IBM

see corrections as marked

KOLSKY

IBM PALO ALTO SCIENTIFIC CENTER
ZZ20-6433, January 1977

MICROCODING APL: THE SYSTEM/370 MODEL-135 APL ASSIST

R. G. SCARBOROUGH
H. G. KOLSKY
N. S. GUSSIN

Declassified

~~IBM INTERNAL USE ONLY~~

× ≠ ∩
π ∫ Σ
∪ ∞ Δ

1974 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6426 March 1974
P. SMITH and K. PRICE — An Architectural and Design Overview of SCAMP (42 p.) IBM Confidential until March 1984, Limited Distribution

ZZ20-6427 October 1974
HARRY F. SMITH, JR. — VM/7 Virtual Memory for the S/7 (68 p.) IBM Confidential until November 1979.

1975 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6428 February 1975
A. HASSITT and L. E. LYON — The APL Assist (RPQ S00256) (115 p.) IBM Internal Use Only

ZZ20-6429 June 1975
M. J. BENISTON, R. J. CREASY, A. HASSITT, J. W. LAGESCHULTE, L. E. LYON — Writing an APL/CMS Auxiliary Processor (with complete example code) (76 p.) IBM Internal Use Only

1976 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6430 January 1976
R. A. BLAINE and H. H. WANG
An Experiment in Parallel Processing in the IBM 370/168 (14 p.) IBM Internal Use Only

ZZ20-6431 June 1976
H. J. MYERS - A Fast Assembly Technique using APL (19 p.) IBM Internal Use Only

ZZ20-6432 December 1976
A. HASSITT, L. E. LYON -
APL Syntax and APL Execution (26 p.)
IBM Internal Use Only

ZZ20-6430 January 1976
R. A. BLAINE, H. H. WANG
An Experiment in Parallel Processing on the IBM 370/168 (14 p.) IBM Internal Use Only

1977 IBM PALO ALTO SCIENTIFIC CENTER REPORTS
IBM CONFIDENTIAL AND IBM INTERNAL USE ONLY

ZZ20-6433, January 1977
R. G. SCARBOROUGH, H. G. KOLSKY, N. S. GUSSIN, Microcoding APL:
The System/370 Model-135 APL Assist (76 p.)
IBM Internal Use Only

ZZ20-6495 August 1975
Abstracts of IBM Confidential and IBM Internal Use Only Palo Alto Scientific Center Reports (43 p.)
IBM Confidential

The availability of reports is correct as of the printing date of this report.

* These reports are available only on the need to know basis, please contact the Scientific Center for information on copies.

• Copies are no longer available from the Scientific Center.

IBM PALO ALTO SCIENTIFIC CENTER TECHNICAL REPORT NO. ZZ20-6433

JANUARY 1977

MICROCODING APL: The System/370 Model-135 APL Assist

Randolph G. Scarborough
Harwood G. Kolsky
Norman S. Gussin

IBM Palo Alto Scientific Center
P. O. Box 10500
Palo Alto, California, 94304

~~IBM INTERNAL USE ONLY~~

ABSTRACT

The APL Assist is a hardware feature which enhances the performance of APL systems by executing a major subset of the APL language directly in microcode. The APL135 Assist is now available on the IBM System/370 Models 135 and 138. It is invoked by a new System/370 instruction called "APLEC". The feature can be used under standard operating systems.

This report describes the details of the APL135 microcode, emphasizing the importance of executors and index tables as a technique that yields high performance code that is easy to understand and maintain. The report also describes the interactive development system designed for this project and contrasts it with the original Mod 135 batch system. The report also discusses some of the lessons learned from the experience of microprogramming on these systems. Perhaps the conclusions and recommendations can save future microcoding groups some time and trouble.

Terms for the IBM Subject Index:

Microprogramming
Machine Language
APL
Performance
Program development
IBM System/370 Model 135

07 - Computing
21 - Programming

IBM INTERNAL USE ONLY

TABLE OF CONTENTS

I. Introduction..1
   A. Objectives of the Project.........................1
   B. Constraints and Interfaces........................2

II. The Original Model 135 Batch System..................4

III. The Interactive Development System..................7

IV. The Importance of Interactive Tools..................9
   A. MPL135 Compiler..................................11
   B. ASM135 Assembler.................................12
   C. SIM135 Simulator.................................13

V. Lessons learned from Experience......................15
   A. Version One and why it was Unsuccessful..........15
   B. Version Two--The Native Mode Approach............17

VI. Overview of the Microcode Emulator..................19
   A. Executors and Index Tables.......................19
   B. Comparison between Mod 135 and Mod 145...........20
   C. Other Programming Techniques.....................23
   D. Documentation....................................24

VII. Detailed Description of APL135 Microcode...........25
   A. Organization of APL135...........................25
   B. Scanning and Execution of APL Programs...........30
   C. Execution of Dyadic Operators....................32
   D. Dyadic Analysis and Control......................33
   E. Dyadic Scalar Execution..........................35
   F. Execution of Monadic Operations..................36
   G. Workspace Resource Management....................36
   H. Management of Variables..........................39
   I. Control of APL Functions.........................42
   J. Indexing Operations..............................44

VIII. Conclusions and Recommendations...................51
   A. Conclusions......................................51
   B. Hardware Recommendations.........................53
   C. Recommendations concerning the Assembler.........55

Table of Contents

Appendix I. The Mod 135 Microcode Simulator (SIM135)....57

Appendix II. Assembler for Mod 135 Microcode (ASM135)....67

Glossary..73

References..76

FIGURES AND TABLES

Figure 1: The Original Batch Microcode System.............5
Figure 2: The Code-and-Test Optimizing System...........10

Table 1: Names of the 52 Microcode CSECTS...............27
Table 2: Organizaton of the Hardware Registers..........28
Table 3: Flags used in the Indexing Routines............47

I. INTRODUCTION

A. Objectives of the Project.

The System 370 Models 138 and 148 have recently been announced and are the leading edge of IBM's intermediate product line. Their architectures contain the most advanced high level language processors on the market. Both of these models, and the Models 135 and 145 from which they came, now have a special "assist" feature to interpret and execute APL programs directly in microcode. This makes them "APL machines" in the same sense that they are "System 370" machines."

This report is an attempt to record some of the experiences, results, and observations concerning the microcode development process. Although it is based mainly on work with the Model 135, the conclusions may have a generality beyond that specific implementation. In addition to the technical details of the APL/135 project, there is recorded here how the project was done and some observations on how such a project should be done.

The original goals of the project were threefold:

(1) To study the problems of microprogramming in the absence of a real target machine.
(2) To develop what promised to be a superior interactive system for the development and testing of microcode.
(3) To bring forth a useful microcoded system to support APL.

The latter goal forced the requirement that standards be established early so that the resulting programs would be compatible with agreed-to interfaces.

There are good reasons to study the experience of microprogramming for its own sake. Microprogrammed computers have become one of the major new thrusts in computer science. It seems obvious that microprogramming will dominate the field of computer architecture and systems design for the next five or ten years.

Moreover, many of the future products being considered by IBM include a heavy emphasis on higher level functions and higher level languages supported by microprogrammed

"engines". One goal of the project was to examine the feasibility of designing, writing, and testing a complex microcoded system in the absence of real target hardware. Such a feasibility has been a major assumption behind much of the future systems work in recent years.

At the beginning of this project, the APL Assist for the Model 145 was under way at the Palo Alto Scientific Center. An immediate demand for a similar feature on the Model 135 was anticipated as soon as the Model 145 APL Assist was available. There would be an advantage if the two programs could be presented for forecasting, development, and marketing as a pair of compatible products. Another important consideration was the fact that the Model 145 group was producing the APL/CMS software product as a part of its effort. Much of it could be used if the APL/135 and APL/145 were made congruent. Finally, some understanding of the general problem of microprogramming a high-level function into a family of future systems could be studied exactly by attempting it on a subset of the current System 370 series.

Much of the APL/135 project work was based upon the earlier work of Tony Hassitt and Len Lyon.(See the References) They had created much of the context for this project. They had solved already the very hard problems of designing and proving an effective APL system. They had worked out the complex interactions between external specifications and internal microprogramming. However, additional problems occured in implementing APL/135 which were not anticipated from the APL/145 work. Since these may be of general interest, they are described in detail below.

The APL/135 project achieved its major objective with the announcement of the APL Assist for the Model 135 on December 3, 1975. This feature is externally congruent with the APL Assist for the Model 145. Because of differences in the internal micro-processors of the two machines, however, the structure and philosophy of the microprograms are radically different. The Model 135 is more modular and more structured and is simpler to maintain and extend. These differences are described in Sections V and VI.

B. Constraints and Interfaces.

The work on the APL/135 microcode was influenced by many factors external to the microcode. These affected the

organization and productivity of the project and the quality and accuracy of the product. There were five main constraints:

(1) A small staff.
(2) There was no Model 135 available nearby.
(3) The program had to be delivered in a standard, rigid format to the customer in SPD.
(4) The control storage allocation was too small to contain all the functions of APL.
(5) The microprogram had to be partitioned into segments to co-reside with any valid Model 135 configuration that might be marketed.

The first three of these constrants affected the organization and control of the project. The unavailability of a local Model 135 reinforced the goal of studying the development process in the absence of real hardware. The latter two constraints were much more serious. They at least doubled the effort required.

Microcode is intricate and internal communication is tricky. The structure of the internal interconnections dominates the writing of the code. The interfaces between microcode sequences are more important in the outcome than the instructions within those sequences which actually execute the useful work for the user. The interfaces must control the structure of the productive code, and yet the interface conventions must be selected to optimize the productive code on both sides. If performance is a goal then coding tricks are mandatory. A microprogram without tricks is either too large or too slow.

The experience of this project leads to the conclusion that a prime factor in the design of a program or an architecture should be the ability of people to implement it. The design should conform to the way people want or need to work, not the inverse. In the inverse situation a design is obtained which has a logical, attractive structure, it is subdivided as necessary for implementation, and programmers are assigned to the pieces.

In this case the opposite approach was used. Each person had skills and was interested in doing a particular kind of work. These skills and interests were merged until a design was found that could be implemented. The project was thereby designed to fit the people, and each person had coherent work.

II. The Original Model 135 Batch Microcode System.

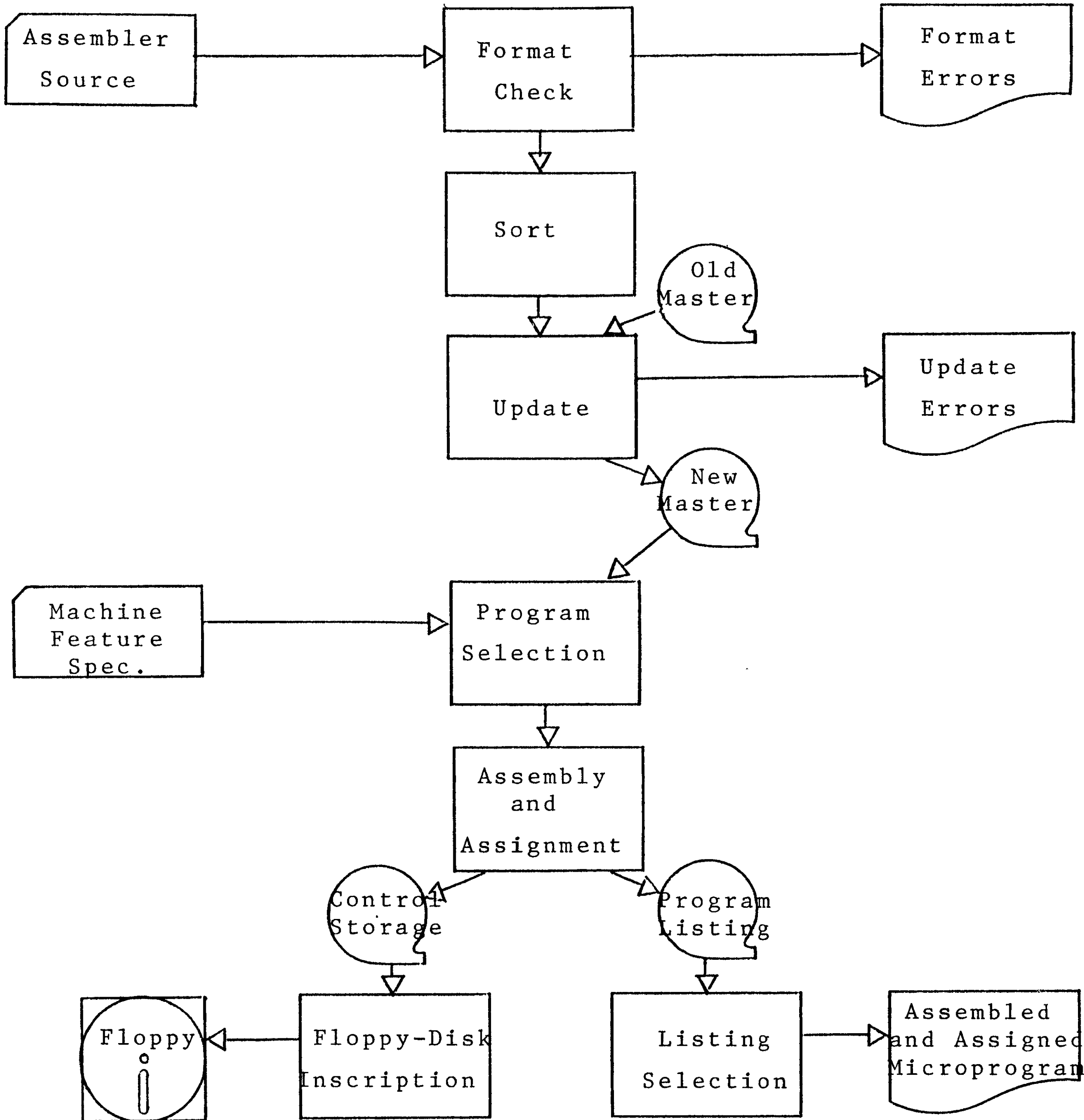
At the beginning of the effort a copy of the microcode production system, APSS, used in developing and manufacturing the 3135 was acquired. Although pieces of it, heavily modified, still remain, nonetheless the philosophy with which they are accessed is radically different.

Figure 1 presents an overview of the standard APSS system. The system, is built around the "microprogram master tape" (MMT) which contains serially by control section all the microprogramming available for the Model 135. It is oriented toward the production of floppy disks containing control-storage contents for customized machine feature configurations. The coding and testing of microprograms was found to be quite difficult under this system.

In order to code a program change, the programmer must generate both update control statements and revised assembly language source statements. He must sequence each altered statement with its eight digit sequence number. If he inserts more than nine cards between two unchanged statements he must repunch and reenter one of these as well. APSS checks each statement individually for format and syntax. If all are correct then they are sorted by sequence field, skeletally assembled, and merged onto the new-master MMT. The revised modules are resequenced during this process. The programmer must check the format-check output to locate syntax and format errors and the merge output to locate sequence (insert, update, delete) errors. Because of resequencing the numbers he must find in the listing are not necessarily the numbers he punched on the cards. The "new master valid" message comes after fifty pages of documentary listings that the system can not suppress.

Next, in order to install a program change, the programmer specifies a Model 135 microcode configuration which contains both the microprogramming in which he is interested and all the code necessary to operate the real machine configuration on which he will be working. This latter category includes the microcode for the System-370 instruction set, the channels, and the native devices and adapters. The required microcode is selected from the MMT and assembled. This produces a listing tape which must be selectively printed (the entire listing is likely to be

FIGURE 1: The Original Batch Microcode System.



eight inches thick) and a tape from which may be made a floppy disk for IMPL-ing the hardware. Alternatively the hardware may be pseudo-IMPL-ed with a program which loads control store from the listing tape. This often requires a patch to control store at the termination of the program. Some program errors, such as omitted or duplicate labels or writing too much code to fit within a section of control store, are detected at this stage.

