2

# MASTER

PREPRINT UCRL- 82652

CONF-790548--1

# Lawrence Livermore Laboratory

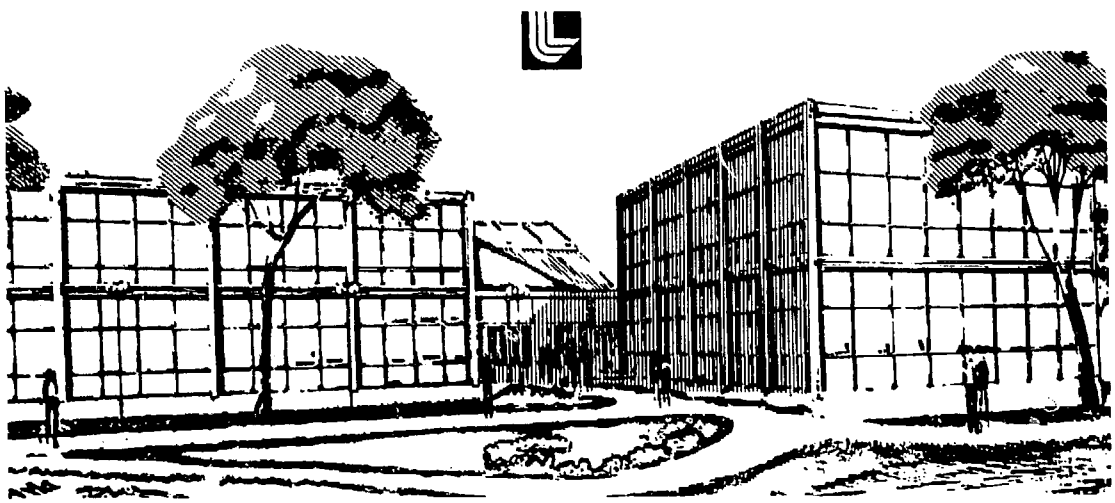
## AP190L and PDP-KI10: A HARDWARE/SOFTWARE MEASUREMENT REPORT

Neil Maron and George G. Sutherland

March 8, 1979

This paper was prepared for presentation at the Third Annual FPS Users Group Meeting, Lake Buena Vista, FL. May 6-9, 1979





# AP190L and PDP-KI10: A Hardware/Software Measurement Report

Neil Maron

and

George G. Sutherland

M-Division
Lawrence Livermore Laboratory
Livermore, California 94550

March 8, 1979

## ABSTRACT

This report discusses an AP190L array processor (manufactured by Floating Point Systems of Beaverton, Oregon) interfaced to a PDP-10 (Digital Equipment Corporation, Maynard, Mass.). After AP software installation, an analysis of the overhead was performed. The results of these measurements and some conclusions will be presented. An AP monitor and software interface were written to minimize the overhead from the PDP-10. A vector extension to the FORTRAN language called "FIVE" was developed to increase user access to the AP. Some the problems associated with defining and implementing FIVE will be discussed. Its successes and limitations will be reviewed.

---

This work was performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore Laboratory under contract number W-7405-ENG-48.

Array processors offer a cost effective alternative for large scale scientific computing. In these days of diminishing budgets, one must seriously consider the AP as opposed to an expensive, general purpose CPU. At Lawrence Livermore Laboratory, California, the Department of Energy is conducting a program of research to develop energy from a process called magnetic fusion. This research, in part, involves the computer modelling of fusion plasma theory. These computer programs are complex and consume large amounts of time. By connecting an AP190L to the PDP-10, the Laboratory has greatly augmented its scientific compute capacity.

The FPS 190L AP was selected because of its unique architecture. The word size of the AP exceeds that of the PDP-10 by two bits and dramatic speed ratios are obtainable for simple arithmetic (see Table 1). However, can one achieve the ratios that are theoretically possible? In the following remarks, the results of this PDP-10 and AP coupling will be reviewed.

Table 1.

| Operation | PDP-KI10[1] | FPS AP[2] |
|---|---|---|
| Addition | 10.8us (2.45-6.26) | 1.3us (.167-.333) |
| Subtraction | 9.9us (2.62-6.43) | 1.3us (.167-.333) |
| Multiplication | 11.1us (3.65-4.84) | 1.3us (.167-.500) |
| Division | 16.4us (7.49-7.95) | 1.7us (3.8) |

(1) The PDP-10 numbers stated were obtained by timing loops 10000 long. The numbers in parentheses are the quoted values from the PDP-10 hardware book.

