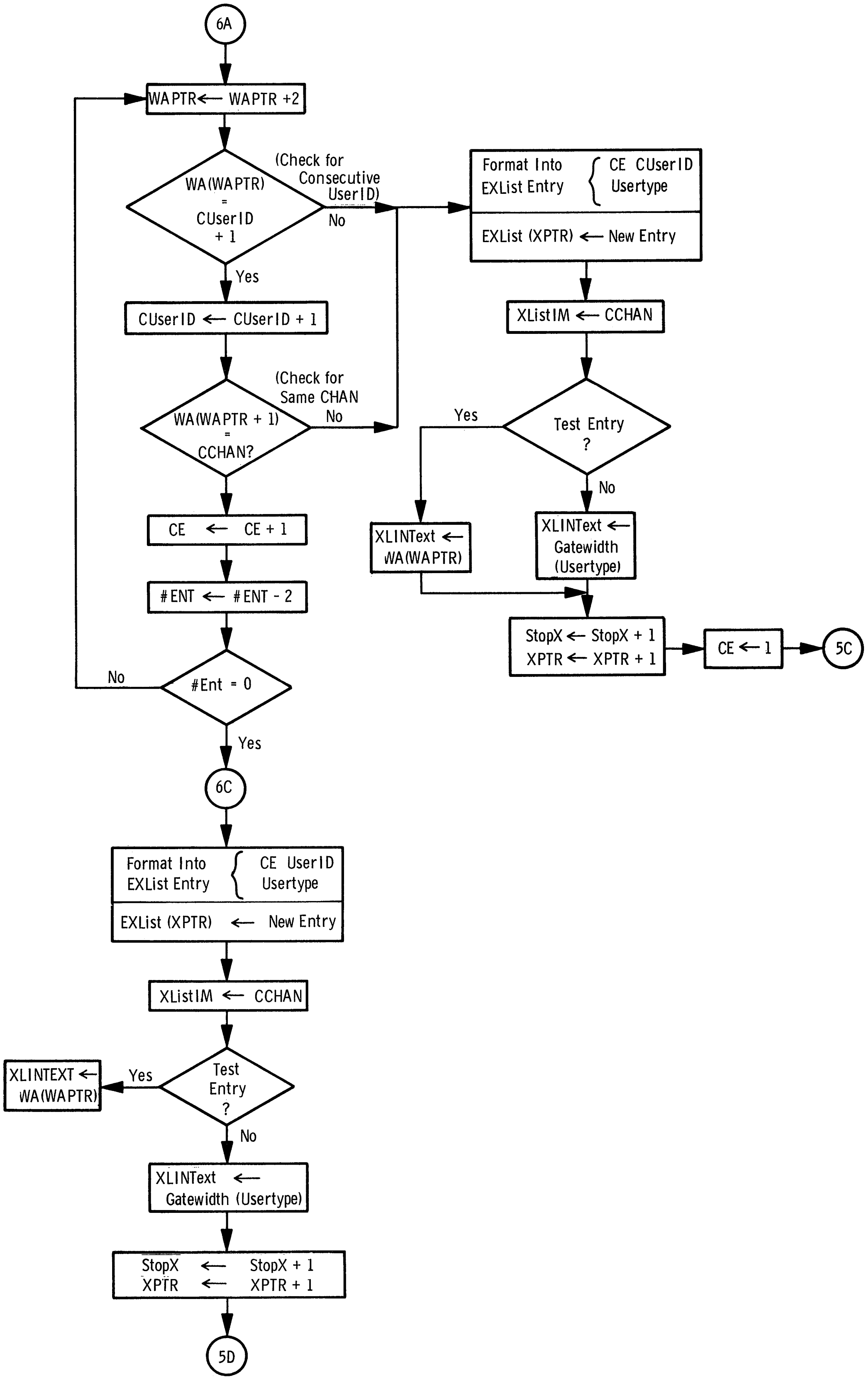
(If the XMITT of the Last Entry + its Gatewidth (or Interval Extent) is Less Than the Adjusted FIStart, Adjustment was Legitimate)