Finally, in order to test the program change, the programmer must obtain time on a dedicated Model 135 and IMPL it with one of the procedures mentioned above. He then may debug, single-cycling and hard-stopping as necessary, from the console. When he finds a bug he must mentally assemble a corrected instruction (no trivial task), manually enter it, and continue. If, as often happens, the patch will not fit in the space of existing code, a branch to a spare section of storage must be inserted, the patch must be coded there, and an additional branch back to the main line must be grafted at the end. It is probable, from the experience of this effort, that one in three or four of these hand changes will be wrong.

In summary, the problems listed below make the standard system inconvenient for experimental development:

(1) All work must be done in batch mode OS job steps. It takes five job steps to make a single test run.

(2) All program changes are made with control and update cards. Not only must the programmer know what he wants to change, but he must express his intent with specification and sequence information which does not directly relate to his intent.

(3) Three separate listings must be examined to ascertain the successful transcription of a program change. In two of these there is large amount of extraneous material printed.

(4) An entire control-storage contents must be selected and assembled to test even the smallest subroutine. The basic machine microcode (central processor and channels) must be functional to test any additional microcode. A real Model 135 must be available and congruent with the selected microcode.

(5) It is difficult to debug microcode on a Model 135. The alter display console functions, for example, do not operate when one needs them in the middle of the tested microprogram. Switches and lights must be used to access main storage, control storage, and registers. Virtual address translation must be done by hand. This involves a control-store access for the segment table origin register and main-store accesses for the segment-table and page-table lookups--all through a single set of switches and lights. Only one facility--address or register--may be monitored at a time, also through that same set of switches and lights.

(6) If a microprogram bug occurs it may destroy the integrity of the machine, necessitating either a real-IMPL or a real-IMPL followed by a pseudo-IMPL followed by a hand-patch IMPL, then an IPL, then the setup of the software test program, and finally the repatching of bugs found up to the point of failure.

(7) A large System 370 is needed preceding the testing to support all the software used to generate the microcode. One needs an OS/VS1 machine with three disk drives and three tape drives. One must be an OS/VS1 operator and IPL the three-drive system. A minimum of three tapes must be mounted during each iteration of the program-changing sequence described earlier. If all this is done on the same real Model 135 as the testing, then one spends ones time in seemingly endless IMPL-ing, IPL-ing, reconfiguring, reallocating, etc.

III. The Interactive Development System.

Study of the standard manufacturing system described in the last section showed that it was necessary to build a system that was more optimized for coding and testing.

Unlike the microcode for the System-370 instruction set, which consists of many short-running nonconnected noninteracting sections, the segments of the APL code would be long-running, interconnected, and interacting. The codes are roughly equivalent in size--15500 bytes for System 370 and 13100 bytes for APL. However on a typical call to the System-370 microcode perhaps only ten microinstructions are executed before the task is complete. (Some instructions, of course, such as floating-point arithmetic and PSW-swapping, take many more.) In all cases there is always a single entry and a single exit defined in architecture,

only a small part of the 15500 bytes will be executed, and no continuity is required across these separate invocations. With the APL interpreter, however, there is no need for the microcode ever to finish, and a single call to the code can pass through a large fraction of the 13100 bytes.

The system to optimize the code-and-test cycle was built on the philosophy of using interactive computation wherever possible. All components are normally executed from the VM/CMS terminal. Source program coding and object program testing--the end points of the cycle--are interactive. Compiling and assembling are invoked interactively and complete automatically. No system operator intervention, and no tapes, are required. The entire system, VM and all, fits on a single disk pack. The time from the beginning of IPL to useful user work is less than thirty seconds. All components of the code-and-test cycle may then be accessed without additional IPL's.

Figure 2 gives an overview of this code-and-test optimized system. It has two basic sections. First is source program coding and changing. Programming is done in a higher-level symbolic microprogramming language, MPL135. (See Walters and McNabb in the references.)

The VM/CMS editor is used to enter and alter programs. Each routine is maintained as a separate CMS file. Using the editor, changes are made directly to the source rather than indirectly via update control statements. After editing, the routine is compiled to produce a listing of the compiler source with errors, if any, and a deck with the assembly-language source program equivalent to the MPL135 program. This ASM135 deck is also preserved individually as a CMS file, so that there is one ASM135 deck for each MPL135 deck in the system. (See Section IV-B and Appendix II).

At this stage all format-check errors have been caught and all omitted and duplicate labels within the routine have been detected--something not possible on the original system until the final assembly. The only errors remaining are either program logic errors or control-storage assignment errors (such as "too much code for core").

The second section of the system is program testing. To prepare for a test, the control sections wanted are named in an exec procedure. The ASM135 decks are then assembled to produce an object deck which may be executed by the simulator. The assembly includes a format check, identical

in function to that in the original system. The format check is still necessary because an internal translation of each statement is constructed for the assembly. On occasion coding may be done directly in assembler source rather than compiler source. A listing of only the relevant programs is produced, since they are the only ones assembled.

To execute the test a hardware simulator, SIM135, is used which is extremely interactive or extremely automatic as desired. (See Section IV-C and Appendix I). This proved to be the most valuable component of the system.

The object deck output by ASM135 and input for SIM135 is a deck in which each card contains, in EBCDIC characters, a pair of hexadecimal numbers which are the control storage address and its contents. It may therefore be easily patched over several testing sessions.

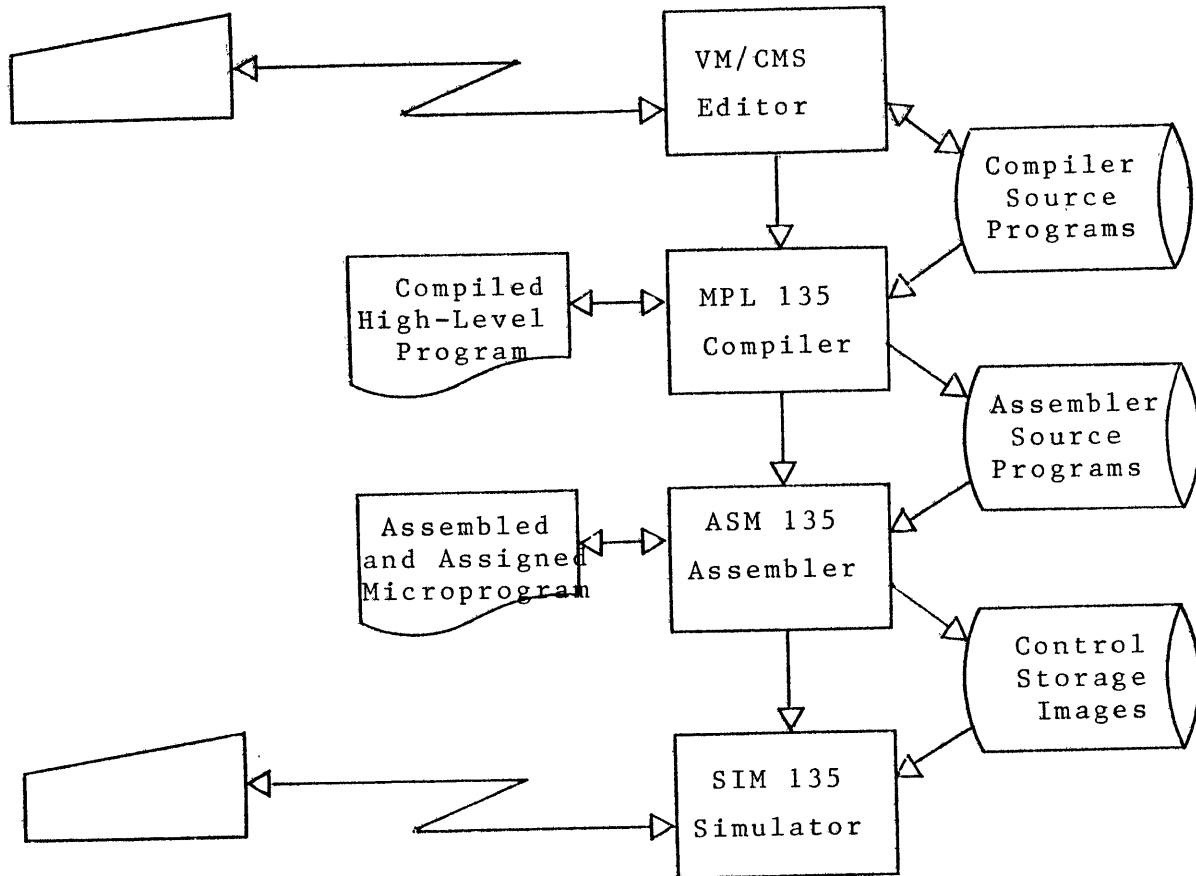
The VM/CMS system continues to provide all its normal support during simulation. The simulator merely executes the microcode presented to it on request from the terminal. Therefore the real machine hardware and software integrity is never endangered by a microprogram catastrophe in test code.

IV. The Importance of Interactive Microprogramming Tools.

Three major programs, mentioned in Section III, were produced to form the code-and-test system: MPL135, ASM135, and SIM135. These support programs made it feasible for a small group to write the microcode. It also vastly improved the quality and accuracy of the final product. If there is one major lesson about microprogramming learned from this project, it is that these support programs are essential.

A microcode compiler, MPL135, was written so that structured and symbolic programming could be used for microcode. A hardware simulator, SIM135, was written not only to duplicate the operation of a real Model 135, but more important, to aid in diagnosing and debugging the executing microcode. This was the most valuable support program. It is in fact far easier to debug microcode on this simulator than on a real machine.

FIGURE 2: The Code-and-Test Optimized System.



A. MPL135 Compiler.

The assembly language for Model 135 microprogramming is complex, not only because it must mirror the myriad features and latches and data paths in the hardware but also because it is itself very low-level and format sensitive. The job would be much simpler if one could do most of the program writing in a higher level language which possessed (1) the power of structured programming technology, (2) had the flexibility and simplicity of the symbolic programming, and (3) was free of burden of card-column fixed-format fields. For these reasons, the special syntax-driven compiler program, MPL135, was prepared for the project.

The MPL135 compiler accepts source programs written in its own high-level language and compiles them into equivalent assembly-language statements acceptable to the standard Model 135 microprogramming system. During most of the effort, however, the virtual assembly-language source decks produced by the compiler were merely a convenient internal staging point.

The compiler provided structured, symbolic, free-form programming. The MPL135 language supports structure statements like "DO", "DO-COUNT", "DO-WHILE", "DO-CASE", "IF-THEN", and "IF-THEN-ELSE". These may be arbitrarily nested. Further, MPL135 permits the coder to provide symbolic names for the registers and latches in the machine (and to provide indirect-address specifications in the symbolic names). A single mnemonic may thereby replace several parsing-oriented tokens. Finally, the absence of the fixed-field card-column assignments enables one to indent the source program to emphasize its logical structure.

An example of the MPL135 language is the following:

```
IF RDESC(B 9)=1 THEN GOTO QUITS
XVALU=XVALU+LVALU
```

The compiler produces the following ASM135 source:

```
QUITS BNZ)W6.1.1
W0-3=W0-3+GPR(W4D0)DW
```

This example illustrates a necessary restriction on the compiler: The generated assembler statements must be completely predictable. The MPL135 compiler does not try to optimize. It does bookkeeping, allows one to write in a logical way, and translates the statements one-for-one into assembly-language statements.

Several features of the MPL135 compiler were very helpful. The logic of IF-THEN-ELSE was inherent in all the programming. The assembler requires otherwise useless labels to support the branching implicit in these structure statements. MPL135 does all this involuted labelwork automatically.

Symbolic names were also very helpful. Writing "LNAME" or "RVALU" for the left-operand name or the right-operand value is simpler and conveys more information than "GPR(W4D2)" or "FPR(W4D0)DW"; "W03 = W03 + LVALU" is better than "W0-3=W0-3+FPR(W4D0)". The free-form source to support the symbolic structure was also valuable.

B. ASM135 Assembler.

The second of the three major software tools was the ASM135 assembler. This assembler combines several functions which are separate in the original system described in Section II.

The ASM135 assembler provides all of its functions in much the same way as a normal system assembler and linkage editor. It is fed assembler source subroutines which are mapped into a resulting control storage image. Each statement is format checked, and the totality is then assigned to addresses in control storage. The output is a deck, akin to an object deck, containing control storage addresses and contents for each instruction or data item in the input source. No linkage editing is required thereafter.

More accurately, no linkage editing is possible. For one reason, the microinstructions for the Model 135 often include nonrelocatable address fields. To completely assemble the instruction, therefore, its actual address in the machine must be assigned. Additionally, the programmer may specify that certain instructions must be assigned at specific addresses, within a specific range, or with specific low-order or high-order address bits.

A further constraint is the limited range of many of the branch instructions. These limits imply not only that the code containing these must be written in small blocks without branches across blocks, but also that the instructions within a block must be assembled into locations which lie within the range of the branch. The assembler assignment tries to resolve all these demands.

The assembler was made to run without tapes or operator intervention. All data is maintained in CMS datasets. The printed output, no longer the eight-inch stack of the original procedure, is passed through an editor which physically prints only requested portions. It also eliminates all the double-spacing, headings, and formatting overhead in the original listing to provide a further two-to-one compression.

One kernal in the assembler was reprogrammed for speed. This one critical routine improved the running time by a factor of ten. The final ASM135 was made into a simple "send a deck in, get a deck out" operation. More detail is given in Appendix II.

C. SIM135 Simulator.

The major component of the code-and-test system is the SIM135 simulator. It was crucial to the success of the project, because it made programming in the absence of a real target machine possible.

As indicated earlier, when the APL Assist is invoked for any given call, a large part of the 13100 bytes of code may be executed. Further, it is not necessary that execution ever terminate (APL programs may validly continue execution indefinitely until interrupted). The Model 135 hardware console is hard to use for debugging such extensive sections of microcode, especially since the inadvertent execution of any wrong code can require a hardware reset. Of course, SIM135 is much less expensive to use than a dedicated Model 135.

SIM135 is an interactive simulator written in System-370 assembler language and executing under VM/CMS. The simulation covers the physical hardware facilities accessible to the microprogrammer. It omits certain obscure facilities, particularly the outboard ends of the native

adapters, not germane to debugging microcode. The simulated machine contains System-370 GPRs, FPRs, and processor storage as well as Model-135 control registers, work registers, some of the external registers, local storage, control storage, and latches.

One of its goals which SIM135 met very well was performance. The simulator is fast. Running on a Model 145 it executes over 4500 microinstructions per second. It takes roughly 550 seconds to simulate one second of Model 135 execution. Since a typical APL test statement might execute between 1500 and 2500 microinstructions, the simulator imposed no bottleneck on the work. It was much faster for debugging than a real Model 135.

SIM135 can be made to run in two modes. In the stand-alone mode it is loaded and started like any other CMS program. The user has access to all its facilities and he may initialize and execute as he pleases. The simulated processor storage is in fact the main storage of the user's virtual machine except for that storage containing the simulator (a simulated address exception will be generated for simulated accesses to this reserved storage). In this mode the simulator is driven solely by the commands entered by the user.

For the bulk of the work, a small interface routine was added to the simulator. This interface initializes itself so that later it will gain control after any program check in the virtual machine. It then passes control to an independent, nonsimulator, user program. In the present case this user program was the software for APL/CMS. The APL/CMS software issues the special opcode 'A0' whenever it wants service from the APL Assist microcode. If the microcode is not present this will cause an operation exception program check. If the microcode is present this will cause a specification exception program check if the active APL workspace is not at the same "EC-level" as the microcode. A version number, contained both in the microcode and in the workspace, is compared at each entrance to the microcode.