(2) The AP numbers stated were obtained from counting cycles since the FPS AP is a syncronous machine and cycle times are absolute, based on the clock frequency of 167ns. The numbers in parentheses are the minimum time per result possible to the time required for one result.

It seems that there should be nearly a factor of 10 speed enhancement by doing arithmetic on the AP. One problem is that the AP and PDP-10 do not share a common memory and hence the data must be shipped back and forth. The AP is connected to the PDP-10 via a channel type interface. Communication between the AP and PDP-10 takes place over the I/O bus side of the interface and DMA transfers of data is done on the memory bus side. The interface was designed to get or put a word of PDP-10 memory at full memory speed (approximately 900ns). The PDP-10 memory is multiported and the AP channel accesses one of these ports. The interface handles all format conversion between the various floating point formats, integer formats, and so on, on the fly.

The PDP-10 is a paged, virtual memory, timesharing machine. This means that the users data may not be physically contiguous. Referencing memory is done by a combination of hardware and software. The effect of virtual is that not all of the user's data has to be in core. Some may be on the disk, i.e. not in physical memory. Timesharing means that the user may not be in core at all if another user is running. Each of these can effect an external hardware device doing DMA to a user's program.

The scenario for getting to the AP from a user's program goes as follows: The user calls one of the FPS software library routines to transfer data to or from the AP and to run a program in the AP. The routines set up the necessary data blocks and then issue a call to the PDP-10 monitor (operating system). The monitor saves the users registers and determines which call was requested. If the call is legal, control is transfered to the proper service routine in the monitor. This routine then proceeds to address check all the user's

addresses and to lock down any data and buffer areas that will be DMA accessed by the FPS interface. On completion of checking and locking, the job is marked as I/O active so that the swapper will not move any portion of the job out of physical memory. If a page was on the swapping space and not in physical core then a page fault is issued to get the page into core. Then the interface is sent a start command. After the interface has completed its requested function or the AP has finished running then the monitor can be sent an interrupt. The monitor will continue to honor requests for AP communication until the list is exhausted. The user is marked as I/O complete and may then run, be swapped, and so on. All of these monitor actions represent many instructions and comprise a large amount of overhead. To measure this overhead, various timing tests were performed with the FPS software that was delivered with the machine.

| NUM | PUT | XFR | DIV | WAT | GET | MC |
|---|---|---|---|---|---|---|
| 1 | 2 | Y | Y | Y | 1 | 15 |
| 2 | 2 | N | Y | Y | 1 | 6 |
| 3 | 1 | N | Y | Y | 1 | 5 |
| 4 | 1 | N | N | Y | 1 | 3 |

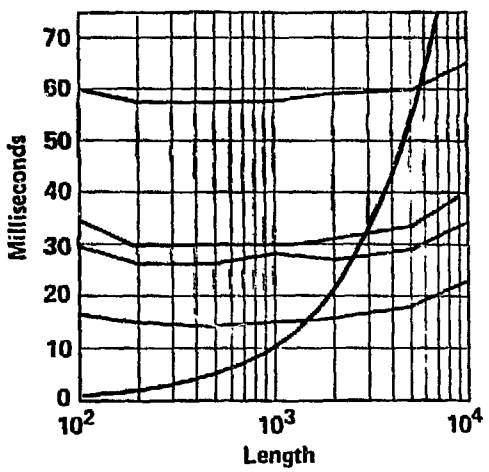



Figure 1

Figure 1 displays various aspects of the timing study. Five curves are shown. The four near horizontal lines represent time versus vector length for AP functions. The curved line is typical of any PDP-10 operation (add, subtract, multiply or divide). The intersection point is the "break even" length for doing the operation on the AP or the PDP-10.

The legend in the upper right hand corner of figure 1 indicates what is represented by each curve. NUM refers to which curve number. PUT is the number of separate calls to the operating system that were done to get the data from the PDP-10 to the AP. XFR (Yes or No) tells whether or not the object code for the AP was shipped to the AP for each execution or not. DIV (Yes or No) tells whether or not the divide routine was actually run in the AP. WAT (Yes or No) states that a wait until completion of running call was issued to the operating system. GET tallies the number of calls to the operating system to get the results back from the AP to the PDP-10. Finally, MC is the total number of calls to the operating system required to accomplish what the previous five columns state. Note that at each length the time required is almost directly proportional to the number of calls to the operating system and is also almost totally independant of the length of the vector. Clearly as the number of calls to the operating system is decreased the "break even" length decreases. The standard software produced by FPS is reflected in curve 2. It does not provide very efficient operation of the AP. Curve 3 was obtained by writing enhancements to the existing software. Curve 4 was also obtained using standard FPS software.