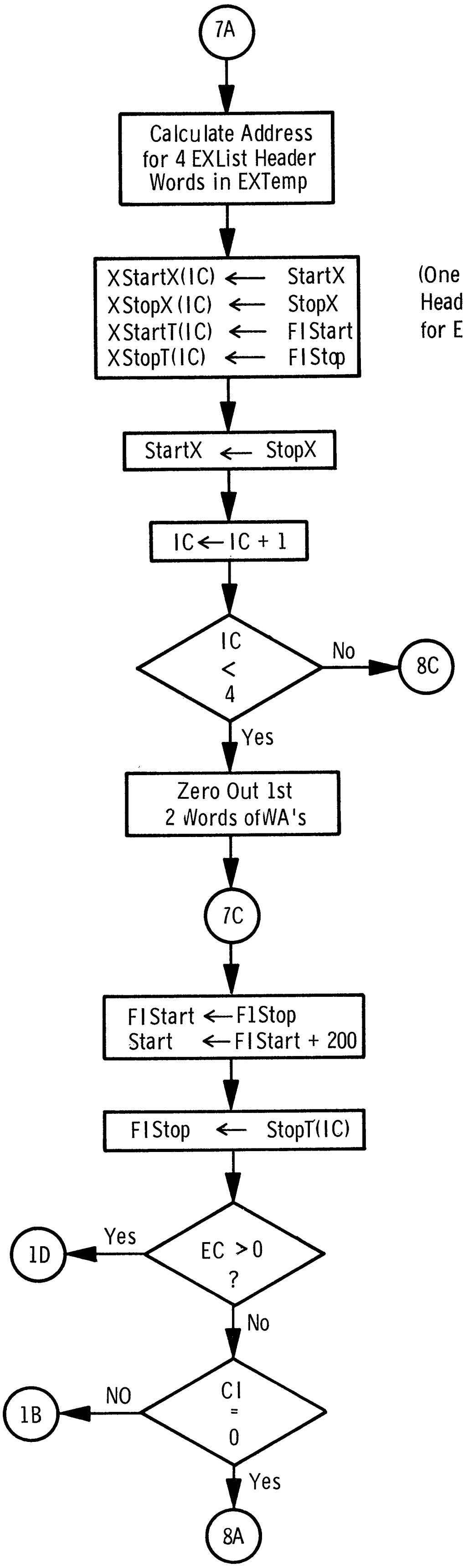
(Return to Either Test Entry Processing or Tactical Entry Processing)


4A
IC = 0
?
Yes
3B
No
Temp ←
FlStart - 200
TestT
>
Temp
No
3B
Yes
FlStart
←Temp
3A