To simulate APL microcode, the workspace version number is changed to an arbitrary number. Thereafter, whenever the APL software wants to execute microcode, a program check results and the simulator interface is invoked at exactly the point where the APL software calls for microcode execution to begin. The interface transfers the real

System-370 registers to their simulated counterparts, simulates the Model-135 actions when the Model 135 initially accesses an 'A0' opcode, and lets the simulated microcode carry it from there. The simulation terminates when the simulated microcode issues a fetch for the next real instruction from main memory--when it therefore has completed the 'A0' operation. The real registers are restored with their simulated counterparts and the native machine regains immediate control.

This interface did not require that all the APL microcode be written before any of it could be tested. Test cases using the APL software were set up, and then the simulated microcode was invoked for just the functions that had been completed to date. The ability to switch between APL versions using the simulator immediately gave values to compare against for all test results. An I-beam function was added to APL to do this switching in APL while APL was executing. I-beam 31, 32, and 33 respectively invoked real (Model 145) microcode, plain software (no microcode), and test (Model 135) microcode; I-beam 30 stated which currently was active. It then became possible to write functions which would test an operation in all versions and compare the results automatically.

V. Lessons learned from Microprogramming Experience.

A. Version One and why it was Unsuccessful.

In the first of the APL coding efforts, the strategy was to take the Model 145 microcode which was being developed concurrently and transform it into Model 135 microcode. The initial objective was to find a method for doing this automatically. This proved to be a more difficult theoretical problem than doing APL itself, so the direct approach of transcribing the code by hand was tried.

If this strategy were successful then only a single development project would be necessary to implement a microcode feature on two--and then also on more--machines in a product line. Whatever had been programmed and debugged on the Model 145 would be translated to execute on the Model 135. The incremental cost would be small.

At first the strategy seemed to be working. The Model 145 internal registers and latches were mapped into Model 135 internal registers and latches and then ~~translated~~ the

code was translated under these conventions. At first the transcription proceeded quickly, but then serious obstacles began to appear. Three main problems with the transcription strategy were discovered:

(1) The fundamental problem was the bulkiness of the 135 code required to shadow the 145 machine facilities. In the Model 145 each microinstruction specifies the execution of several things in parallel. Most microinstructions include the address of the next instruction to be executed, and many of these permit testing of condition codes or data bits to perform four-way branching without timing penalties. Much of the APL/145 microcode takes advantage of this embedded branching flexibility. The Model 135, however, has no such parallel capability. Each of the four-way branches had to be replaced by separate branch-on-bit instructions to get the same logical effect. This is a three-for-one increase in transcribing the branch field alone.

(2) The basic arithmetic and logical operations require more instructions on the Model 135 than on the Model 145. Processor storage accesses need two rather than one. Register references often require a preceding instruction to initialize a register address pointer. Literal fields must be developed digit by digit where the Model 145 can build a byte at a time.

(3) The shadowing of registers proved very costly. The Model 135 has only seven half-words of registers in which arithmetic can be done. This shortage of computational registers forced many loads and stores of shadowed Model 145 registers and of intermediate or partial results.

The programs resulting from this direct transcripton were so large and slow that it had to be abandoned. The control storage limit assigned to APL135 was 13100 bytes (necessary to permit it to merge with other machine features). At the end of this first attempt the code exceeded 26000 bytes. It was clear that it was necessary to

tailor the program more closely to the target machine.

B. Version Two--The Native-Mode Approach.

The strategy underlying the second version was to write the APL Assist as finely tuned to the Model 135 hardware as possible. This promised the necessary increase in performance and decrease in control-storage size. Concurrently, the program would be forced into a structure of independent modules and the quantity of test-and-branch instructions would be restricted.

The Model 135 has very few computational elements internally. In addition to the System-370 GPRs and FPRs there are eight directly addressable scratch registers. All arithmetic, however, is done in a separate set of seven halfword registers. These computational registers may be accessed as halfwords, fullwords, or doublewords during the simple arithmetic operations.

However, these registers are also used for addressing and controlling the other machine facilities. To address a GPR or FPR one must have its hex-digit register number in one of the even-numbered work registers, thus destroying one of the fullword slots. One may branch and link only with two specific work registers. Transfers from storage require that both storage address and storage data be placed in work registers. These computational elements, so few in number yet so central to the machine, proved to be the real bottleneck.

The programming method therefore became almost the opposite of top-down program design, since it began with the lowest-level hardware as the focus. This hardware had to be driven efficiently in speed and space if the program was to be successful. From this level there was a search for a structure which would produce the lowest-level optimized environment efficiently. Obviously some initial top-down analysis was done to locate the heavily used parts of APL processing. Many top-down structures are possible implementations for the central processes of APL. These processes are themselves quite invariant when properly chosen.

The strategy chosen was to make the central, or lowest-level, loops in the code be as optimized as possible. Several important consequences came from this approach:

(1) Testing and branching disappear from the inner parts of the program. It is redundant to interpret something on each iteration of a loop, and clearly it is inefficient on a machine where each test and each branch costs a cycle. The case-by-case and bit-by-bit testing, which were so costly in the earlier version were purged from inner loops. Instead, the analysis was done once, during initialization, and the resulting decisions were preserved in a directly accessible manner. This is equivalent to a local compiling of the inner loops.

(2) The control of execution now becomes separable from the actual execution itself. The lowest-level subroutines do very little interpretation. They work very much like opcodes on a normal machine. The caller governs all the variable parameters of execution, but does not engage in the execution itself. These calling routines are quite analogous to dynamic assemblers. They accept input, compile calls to the opcode-like subroutines, and issue an "execute" instruction to get work done. The task of the calling routines is simply control--they generate, maintain, and modify parameters.

(3) Since all the control has been collected into one place, the control itself can then be implemented with almost no explicit decision-making. When all the case-by-case bit-by-bit testing has been moved onto a single page, the redundancy and parallelism of most of the tests is apparent. Often a table look-up, using the bits which had been tested, can be performed instead of testing to yield a result directly. The table entry can be a value or the address of a subroutine for generating a value or executing a case. The decision-making, rather than being performed by explicit instructions, is inherent in the design of the table and index.

(4) A surprising consequence was that the higher-level routines in the structure were simplified even though they were written to optimize the interface to the lower-level subroutines. The caller issues orders--commands--to the lower levels. These lower levels, rather than interpreting the environment and deciding their purpose dynamically, simply perform as directed. There is only one copy of control information; it is analyzed and specified by the caller and modified by execution by the subroutine. What the caller needs as data in its analysis commonly becomes the controlling parameters to the subroutine. The result is

an interface scheme which allocates registers and other machinery in a manner not only optimized for the lowest level but also patterned in a rational and useful way for the work done at the higher levels as well.

(5) This approach results in a program that is strongly modular. The lowest level subroutines stand as unconnected individuals. The higher level routines have a similar character, however, since the programming is done with a microinstruction set augmented by a subroutine set. There is little cross-talk between routines. Where complex code does occur in a routine it tends to be rigorously bounded.

(6) The modular design has the effect that physical interfaces can change very late in implementation as a matter of convenience--mainly because the conceptual interface is unaffected. Team programming to write the subroutines is then clearly feasible.

In conclusion, this strategy paid off very well indeed: the resulting microcode runs twice as fast in half the core! Where the execution of inner loops dominates performance, the improvement was in fact much greater.

VI. Overview of the Microcode Emulator.

A. Executors and Index Tables.

The major effort of the second APL implementation was the search for interfaces which would optimize the hardware and which would also logically partition the application in a structured programming sense. Writing the application oriented instructions within a module posed no basic problem. To find clean divisions between these modules, and to create frictionless communication between them, was the essence of the task.

The "executors" and "index-tables" are the most fundamental structural elements in the APL/135. APL operations are performed wherever possible in two disjoint phases. First, parameters are analyzed to determine exactly what must be done. This information is reduced to two data fields for each action necessary: (1) The address in control storage of a specific routine which will without additional interpretation do exactly the required operation;

and (2) A field of register pointers indicating the registers which hold the data required in the operation.

Control is then passed to the executor routine to do the processing required. The addresses of the executor routines are located by using the variable parameters of the operation to index into tables of executor addresses. The intent was to eliminate explicit decisions. Instead parameters are manipulated to yield the address of a routine which does the specific task.

B. Comparison between Mod 135 and Mod 145 Microcode.

The APL/145 microcode was written in what can be called an "interpretive scheme" as contrasted to the "index and execute" method discussd above. Each scheme has been designed for optimizing the hardware implementing it. Neither works well when it is implemented on the other machine.

For example, consider the processing for a dyadic scalar operator (such as '0 1 2 3 4 - 5 6 7 8 9') starting where the scan invokes the dyadic scalar routine with the names of the two operands and the opcode for the operation. The dyadic scalar routine is to return the name of a variable with the result value.

Both machines begin by initializing the left and right operands. A subroutine is called to obtain the descriptors, the address of the data values, and the first data value. Both machines then check that the operands are conformable (have compatible shapes) and they then obtain space for the result. There is little basic difference in this processing.

Part of the above analysis, however, is to determine the arithmetic mode for the operation and for the result. If both operands are logicals or integers then integer arithmetic will be performed and an integer result (barring overflows) produced. If one or both operands are real then floating point arithmetic will be required.

The Model 145 records this analysis in some immediately settable-resettable-testable latches. The Model 135 records it by writing a numeric literal value into a work register. This value will later be used to index into a table to locate the specific executors required for the data mode.

At this point the Model 145 begins an element by element execution of the inner loop. It calls a subroutine to fetch and convert the left operand. The subroutine fetches the variable descriptor to determine what mode it has, branches to some statements ~~with~~ which fetch a variable in that mode, branches to other statements which contrast the actual mode with the desired mode (recorded in latches during the initialization), and branches to statements to do the conversion if required. The dyadic routine then calls the same subroutine to fetch and convert the right operand. With both operands then available and in the appropriate mode the dyadic routine itself enters a network of branches which interpret the opcode, bit-pair by bit-pair or bit-by-bit, to arrive finally at the statements which implement the operation in the appropriate precision.

When the operation for one element is complete, a subroutine is called to store the result. It interrogates the result descriptor and effects the storage. In all of these four operations--fetch left, fetch right, execute opcode, and store result--the process is implemented by branch networks which fan out, decoding descriptors and parameters, to cover the set of known cases. This procedure is then repeated for each element in the vectors.

This scheme is fast and compact on the Model 145 since the testing and branching is done in parallel with other work. A remarkably few instructions implement the large variety of cases which arise. This is critical in the Model 145, where each instruction costs four bytes and the allocation for the microcode is small.

However, as discovered in the earlier microcode version, this scheme causes problems on the Model 135. For each Model 145 instruction executed in the branch networks the Model 135 has to execute four--the actual operation in the microinstruction and the parallel decode of two data bits. For each four-byte Model 145 instruction at least eight bytes would be needed in the Model 135. Additional instructions would be required if the data accessed by these instructions had to be fetched from a shadow register.

The index and execute method eliminates the need for these branches. Consider the examples of the tables for opcode executors and for the fetch-and-convert executors. Opcodes are six-bit entities with two more bits added implicity to indicate the mode of arithmetic. The Mod 145

interpretively decodes these eight bits, whereas the opcode is added as an index to a table base address and the address of the code specific to the opcode and mode is fetched directly. The fetch-and-convert executors are similar. The table has entries for each combination of source and target data type (such as integer-to-real or logical-to-logical), and it is indexed with the actual data type and the desired data type for the operation. The tables cost core, but not as much as the branching instructions on the Mod 135.

The main payoff is in repeated iterations of the inner vector loops. Once the table index has been performed and the address of the necessary executor obtained, it can be saved and used repeatedly. No decoding at all need be done in the inner loop. Consequently these table indexings are performed as initializations. They prepare four registers with the addresses of the four necessary executors (left-fetch, right-fetch, opcode-execute, and result-store). The innermost loop then looks exactly like 'BALR, BALR, BALR, BALR, BCTR' implemented with microinstructions.

The advantages of the index and execute approach may be summarized as follows:

(1) There are no branch networks to code, debug, and maintain. Instead one must only discover a satisfactory format for the table and method for the index.

(2) The executors promote extreme modularity and simplicity. The table-index routines, which are actually the decision routines, are separate from the executors, so that the code which ultimately does the work is not mixed with decision-making requirements. The executors may be written independently without diminishing efficiency.

(3) The executors can be used like subroutines by many callers. The dyadic scalar opcode executors, for example, are also used by the inner-product, outer-product, reduce, and iota-epsilon routines under the same conventions.

(4) It is easy to maintain an executor, since each is simple and stands apart from the others. It is also easy to add new ones to the system. A new operator is supported simply by replacing the 'system error' executor address presently in the table with the address of a new executor to support the operator.

(5) Finally, the executor structure is fast on the real

Model 135. With the exception of the four BALRs and the BCTR, almost all instructions in the inner loop contribute directly to the work cf that loop.

C. Other Programming Techniques.

Usually branch networks were replaced with tables and indexing even where the ultimate target of the branching was not executable code but the identification of a data item. Index tables were used to select executors or control information for processing monadic and dyadic scalar and mixed operators, reduction operators, arithmetic overflowing operator replacement, subscripting operations, variable fetches, variable stores, and several local minor functions. In each case this saved time, core and work and made it easier to expand and maintain the microcode.

In indexing (subscripting) routines, three executors are invoked for each element in the central loop. The first fetches the next subscripting variable and converts it to an integer. The second and third respectively fetch and store the subscripted variable. Whether the fetched or the stored item is the subscripted item depends on the APL statement. There are separate executors for each data type and direction. All the executors (with a few exceptions) for these three actions were constructed within sixteen-way branch spaces in the Model 135. The sixteen-way branch uses a hex digit in a work register to select one of the sixteen targets four instructions apart. Most of the executors could be constructed within this space. Since all sixteen values in the table were not needed, the vacant targets could be used to complete the larger executors. What the table index produced in this case was not the address of the executors but a data word containing the hex-digit numbers of the necessary executors. Then the sixteen-way branch was entered and each executor, instead of returning to its caller, just branched sixteen-ways on the digit from the table for the next executor in line.

Of course, it is not possible everywhere to reduce bit-testing and decision-making to automated table lookups. The language itself demands some highly asymmetric involuted examination of strange cases. One of the worst cases is the conformality testing, in the dyadic scalar operators, of operands with different sized arrays where both are single valued.

Other sections of the code are highly serial. The scan, for example, examines each new token in turn and decides what to do as each new token is examined. Function calls, similarly, localize variables one by one, and each may be different in kind from its precursors.

The scan of the top two tokens is a sixteen-way branch on the first followed at each target by an eight-way branch on the second. Each possible combination of the top two tokens thereby is routed to a specific address, and this is simply a branch to the appropriate routine for the combination. The entire logic is representable by a two-way table and the correlation between table and code is immediately clear. This branching was fully populated even though for some top tokens only two or three next tokens actually may occur. These could have been isolated by specific testing in fewer than the eight branch targets now used. However, this was precisely the "spaghetti" one tries to avoid.

The interfaces to subroutines were evaluated to optimize performance in the subroutine. Often the output of a subroutine became the input of a following subroutine or process. Therefore the computational registers were carefully allocated to the active fields, and each active field was forced into the same register for all subroutines. As a result the shuttling of arguments almost disappeared.

D. Documentation.

Aside from this report, APL/135 is documented solely by the comments included in the listings. These comments fall into two categories:

(1) Line-by-line comments which describe the local effect of the individual instructions. These comments speak more in terms of APL data items than in terms of registers and arithmetic.