Note that for simple diadic operations the time required to obtain results is almost independent of length for the AP. For the PDP-10 the time is proportional to the length of the vector. Addition, subtraction and multiplication are closely grouped together. Division takes a little longer. With the standard, delivered FPS software the "cross-over" point is seen to be vectors on the order or 2500 to 3000 long. This is only the break even point. Factor of two execution rate improvements are achieved around 7000. Thus very long vectors are required in order to obtain break even values. Another approach might be to do a lot of computation in the AP, such as running a whole program. It turns out that if one is not careful, multiple computations with little data transfer will require more time. Although not shown, this phenomena was observed for the equation V1=V2*V2+V3*V3. This was run on the PDP-10 as one DO loop and on the AP by shipping V2 and V3, then separately doing the two multiplies and one addition, then sending back V1.

A more thorough investigation revealed some of the problem areas. One of these is FPS' strategy of frequent calls to the operating system.

The FPS array processor can be a viable option for providing increased number crunch power to a PDP-10. The software delivered by FPS does not maximize the cost effectiveness and shows that for the greatest advantage to be gained, improved software would have to be developed which would cut down on the operating system overhead associated with using the array processor. The specific improvements would be to chain the DMA transfers and the vector functions together to minimize calls to the operating system.

## Fortran Interface to VEctor programming

In this second part, a software interface to bridge the gap between the user and the AP will be discussed. Very few users at Livermore have the time or incentive to acquaint themselves with a new computer environment. FORTRAN is the major implementation language at the Laboratory. At the time of installation, there existed no AP-FORTRAN so an extension to PDP-10 FORTRAN was defined with the goal that users could program the AP in a natural fashion that would be efficient both for them and the AP. The following is a software interface that is based on the vector and the Scientific Library.

### INITIALIZATION

```
CALL IFIVE(STRING,NUM,IER)
```

STRING is a FORTRAN-like statement which expresses the vector operations to be performed. NUM is an integer which uniquely identifies the STRING. IER is a variable which will be set on encountering an error condition during execution. The following is an example of an IFIVE statement:

```
CALL IFIVE('A=B*(C+D)',28,JERR)
```

The expression is enclosed in quote marks so that it will be treated as a Hollerith string and not be decoded by the compiler. Multiple statement strings are permitted; and results of previous computations may be used in subsequent ones. For instance:

```
CALL IFIVE('A=A+B;VRTA=INVRT(A);C=(A+VRTA)/2.0',7,KER)
```

## ARRAY AND SUBSCRIPT RANGE DEFINITION

CALL DFIVE('ARRAYNAME(DIMENSIONS)',NUM,LER)

CALL DFIVE('R.ARRAYNAME(LOWERBOUND:UPPERBOUND:INC)',NUM,MERR)

DFIVE indicates a definition statement. Dimension and range information about arrays is communicated in this fashion. Variables encountered in IFIVE statements that are not previously defined as arrays will be treated as scalars or simple variables. The R. defines which elements are to be referenced in the vector computations. DFIVE examples follow:

CALL DFIVE('A(0:63),B=C(-J:J,-K:K)',39,NERR)

The array A is defined as one dimensional with 64 elements. The arrays B and C are two dimensional with the values J and K being defined elsewhere. The equal sign separating B and C provides a shorthand for arrays of equal dimensions. Another form permitted in DFIVE defines simple variables:

CALL DFIVE('J=',JX,'K=',KY,41,NFAIL)

This argument list identifies and locates the variables J and K. The specifications given in DFIVE statements are global across all IFIVE statements. All simple variables must be defined prior to their appearance in a range or dimension DFIVE statement. In the example given below, elements number 1 thru 62 would be referenced in the one dimensional array A.