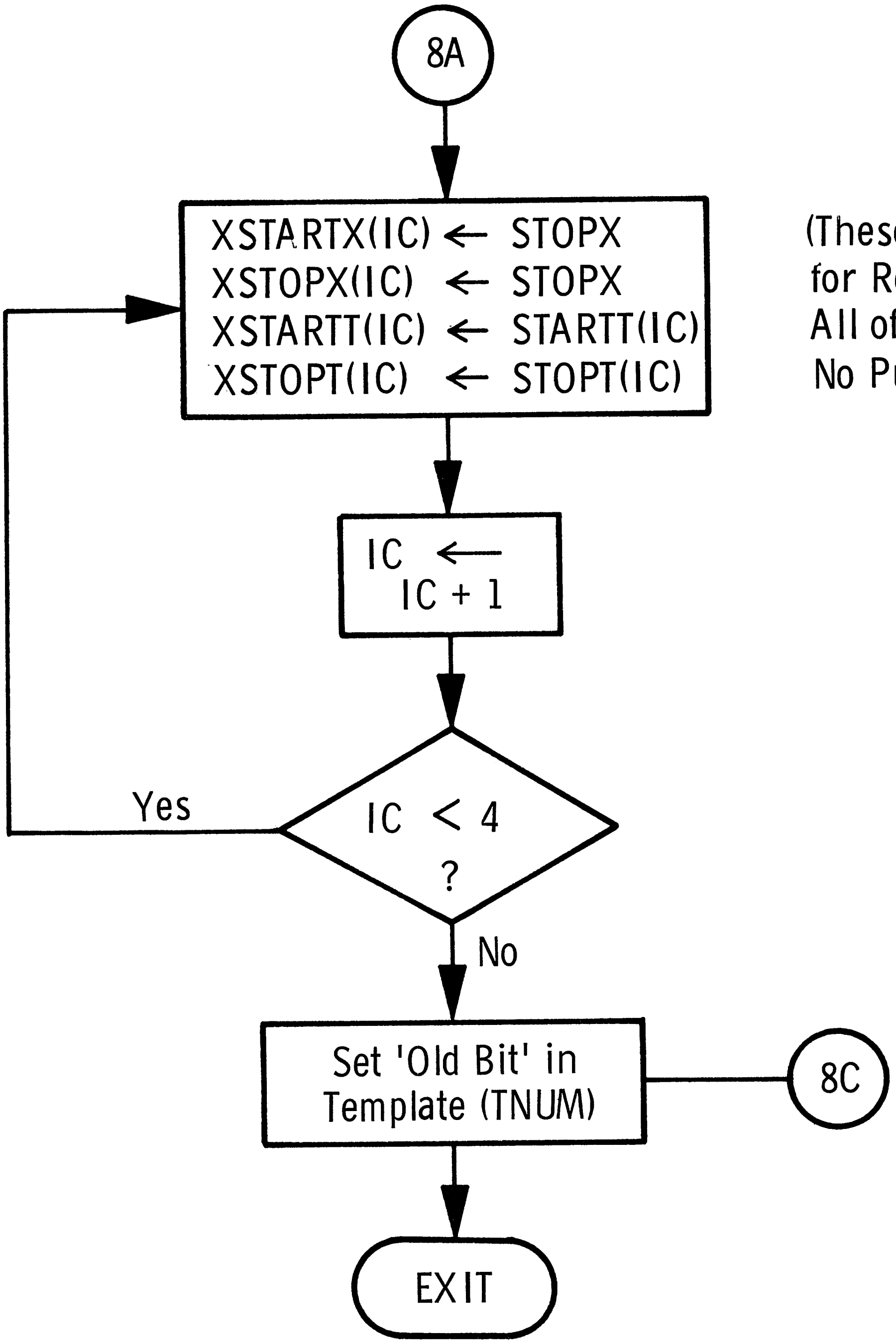

4B
IC = 3
?
Yes
3B
No
Temp ←
TestT + TINTVL + 15
Temp
StopT (IC + 1)
?
No
3B
Yes
FlStop
←Temp
3E


5A
WAPTR
A(1st Wk. Area)
5D
Any Work Areas Left?
No
7A
Yes
WAPTR ←
A(Next WA)
Get # ENT, Time of Last From WA(0), WA(1)
CE ← 1
ENT = 0?
Yes
No
5B
WAPTR
WAPTR + 2
5C
CUSERID ← WA(WAPTR
CCHAN ← WA(WAPTR + 1)
ENT ← ENT - 2
ENT = 0 ?
Yes
6C
No
6A


6A
WAPTR ← WAPTR +2
WA(WAPTR) = CUserID + 1
(Check for Consecutive UserID)
No
Yes
CUserID ← CUserID + 1
WA(WAPTR + 1) = CCHAN?
(Check for Same CHAN
No
CE ← CE + 1
#ENT ← #ENT - 2
#Ent = 0
No
Yes
6C
Format Into EXList Entry { CE CUserID Usertype
EXList (XPTR) ← New Entry
XListIM ← CCHAN
Test Entry ?
Yes
No
XLINText ← WA(WAPTR)
XLINText ← Gatewidth (Usertype)
StopX ← StopX + 1
XPTR ← XPTR + 1
CE ← 1
5C
Format Into EXList Entry { CE UserID Usertype
EXList (XPTR) ← New Entry
XListIM ← CCHAN
Test Entry ?
Yes
No
XLINTEXT ← WA(WAPTR)
XLINText ← Gatewidth (Usertype)
StopX ← StopX + 1
XPTR ← XPTR + 1
5D



(One Complete EXList Header Stored in EXTemp for Each Format Interval.)



(These entries made for Remaining Intervals. All of Which Have No Pulses)

EXECUTION LIST GENERATOR - DATA STRUCTURE