(2) Descriptive paragraphs preceding each section of code and each subroutine in the emulator. These describe the intent and method of the program and identify specifically the inputs and outputs.

Putting the documentation directly into the code greatly reduces the problem of keeping two separate

resources (the document and the program) synchronized.

Preparing comments imposes a considerable burden on the productivity of the programmer. In Version 2 all 13100 bytes of the microprogram was produced initially without a single comment. The MPL135 symbolic nomenclatures and structured indentation provided all the memory refreshing required. Toward the end of the project, however, when comments were being added as precursor of completion, it became much more difficult to modify the code. It took two, three, or four times as long, since not only the instructions but also the descriptions had to change.

VII. Detailed Description of the APL135 Microcode.

A. The Organization of APL135

1. Invoking the Microcode.

The APL emulator is envoked when the special 'A0' operation code (The APL opcode, 'APLEC') is detected in an instruction sequence by the System 370. The emulator does not modify any other System 370 instruction, so it may be used in the presence of standard operating systems, such as VS and VM/370.

The specification of the 'A0' opcode or 'APLEC' is given by Hassitt and Lyon in "The APL Assist". (See References). The details are not repeated here, since the intent is to limit this report to the details of the implementation of the Mod 135 microcode only.

The following quantities are assumed to be correctly implemented on entrance to the emulator. They are not verified within the Mod 135 microcode:

(1) Base registers: GPRB (the base used by software) is assumed to contain an address ending in x'000' and GPR3 (the base used by microcode) is assumed to contain that same address ending in x'8F0'. The microcode destroys GPRB during execution, but recreates it from GPR3 before termination. The high-order byte of both addresses is assumed to be '00'.

(2) Control words: All control words and microstore save areas are assumed to lie within the same DAT page as

the workspace validation checkword. This guarantees that, having successfully accessed the checkword, the microcode environment can be preserved in the workspace for relocate exceptions and hardware interrupts without generating a page fault.

(3) Workspace format: The workspace format is assumed to be valid and execution of an invalid workspace will produce unpredictable results.

2. Organization of the Microcode Routines.

The APL emulator consists of many routines (CSECTS) which may be grouped for conceptual convenience. The names of the routines contain only three letters (a restriction imposed by the APSS assembler). The routines begin with the letter 'H'. This letter was assigned to APL135 so that none of its routines would conflict with any other CSECT names on the Model 135 manufacturing master tape. The remaining two letters are chosen to be mnemonic. (See the Glossary for the meaning of the acronyms.)

Throughout most of the execution of the APL emulator the microcode is dealing with an operator 'OCODE' on two operands, the left operand 'LITEM' and the right operand 'RITEM', to produce a result variable 'ZITEM'. These operands or variables--'XITEM's--generally have a name 'XNAME', an address 'XADDR', a description 'XDESC', a type 'XTYPE', a shape 'XSHAP', and a value 'XVALU'. Comments within the microcode refer to these xitems with these abbr abreviated xterms. Some of this information is regularly contained in specific registers while other information does not possess a customary registration.

3. The Hardware Trap Required for APL Relocation.

A hardware pseudotrap--EXT3 bit 4--is required to execute APL emulation. Setting the pseudotrap causes a hardware logout and a transition into the console zone (exactly like program-event-recording except that a flag bit is flipped to distinguish these two cases). This pseudotrap has three specific necessary attributes: It preserves all CPU-zone registers (intact in zone-0), it retains the fault address (logged into zone-1), and it branches to a trap routine (location '0004'). These pseudotrap instructions are placed following each main-storage reference (each with its skip) and are executed when the storage access fails--

TABLE 1: THE NAMES OF THE 52 MICROCODE CSECTS

HAA: Analysis and interface between software and microcode.

HDA,HDM,HDS: Dyadic mixed and scalar operators (except mixed operators requiring separate routines (see HQ*)).

HEP,HER: Inner-outer products and reduce-scan operations.

HFA,HFE,HFI,HFX: Function control--invoke, branch, un-invoke, and software-execute.

HIA,HIE,HIF,HII,HIS,HIV,HIX: Indexing operations.

HMA,HMM,HMS: Monadic mixed and scalar operators (except mixed operators requiring separate routines (see HP*)).

HOF,HOP: Arithmetic routines--fixed-point and progression-vector.

HPC,HPI,HPO,HPR: Monadic-mixed operators.

HQC,HQE,HQI,HQO,HQR,HQT: Dyadic-mixed operators.

HRA,HRE,HRI,HRS: Resource management--obtain and release names and cores and assign and synonym values.

HSA,HSI: Scanning and execution selection.

HUX: Utilities.

HVA,HVC,HVF,HVP,HVS: Variable management--initialize, fetch, store, point, and copy.

HWC,HWD,HWE,HWF: Linkages between microcode blocks.

HZI,HZR,HZT,HZX: System-support--interrupt, dynamic-address-translation, and tables.

TABLE 2: ORGANIZATION OF THE HARDWARE REGISTERS.

Hardware registers are used by the emulator generally for the following purposes: (The GPR's and FPR's are the standard S-370 general purpose registers and floating point registers. The CWR's and WWR's are internal to the Model 135 microcode.)

```
GPR0:  Lvalu if ltype=(char,logi,inte)
GPR1:  Litem=(ldesc,lname)
GPR2:  Laddr
GPR3:  '00WWW8F0':  Workspace base
GPR4:  Scratch
GPR5:  Scratch
GPR6:  Rvalu if rtype=(char,logi,inte)
GPR7:  Ritem=(rdesc,rname)
GPR8:  Raddr
GPR9:  Ocode or zitem=(zdesc,zname)
GPRA:  'FASSSSSS':  'ASCAN' APL scans pointer
  (address of next scannable function token)
GPRB:  '2BSSSSSS':  'ASTAK' APL stack pointer
  (address of next fillable stack position)
GPRC:  Zaddr
GPRD:  Scratch
GPRE:  Token preceding 'LITEM OCODE RITEM' in scan
GPRF:  '197ERRII':  Where 197E='XSTAK', the
       register pointers, RR=external-function
       return code, and II=operator index.
FPR0-FPR1:  Lvalu if ltype=real
FPR2-FPR3:  'FCORE' free-core anchor and 'FNAME'
            free-name anchor
FPR4-FPR5:  Scratch
FPR6-FPR7:  Rvalu if rtype=real
CWR0-CWR7:  Scratch
WWR0-WWR7:  Scratch
```

Control storage is used by the emulator generally for the following purposes:

```
DSC08:  Address of relocation routine or relocate-
        exception interrupt code
DSC17:  Bf:  APL-active flag (tested by System-370
        Machine-check and program-interrupt
        Microcode):
             '0'--set on APL termination
             '1'--set on APL initiation
```

DSC48-DSC49: System-370 instruction counter
DSC50-DSC53,DSC56-DSC57: Scratch (save area during dynamic address translation)
C(DSC17+('00'-'0C')) (lex-mode hard save list): Save area during timer-update service.

Hardware controls (external-immediate addresses) are used by the emulator generally for the following purposes:

```
X0:  B1:  Irequ:  Quantum-end request pending.
X1:
     B0:  Dalow:  '1'--mandatory
     B1:  Dpurg:  '0'--mandatory
     B2:  Dtrap:
                  '0'--set on APL initiation
                  '1'--set on APL termination
     B3:  Dlexe:  Signal visibly on the system
          control console APL emulation (but
          sometimes a flag--see the comments
          in routine HZA):
                  '0'--set on APL termination
                  '1'--set on APL initiation
     B6:  Dtalu:  '0'--mandatory
     B7:  Dwrit:
                  '0'--set on APL termination
                  '1'--set on APL initiation
X2:  B1,B3:  CCODE:  External return code and
          internal flag:
          '0'--remove normal termination and inter
               zitem=(operator-result)
          '1'--remove error termination
          '2'--inter zitem=(assignment-result)
X3:
     B3:  Idump:
          '0'--set on APL termination (by IFETCH)
          '1'--set on APL initiation
     B4:  Atrap:
          '0'--execute microcode
          '1'--execute pseudotrap
X5:  B2:  Bwrit:
          '0'--set on APL termination (by IFETCH)
          '1'--set on APL initiation
```

normally because of a relocation exception. Such storage access failures can then be serviced completely transparently to the local microcode and registers.

B. The Scanning and Execution of APL Programs.

The execution of APL programs is done by the scanning of tokens from an expression within an APL workspace. The line is scanned from right to left (viewed in its original typed-in form). The current value of the scan address (ASCAN) is the equivalent to the instruction counter in System 370.

Scanning is the cyclic activity of fetching, decoding, and executing APL tokens. A token is read and expanded into its fullword stack-token format. The top two stack tokens are jointly decoded. An act specific to the top tokens is performed (Examples: to add the new token to the stack and continue, to execute an operator, and to invoke a function). Each of these acts leaves at least one token on the stack. Scanning then continues with the next APL token. The syntax decision table is similar to that of APL/145. It differs only in minor details of the decoding.

Variables placed on the stack represent a value and not a name. When the variable is finally accessed it must have the value it had when it was stacked. It is, however, possible to reassign the value of a stacked variable before the stacked variable is accessed. Consequently, whenever a stacked variable is followed by syntax which makes it possible to change the value of the variable, the value is copied and placed on the stack either as a stack-immediate or as a temporary variable. These ~~may not~~ cannot or must not be changed in value.

The decoding of the top two stack tokens is accomplished by two sixteen-way branches. The first decodes the top stack token on its syntax class; each of its sixteen targets is the second branch which decodes the next-to-top stack token on its syntax class. The targets of this second group of branches (one for each first-branch target) are the instructions labelled 'XXXYZ' ('YZ' hexadecimally numeric). 'Y' is the syntax of the top token and 'Z' is the syntax of the next-to-top token: XXX15, for example, is reached when the first token is '1XXX' and the second token is '5XXX'. These XXXYZ instructions finally branch to the specific 'ACTXX' routine which will handle properly the tokens on the

stack. Note that the top token can have syntax '0'-'F' but that the next-to-top token can have syntax '0'-'7'--only the necessary targets in the branches are supplied.

The top four stack tokens are maintained in general registers 1, 9, 7, and E logically in that order. On entrance and exit the tokens are physically in their logical positions but during scan only the logical precedence is preserved. The ordering is controlled by the 'XSTAK' register (normally W6) whose digits respectively address--digit-0 points at stack-0--the top four stack positions. On entrance and exit, therefore, xstak will contain '197E'. When a newly-fetched token is placed into a register it will be placed into the register addressed by digit-0. When a token must be moved to main storage it will be the token in the register addressed by digit-3. When the three register-remaining tokens must be pushed down one logical position in the stack this is done simply through a "ring shift right four" of the xstak register. (note that the cyclic ordering of '197E' must be preserved ('179E' for example is invalid)--several subroutines depend on this ordering and the xsave routine locates the home position by looking for the 'E'.)

The detailed documentation of APL/135 uses the following nomenclature and registers. The top four stack tokens are called stak0, stak1, stak2, and stak3. The current scanning address 'ASCAN' is maintained (usually) either in W2-W3 or in GPRA and its first byte always contains the register pointers 'FA'. The current stacking address 'ASTAK' is maintained (usually) either in w0-w1 or in GPRB and its first byte always contains the address-table bits and register pointers '2B'. The current logical ordering of the stack registers 'XSTAK' is (usually) contained in W6 and its home position ('197E') is contained in GPRF.

Some microinstructions in the scan routine are hard-code, that is, represented by numeric constants and not assembled. This is necessary because the assembler will not accept sixteen-way branches for which all legs are not defined. In the decoding of the second stack token there are twelve sixteen-way branches for which only the first eight paths can ever be taken and therefore, to save 96 micro instructions, only the first eight paths are provided.

C. The Execution of Dyadic Operators.

The execution of dyadic mixed operators is accomplished by separate execution subroutines. The dyadic analysis routine (HDA) detects mixed operators, obtains the address of the proper execution subroutine, and transfers control to the execution subroutine which performs the operation.

The execution of dyadic scalar operators is organized around the use of executors. These are short single-purpose subroutines which execute small specific pieces of work: fetch-and-convert an input variable, or execute a scalar operation, or store an output variable. All executors of a given type (fetch, execu, store) have a common interface with respect to inputs and outputs. Only the data type (char, logi, inte, real) used with the specific executors differs. All executors of all types have a common interface with respect to linkage:

W4--register pointers
W5--executor address
W6--return address

Given a fullword register containing the register-pointers and executor-address, this linkage interface is very much like a System-370 BALR. This technique eliminates opcode-operand decoding during execution iterations, enables multi-routine accessing of executors without requiring the executors to determine the correct return point, and permits a one-instruction inline linkage in the calling routines. This has proved to be a very important concept in giving performance to the Model 135 APL emulator.

The task of dyadic scalar analysis then becomes threefold: First it must validate the operation and obtain result storage. Second it must determine the fetch, execu, and store executors appropriate to the operands and operation. Third it must invoke the selected executors, element-by-element in vector-array cases, to evaluate the operation.

Operation validation is done with branch-laden routines which determine the mode of arithmetic and result, check the conformality of the operands, and obtain storage for the result (if vector or array).

Executor selection is done by indexing into tables of executor addresses. All such index-tables have a base

address. Offsets are added to this base to obtain the address of an element within the table. The element is fetched to obtain the desired executor address. The offsets into the table are generated by manipulations of various data items--there is no general rule and the individual rules are matters of convenience and efficiency. Each table is described elsewhere in detail. When an executor address is obtained it is retained, with the requisite register pointers, in a fullword register.

Executor execution is done by calling a subroutine to effect a pseudo-BALR as mentioned earlier. The subroutine restores the executor-invoking fullword and goes to the address in that fullword. Consequently the specific instructions in the calling routine to execute a vector-operation consists of five pseudo-BALRs: Fetch rvalu, fetch lvalu, execu xexec, store zitem, and count-and-branch until finished.

D. Dyadic Analysis and Control.

This section summarizes in more detail the processing of dyadic operations by the HDA routine. Portions of the HDA routine are also used for outer-product and for dyadic iota-epsi. These other operators are initialized elsewhere, they enter HDA with case-switches set, and are completed there. The paragraph names below correspond to the main parts of the HDA routine.

Begin: Operand and operation descriptors are accessed. Mixed operators are detected and, through a table-index, are sent to specific subroutines for execution. Scalar operators are analyzed to determine the mode of arithmetic during execution and the mode of the result to be generated.

Valid: The operands are checked for conformality. Operands involving true scalars are isolated and expedited; otherwise separate shape-check routines are entered for the various shape-shape combinations. Arithmetic-progression-vectors versus (pseudo) scalars are sent into immediate apv arithmetic execution if the opcode is suitable.

Iopie,iprod,iioep: Special initialization is performed for outer-product and iota-epsi.

Space: Space for nonscalar results is obtained.

Opera: A table-index is performed to obtain the executor address of the dyadic-scalar executor and of the variable-store executor. Nonvector operations requiring no operand conversions are expedited to execution. Otherwise a further table-index is performed for both the left and right operands to obtain the executor address of the variable-fetch (convert) executors. All executor-addresses are retained with register-pointers for use during operation execution.

Xloop: Vector operations are executed, element by element, until complete.

Ximed: Nonvector nonconverting operations are executed expeditiously.

Xopie,xprod,xioep: Final initialization of outer-product and iota-epsi is performed. The separate cases are then executed in specific element-by-element loops.

Quits: Operands are released and control is returned to scan analysis.