CALL DFIVE('R.A(1:62)',43,JAIL)

## RUNNING

CALL RFIVE(NAMES,NUMBER,IFAIL)

After proper initialization with IFIVE and DFIVE subroutines calls one may supply an argument list consisting of the actual arrays and scalar variables to be operated on as specified by the function defined as NUMBER. If no error occurs, the value of IFAIL will be unchanged. In the multi-computational initialization example, which is repeated below, there must be six arguments supplied. Input array names for A and B as well as output names for VRTA and C are required. Array A is both an output as well as input array name. With few exceptions, vectors appearing in an IFIVE arithmetic string must reference an equal number of elements.

```
CALL IFIVE('A=A+B;VRTA=INVRT(A);C=(A+VRTA)/2.0',7,KER)
CALL RFIVE(XA,XB,XVRT,XC,7,IHOPE)
```

INVRT is a matrix inversion routine. Arguments found in the RFIVE statement must correspond to the dummy arguments in the IFIVE statement in order of their appearance from left to right.

## IMPLEMENTATION

Two approaches to providing the software interface, FIVE, between the PDP-10 and the AP 190L were considered. The first approach ran entirely at execute time. All range, dimension, and function definitions were scanned and the information derived was tabled. When an RFIVE statement was encountered, the information was used to supply the AP with the correct data for computation. It was observed early in this effort that the overhead both to execution speed and table space would be excessive. Therefore, this strategy was abandoned in favor of a pre-compile scan on

the FORTRAN source program containing the FIVE subroutine CALL statements.

The pre-compile scan method looks at the source program and digests the information found in FIVE statements. This information is organized as found in Table 2. Using part of a DFIVE given previously:

```
CALL DFIVE('A(0:63)',59,NERR)
```

an entry for A would be made in the Vector Symbol table with one dimension. The Dimension table would have an entry pointing to the Vector Symbol. The ID Number, 59, and a pointer to the Bounds table would be found. The Bounds table contains an ordered pair for each dimension found in the define statement. The ordered pair entry consist of a pointer to the Scalar Symbol table and a sign. A similar entry would be found for the following range definition:

```
CALL DFIVE('R.A(1:62:2)',60,MYER)
```

The Range table would contain an entry with the ID Number, 60. The Bounds table would contain an ordered triplet for the lower, upper, and increment information and their signs. A simple IFIVE statement will help illustrate other table entries:

```
CALL IFIVE('A=B+C',67,NIX)
```

The vectors A, B, and C are presumed to be defined elsewhere and appropriate entries made. The string A=B+C will be translated to postfix ABC+= and tabled along with the ID Number 67. The string ABC+= is scanned left to right for operators while stacking operands. In the example, A, B, and C would be on the stack when the + operator is encountered. The + operator is binary with two operands, B and C. Therefore, a call to the vector add routine in the Scientific Library is

generated. B and C removed from the stack and replaced with a temporary result. Scanning the postfix resumes and the equal sign is found. This ends the process. A, B, and C are entered in the Argument table. Since A appears on a left hand side of an equal sign, it has a result pointer. The result pointer is from B+C. When the pre-compile scan is complete, all the table information is compressed and written to a disc file. The disc file is then read by the executing program during an initialization phase. During execution, the subroutine calls to the FIVE package are treated differently since some of the information was digested during the pre-compile scan. When a DFIVE call is made, all vectors defined with range and dimension information that match the ID number are activated. The bounds pointers to range and dimension information are placed in the Vector Symbol table. Any simple vaiable definition found is also supplied the current value. IFIVE statements are ignored. When an RFIVE statement is called, the ID number is matched against those in the ID Function Number tab . Pointers to the Argument table and list of AP Scientific Library calls are found. All management of the AP memory is done on the basis of length of vectors, both input and generated temporaries, and number of scalars. The vector lengths are obtained from the dimension and range data pointer to by the dummy variables in the IFIVE statement. When all relocation for the AP Scientific Library calls is complete, transfer of the input is begun. Instead of an individual APPUT for each input, all information about input is collected. This includes the list of relocated Scientific Library calls. The host location, AP location, length, and type of format for each input is sent in one access from the host to the AP. The AP monitor is started and the calling program in the host is

suspended in a wait state. The AP monitor executes all the Scientific Library routine in the relocated list provided by the host. On completion, control is returned to the host and the program waiting is restarted. The input process is now reversed. All information about output is collected and one access is made to transfer the results from the AP to the host.