Addresses in the HDA routine are hard-assigned because it calls subroutines the parameters for which are passed in the return address. Micro instructions which must be fixed parametrically (or immediately adjacent micro instructions) are identified by labels of the form 'ZYYXX': 'Z' flags these micro instructions; 'YY' groups related fixed micro instructions; and 'XX' is the parametric value which must be contained in the low-order byte of the address of the labelled micro instruction.

One case of this address-fixation deserves further comment. A subroutine is called to execute the count and branch which terminates the iterative execution sequences mentioned earlier. This subroutine, HUX-ITERX, has parameters passed as described. It does not return to the next-sequential-instruction. If the count is not exhausted it returns upstream a specified number of instructions. If the count is exhausted it returns either down stream a specified number of instructions (prod and iota-epsi or to a fixed offset within the calling control storage module (dyad vector). The entire procedure seems unnecessarily

involuted, but it conserves control storage.

E. Dyadic Scalar Execution.

The executors employ one of three general procedures to evaluate the scalar operation:

(1) Most executors execute inline microcode which directly produces the result.

(2) Some executors (such as those for trigonometric and logaritmic functions) exit to software which produces the result. The software is responsible for returning with the result through a standard interface. The executor then resumes as if no exit had occurred.

(3) Some executors (fixed-point divide and floating- point arithmetic) branch directly into the System-370 microcode which produces the result. These executors preserve the necessary registers and simulate the internal environment generated by the execution of an appropriate System-370 instruction. The APL-active switch remains set and the simulated PSW-address is made odd. This odd-address generates a PRI-trap when the System-370 instruction microcode terminates with an IFETCH. The System/370 PRI-trap microcode tests if the APL-active switch is set and if so then it returns to the invoking executor. The executor restores its registers and then resumes as if no exit had occurred.

Arithmetic overflows may occur in some scalar operations. These cause a branch to the dyadic analysis overflow subroutine. The overflow code passed to that subroutine is generated through the address assignments of the overflowable executors.

The address-assignments of certain executors are significant. In all cases the specific bit assignments are arbitrary and respecifiable provided that the assignments and the examinations are consistent. The following summarizes the uses of the address.

(1) The low-order byte contains the overflow code for those executors which can overflow. This is tested by HDA.Ovflo for overflow processing.

(2) The high-order byte contains two bits to

control processing of executors which branch to the System-370 microcode. These are tested by HDS.Error for program-exception processing, or HDS.Zexex for fixed-float initialization.

(3) The high-order byte contains two bits to control initialization of executors of fullword logical operators. These are tested by HVF.Inikk during initialization.

F. The Execution of Monadic Operations.

Monadic operations are recognized in the Scan analysis routine and sent to the HMA routine for further analysis. The right operand has been initialized. The monadic operation is evaluated, as mixed or scalar and in microcode or software, the operand is released, and the result is placed on the stack. Control is then returned to scan analysis.

The execution of monadic operators is accomplised by separate execution routines. Mixed operators are each handled either in distinct routines or in software. Scalar operators are reinitialized to appear as dyadic and are then executed as dyadic operators. The HMA routine merely routes the operations to appropriate execution procedures.

G. Workspace Resource Management.

A group of routines--HRA, HRE, HRI, HRS--provide the essential management of resources during the execution of the emulator. These tasks--to obtain and to release names and storage, and to assign and synonym values-- are those which give APL its unique dynamic features.

1. Obtaining Names and Space.

The HRI routine contains two separate subroutines which obtain names and storage, respectively, from the pools of available names and storage. The names subroutine is straightforward and uncomplicated.

The space subroutines are manifold, but differ primarily in the invocation interface. To simplify the requirements in calling routines, space may be requested under several different interfaces. The most common of

these are:

(1) Refer,shape: Obtain a variable with a parametric datatype but with the shape of an existing variable. In the refer case the shape-vector has not yet been located while in the shape case the shape-vector has already been located.

(2) Elemx: Obtain a variable with a parametric datatype containing a parametric number of elements and dimensions. Insure the validity of the element count and reset the pseudoscalar description bit if the variable has only a single element.

(3) Bytez: Obtain a variable with a parametric description containing a parametric number of bytes.

These entries, gradually merging, all funnel into the label "bytex" which acquires a specified amount of storage and initializes it based on parameters set by the caller and by the entries.

Several levels of initialization may be requested for the acquired storage (although not all combinations of invocation-initialization are valid). An element count or an entire shape vector can be copied into the new space. A value-vector can be copied into the new space. The space can be initialized for use by the variable store-executors in routine HVS. The space name can be placed on the stack as the result stack entry for a monadic or dyadic operation. These services are provided to simplify the processing and reduce the control-storage requirements in the invoking routines.

Two special cautions are in order: (1) The return address for the space subroutine contains embedded parameters. (2) Since both the names and the space subroutines can exit to software (for name-table extension and for garbage-collection), no information except the xscan registers (described in the scan-analysis routine) may be preserved in GPRA-GPRB.

2. Use of Synonyms in Resource Management.

A crucial procedure for conserving resources during APL execution is the use of synonyms. It not only saves space, by not duplicating arrays, but saves the time which copying

the arrays would take.

The HRS subroutine sets the value of a given variable equal to that of a second variable. It is invoked mainly (1) to execute the assignment operator when the right variable is not an immediate variable, (2) to set the result of an operation equal to one of the operands in that operation, (3) to shadow during function invocation the arguments into and the local-variables within the function, and (4) to shadow permanent variables on the token stack when subsequently-scanned tokens hazard the value of the variables.

Control reaches HRS either through branches in other routines (zisrx) in which cases control passes to the scan analysis routine at the completion of processing, or through subroutine calls in other routines (synon) in which cases control returns to the calling routine at the completion of processing.

The synonym generation may be done in one of several ways. If xitem is temporary then it will be renamed into zitem. If xitem is immediate then zitem will be made an equivalent immediate. If xitem is already a synonym then zitem will be made an additional synonym. If xitem is an addressed (pseudo)scalar or APV-vector, or if xitem requires less than 64-bytes of storage then zitem will be made a copy of xitem. Otherwise zitem will be made a synonym of xitem.

The following illustrates the possible cases before and after linking. The entries represent the contents of the link-name field in the synonym block.

Litem and Ritem are the left-and-right links of xitem. The link-names for litem, xitem, ritem, zitem, dont-care-items, and null-link-items (end-links on the chain) are represented respectively by 'L', 'X', 'R', 'Z', 'Q', and '0'.

Before linking			After linking			
litem	xitem	ritem	litem	zitem	xitem	ritem
Q-X	L-R	X-Q	Q-Z	L-X	Z-R	X-Q
Q-X	L-0		Q-X	X-0	L-Z	
	0-R	X-Q		0-X	Z-R	X-Q
	0-0			0-X	Z-0	

The zitem and xitem link-name fields are constructed into and initialized from WWR2-WWR3 and WWR4-WWR5,

respectively. Then they are initialized by subroutine if litem requires alteration.

Resource synonyms are released by the HRE routine. It consists of two separate subroutines which release names and storage, respectively, and return them to pools of available names and storage.

If the released storage was a synonym block in a synonym chain, then the synonym chain must be relinked to exclude the released block. The following illustrates the possible cases before and after relinking. The illustration entries represent the contents of the link-name field in the synonym block: Litem and ritem are the left-and-right links of the current item xitem; link-names for litem, xitem, ritem, dont-care-items, and null-link-items (end-links on the chain) are represented respectively by 'L', 'X', 'R', 'Q', and '0'.):

Before relinking			After relinking	
litem	xitem	ritem	litem	ritem
Q-X	L-R	X-Q	Q-R	L-Q
0-R	X-Q		0-Q	
Q-X	L-0		Q-0	
0-R	X-0		make ritem nonsynonym	
0-X	L-0		make litem nonsynonym	
	0-0		release value block	

In the two next-to-last cases the single remaining synonym is desynonymed by releasing the synonym block and making the variable a free-standing variable resident in the value block. In the last case--a synonym format without sibling entries (generated only when a workspace-full error occurs in the synonym-allocation process)--the value block is simply released.

H. The Management of Variables.

A group of routines, HVA, HVC, HVF, HVP, HVS provide for the management of variables. The tasks include: initialization, fetching, storing, converting and copying.

1. Variable Initialization.

The initialization is done by HVA which prepares the variable for accessing by the fetch executors. The following summarizes the processing of this subroutine:

If requested then the register pointers for the standard variables litem or ritem are obtained. The variable is analyzed. Immediate variables are made to appear as permanent-immediate and their values are expanded into standard representation. Addressed variables are obtained from storage. Their descriptions are marked permanent or temporary. Their addresses are made to point at the value block and are modified by the addition of control information ~~mation~~ in the hi-order byte. Their initial value is fetched. The information generated for both immediate and addressed variables is placed in the parametric GPRs and FPRs.

The following illustrates the completed initialization for various variables (assuming input W4='QRS3').

	GPRr	GPRs	GPRq	FPRr	FPRs	
Immediate						
Char	2E0400CC	000000CC	0604XXCC	XXXXXXXX	XXXXXXXX	XXXXXXXX
Logi	2E000001	00000001	06000001	XXXXXXXX	XXXXXXXX	XXXXXXXX
Inte	2E012222	00002222	06012222	XXXXXXXX	XXXXXXXX	XXXXXXXX
Inte	2E813333	FFFF3333	06013333	XXXXXXXX	XXXXXXXX	XXXXXXXX
Addressed						
Char	29XXNNNN	000000CC	TTTTNNNN	XXLLLLLL	XXXXXXXX	XXXXXXXX
Logi	29XXNNNN	00000001	TTTTNNNN	01LLLLLL	XXXXXXXX	XXXXXXXX
Inte	29XXNNNN	00000002	TTTTNNNN	XXLLLLLL	XXXXXXXX	XXXXXXXX
Apvv	29XXNNNN	00000003	TTTTNNNN	QVLLLLLL	XXXXXXXX	XXXXXXXX
Real	29XXNNNN	XXXXXXXX	TTTTNNNN	QRLLLLLL	44444444	44444444

'TTTT' is the variable descriptor plus '0200' if it is a permanent variable, 'NNNN' is the variable name, 'LLLLLL' is the variable value-block (first-element) address, and 'V'='R'+'1' ('R' assumed even for apvv and real).

2. Variable Fetching--Analysis.

The analysis subroutine is called when an initialized variable is to be prepared for fetching and converting to a specific internal format. The subroutine determines the element count of the variable (useful information for many callers), obtains the address of the fetch-executor appropriate to the into-from conversion request, and constructs a fullword which can be used to invoke the executor with the executor execution subroutines in routine HUX. This analysis subroutine is called by routines which

need to fetch a variable in a standard arithmetic mode.

3. Variable Fetching--Execution.

The executors execute all variable fetch-and-convert operations. There are two executors, one for scalar variables and one for vector variables, for each valid (char, logi, inte, real) into-from conversion combination. All of these executors have a common interface. All are given an initialized xitem from which they obtain the next value. the converted xvalu and the incremented xaddr (if vector) are returned.

In addition to the standard executors are two sets of nonstandard executors which support the execute-reduction routine. One of these sets is the reduction fetch-executors. These serve the same purpose as the standard executors but fetch the variable elements in descending, not ascending, order. The other of the sets is the reduction store-executors. These serve two functions--they store the result of an operation and they fetch the rite-operand (identical to the result but perhaps in a different arithmetic mode) for the next operation. The three sets of executors are distinguished by the naming convention detailed in HVF-Enter.

4. Variable Storing--Analysis.

The HVS routine handles the analysis and storing of variables. The analysis-and-store scalar-variable subroutines are called when various kinds of results must be stored. Many entries are provided (all matters of convenience) and these are detailed in HVS. The stored variable is placed on the stack as a stack immediate if possible; otherwise it is stored as a temporary variable in main storage. Two entries, however, make use of permanent workspace variables.

5. Variable Storing--Execution.

The executors execute all scalar and vector store operations. These executors are accessed indirectly through index-tables in various routines. The vector executors are all given a zaddr and a zvalu which they store at zaddr which they increment to address the next element position.

The scalar executors, overlaping heavily with the subroutines described above, produce the same form of stored-and-stacked variable as those subroutines. The scalar executors are intended to be invoked once per (monadic or dyadic) operation while the vector executors are intended to be invoked once per (monadic or dyadic) operation execution element.

I. The Control of APL Functions.

The routines HFA, HFE, HFI, HFX control the use of functions. This includes the branch arrow (GO TO) as well as the invoking and releasing of functions and the calls for software function execution.

1. The RIGHT ARROW (GO TO) Operation.

The HFA routine executes the GO TO operator which may be represented internally either as the monadic scalar right arrow with the right operand containing the value of the branch target, or as a single APL GOTO token integrally containing the value of the target.

There are four entries to the HFA routine corresponding to the four cases where branches are detected:

(1) Begin: Scalar-operator branches reach here through a direct branch from the scan analysis routine. The right operand has been initialized.

(2) Begis: Special-token branches reach here through a direct decode from the scan analysis routine. The branch-target value has been extracted.

(3) Begcc: 'GOTO SCALAR COMPRESS SOMETHING' branches reach here in a special optimization through a direct branch from the execute-compress routine. Conditions are substantially the same as in the begin case above.

(4) Begix: Scalar-operator branches with complex syntax context reach here through a table index from the moadic analysis routine; the right operand has been initialized.

All branches in the first three cases which reach the HFA are executable branches. Any unusual conditions which

inhibit branching in the microcode have been detected elsewhere and any branching under these conditions is executed in software. Specifically regarding the branches executed in microcode: The branch is not traced; the branch does not appear in an immediate (directly entered terminal statement) function; and the branch has simple syntax (nothing else is stacked).

While these branch-inhibit conditions are explicitly checked by the branch sources described above, the check of the stack syntax status is less explicit. The single token goto appears only in valid syntax (where it will be the only token in the APL statement). The scalar-operator simple-syntax branch is entered from scan analysis only when 'END-OF-LINE RITE-ARRO' is decoded and it is guaranteed by the operation of scan-analysis on software-provided statements that the remaining stacked tokens are 'VARI NULL' only. The scalar-operator complex-syntax branch is entered when the rite-arrow appears with any other token on its left. The scalar compress does, however, examine its context to insure simple syntax.

Three outcomes are possible from the HFA routine. If the branch-target value is a null vector (this can not occur in the single-token case) then no branch is taken and the scan continues in sequence. If the branch-target value is equal to the number of a statement in the currently invoked function (other than statement zero) then the branch is taken and the scan continues at that statement. If, otherwise, the branch-target value is outside the range of the function then a function release is effected by passing control to the function release routine. In the branch cases a null result is supplied to permit a resource-release attempt for zitem to succeed without problems. In the scalar-operator cases the rite operand is released.

2. Invoking and Releasing APL Functions.

The HFI routine performs the invocation of an APL function. Control reaches the routine through a direct branch from scan analysis after the function and its arguments have been detected. This routine preserves the values of the arguments, constructs a function invocation block (FIB) on the APL stack, shadows the values of all local variables, and assigns to the function local-named arguments their preserved invoking values. Control is returned to scan analysis with the APL-scan address pointing

at the start of the invoked function.

The HFE routine releases an active APL function. HFE is called when it detects a branch to a statement outside the range of the function. The routine preserves the function result value (or generates a nonvalued result), releases and unshadows local variables, removes the function invocaton block (FIB) from the stack, and adjusts the APL-scan address to point at its previous position in the function which invoked the currently-releasing function. Control is then passed to scan analysis.

J. Indexing Operations.

1. Terminology and Organization.

Indexing is one of the most complicated parts of the APL emulator. The performance on application programs depends to a large part on how efficiently indexing is done. The array cases in the Model 135 emulator follow the method given by A. Hassitt and L. E. Lyon (See References).

The following describes the indexing operations. The emphasis has been on speed, even when it costs storage and complexity. The indexing operations involve three distinct variables:

(1) Ritem: The randomly-accessed (hence 'R') variable--the variable which is indexed.

(2) Sitem: The sequentially-accessed (hence 'S') variable--the variable which is the input into or output from ritem.

(3) Qitem: The indexing variables--one variable or semicolon for each dimension of ritem.

Indexing operations have two different forms depending on the direction of assignment. The first plucks values from an indexed variable while the second sticks values into an indexed variable:

(1) Sitem gets ritem(indexed by qitem)
(2) Ritem(indexed by qitem) gets sitem

These are called respectively fetch indexing (ritem=fech)

and store indexing (ritem=stor). In both cases the zitem supplied as the result of the indexing is sitem.

The basic method of the indexing routines is to reduce qitem, which may be a complex sequence of variables and semicolons, into a compressed, accessible, and optimized form. In all cases the index qvalu resulting from this reduction will be an origin-zero index on the ravel of ritem. The final indexing routines, the executors which actually fetch and store the indexed elements, therefore are presented with an ritem which they treat as a vector and with a series of qvalus which identify the specific elements which are indexed in that vector.

The final qvalu sequence will be presented to the executors in one of four basic modes. The executors are not actually given the qvalus themselves; they are given initialized addresses or counters in one of these four modes and the executors fetch the indexes using these parameters. Reducing qitem into one of these four modes is in fact therefore the reduction of qitem into one of four parameter forms which will control the execution of the final specific executors.

2. The Four Basic Modes of Indexing.

The four basic index modes, known as 'XMODE's, are:

(1) Xmode=scal: If all dimensions of ritem are indexed by scalars then qitem will be reduced to a single number contained in registers and all subscripting variables will be released. The parameters inherent in this xmode are:

Xbase: Index of single indexed element

(2) Xmode=apvv: If ritem is indexed, in all dimensions not indexed by scalars, either by a single arithmetic-progression-vector or by a single contiguous set of elided subscripts then qitem will be reduced to a single progression vector contained in registers and all subscripting variables will be released. The parameters inherent in this xmode are:

Xbase: Index of initial indexed element. The effect of all scalar qitems and of the base qvalu for the qitem=apvv is incorporated into this xbase.

Xstep: Increment to index for each subsequent indexed element. This is the step value for generating the indexes. For cases where qitem=apvv indexes the innermost ritem dimension then xstep equals qstep and for other cases then xstep equals qstep times xwate, the dimension multiplier weight for the indexed dimension.

Xlimi: Number of indexed elements.

(3) Xmode=vect: If ritem is indexed, in all dimensions not indexed by scalars, by a single vector or array then qitem will be reduced to this single vector and all other subscripting variables will be released. The case where both ritem and qitem are arithmetic-progression-vectors will be forced into this mode. The paramaters inherent in this xmode are:

Xbase: Index of initial indexed element (ignoring xitem). The effect of all scalar qitems is incorporated into this xbase.

Xwate: Multiplier for each xvalu in xitem. For cases where xitem indexes the innermost ritem dimension then xwate equals zero (a flag for this case) and for all other cases then xwate equals the dimension multiplier weight for the indexed dimension.

Xlimi: Number of indexed elements.

Xitem: The nonscalar qitem (initialized).

(4) Xmode=list: If ritem is indexed by anything outside of the above three cases then a special indexing variable will be constructied and all subscripting variables will be released. These lists must be differentiated occasionally by the nature of the innermost dimension subscript: It may be either a progression vector (xlist=prog) or a vector of distinct index values (xlist=valu). The parameters inherent in this xmode are:

Xbase: Index of initial indexed element (ignoring vector or array qitems). The effect of all scalar qitems and of the base qvalus of all progression-vector qitems is incorporated into this xbase.

Xitem: The special indexing variable.

TABLE 3: FLAGS USED IN THE INDEXING ROUTINES

Throughout the indexing procedures a collection of sixteen flags--'XFLAG'--is used to record and decode the status and nature of the operation. These flags are contained in a single halfword. The flag name in general contains the name of the item (q, r, or s) described by the flag and the name of the alternative states ('1' or '0') for the described condition. The flags are:

```
B0--xqfr:  (if xrav=aray) qitem is fake or real:
    '1'--fake:  Xmode=(scal,apvv,vect)
    '0'--real:  Xmode=list
B1--xqvs:  Qitem is vector or scalar:
    '1'--vect:  Xmode=(apvv,vect,list)
    '0'--scal:  Xmode=scal
B2--xoev:  Index origin is error or valid:
    '1'--erro:  Index origin is erroneous
                (forces xooz=ones)
    '0'--vali:  Index origin is valid
B3--xooz:  Index origin is ones or zero:
    '1'--iorig is '1'
    '0'--iorig is '0'
B4--xrcs:  (if xrsf=stor) ritem is copy or same:
    '1'--copy:  Ritem must be copied before
                indexing (ritem=(syno,apvv) or
                stype is higher than rtype)
    '0'--same:  Ritem may be indexed directly
B5--xrds:  (if xrsf=stor) rtype-stype are diff
           or same (after any rtype conversion):
    '1'--diff:  Rtype and stype differ
    '0'--same:  Rtype and stype are the same
B6--xsvs:  (if xrsf=stor) sitem is vect or scal:
    '1'--vect:  Sitem is not (pseudo)scalar
    '0'--scal:  Sitem is (pseudo)scalar
B7--xrcd:  (if xrsf=stor and xrcs=copy) rtype-
           stype are diff or same (before any
           rtype conversion) (forces xrcs=copy):
    '1'--diff:  Rtype and stype differ
    '0'--same:  Rtype and stype are the same
B8--xrav:  Ritem is aray or vect:
    '1'--aray:  Ritem is an array or is a vector
                indexed with an elision
    '0'--vect:  Ritem is a vector
B9--xqpv:  (if xqvs=vect) qitem is prog or valu:
    '1'--prog:  Xmode=apvv or xmode=list=prog
```

'0'--valu: Xmode=vect or xmode=list=valu
Ba--xrpv: Ritem is prog or valu:
'1'--prog: Rdesc=apvv
'0'--valu: Rdesc=(vect,aray)
Bb--xrss: (if xrsf=stor) ritem is shared assign
'1'--shar: Ritem is shared variable
'0'--solo: Ritem is not shared variable
Bc--xrsf: Ritem is stor or fech:
'1'--stor: Ritem is stored
'0'--fech: Ritem is fetched
Bd-bf--xtype: Type of indexing (if xrsf=fech then rtype, and if xrsf=stor then rtype after any conversion)

---

3. The Routines Used in Indexing.

space

The objective of these indexing routines is speed in the execution of indexing at the expense of program storage or simplicity in initialization. Much of the complexity of indexing operations results from the attempt to detect, decode, and execute the four distinct xmodes. But each of these modes invokes only the minimum initialization processing required. For example, no special indexing variable is constructed unless there are two or more non-scalar indexing variables and in these cases such an item is mandatory. And the executors, given the compacted and optimized parameters which this initialization generates, are extremely compact in control storage and extremely optimized in execution iteration.

Indexing operations are executed by a sequence through several routines. HII analyzes the operation, collects and records information about it, and passes control either to routine HIA if ritem is an array or to routine HIV if ritem is a vector. These routines reduce the indexing information into one of the standard xmodes, check the conformality of ritem and qitem, and pass control either to routine HIF if ritem is fetched or to routine HIS if ritem is stored. HIF obtains space for the result and HIS, after obtaining a copy of ritem if ritem may not have sitem stored into it, checks the conformality of sitem and qitem. These both then pass control to routine HIE. HIE executes directly scalar indexings and invokes HIX to execute batches of vector indexings.

Three other routines--monadic reverse, dyadic reverse,

and dyadic rotate--branch into the indexing routines to complete their operations. These other routines initialize flags and registers as required. Their processing is then identical to that of standard indexing operations.

4. The Indexing of Arrays.

The routine HIA handles the case of array indices. It reduces the index, Qitem, from its external form to one of the four standard index modes. The following describes the procedure:

The xmode to be used is already known because of the scanning performed by HII and is determined by the xflag:

Xflags			Xmodes
Xqfr	xqpv	xqvs	
Fake	....	Scal	scal
Fake	valu	vect	vect
Fake	prog	vect	apvv
Real	valu	vect	list=valu
Real	prog	vect	list=prog

The special indexing list 'XLIST' is a compacted and optimized representation of the separate index tokens specified for the indexing operation. It collects in one place all the information required to execute the requested fetching or storing. Xlist has the description and shape vector of the concatenated nonscalar indexes (the shape of the result for fetching and the conformality requirement on sitem for storing).

The data space within xlist has two sections: A set of line items 'XLINE's, each representing an execution iteration loop (much like a set of nested do-loops) for an indexed dimension; and a set of index values 'XVALU's copied form the qitems for which qdesc=(vect,aray) and not qdesc=(scal,apvv) and adjusted for the index origin and the dimension in which they index.

The xlines are the first data items following the desc-name word in xlist storage. Xlines are ordered with the outermost (left-most) dimension appearing first and the innermost (rite-most) dimension appearing last. Each xline contains four words. There are four forms of xline (depending on whether the line represents a progression vector (xline=prog) or a vector of values (xline=valu) and

on whether the line is the innermost (xline=inne) or not (xline=oute)):

```
Oute-valu:  80MMMMMM XXXXXXXX XXXXXXXX XXVVVVVV
Oute-prog:  40MMMMMM XXXXXXXX XXXXXXXX 00SSSSSS
Inne-valu:  00NNNNNN XXXXXXXX XXXXXXXX XXUUUUUU
Inne-prog:  00NNNNNN XXXXXXXX XXXXXXXX 00SSSSSS
```

The data within the xlines is:

'MMMMMM': (qelem-'1')*'4' iteration counter for the dimension
'NNNNNN': Qelem iteration counter for the dimension
'SSSSSS': Xstep index increment in elements on ravel(ritem) represented by an increment of qstep in qvalu on the dimension
'UUUUUU': Offset from xaddr at init to first xvalu for the dimension
'VVVVVV': Offset from xaddr at init to final xvalu for the dimension

The xvalus are the final data items preceding the shape vector in xlist storage. Xvalus are stored in descending order within each dimension and the innermost dimension is stored first. Each xvalu is the origin zero index in ravel(ritem) of the element implicitly addressed by that qvalu in the indexed dimension--it is the origin zero product of the qvalu and the dimension multiplier xwate for the indexed dimension.

Dimensions are processed from the innermost to the outermost and certain data items are accumulated during this process. Zelem is the number of elements involved in the indexing operation and it is the product of the qelems for the nonscalar qitems. Xwate is the number of elements in ravel(ritem) represented by an increment of one in an index value in a dimension--it is the dimension multiplier applied to all indexes in a dimension to alter them from indexes on a dimension in an array into indexes on the ravel of the array--and it is the product of any innermore dimension sizes. Those of these accumulators which are equal to '1' on the innermost dimension are instead initialized with a negative number--this is tested at all points where a reference is necessary and this obviates many

multiplications.

VIII. Conclusions and Recommendations.

A. Conclusions.

In this section are identified the specific hardware, software, and environmental features which affected this effort, either negatively or positively. The intent is to make comments which can be relevant to other complex programming projects. The main conclusions are:

(1) Feasiblity.

It is possible to write a complex machine feature in microcode, given the right environment and tools. The System 370 Model 135 APL Assist met its goals of speed and space.

(2) The Importance of Interactive Development.

Interactive development using VM/CMS provides an excellent vehicle for almost all aspects of research and development activities.

(3) Direct Translation of Microcode is Risky.

A direct microcode translaticn from one machine to another is not a straight-forward process. This is particularly true if optimized results are desired. The direct MPL145 to MPL135 translation resulted in a program that was twice as large and twice as slow as a design native to the Model 135.

(4) The Importance of the use of Simulators.

It is not only possible but even desirable to develop and debug microprograms using only simulators. A well-written simulator magnifies the microprogrammer's productivity beyond all reasonable expectations. The fact that certain peripheral aspects of the machine are not simulated is of little consequence compared to the many additional central facilities offered by the simulator.

(5) The Importance of High Level Languages.

A higher-level-language structured compiler is an

important tool for the production of microcode. The use of structured programming prevents many label and branch errors and improves the legibility and modifiability of the code. The compiler simplifies the programming job since one may use application symbolic names for machine facilities and system variables.

(6) The Importance of Optimized Tools.

The software supporting microcode development should be optimized for the code-and-test environment rather than the manufacturing environment. This can also result in faster manufacturing since the quality of the resulting code should be better.

(7) "Inside-out" Programming is best for Performance.

The best implementation strategy for microcode seems to be to optimize from the bottom to the top (or inside out). In doing so one examines the interfaces thus created until a simple, consistent set is found. The subroutines thus created then look like new and more powerful op codes in which the upper levels of the program may be implemented. If the interfaces in each case consist only of the variable parameters for the subroutine, and these are maintained in accessable and invariant locations, then the structuring will be efficient in speed and space, and will be logically and physically congruent.

Separating the code into small free-standing subroutines works excellently in terms of productivity, maintainability, and expandability. The executor concept involving independent non-decision-making single-purpose subsubmodules especially facilitates development. Partitioning the remaining processing into two disjoint sections which respectively select executors and invoke executors greatly reduces the complexity of the control logic.

(8) Small Programming Teams yield the best Code.

If a highly optimized program is desired, as was the case with APL/135, a small team is more productive than a large group. A small team can make the necessary design tradeoffs much more readily. Even major changes in design topology are possible at a late date.

(9) Documentation should be done late in the Project.

The final code including its comments should be the main documentation of a program. The comments should be done as late in the devlopment cycle as possible to make sure that the documentation is consistent in format and content with the delivered programming. This is a dangerous recommendation and could cause serious trouble if the main structure of the program itself were not well organized and well written.

(10) Accurate Time Estimates are still hard to make.

In spite of all advanced programming tools and computer science, in spite of having a well-defined program and a well-defined goal, it was still difficult to make a accurate estimates as to how long various tasks would take.

B. Hardware Recommendations.

(1) The Need for a Relocation Trap.

The hardware which underlies an emulator in a virtual-memory machine must have a competent trap for relocation exceptions. When a storage reference to an untranslatable main-store address occurs, the hardware should trap to a translation routine, recording the address from which the trap was made, but modifying none of the active working registers. These latter can be saved by the relocation routine if necessary but they must not be altered by the trap itself. Through a small hardware modification an adequate trap for the Model 135 was created. The bare Model 135 does not record the location of the trapping microinstruction on a relocation exception. It was designed to support only the "trial fetch" philosophy needed for emulation of System-370.

(2) The Need for extra Hardware Latches.

The hardware should have immediately accessable settable-resettable-testable latches for microprogram use. Often one needs to remember some condition for a short while. There rarely is a register available to hold this scratch information. Some individually maintainable hardware latch bits (eight is a good number) would simplify this greatly. They must, however, be allocated solely by the local-level microprogram. There should be no demand to save and restore them in subroutines.

(3) The Need for better Bit and Byte Instructions.

The hardware should have reasonably powerful instructions for masking and shifting data bits embedded in bytes and halfwords. The mask and shift values should be immediate literals within the instruction. Some of the table indexing was awkward because it is difficult on the Model 135 to extract from the operation and variable descriptions the data bits needed as table indexes, and then to position them in the appropriate position of the index field. Some of this was solved by building awkward tables, aligning the entries to the vagaries of the initial data-bit alignment, but this scatters the table elements through sections of control storage and decreases legibility and maintainability.

(4) The Need for an Adequate "Branch and Count".