In an effort to improve the speed of the software package, an attempt is made to perform relocation and memory management of the AP at pre-compile scan time. This can only be done is there are no variable dimension or range definitions. When these conditions are met, much of the overhead due to the FIVE package in the executing program can be eliminated. Only the actual location of the variables in the RFIVE statement need be supplied to start an AP run.

Table 2.

VECTOR SYMBOL

| Name | Dimension ID | Range ID | Number of Dimensions |
|---|---|---|---|

DIMENSION

| Vector Symbol Point | Bounds point | ID Number |
|---|---|---|

RANGE

| Vector symbol Point | Bounds Point | ID Number |
|---|---|---|

SCALAR SYMBOL

| Name | Value/Pointer |
|---|---|

BOUNDS

| Scalar Symbol Point | Sign |
|---|---|

ID FUNCTION NUMBER

| ID Number | Postfix Point | Postfix Length | Arg Point | Arg Length |
|---|---|---|---|---|

ARGUMENT

| Name | Point | Result Point | Loc in Host | Loc in AP | Length |
|---|---|---|---|---|---|
| | | | | | |

FUNCTION

| Name | Number of Inputs | Number of Outputs |
|---|---|---|
| | | |

Timing the FIVE Interface

(1)
```
A=B+C*D
E=F-A*G
H=(G+G)/(1.0+G*G)
P=A+E*H
Q=E-P*G
R=Q+C*D
S=T+Q
T=SQRT(S*S+R*R)
```

and
```
F=E*U+P*W
B=P*U+E*W
```

(2)
```
X=PBC(X+VX,XN)
Y=Y+VY
RHO=SQRT(X,1.0-RPI*(YGC-Y),RHO,QDX)
```

The two examples above were coded in FORTRAN for the PDP-10. Example 1 consists of two parts and represents a portion of a 1-dimensional Boris particle mover. The second is an extract from a 1-dimensional electrostatic particle program. Both benchmarks exhibited the same speed characteristics. A speedup of approximately 6 times was noted when relocation was performed at execution time. With relocation at precompile scan time, a speed up of 17 to 19 times was observed. This includes all overhead. Measurements of the Livermore AP monitor overhead indicates that about 4% AP time is spent in that function for vector lengths of 1000 (see fig. 2).

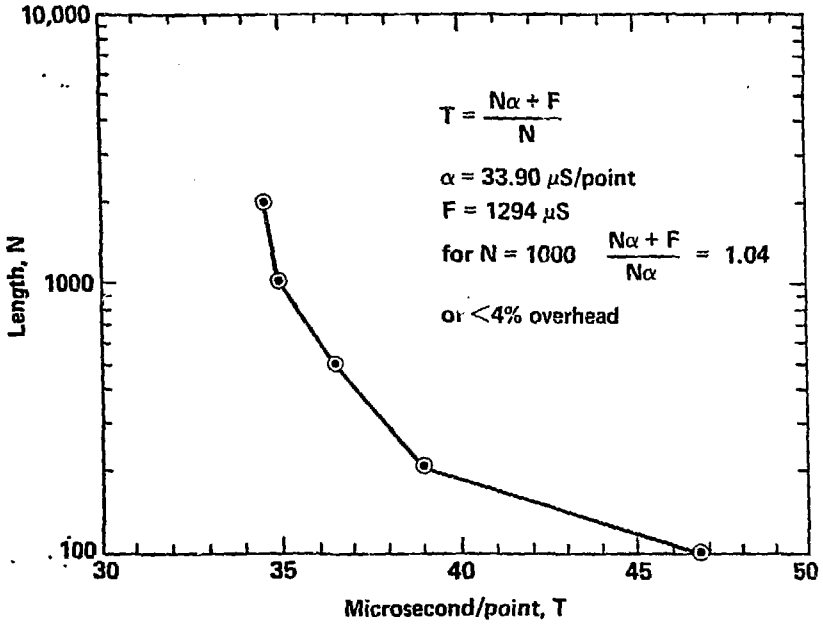



Figure 2

Discussion and Conclusion

The FIVE package has its limitations. Relocation is expensive at execution time. Vectors which are multidimensional and have irregular range variations must be moved into a vector whose elements are evenly spaced. This of course increases overhead, lowering efficiency. There is no common subexpression analysis in the precompile scan. To be fully useful, the AP needs an efficient FORTRAN from a reliable commercial source. We eagerly await the FPS product.