The hardware should have a hardware loop-counter which can be decremented and tested in a single branch instruction (exactly analogous to a System-370 'BCT'). The Model 135 has such a counter, but its value at any instant can not be recovered from the counter itself. Since on an interrupt the emulator needs to save the value of all the hardware components it employs, the counter was useless. (The Model 135 System-370 emulator uses the counter when it knows no interrupt is possible or when it can recover the value from other data conveniently available.)

(5) The Need for a "Branch and Link" with Return.

The hardware should have an efficient BALR microinstruction. In the central loops the code looked like 'BALR BALR BALR BALR BCTR'. Unfortunately each of these instructions actually was a pseudosubroutine call. The BALR is implemented on the Model 135 by

```
          EXEC7   BAL)W6

EXEC7         W4/W5=CWR7
              RETURN)W5
```

where the first instruction sets the return address (executes the '.AL.') and the pseudosubroutine EXEC7 loads work storage (from which a branch can be made) and branches (executes the 'B..R'). A separate EXECn subroutine was needed for each of the CWRns holding the branch-target field

used in a pseudo-BALR. Worse, executing three instructions instead of one for each action in the innermost loops of an emulator is costly. (It still was much faster, and much smaller, than the interpretive execution that it replaced.)

(6) The Need for Ample Microcode Control Storage.

Control storage should be large enough for the project. In this case the lack of adequate storage greatly increased the total amount of work necessary. Three methods were used to compensate for the lack of storage:

(1) Some functions which might have been microcoded were left to software. The rarely used routines were omitted whenever possible.

(2) All microcode was carefully optimized for storage utilization. This was a tedious waste of programmer time.

(3) Coding artifices which served no other useful purpose were invented to compress the code further. They cost time and reduced the flexibility of the code.

C. Recommendations Concerning the Assembler.

(1) The Need for Convenient Handling of N-Way Branches.

The assembly language for microprograms should facilitate, rather than refuse to accept, some of the coding techniques needed to save space or gain speed. For example, the APSS assembler requires that all sixteen targets of a sixteen-way branch be defined even if the data underlying the branch makes it impossible for many of them to be reached. This is wasteful of space (or the programmer writes as hard hexadecimal constants the instructions required to implement the branch).

(2) The Need for more control over Branch Address Assignment.

The assembler should be able to float the address assignments of assembled microinstructions to meet the branch limits of the instructions and the specifications of the microprogrammer. The programmer often must use bits in an address as flags. For example, bits in the address of a subroutine obtained by a table index can be used to indicate attributes about that subroutine (or the cases it is

designed to handle). Bits in the address from which a subroutine is called (via a branch-and-link) can be used to contain parameters for the subroutine. Often only certain bits need be fixed in these cases. Some combinations of fixed bits the assembler will allow to float (it will assign the others arbitrarily to meet the requirements of other code modules) but others it will not. The programmer must fix these addresses absolutely, with the result that control storage becomes unnecessarily chunky, discrete, and fragmented.

APPENDIX I:

SIM135 - SYSTEM 370 MODEL 135 MICROCODE SIMULATOR

SIM135 is an interactive simulator written in System-370 assembler language and executing under VM/CMS. The simulation covers the physical hardware facilities of the Model 135 accessible to the microprogrammer. It omits certain obscure facilities, particularly the outboard ends of the native adapters. The simulated machine contains System-370 GPR's, FPR's, and processor storage as well as Model 135 microcode control registers, work registers, some of the external registers, local storage, control storage, and latches.

The description of the individual commands which follows will explain SIM135 from the user's point of view.

High performance was one of the main goals achieved by SIM135. Running on the Model 145 it executes over 4500 simulated microinstructions per second. It takes roughly 550 seconds to simulate one second of Model 135 execution time. Since typical APL test statements execute only a few thousand microinstructions, the simulation time was very small. Even when long traces of complete functions were run (see the TRAX command), the running time was never excessive.

A. Invoking SIM135

The simulator is executed under CMS, and exists as a module 'SIM135 MODULE'. It is invoked by issuing the name of the CMS EXEC file "APLSIM".

The simulator will load its simulated control store from a CMS file called CONTROL MEM135. If the file does not exist, then simulated control store will be set to all zeros.

The simulator indicates when it is ready for use by displaying the message:

CONTROL MEM135 LOADED

following which a CLEAR WS will be loaded. APL mode will

then be entered. The following table describes the simulator commands :

TABLE OF SIMULATOR COMMANDS

(note- "display" can mean to type at user's terminal):

DI	display next microinstruction address
DW	display work store,
DA	display aux store
DC	display control store
DM	display main store
DP	display program (simulator)
DT	display time (accumulated microinstruction time)
AI	alter next microinstruction address
AW	alter work store
AA	alter aux store
AC	alter control store
AM	alter main store
EXEC	execute microinstruction(s)
OLDI	display last microinstruction address
PRINT	print contents of listed facilities, offline
QUIT	terminate simulation program
STOP	interrupt simulation on condition
SHOW	automatically display changed facilities
INIT	initialize facilities
LIST	execute a list of stacked commands
TRAX	write a trace of executed microprogram to CMS
CODE	source statement entry mode (for patching)

B. Format of SIM135 Commands

All commands may be entered using CMS conventions, i.e. the command may be entered in lower case, and the character-delete and line-delete symbols may be used.

TABLE OF COMMAND ARGUMENTS

DI	no arguments
DT	no arguments
AI	new value for microintruction address
Dx	a single hex value, or two values separated by a dash (-).
Ax	a single hex value, or two values separated by a dash (the second value is ignored)

EXEC	n (decimal), or no argument, in which case 1 is assumed.
OLDI	no arguments
PRINT	facility list
QUIT	(CLEAR)
STOP	stop address list
SHOW	ON or OFF
INIT	facility or file, see below

For commands which may refer to aux store (DA, AA, PRINT), the mnemonics GPRn, FPRn, and CWRn are accepted, where GPR0-GPRF correspond to aux 00-0F, FPR0-FPR7 correspond to aux 10-17, and CWR0-CWR7 correspond to aux 18-1F.

C. Description of Individual Commands.

(1) The Display Commands (DI,Dx, etc.)

Workstore is always addressed at the halfword level, and aux store at the full word level. Work store and control store are displayed in units of a half word, while aux store and main store are displayed in units of a full word. Note that where a pair of addresses is used to specify an address range, e.g. DW 3-5, if the second argument is less than the first, then the second argument is ignored. Aux store mnemonics are individually converted to absolute aux store addresses, e.g. DA CWR0-CWR3 is equivalent to DA 18-1B, which displays 4 words, whereas DA CWR0-3 is equivalent to DA 18-03, which displays one word.

(2) The Alter Commands (AI, AW, etc.)

The "alter" commands use only the first argument to determine the starting address of the data to be entered. In the cases of main and control store, the address may be any byte in legal range. Upon entering an ALTER command, the keyboard is unlocked for entry of up to 16 bytes (32 hex digits), which will be stored consecutively starting at the address specified in the command. The digits may be entered with arbitrary spacing. If the number of digits entered is over 32, or if the data would extend beyond the limit of the facility, the excess is ignored. If the digit count is odd, or if invalid digits are entered, an error message is displayed, and the keyboard is again unlocked for re-entry of the data. A carriage return without data cancels the alteration request.

(3) The Print Command.

The facility list of a PRINT command is:

W A a-a C c-c M m-m

where W refers to work store (always 8 registers), a-a is an aux-store address or pair of addresses, c-c is a control-store address or pair of addresses, m-m is a main-store address or pair of addresses. The facility list may be followed by an apostrophe (preceded by at least one blank); up to 75 bytes of commentary may follow the apostrophe and will be printed in place.

The arguments are separated by blanks, and any argument is optional. If the same facility type appears more than once, only the last specification is used.

(4) The EXEC Command

The EXEC command argument is n, a decimal number, which defaults to 1 if the argument is omitted.

When the EXEC command is entered, the simulator simulates n microinstructions, unless it reaches a simulation interruption condition (to be defined later). Upon completion of n microinstructions, the message

LAST CYCLE: k, I=iiii NEXT=jjjj

is displayed, where (1) k is the cycle number of the last microinstruction executed, (2) iiii is the control-store address of the last microinstruction, (3) jjjj is the address of the next microinstruction.

The keyboard is unlocked for further command entry.

(5) The STOP Command

The STOP address list is a list of control store addresses, separated by blanks, at which the simulator is to stop and unlock the keyboard for command entry. Existing stop flags can be turned off with this command. There are three subargument formats:

S addr for stopping after microinstruction

execution.

P addr   for stopping before microinstruction execution.

OFF addr   to turn off stop flags (both types)

'P' and 'OFF' must precede each address to which they apply. 'S' is the default and may be omitted.

For example: STOP 8 24 P 30 OFF 16

would mean stop after instructions at addresses 0008 and 0024, stop before instruction at address 0030, and turn off STOP flags at instruction 0016.

(6) The INIT Command

The "INIT" command may have three forms of argument - the facility form, the file form, and the list form.

The facility form is:

INIT x addr data,data,...

or

INIT I addr

where x may be W, A, C, or M for work store, aux store, control store, or main store, respectively. Addr represents an address at which the initialization is to begin. 'I' refers to the next address from which to fetch a microinstruction for execution when an "EXEC" command is issued.

The command argument data (following the facility address) consists of "units" of hex digits separated by commas. A unit is up to four digits for work and control store, and up to 8 digits for aux and main store. The data is stored starting at the specified address. A comma (with no digits) indicates skip a unit. If the digit count is low in a unit, then the unit is zero-filled from the left.

The file form of the "INIT" command is:

INIT FILE filename

where filename is the name of a CMS file (maximum 7

characters) with file type "DATA". The file may contain images of "INIT" commands, exactly as above (beginning with the word INIT), and may also contain images of "STOP" commands, exactly as above (beginning with the word STOP), and may contain LIST initialization commands as described below.

The LIST form (in an INIT FILE) is:

INIT LIST n

where n is a decimal number from 1 to 127. Following this command may be images of commands as they would be entered from the terminal. The legal commands in a list are DI, DA, DW, DC, DM, PRINT, SHOW. The list must be terminated by an "END" command. At initialization time, only the command keywords are examined for legality; the arguments are checked when the commands are executed.

If INIT LIST n is entered from the terminal, then a message will be displayed indicating the number of commands currently in list n. If list n is empty, the number 1: will be displayed, at which time a command image may be entered into the list. If the command keyword is legal for a list, it will be stacked, and the number 2: will be displayed, etc. To terminate list initialization mode, enter the command "END". If the list is not empty, the first command will be displayed, after which it may be overwritten by entering a new command, or it may be deleted from the list by entering "DELETE", or the next command may be displayed by entering a carriage return. The list edit mode is exited by entering "END".

The "LIST" command has five formats:

```
LIST n addr addr                                  (1)
LIST 0  addr addr                                 (2)
LIST n OFF                                        (3)
LIST n ON                                         (4)
LIST n                                            (5)
```

Format (1) causes the execution of list n whenever any of the addresses in the argument list are fetched as microinstruction addresses during simulation; the list is executed before the microinstruction. Format (2) has the effect of turning off the list flags at the indicated addresses. Format (3) causes list n to be deactivated during simulation. Format (4) causes list n to be

reactivated. Format (5) causes immediate execution of list n regardless of whether it is deactivated.

(7) The SHOW Command

The SHOW command will cause a display of the cycle number and microinstruction address of any microinstruction which causes a change in the contents of work or aux store, or causes a store operation to main or control store (whether or not the new value differs from the old), along with the value of such changes.

SHOW OFF disables the SHOW function.

SHOW ON may be followed by a facility list for selective tracing, for example:

SHOW ON W 0 W 4-6 A 3 A 10-1A C 600-5FFF M 0-7FFFF

which monitors work regs 0, 4, 5, 6, aux regs 3 and 10 through 1A, control store in the range 0600 through 5FFF, and main store in the range 000000 through 7FFFF.

Notice that only one control store range and one main store range may be specified in the argument list.

In addition, the argument list may include "P", for offline print, and/or "T", for full trace, i.e. trace branches. If "T" is used, it should precede the list of facilities whose changes are to be displayed.

The command "SHOW ON T"

would produce an instruction trace without showing the values of any facilities.

(8) The QUIT Command

The QUIT command signals the end of the session. The QUIT command may take the argument "CLEAR", which causes a CLEAR WS to be loaded, provided the simulator was entered via 'AO'. If it as entered by an external interrupt then QUIT, with or without argument, is equivalent to "APLX".

(9) The TRAX Command.

The TRAX command provides a trace of microaddresses in an external CMS file (e.g. disk). This provides a faster tracing facility than SHOW. Also one can use other programs to study the results of a long trace. It was used to locate high use kernels in the APL emulator for special optimization. It was also used to verify that all the correct branch paths were executed in test cases.

(10) The CODE Command

The CODE command put the simulator in source-statement entry mode. It thus provides a direct capability to patch the microcode being simulated by providing the hex code corresponding to the source statements entered. The inputs are regular assembly language statements. "END" terminates the use of CODE. For example:

```
User:     CODE
User:     W0/W1=MS(W01)FW SK
 SIM:     0246
User:     W0-3=W0-3,OR,GPR(W6D0)DW
 SIM:     0793
User:     END
```

D. THE SIM135 TO CMS/APL INTERFACE

The simulator is interfaced to CMS/APL so as to allow simulation of microcode while using the APL software. The interface allows terminal editing while in the simulator, but disallows it in APL mode.

A clear workspace will be loaded, and the system will be APL mode. At this point (assuming the 145 APL emulator is loaded) the system should behave as a normal APL system.

The simulator is entered by depressing the "attn" key and issuing the CP command "EXT", for external interrupt. This causes the simulator to be entered, at which time all ordinary simulator functions are possible.

Three commands have been added for use with APL:

APCT set microinstruction count (on "A0" entry) (requires decimal argument).
APSM activate simulation of "A0" op code

APLX exit to APL

APLX has three forms:

APLX (no argument): resume APL from external interrupt

APLX addr addr ...: return to APL upon execution of microinstruction at any indicated address (the microinstructions at these points should be "IFETCH" microinstructions)

APLX TO addr: immediate return to APL via micro instruction at indicated address (which should be an IFETCH).

Note that the no-argument form is valid if and only if the simulator was entered via external interrupt, while the "TO" form is valid if and only if the simulator was entered via the 370 op code "A0".

APLX (second form) may be used in an INIT file.

E. LINKAGE CONVENTIONS.

Entering the simulator via the external interrupt causes the real general purpose registers to be saved. They will be reloaded upon exit from the simulator via "APLX" command (except register 14).

Entering the simulator via 370 op code "A0" causes the real GPR's to be stored in the simulated GPR's, the real floating point registers to be stored in the simulated FPR's, and the real condition code to be set in the simulated condition code. When returning to APL, the real GPR's are loaded from the simulator, the real FPR's are loaded from the simulator, and the simulated condition code is set in the real machine.

It is important to note that entering the simulator via "A0" causes the microword at control store location 0002 to be overlaid with an IFETCH which is used to begin simulation; therefore, location 0002 should be avoided in the assembly of microcode to be simulated together with the APL system.

Note that the branch address following an IFETCH is

determined by hardware, and in the case of "A0" the branch address is 0304.

Upon returning from an "A0" entry the address at which to resume APL execution is calculated from the contents of work registers W4/W5. If these registers contain the resume address, and the simulator is exited via IFETCH, the correct address will be calculated. IFETCH updates W4/W5 by +2 or +4 depending on whether the 370 op code which is fetched is an "RR" type or not. Therefore, great care should be exercised in exiting via any microinstruction other than IFETCH. Note also that IFETCH sets the condition code to be reflected to the real Model 135.

The APSM command initiates simulation by invalidating the checkword used by the 145 APL emulator for "A0" op codes. This checkword is part of the workspace, and is reloaded when a workspace is loaded.

APPENDIX II:

THE ASSEMBLER FOR MODEL 135 MICROCODE (ASM 135)

One of the major microprogrammning development tools was ASM135, the microcode assembler and linkage editor. It was part of the interactive code-and-test system described in Sections III and IV. Since it was built on the original APSS batch system, it is proper to start with a description of APSS, then describe how it was modified to become ASM135.

A. The Original APSS.

1. Organization and Purpose.

APSS (A Processor Support System) is a library of OS/360 application programs designed to aid the development and generation of microprogram control store information for the 370/3135. The programs were written at the IBM UK Laboratories in Hursley.

APSS provides two important engineering services, namely the maintenance of program records and generation of the data that is to be written onto the initial microprogram disk of the 3135 console file. (See Figure 1). It also produces the associated field engineering microprogram documentation. The various APSS programs are normally involked by catalogued procedures of job control language (JCL) statements.

2. Use of the APSS Microprogram Master Tape

The 370/3135 microinstruction statements are released by SCD to the manufacturing plants on an OS/360 magnetic tape data set. This data set is referred to as the microprogram master tape (MMT). The MMT contains data comprising the source microinstruction statements, encoded control word bit patterns and various tables relating the feature bill of material (BM) numbers to microprogram names that are recognised by APSS.

The MMT can be altered using the APSS maintenance

programs. Such changes would normally be made as a result of a request for engineering action (REA). The maintenance programs merge the data obtained from the current MMT with the REA information provided on punched cards and produces as an output a new MMT; the current MMT remaining unchanged.

The incoming micro instruction changes to the MMT are translated into a corresponding two byte control word bit pattern and recorded with the eighty byte card onto the new MMT. Microinstruction branches are not completely translated because the absolute location of the microprogram is nct known at this stage of processing. The translator program scans each symbolic microinstruction and searches a dictionary of microinstruction primitives and prototype templates contained within the APSS program library.

3. How APSS is used in Manufacturing.

The MMT contains all of the 3135 basic and feature microprogram data. The console disc file however, must contain microprcgram corresponding to a customers individual control store requirements. The selection of this microprogram is specified by punched cards containing production control information. This includes feature BM numbers corresponding to the customers individual feature requirements, CPU serial and order number. APSS attempts to match each BM number from the production control cards with a corresponding feature BM number recorded in a table on the MMT.

The entries in this table are derived from the 3135 microprogram bill structure and represent every possible feature that requires microprogram. If a match occurs, further information is obtained from the MMT which identifies the micro program associated with the matching BM number. This information is accumulated until every BM number present on the production control input is matched. APSS then commences to read the desired microprogram from the MMT: the selection is written onto an intermediate data set.

While the program is reading the MMT, various conditional selection statements will be encountered interspersed with the microinstructions. These statements contain mnemonic operands that are exact counterparts of the BM numbers that occur in the BM tables mentioned above. If

IBM INTERNAL USE ONLY

one of these BM's is present on the production control cards then the conditional selection statement is obeyed and the following microinstructions are either skipped, or included in the selection.

4. The Assembly Process.

The microinstructions recorded on the intermediate data set represents a sub-set of the total MMT and exactly suit the customer's individual control store requirement. During the selection process, the microinstruction branches are completely cross-referenced. This is a necessary preliminary to the allocation of control store addresses and generation of the microprogram listing.

The control store address assignment programs analyse the selected source microinstructions and allocate the necessary control storage space while checking the network of branch microinstruction connections. Subsequent programs complete the partially assembled bit patterns of the branching microinstructions mentioned above and sort the microinstructions into control store address order. The final stages of the generation creates the disc data by attaching track/sector control information to the sorted and encoded microprogram. The print data comprising the micro program listing and control store contents map are also produced at this time.

The output from APSS is recorded on the print image data set and the microprogram image data set. These data sets are subsequently used to write an disc for the console file and the supporting field engineering documents. The actual writing of the disc and production of the documents is not a part of the APSS system. On-line output provides detailed status information about the run and indicators and failures that may have occured.

5. Summary of the Data used and produced by APSS.

An MMT contains a leader (X) and trailer (Z) record, as well as CSECTs, which are of the following types:

Bill of Material CSECT
Load CSECT
History CSECT
Configuration CSECT

Routine CSECT

The CSECTs are made up from cards, which are of the following types:

Title card
End Card
Comment card
Routine card (Microinstruction)
Load CSECT card
Bill of Material card
Configuration card
Assignment Control card
Selection Control card

These cards are CSECT detail cards. These cards plus control cards (CREATE| UPDATE| DELETE| AND LIST) provide input for the create/ update phase.

Input for the assignment phase consists of a MMT and Production Control (PC) cards. There are 6 types of PC cards. They permit specifying BM, system identification, CSECT| specific MMT, and feature. Those cards are used to select some or all of the cards on a MMT to create a selected file corresponding to a specific 3135 being manufactured. This selected file then ha assigned to all instructions. Program listings and console file IMP disk images are produced.

B. Modifications made to the APSS System.

1. The Creation of the OS Version.

The first assembler used was an OS version. Since APSS was available in both symbolic and object module form, the main effort involved deleting functions and packaging the program. APSS contains 2 procedures which in turn contain 4 job steps. It utilizes 40 data sets, some in multiple job steps.

The first decision was to eliminate the use of the MMT. Most of this project involves assembly of new or major reworks of present code. The code is very unstable, and the concept of a MMT did not seem practical in a development atmosphere. This decision permitted the deletion of the UPDATE job step and the PROCON and SELSIN phases of XASSIGN. It also permitted deleting the SORT job step.

The second decision was to eliminate the microprogram image data set, and produce instead a data set for input to the SIM135 simulator. This permitted deleting the MINFORM phase of XASSIGN.

The next decision was to imbed FORMAT as a subprogram of XASSIGN to produce a single job step. It was now a single program which accepted a subset of APSS input and produced as output listings and simulator input, i.e., an assembler. The number of data sets had been reduced to 10. This system ran under OS and was in use for several months.

2. The CMS Version.

The second version of the assembler was a CMS version. As the use of the assembler grew, the OS overhead became excessive, and a CMS version offered the prospect of less overhead and greater speed. A second advantage was the prospect of several people being able to use ASM 135 simultaneously. Only one OS machine was active in the CP environment, which meant only one 135 microcode assembly at a time was possible.

Two major changes were made. The first was the replacement of OS I/O by CMS I/O. The second change was in the method of main storage allocation. In the OS version, each subroutine is dynamically loaded on demand, and the space released when the subroutine is finished. This is the standard method of conserving main memory in a standard system. However, if CP/CMS is to be utilized, there is no advantage to this dynamic loading and releasing. The same effect is obtained by the paging performed by CP. Thus it was decided to link all subroutines together as one large core load and let CP perform store management.

Another effort was made to incorporate subroutines inline. Again simplification and common notation were the desired objectives. It had been observed that code was present in subroutines which was needed in the original system, but which was never executed in the new system since some functions had been deleted at a higher level. Multiple independent subroutines make detection of this redundant code difficult. The number of subroutines was cut to 13 and the number of data sets to 6.

The programming techniques used in APSS made its

modification relatively easy. Common labeling schemes, common names for identical constants and variables, and the many comments made the work easier. The general procedure involved in substituting subroutines in line consisted of replacing the CALL by the body of the subroutine, adding the variables and constants to the calling programs variables and constants, and adding DSECT's to the calling DSECT's. Global changes had to be made to convert parameters into arguments. Addressibility and register usage had to be arranged for, as did exiting. In many cases, the exit of a subroutine was physically at the end of the code, so that it could be deleted, producing a fall-thru case. In other cases, a branch to the return point was necessary. The parameter initialization was removed also. The work was greatly facilitated by the use of the CMS editor.

The last version incorporated only one basic change. It was observed that most of the time in the assembly process was being spent in the statement scan during the first pass. This was a subroutine which had never been modified since it provided a necessary function, and did not superfically have redundant facilities. However a replacement subroutine was coded which turned out to be faster by a factor of greater than 10. This was imbedded.

Many minor changes had been made in other sections also, but not actually incorporated. With the replacement of the scanning subroutine, however, the other changes were incorporated. The result is ASM135.

GLOSSARY OF ACRONYMS AND ABBREVIATIONS

ABEND - Abnormal end.
ACTxx - Actions taken as a result of SCAN.
APLEC - S/370 opcode invoking the APL assist. (see A0)
APSS - "A Processor Support System". Mod 135 assembler.
APV - Arithmetic progression vector.
ASCAN - Current scanning address.
ASM135 - VM/CMS Asembler for Mod 135 microcode.
ASTACK - Current stacking address.
A0 - Special S/370 opcode. Invokes the APL assist.
BALR - Branch and link with return op code.
BM - Bill of materials. List of machine features.
CCODE - Condition code.
CHAR - Character type variable.
CONVE - In copying--type to which variable is converted.
CPU - Central Processing Unit.
CSECT - Control section. A named microcode routine.
CWRx - Mod 135 microcode 8 working registers. (CWR0-7)
DAT - Dynamic address translaticn.
DLEXE - A Mod 135 external immediate. Signal on the control console.
DSC - Directly addressable common control store.
DTRAP - A Mod 135 external immediate. Set when APL is being emulated.
EC-Level- Engineering change level.
EXECU - Execution. Part of the FETCH-EXECU-STORE loop.
FAKE - Short name for "pseudo-scalar".
FETCH - Fetch. Part of the FETCH-EXECU-STORE loop.
FIB - Function Invocation Block. Is put on the stack for a function.
FINIS - End of the FETCH-EXECU-STORE loop.
FNAME - A function's internal name.
FPRx - S/370 8 Floating point registers. (FPR0-7)
GPRx - S/370 16 general purpose registers. (GPR0-F)
Hxx - Three letter names used for the APL-135 microcode routines or CSECTs (See Table 1).
HIPE - Highest operand precision (of logi, inte, real)
IFETCH - A Mod 135 microcode instruction. Beginning of a S/370 operation.
IMPL - Initial micro-program load.
INTE - Integer type variable.
IPL - Initial program load.
IORIG - Index origin in APL. (0 or 1)
IREQU - Mod 135 external immediate. Quantum-end request.
JCL - Job control language statements for OS.
LBRAK - Token representing the left bracket of index.

LITEM - Left operand for a dyadic APL operator.
LNAME - Name of left operand of an APL operator.
LOGI - Logical type variable.
MMT - Microprogram master tape.
MPL135 - High level language compiler for Mod 135.
OCODE - Op code. Usually means an APL operator.
OCODU - Op code. Second step of inner/outer product.
OCODV - Op code. First step of inner/outer product.
OMODE - Type of arithmetic for execution of an APL operator (logi, inte, real).
PRI - Pending real interrupt (hardware interrupt)
PSW - S/370 Program Status Word.
QITEM - An intermediate operand in the execution of a complicated operation.
QITEM - In Indexing--the indexing variables.
RBRAK - Token representing the right bracket of index.
REAL - Real type variable.
RITEM - In Indexing--randomly accessed variable.
RITEM - Right operand of an APL operator.
RNAME - Name of right operand of an APL operator.
SIM135 - Simulation program for executing Mod 135 microcode programs.
SITEM - A shared or distinguished variable.
SITEM - In Indexing--a sequentially accessed variable.
SITEM - In copying--the source item.
STORE - Store. Part of the FETCH-EXECU-STORE loop.
TITEM - In copying--the target item.
TLB - Translation look-aside buffer.
TNAME - Target name.
VDELT - Delta--no. of elements from the first to final value of an APV.
VELEM - Number of elements in APL indexing.
VM/CMS - Virtual machine conversational monitor system
VSAPL - Version of APL running under VS.
VSTEP - Step or increment in APV indexing.
WS - Workspace in APL.
WWR - Mod 135 microcode 8 working registers. (WWR0-7)
XADDR - Internal address of an API Variable.
XARGn - Internal synonym names for arguments of functions.
XBASE - Index of the first indexed element.
XDESC - Internal descriptor of an APL Variable. (Scalar, Vector, Array).
XELEM - An element (value) of an APL vector or array.
XFAKE - Dummy variable used to make a monadic operator into a dyadic operator.
XFLAG - 16 index flags. Used to record the status of and to decode index ops.

- XITEM - Any operand of an APL operator. (e.g. LITEM, RITEM, LNAME).
- XLIMI - Number of Indexed elements.
- XLINE - Indexing--represents an index iteration loop.
- XLIST - Indexing-- control list.
- XMASK - A byte used in logical arithmetic operations.
- XNAME - Internal name of an APL variable.
- XPARM - Parameter for an indexing operation.
- XSHAP - Shape of an APL vector or array variable.
- XSTAK - Logical order of the top 4 stack items. (Normally '197E').
- XSTAT - Registers to maintain address of a third variable in an operation.
- XSTEP - Index step or increment as computed from XLINE.
- XTYPE - Type of an APL Variable (part of XDESC)
- XVALU - Value of an APL variable. Sometimes means first value or next value of a vector or array.
- XWATE - Indexing weight. The no. of elements represented by a step of one in an index.
- ZITEM - Result variable from the execution of an APL operator.

REFERENCES

A. Hassitt and L. E. Lyon, "Efficient Evaluation of Array Subscripts of Arrays", IBM Journal of R. & D., p.45-57, January 1972.

A. Hassitt and L. E. Lyon, "The APL Assist", IBM Palo Alto Scientific Center Report Number ZZ20-6428, Feb 1975. (IBM internal use only).

IBM Corporation, "APL/CMS User's Manual", IBM Order Number SC20-1846, July 1974.

IBM Corporation, "IBM Virtual Machine Facility/370: Introduction", IBM Order Number GC20-1800.

IBM Corporation, "VS APL for CMS: Terminal User's Guide", IBM Order Number SH20-9067, January 1976.

John R. Walters and Daniel L. McNabb, "MPL/135--A Programming Language for System/370 Model 135 Microprogramming", IBM Palo Alto Scientific Center Report ZZ20-6415, March 1972. (IBM internal use only).

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) : Randolph G. Scarborough
Harwood G. Kolsky
Norman S. Gussin

2. TITLE : MICROCODING APL: The System/370 Model-135 APL Assist

3. ORIGINATING DEPARTMENT
IBM Palo Alto Scientific Center

4. REPORT NUMBER
ZZ20-6433

5a. NUMBER OF PAGES
76

5b. NUMBER OF REFERENCES
6

6a. DATE COMPLETED
August 1976

6b. DATE OF INITIAL PRINTING
January 1977

6c. DATE OF LAST PRINTING

9. SUBJECT INDEX TERMS
Microprogramming
Machine Language
APL
Performance
Program Development
IBM System/370 Model 135
07 - Computing
21 - Programming

7. ABSTRACT :

The APL Assist is a hardware feature which enhances the performance of APL systems by executing a major subset of the APL language directly in microcode. The APL135 Assist is now available on the IBM System/370 Models 135 and 138. It is invoked by a new System/370 instruction called "APLEC". The feature can be used under standard operating systems.

This report describes the details of the APL135 microcode, emphasizing the importance of executors and index tables as a technique that yields high performance code that is easy to understand and maintain. The report also describes the interactive development system designed for this project and contrasts it with the original Mod 135 batch system. The report also discusses some of the lessons systems. Perhaps the conclusions and recommendations can save future microcoding groups some time and trouble.

8. REMARKS :

IBM INTERNAL USE ONLY

Palo Alto Scientific Center, P. O. Box 10500, Palo Alto, California 94304

10267794